On the Complexity of BWT-runs Minimization via Alphabet Reordering

Jason Bentley∗ Daniel Gibney† Sharma V. Thankachan‡

Abstract

We present some of the first known results on the computational complexity of minimizing the number of runs in the Burrows-Wheeler Transform (BWT) of a string. As fully functional text indexing in space proportional to the number of runs in the BWT of the text has now become a reality, this is currently of great interest. We consider the following problems:

1. Given a string $T$, of length $n$, over alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ and integer $t$, decide if there exists an ordering on $\Sigma$ such that the number of runs in $BWT(T)$ is less or equal to $t$.

2. Given a set of $d$ strings $T_0, T_1, \ldots, T_{d-1}$ each of length roughly $n$, append to each a unique delimiter $\$_i$ and then concatenate the strings, creating the new string $T_0\$T_1\$ \ldots T_{d-1}\$. The aim is then to order the set of delimiters $\{\$0, \ldots, \$_{d-1}\}$ such that the delimiters are lexicographically first in the alphabet $\{\$i : 0 \leq i \leq d - 1\} \cup \Sigma$ and the number of runs in $BWT(T_0\$T_1\$ \ldots T_{d-1}\$)$ is minimized.

3. For Wheeler graphs, we give a natural formulation of the transformation of the directed edge labeled graph to a string. This mapping corresponds to the Burrows-Wheeler Transform when the graph is a path. Now, given a Wheeler graph $G$ with $n$ vertices and an edge alphabet $\Sigma$ and an integer $t$, decide whether there exists an ordering on the sources of $G$ such that the number of runs in $BWT(G)$ is less or equal to $t$.

We prove that the first problem is NP-complete and cannot be solved in time $2^{o(\sigma)n}$ unless the Exponential Time Hypothesis fails. We define a minimization variant of this problem where the cost function is defined as the number of extra runs, which is the number of runs in $BWT(T)$ minus some constant multiple of the obvious lower bound $\sigma$. The results on the first problem then follow from a series of L-reductions from the Minimum Path Cover Problem. These L-reductions demonstrate that the optimization problem is APX-hard and does not admit a $n^{1/2-\varepsilon}$-approximation for all $\varepsilon > 0$ (assuming $P \neq NP$). Following similar techniques, we obtain that the third problem is NP-complete as well. On the other hand, the second problem can be solved in time $O(nd + \sigma^2d)$ time. This is optimal for fixed $\sigma$. We emphasize that this is a worthwhile preprocessing step by demonstrating a case where this algorithm removes a factor $\Omega(\log_\sigma d)$ from the number of runs.

Owing to the rapid growth of large, yet highly-repetitive data sets and the ever increasing interest in its compressed indexing using versatile BWT-based techniques, our problems are of fundamental and practical importance. However, the theoretical bottle necks suggest that heuristics are perhaps the more plausible way to approach them. The significance of the second problem is well understood in the bioinformatics community, however the best attempts are still heuristics. To this end, our new optimal-time algorithm might be able to speed-up some of the existing softwares.

∗Dept. of Mathematics, University of Central Florida, Orlando, USA. e-mail: jason.bentley@ucf.edu
†Dept. of CS, University of Central Florida, Orlando, USA. e-mail: daniel.j.gibney@gmail.com
‡Dept. of CS, University of Central Florida, Orlando, USA. e-mail: sharma.thankachan@ucf.edu
1 Introduction and Related Work

The Burrows-Wheeler Transform (BWT) [6] and the FM-index [10] are ubiquitous in bioinformatics and other text processing applications [20, 22, 23, 28]. Improvements to the algorithmic aspects of this transformation and related data structures can therefore have a significant impact on the research community. With the most recent developments, the number runs in transformed texts has become of great importance [5, 3, 13, 18, 19, 30]. One reason for this is that BWT of a highly repetitive text contains long runs of repeated characters, making it amenable to Run-Length compression. Another reason is the most recent advancements made by the data-structure community.

Although BWT based indexes have existed for many years [29], only recently has the property that the BWT of text frequently contains long runs been exploited to such a great extent. The first Run-Length FM-index was developed by Mäkinen and Navarro in 2005 [24]. However, this index lacked the ability to locate the occurrences of a pattern within space bounded in terms of the number of runs. Only in 2018 was the more functional Run-Length FM-index developed by Gagie, Navarro, and Prezza introduced [13]. It is among the most powerful new results, providing most of the functionality of the traditional FM-index with space bounded in terms of the number of runs. Since its introduction, fully functional indexes using space proportional to the number of runs have also now appeared in [12]. The complexity of a recent almost-optimal BWT construction algorithm is also parameterized on the number of runs [16]. Given the importance of space efficient indexes and increasing amount of text repetition found in most applications, this line of research will undoubtedly continue to grow in the coming years. One effort which will directly contribute to this research is determining which techniques can be used to reduce the number of runs in the BWT.

Perhaps the most natural way to try to minimize the number of runs is to change the lexicographic ordering assigned to symbols from its alphabet. This has the advantage of not requiring any changes to the text nor requiring modifications to the indexing algorithm and data structure. To see that this can have an impact on the number of runs in the BWT of text $T$, consider as an example the string *mississippi* with the usual ordering ($i < m < p < s$). In this case, its BWT has eight runs. But, with the order $s < i < p < m$, its BWT has only six. This problem of reordering the alphabet is clearly fixed-parameter tractable in $\sigma$, the alphabet size and has a trivial $O(\sigma! \ n)$ time solution. This may be adequate for DNA-sequences, but when the alphabet is even slightly larger as in protein sequences or in English language, one would desire something more efficient.

A second, more constrained, way of attempting to minimize the number of runs arises when we consider the BWT of $T = T_0 \$_0 T_1 \$_1 \ldots T_{d-1} \$_{d-1}$, formed by appending the $d$ strings $T_0$ through $T_{d-1}$ with *distinct delimiters* and then concatenating them together. By finding an optimal ordering on these delimiters we can again reduce the number of runs in $BWT(T)$. We will see a simple example in Section 5 in which an optimal delimiter ordering removes a factor of $\Omega(\log_\sigma d)$ over the naive delimiter ordering $\$_0 < \ldots \$_{d-1}$. The trivial solution takes $O(d! \ n)$ time. However, several applications in bioinformatics call for a faster solution. E.g. compression/indexing of sequencing data, which is a collection of billions of reads, where tools like BEETL currently rely on heuristics [7].

Finally, we consider the nascent subject of Wheeler graphs [11]. They are edge-labeled directed graphs which can model many BWT index-able structures. We define from a linear layout of the vertices of a Wheeler graph $G$ a string $BWT(G)$. This string generalizes the BWT on a text, which in terms of Wheeler graphs forms a path. With this definition, the problem of ordering the alphabet to minimize the runs in $BWT(G)$ can be posed for Wheeler graphs. For the problem of delimiter ordering, the most natural formulation is to try to find an optimal ordering on the sources of $G$. 


2 Results

For the following problems we consider all strings to be over an alphabet \( \Sigma = \{0, 1, \ldots, \sigma - 1\} \). A run in a string \( T \) is a maximal unary sub-string. Let \( \rho(T) \) be the number of runs in a string \( T \).

**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 \( \Sigma = \{0, 1, \ldots, \sigma - 1\} \) such that \( \rho(BWT(T)) \leq t \).

**Theorem 1.** AO is NP-complete.

As one would guess, we construct this hard instance by employing an extremely large alphabet. However, for general alphabet, we obtain the following result by combining the reduction used to prove Theorem 1 and the widely accepted (yet unproven) Exponential Time Hypothesis (ETH).

**Corollary 1.** AO cannot be solved in time \( 2^{\omega(\sigma)n} \), unless the Exponential Time Hypothesis fails.

For the minimization problem corresponding to AO we consider the cost function to be the number of extra runs, which is \( \rho(BWT(T)) \) minus a multiple of \( \sigma \), the lower bound on the number of runs. Specifically the cost function is \( c_{AO_k} = \rho(BWT(T)) - k\sigma \) for a parameter \( k \). We denote this optimization problem as AO\(_k\) and we have the following result.

**Theorem 2.** AO\(_{1+\delta}\) is APX-hard for any constant \( \delta > 0 \).

This implies that every problem in the complexity class APX can be reduced to AO\(_{1+\delta}\). One consequence is that no polynomial-time approximation scheme (PTAS) is possible, assuming \( P \neq NP \). In fact, we can prove an even stronger result regarding the inapproximability of AO\(_{1+\delta}\).

**Theorem 3.** For any constants \( \varepsilon > 0 \) and \( \delta > 0 \) there does not exist a polynomial time \( n^{1/2-\varepsilon} \)-approximation algorithm for AO\(_{1+\delta}\), assuming \( P \neq NP \).

We now present another problem on run minimization and provide a positive result.

**Problem 2** (Delimiter Ordering). Given a set of \( d \) strings \( T_0, \ldots, T_{d-1} \) each of length most \( n \), append string \( T_i \) with the new symbol \( \$_i \) \( (0 \leq i \leq d-1) \) and find a ordering \( \pi \) on the symbols \( \$_i \) such that \( \$_\pi(0) \prec \$_\pi(1) \prec \ldots \prec \$_\pi(d-1) \prec 0 \ldots \prec \sigma - 1 \) and \( \rho(BWT(T_0\$_0 T_1\$_1 \ldots T_{d-1}\$_{d-1})) \) is minimized.

In Section 5, we provide an example where an optimal ordering of delimiters removes a factor of \( \Omega(\log_\sigma d) \) in the number of runs, demonstrating that this can be a worthwhile preprocessing step.

**Theorem 4.** Delimiter Ordering can be solved in \( O(nd + \sigma^2d) \) time.

Note that our algorithm runs in optimal linear time when all strings are of length roughly the same and \( \sigma = O(1) \), as in a typical high-throughput DNA sequencing experiment data, which consists of several billions of short DNA fragments called reads. Therefore, we refer to [7] for an immediate use case of our result in bioinformatics. Also, we note that if we modify Problem 2 to have the objective of minimizing the number of runs in the extended-BWT defined in [25], that is \( eBWT(T_0, \ldots, T_{d-1}) \), we obtain the same time bounds for that transformation as well.

As an extension of Delimiter Ordering we consider the Source Ordering Problem (SO) on Wheeler graphs. Here the goal is to order the source vertices of a Wheeler graph \( G \) in order to minimize the number of runs in the induced string denoted by \( BWT(G) \) (the exact definition of \( BWT(G) \) is deferred to Section 4 and Appendix A). In contrast to the Delimiter Ordering Problem on the BWT, Source Ordering on Wheeler graphs is computationally difficult.
Problem 3 (Source Ordering(SO)). Given a Wheeler graph $G$ and an integer $t$, decide whether there exists an ordering of the sources such that $\rho(BWT(G)) \leq t$.

Theorem 5. SO is NP-complete.

The techniques used for the reductions in this paper may be of independent interest. They make (as far as the authors are aware) novel use of a parameter $\delta$ within the minimization problems. By manipulating $1 + \delta$ within the cost function as a coefficient on the natural lower bound of a solution we gain further insight into the inapproximability of the optimization problems.

Roadmap. We start with preliminaries in Section 3. We prove Theorems 1, 2, 3, 5 and Corollary 1 in Section 4. In Section 5 we provide an algorithm proving Theorem 4. Finally, in Section 6 we discuss some open problems related to Alphabet Ordering.

3 Preliminaries
3.1 The BWT

The BWT of a string $T$ 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 $BWT(T)$. See Figure 1 for an example. To make the transformation invertible a new symbol $\$ is appended to $T$ prior to sorting the circular shift. Historically, the BWT was introduced by Burrows and Wheeler for the purposes of compression [6]. It was later adapted into a full-text index in the seminal work of Ferragina and Manzini [10] and has since become an essential tool in bioinformatics. The key observation which makes the FM-index possible is the LF-mapping. The LF-mapping maps a symbol in the column $L$ to its corresponding position in the column $F$. By exploiting the property that the range of rows becomes smaller as we match a pattern from right to left, a full-text index of size (in bits) proportional to $n \log \sigma$ [10] or even $\rho(BWT(T))$ can be obtained [13]. Both the BWT of a string and the FM-index can be constructed in time linear or even faster [4, 16, 17, 27].

One observation we make now is that if we were to write out the directed path obtained by following the LF-mapping from row to row, creating a vertex for each row, and marking each vertex with the symbol seen in the $L$ column, we would get a directed path with vertices labeled with the reverse of the string $T$. For example, in Figure 1 if we start from the first row, we obtain the path $i \rightarrow p \rightarrow p \rightarrow i \rightarrow s \rightarrow s \rightarrow i \rightarrow s \rightarrow s \rightarrow i \rightarrow m \rightarrow \$.

Figure 1: $BWT(\text{mississippi}\$) is in column $L$. The LF-mapping is shown in the right figure.
3.2 L-reductions

Many of the hardness results related to approximation in this work use L-reductions [8]. For the optimization problems presented in this paper, the goal is to minimize the cost of the solution. We will use the following notation throughout this paper:

- \( \text{OPT}_A(x) \) 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 \( R_A(x,y) = \frac{c_A(y)}{\text{OPT}_A(x)} \), which is \( \geq 1 \).
- \( f_A(x) = x' \) is the mapping of problem instance \( x \) of Problem \( A \) to instance \( x' \) of Problem \( B \).
- Letting \( y' \) be a solution to problem instance \( x' = f_A(x) \), we let \( g_B(y') = y \) denote the mapping of a solution \( y' \) to a solution \( y \) for instance \( x \).

As a result, \( R_B(x',y') = 1 + r \) implies \( R_A(x,y) \leq 1 + \alpha \beta r = 1 + O(r) \). The L-reductions preserve APX-hardness [31]. Note that if \( c_A(y) = c_B(y') \) then Conditions (i) and (ii) are satisfied.

4 Hardness of Alphabet Ordering

Technical Overview: The hardness results are rooted at hardness of Minimum Path Cover (MPC). We start by showing that the decision version of MPC is NP-complete, and the minimization variant is APX-hard and inapproximable. Using a modified version of problem called Column Ordering\(^{†} \) (CO\(^{†}\)) as an intermediate step, we develop an L-reduction from MPC to AO\(_{1+\delta}\). The NP-completeness, APX-hardness and inapproximability results all follow from this L-reduction.

Problem 4 (Minimum Path Cover (MPC)). Given an undirected graph \( G = (V,E) \) find a minimum sized set of vertex-disjoint paths such that every vertex of \( G \) is in some path.

We begin by proving some hardness results related to MPC. The NP-completeness of MPC is immediate from a simple reduction from the Hamiltonian Path Problem to MPC. The cost function for MPC, denoted \( c_{MPC} \), is the number of paths in the path cover.

Lemma 1. MPC is APX-hard.

Proof. By a result by Papadimitriou and Yannakakis, the Traveling Salesperson Problem on an undirected graph with edge weights 1 and 2 (known as (1,2)-TSP) is Max-SNP-hard [32]. Being Max-SNP-hard implies being APX-hard [26]. Let \( c_{TSP} \) be the sum of edge weights in the Traveling Salesperson tour. Given an instance \( x \) of (1,2)-TSP in a complete graph \( G = (V,E) \), we construct a graph \( G^* \) by deleting the edges with weight 2. We consider the graph \( G^* \) as an instance \( f_{TSP}(x) \) of MPC. We transform a solution \( y' \) of \( f_{TSP}(x) \) to a solution \( g_{MPC}(y') \) of (1,2)-TSP by connecting the solution’s paths with edges of weight 2 to obtain a tour. Then \( f_{TSP} \) and \( g_{MPC} \) form an L-reduction since letting \( p^* = \text{OPT}_{MPC}(f_{TSP}(x)) \) and \( p = c_{MPC}(y') \) we obtain
• \( \text{OPT}_{\text{MPC}}(f_{\text{TSP}}(x)) = p^* \leq |V| + p^* = 1(|V| - p^*) + 2p^* = \text{OPT}_{\text{TSP}}(x) \).

• \(|\text{OPT}_{\text{TSP}}(x) - c_{\text{TSP}}(g_{\text{MPC}}(y'))| = ||V| + p^* - (|V| + p)| = |p^* - p| = |\text{OPT}_{\text{MPC}}(f_{\text{TSP}}(x)) - c_{\text{MPC}}(y')|\).

\[
\]

Once we provide the appropriate L-reduction, the next lemma will be used to prove Theorem 3.

**Lemma 2.** There exists no polynomial time \(|V|^{1-\varepsilon}\)-approximation for MPC, given \( P \neq NP \).

![Figure 2: \((G')^n\) has a path cover of size one if \( G \) has Ham. path and at least \( n^j + 1 \) otherwise.](image)

**Proof.** Fix \( \varepsilon > 0 \). We reduce the Hamiltonian Path Problem to a \(|V|^{1-\varepsilon}\)-approximation for MPC. Take \( j \) large enough that \((j - 1)/(j + 1) > 1 - \varepsilon \). Given a graph \( G = (V, E) \) as input to the Hamiltonian Path Problem with \(|V| = n\), we create a new graph \( G' \) by adding vertices \( s, s', t, t' \), adding edges to make \( s' \) and \( t' \) adjacent to all vertices in \( G \), and creating two edges \((s, s')\) and \((t', t)\). Duplicate this graph \( n^j \) times and create a graph \((G')^n\) by chaining these graphs together, bringing the \( s \) vertex into correspondence with the \( t \) vertex in the preceding graph (for all but the first graph). See Figure 2. Let \( n' = O(nn^j) \) be the number of vertices in \((G')^n\). If \( G \) has a Hamiltonian path then the minimum path cover of \((G')^n\) contains only one path. Otherwise, it contains at least \( n^j + 1 \) paths, which is greater than \( n^{n^j-\varepsilon} \) for \( n \) large enough.

Let \( G \) be an instance \( x \) of MPC. Let \( G' \) be constructed from \( G \) as in the proof for Lemma 2. We label the modified instance of \( x \) as an instance \( f_{\text{MPC}}(x) \) of Minimum Path Cover\(^\dagger\)(MPC\(^\dagger\)). From a solution \( y' \) of \( f_{\text{MPC}}(x) \) we obtain a solution \( g_{\text{MPC}^\dagger}(y') \) for \( x \) by removing from the path cover in \( y' \) any edges and vertices not in \( G \). Let \( c_{\text{MPC}} \) and \( c_{\text{MPC}^\dagger} \) be the number of paths in the path covers of their respective graphs.

**Lemma 3.** The functions \( f_{\text{MPC}} \) and \( g_{\text{MPC}^\dagger} \) form a L-reduction, i.e., MPC is L-reducible to MPC\(^\dagger\).

**Proof.** It can be easily verified that if \( G \) can be covered by \( p \) paths then \( G' \) can also be covered by \( p \) paths, and vice versa. Therefore \( c_{\text{MPC}^\dagger}(y) = c_{\text{MPC}}(y') \).

Let \( G' = (V', E') \) be an instance of MPC\(^\dagger\). We construct an incidence matrix with \( m = |E'| \) rows and \( n = |V'| \) columns. Let \( c_s \) and \( c_t \) refer to the columns for vertices \( s \) and \( t \). Let \( \ell \) be any value greater than \( 4m \). Add to this matrix \( \ell \) rows with only 1’s in \( c_s \) and \( \ell \) rows with only 1’s in \( c_t \). The other entries in each of these rows are each 0. We arrange the rows of the matrix such that these added \( 2\ell \) rows are the bottom-most and alternate between 1’s in column \( c_s \) and 1’s in column \( c_t \). Additionally, we ensure that the alternation begins with a row containing 0 in \( c_s \). Also, we arrange the rows of the matrix such that the first row and the second row are for the edges \((s, s')\) and \((t, t')\) respectively. We call this matrix \( M \). See Figure 3 for an illustration.

**Definition 1.** Let \( L(M_\pi) \) denote the string created by ordering \( M \)'s columns by \( \pi \) and then concatenating the rows from top to bottom.
We have constructed the input to a special instance of the problem we call Column Ordering.

**Problem 5 (Column Ordering (CO)).** Find an ordering \( \pi \) of the columns of matrix \( M \) such that \( \rho(L(M_\pi)) \) is minimized.

We call a special instance of Column Ordering constructed from an instance of \( \text{MPC}^\dagger \) as Column Ordering\(^\dagger\)(CO\(^\dagger\)) and assume in addition to \( M \) that we know the parameters \( m, n, \) and \( \ell \). Let \( x \) be an instance of \( \text{MPC}^\dagger \) and let \( f_{\text{MPC}}(x) \) be the reduction to an instance of CO\(^\dagger\) as described above. We say an entry in \( M \) contributes a run if it is the start of a run in \( L(M_\pi) \). A solution \( y' \) of \( f_{\text{MPC}}(x) \) is mapped to a solution \( g_{\text{CO}}(y') \) of MPC\(^\dagger\) by taking adjacent columns which contain two 1’s in the same row as indicating vertices adjacent in a path. The cost function \( c_{\text{CO}} \) is defined as \( \frac{1}{2}(\rho(L(M_\pi)) - 4m - 2\ell + 2n) \).

This is possible only because we know an instance of CO\(^\dagger\) is derived from an instance of MPC\(^\dagger\) so that \( m, n, \) and \( \ell \) are defined. We now employ a similar tack to \([21]\).

**Lemma 4.** The functions \( f_{\text{MPC}} \) and \( g_{\text{CO}} \) form a L-reduction, i.e., Minimum Path Cover\(^\dagger\) is L-reducible to Column Ordering\(^\dagger\).

**Proof.** We divide this into two cases.

**Case:** \( c_s \) and \( c_t \) are on the boundaries of \( M \). For a column ordering \( y' = \pi \) with \( c_{\text{MPC}}(g_{\text{CO}}(y')) = p \) paths, each edge belonging to a path causes its row to contribute two runs (except the second row has one fewer contribution and another row has an additional contribution). Each edge not belonging to a path causes its row to contribute four runs. Each of the \( 2\ell \) added rows contribute one run. This gives \( \rho(L(M_\pi)) = 2(n - p) + 4(m - (n - p)) + 2\ell = 2\ell - 2n + 4m + 2p \), implying that \( c_{\text{CO}}(y') = p = c_{\text{MPC}}(g_{\text{CO}}(y')) \).

**Case:** at least one of \( c_s \) or \( c_t \) are not on the boundary of \( M \). There are at least \( 3\ell > 2\ell + 4m \) runs, implying a cost that exceeds \( n \), the maximum number of paths in any path cover of \( G' \). Hence, \( \text{OPT}_{\text{MPC}}(x) = \text{OPT}_{\text{CO}}(f_{\text{MPC}}(x)) \) (with an optimal solution having \( c_s \) and \( c_t \) on the boundaries) and \( c_{\text{MPC}}(g_{\text{CO}}(y')) - \text{OPT}_{\text{MPC}}(x) \leq n - \text{OPT}_{\text{MPC}}(x) \leq c_{\text{CO}}(y') - \text{OPT}_{\text{CO}}(f_{\text{MPC}}(x)) \).

The transformation \( f_{\text{CO}} \) from an instance \( x \) of CO\(^\dagger\) to an instance \( f_{\text{CO}}(x) \) of AO\(_{1+\delta}\) is described next. Given a modified \( m' \times n \) matrix \( M \), where \( m' = m + 2\ell \), we will output a string \( T \). We describe \( T \) in terms of its sub-strings. The substrings are created by iterating through the matrix \( M \) by column then row and outputting substrings as follows:

- For \( 1 \leq j \leq n, 1 \leq i \leq m' \): if \( M_{i,j} = 1 \) output the substring \( 10^{i+2}2C_j \)
- For \( 1 \leq j \leq n \): output the substring \( 0^{m'+2}2C_j \)
- Append to each substring created above a unique \( S_i \) symbol (\( 1 \leq i \leq 2m + 2\ell + n \)).
- Create new symbols \( b_1, b_2, \ldots, b_z \) (\( z \) to be specified later).
The string $T$ is the concatenation of these substrings and symbols in any order. The alphabet set $\Sigma$ is
\{0, 1, 2\}∪\{C_1, C_2, \ldots, C_n\}∪\{$\$\}$∪\{b_1, b_2, \ldots, b_x\} and its size $\sigma = 3+2n+2m+2\ell+z$.

The idea behind this reduction is that the symbol $C_j$ corresponds to the $j^{th}$ column of $M$. In an
optimal ordering of $\Sigma$ the ordering given to the symbols $C_1, C_2, \ldots, C_n$ gives an optimal ordering
of the columns of $M$. This establishes Theorem 1, the NP-completeness of $AO$. For inapproximability
we establish Lemma 5 using the mapping $g_{AO_{1+\delta}}$ from a solution $y'$ of $CO^\dagger(x)$ to a solution $y$ of $x$
which takes the relative ordering given to the symbols $C_j$ as the ordering on the columns of $M$.

**Lemma 5.** For any constant $\delta > 0$, the functions $f_{CO^\dagger}$ and $g_{AO_{1+\delta}}$ form an L-reduction, i.e., $CO^\dagger$
is $L$-reducible to $AO_{1+\delta}$.

**Proof.** The following terminology is used within the proof (see Figure 4 for an illustration).

- We refer to the portion of the $L$ column where the corresponding rows of $F$ contain all of the
  ‘a’ symbols as the $a$-block.
- Let $C_s$ and $C_t$ be the symbols in $\Sigma$ corresponding to the columns $c_s$ and $c_t$. We say that
  $C_s, C_t$ are *extremal* (in the alphabet ordering) if for all $j$ either $C_s \leq C_j \leq C_t$ or $C_t \leq C_j \leq C_s$.
- A *simulated row* refers to the portion of the $L$ column which corresponds to a row of $M$. This
  is equivalent to the row of $M$, with the same number of runs but differing in the lengths of
  runs of 0. Alternatively, it can be seen as all 0 and 1 symbols (across all substrings as defined
  above) which have the same distance from the next $\$$ symbol on the right.

We will show that the following properties can be assumed about a solution $y'$ to $f_{CO^\dagger}(x)$: In
general, we only need to consider solutions $y'$ with the following properties:

- For a fixed $j$, all $C_j$ are placed adjacently in $BWT(T)$;
- All 2’s are placed adjacently in $BWT(T)$;
- The symbol 2 is ordered as the character preceding 0;
- For any ordering on the column symbols $C_j$, the distinct $\$$ symbols are ordered in such a way
  as to minimize the number of runs of 1 in the 0-block.

Suppose there’s some solution $y''$ where one of the above properties does not hold. Then we can
replace the arbitrary solution $y''$ with some better solution $y'$ (w.r.t the cost function $c_{AO_{1+\delta}}$) while
maintaining that $g_{AO_{1+\delta}}(y'') = g_{AO_{1+\delta}}(y')$. Therefore these conditions always hold in an optimal
solution of $f_{CO^\dagger}(x)$. Also, if the L-reduction Condition (ii) holds on $y'$ it holds on $y''$ as well.

We consider each of the properties mentioned above and show that this is the case. If $y''$
is a solution where for a fixed $j$ the symbols $C_j$ are not placed adjacently, by reordering the
unique $\$$ symbols we can obtain $y'$ that has matching $C_j$ symbols placed adjacently in the BWT
while at least maintaining the same number of runs elsewhere, causing $c_{AO_{1+\delta}}(y'') > c_{AO_{1+\delta}}(y') \geq
OPT_{AO_{1+\delta}}(f_{CO^\dagger}(x))$. By similar reasoning, we can assume all 2’s in the BWT are placed adjacently.
We may assume that symbol 2 is ordered adjacent to 0 in the alphabet ordering due to the run of
0’s occurring at the corresponding side of the 0-block. The final property can be assumed since
each $\$$ symbol contributes one run regardless of its place in the ordering.
Figure 4: Directed paths corresponding to the matrix $M$ traversing the LF-mapping. With $z = 4$, the string could be $T = b_4b_3b_2b_1000002C_2D_10002C_2D_3000002C_1D_210002C_1D_1$. We consider the $b$ symbols as ‘pumping’ the alphabet and the symbol 2 has forcing the simulated rows to move in the same direction across the 0-block. Note that in an optimal solution $D_3$ and $D_4$ would switch their relative order to create a single run of 1’s in the row simulation.
Observation 1. The Column Ordering problem on arbitrary binary matrices cannot be reduced to Alphabet Ordering using the techniques presented here, because the simulated rows cannot simulate arbitrarily long runs of 1. However, the specifically designed instances of Column Ordering, CO†, given here circumvent this issue. When $C_s$ and $C_t$ are extremal with $C_s < C_t$, our matrix linearization will only have 1-runs of length two or less; hence, the only difference between the 0-block and $L(M_\pi)$ are the lengths of the 0-runs. Suppose we have a solution $y''$ with $C_s, C_t$ extremal and $C_t$ ordered before $C_s$. Then there also exists a solution $y'$ where $C_s, C_t$ are extremal, $C_s$ is ordered before $C_t$, $c_{AO_{1+\delta}}(y') \leq c_{AO_{1+\delta}}(y'')$ and $g_{AO_{1+\delta}}(y'') = g_{AO_{1+\delta}}(y')$. This ordering can be found by reversing the ordering given to the alphabet in the solution $y''$. This follows because the first two rows may create a (non-simulatable) run of four 1’s in $L(M_\pi)$ with $C_t$ at the beginning, whereas no such run can exist with $C_s$ at the beginning. The remaining number of runs is maintained. Hence, we can always consider $C_s < C_t$ whenever $C_s$ and $C_t$ are extremal.

Cost function: Let $r_0$ be the number of runs created in the 0-block with a solution $y'$. By the properties established above we can assume every solution $y'$ to $AO_{1+\delta}$ that we will consider has $\rho(BWT(T)) = r_0 + \sigma - 1$. This is since every character except 1 contributes exactly once to the run count outside the 0-block. This implies

$$c_{AO_{1+\delta}}(y') = r_0 + \sigma - 1 - (1 + \delta) \sigma = r_0 - \delta \sigma - 1.$$

Condition (i) of L-reductions: Let $\pi^*$ be an optimal solution to the instance $x$ of $CO^\dagger$ and $\rho^* = \rho(L(M_{\pi^*}))$. In an optimal solution of $f_{CO^\dagger}(x)$ the symbols $C_s$ and $C_t$ must be extremal. Otherwise the 2ℓ bottom simulated rows contribute at least 3ℓ runs to $r_0$ which is more than the worst case $2\ell + 4m$ contributed otherwise. Observation 1 then implies

$$OPT_{AO_{1+\delta}} = \rho^* - \delta \sigma - 1.$$

We need to show that there exists a constant $\alpha > 0$ such that $\rho^* - \delta \sigma - 1 \leq \alpha OPT_{CO^\dagger}(x) = (\alpha/2)\left(\rho^* - 4m - 2\ell + 2n\right)$. Set $\alpha = 2$ and ‘pump’ the alphabet size by making $z > (4m + 2\ell)/\delta$. Then it holds that $4m + 2\ell - 2n - 1 \leq \delta \sigma = \delta(3 + 2n + 2m + 2\ell + z)$.

Condition (ii) of L-reductions: Let $\pi = g_{AO_{1+\delta}}(y')$ and $\rho = \rho(L(M_\pi))$. When $C_s$ and $C_t$ are not extremal in $y'$ the number runs $r_0$ created in a simulated row layout in the 0-block for $y'$ is at least $\rho$. When $C_s$ and $C_t$ are extremal in $y'$, by Observation 1 $r_0$ is equal to $\rho$. Therefore,

$$c_{CO^\dagger}(g_{AO_{1+\delta}}(y')) - OPT_{CO^\dagger}(x) = \frac{1}{2}(\rho - 4m - 2\ell + 2n) - \frac{1}{2}(\rho^* - 4m - 2\ell + 2n) \leq (r_0 - \delta \sigma - 1) - (\rho^* - \delta \sigma - 1) = \beta(c_{AO_{1+\delta}}(y') - OPT_{AO_{1+\delta}}(f_{CO^\dagger}(x))),$$

where $\beta = 1$.

This completes the proof of Lemma 5.

4.1 Wrapping Up

An L-reduction from MPC to $AO_{1+\delta}$ combined with Lemma 1 completes the proof for Theorem 2. To prove Theorem 3, recall that (1,2)-TSP was only required for the APX-hardness of MPC.
The following results depend on only the NP-completeness of Hamiltonian Path problem. The Hamiltonian Path problem is still NP-complete on planar graphs, hence we can assume $|E| = m < 3|V| = O(n)$. Recall in the proof of Lemma 5 we create a string $T$ of length $\Theta(mn + m/\delta)$ which is now $\Theta(n^2)$.

An $(1 + n^{1-\epsilon})$-approximation for $AO_{1+\delta}$ provides a $(1 + O(n^{1-\epsilon}))$-approximation for $MPC$. By Lemma 2 for $0 < \epsilon < 1$ having a polynomial time $n^{1-\epsilon}$-approximation for $MPC$ is not possible (if $P \neq NP$), it follows that having a polynomial time $n^{1-\epsilon}$-approximation algorithm for $AO_{1+\delta}$ is not possible. Since $|T| = \Theta(n^2)$, a $|T|^{1/2-\epsilon/2}$-approximation for $AO_{1+\delta}$ provides a $n^{1-\epsilon}$-approximation for $MPC$. This completes the proof of Theorem 3.

We finally prove Corollary 1. Assuming the exponential time hypothesis (ETH), there exists no $2^{o(n+m)}$ time algorithm for Hamiltonian Path Problem [9], and hence no $2^{o(n+m)}$ time algorithm for $MPC$. Since the alphabet size $\sigma$ used in our reduction to $AO_{1+\delta}$ is linear in $n+m$, an $2^{o(\sigma)}n$ time algorithm for $AO_{1+\delta}$ would imply an $2^{o(n+m)}$ time algorithm for Hamiltonian Path.

**Source Ordering on Wheeler Graphs** We present a brief overview here and details are deferred to Appendix A. To define the BWT transform from a Wheeler graph $G$ to a string, $BWT(G)$, we assume a proper ordering on the vertices. We label each vertex in $G$ by its departing edge labels. If a vertex has multiple edge labels leaving it, we consider all possible orderings of its labels and take the one which gives the minimal number of runs. In the proof of Lemma 5, by replacing the $C_i$ symbols with sources and constructing the same paths leaving each source as the paths leaving $C_i$ we can obtain Theorem 5. An optimal ordering on the sources provides a minimum ordering of the columns of $M$.

**5 Delimiter Ordering**

Recall that we wish to find an ordering on the symbols $\$0, \ldots, \$d-1$ such that the number of runs in the BWT of $T = T_0\$0 \ldots T_{d-1}\$d-1$ is minimized. We transform this problem into the problem of ordering a set of paths. This is done by creating a directed path for each substring, $T_i$, and placing these paths in a tree structure where the vertices are grouped into ‘blocks’ which are determined by the labels on their incoming path. The Delimiter Ordering problem then becomes equivalent to finding the starting position of these paths in the root block of the tree. See Figure 5 for an example of such a tree. We wish for the ordering given to the paths to minimize the number of runs within blocks, as well as the number of runs between adjacent blocks. We formally define these ideas.

For the string $T_i = t_1t_2 \ldots t_n$ we consider the directed path $P_i$ with vertices labeled from beginning to end with symbols $t_n \rightarrow t_{n-1} \ldots \rightarrow t_1 \rightarrow \$i-1$, where $\$i-1 = \$d-1$ when $i = 0$. For a vertex $v$ in $P_i$, let $str(v)$ denote the string formed by concatenating labels on path $P_i$ from beginning up to, but excluding $v$ (the empty string is possible). The block with label $s$, or $B_s$, is defined as $B_s = \{v \in V : str(v) = s\}$. We consider a block $B_s$ as

\[\begin{figure}
\text{Figure 5: The tree for the sting $BWT(T)$ where $T = 00\$010\$111\$2021\$30002\$4202\$5$. The order of each block is shown in larger font.}
\end{figure}\]
having a block child $B_{sa}$ which consists of all the vertex being mapped to $B_{sa}$ from $B_s$ with the label $a$. The root of the tree is the block with the empty string as a label. The blocks are ordered by the lexicographic order of the reverse of their strings.

The blocks of $T = T_0s_1 ... T_{d-1}s_{d-1}$ can be determined in linear time. This can be done by (i) choosing an arbitrary order on the delimiters, (ii) constructing the BWT while maintaining the original text position in $T$ for each entry, and (iv) constructing the longest common extension structure for $T$. A longest common extension query takes as arguments two indices $i$ and $j$ and returns the length of the longest common sub-string of $T$ starting at $i$ and $j$. The data structure can be constructed in linear time and performs queries in constant time [15]. In a linear scan of $BWT(T)$, we can identify where blocks begin using the longest common extension structure. See Algorithm 1 for more details.

Lemma 6. For any tree as constructed above with $d$ paths, the number of blocks with multiple block children is at most $d - 1$.

Notice that within each block we may permute the ordering of the vertices so that vertices with the same label are consecutive within a block with no effect on the ordering of vertices in the rest of the tree. Therefore, in finding the optimal ordering, we may view each block as a “tuple”, each holding a subset of alphabet symbols appearing only once per tuple. The order in which the tuples are listed is determined by the ordering on the blocks. For example, in Figure 5 the resulting tuples are $(0, 1, 2)(0, 1)(5)(3)(0)(2)(1, 2)(8)(1)(0)(4)(0)$. We define a new problem:

Problem 6 (Tuple Ordering (TO)). Given a list of tuples $t_1, ... , t_q$ in a fixed order, each containing a subset of symbols from $\Sigma$, order the elements in each tuple such the total number of runs in the string formed by their concatenation $t_1 \cdot t_2 \cdot ... \cdot t_q$ is minimized.

Clearly, the problem of minimizing the total number of runs in $BWT(T)$ is equivalent to maximizing the total number of adjacent matches between the tuples formed as above. To maximize the number of adjacent matches we start by constructing a DAG $G$ which we call the tuple graph. We construct $G$ as follows: for each tuple $t_i$, create two identical sets of vertices $L_i$ and $R_i$. Both $L_i$ and $R_i$ contain vertices corresponding to all of the symbols in $t_i$. If $t_i$ has more than one element we create a directed edge between a vertex in $L_i$ and a vertex in $R_i$ if their labels are distinct. If $t_i$ has only one element, then we create an edge between the corresponding vertex on both sides. We call these consistency edges. Between tuples $R_i$ and $L_{i+1}$ for $1 \leq i \leq q - 1$ we create edges between the vertices in $R_i$ and $L_{i+1}$ if they represent the same symbols. We call these adjacency edges. The aim is now to maximize the number of adjacency edges possible in a set of paths whose union contains exactly one element from every $L_i$ and $R_i$. Once such a set of paths is found we obtain the ordering for each tuple by observing which vertex is taken for each $L_i$ and $R_i$. The vertex taken for $L_i$ ($R_i$ resp.) corresponds with the tuple element which should be placed on the far left (far right resp) of $t_i$. See Figure 5 for an illustration.
The algorithm to find such a set of paths is as follows. Find the longest path, \( Q_1 \), starting from \( L_1 \). Notice that this path must end at some \( R_j \), otherwise we could always expand it with a consistency edge. If \( j < q \), we find the next longest path \( Q_2 \) starting from \( L_{j+1} \). We continue this process until a vertex from every \( L_i \) and \( R_i \) is contained in some path. It is easy to see that this algorithm does indeed return a set of paths maximizing the number of adjacency edges.

**Complexity:** The whole algorithm is summarized by the pseudo-code in Algorithm 1 and Algorithm 2. By Lemma 6 there are at most \( d \) tuples consisting of more than one element. Each of these contributes \( O(\sigma^2) \) edges to the tuple graph. The remaining tuples contribute at most three edges. Since there are at most \( nd \) blocks in total, the total number of edges and vertices in the graph is \( O(\sigma^2d + nd) \). The tuple graph is a DAG so finding these longest paths can be done in linear time.

**An Example:** We will now show an example where the delimiter 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 delimiters to be ordered such that the number of runs is \( \Omega(nd) \). Let \( n = \log_\sigma d \) and consider the \( \sigma^n \) distinct binary strings concatenated with delimiters in lexicographic order. For example, with \( n = 3 \) we would have \( T = \text{000}_0\text{001}_1\text{100}_2\text{101}_3\text{110}_4\text{111}_5\text{011}_6\text{101}_7 \) with \( _0 < _1 \ldots < _7 \). The string \( BWT(T) \) alternates between the delimiters, 0’s, and 1’s yielding \( \Omega(nd) \) runs. On the other hand, for this same case, arranging the delimiters in the optimal ordering allows for at most two runs per block giving \( O(d) \) runs in total.

**Pseudo-code.** We make some comments on the pseudo-code given for this section.

- Within Algorithm 1 the tuples, whose construction uses the variable \( S \), are constructed using a set data structure and then converted to a tuple as needed. This set structure can be easily realized using a bit vector. This tuple is then concatenated with the list \( \text{tuples} \). We consider pointers to these set structures as being stored within the \( \text{Block} \) array, so that multiple elements of the \( \text{Block} \) array can store a reference to the same set structure. Hence we can access and modify the same set structure through different elements stored within the \( \text{Block} \) array. The new set operation creates a new set, initialized with whatever set is passed to it. References to the previously used sets are maintained in the \( \text{Block} \) array, as these references are stored in Line 20. For creating the tree structure needed later, we augment the set structure with members storing references to the parents and child sets (if they exist).

- Within Algorithm 2 we make use of a function denoted as \( \text{LONGESTPATHFROMLEVEL}(i, G) \). The function \( \text{LONGESTPATHFROMLEVEL}(i, G) \) takes a DAG \( G \), equipped with a leveling on its vertices, as well as an integer \( i \). The function returns a path starting from level \( i \) in \( G \) with the longest length amongst all paths starting at level \( i \). This can easily be done in linear time with breadth-first search by appending a fan-in to the vertices on level \( i \).

### 6 Discussion and Open Problems

The following generalization of the Delimiter Ordering Problem is interesting and has unknown computational complexity. Like Delimiter Ordering, the following problem seems to be of more significance when the number of strings \( d \) is far more than the length of strings \( n \).
Algorithm 1: Delimiter Ordering

/* Preprocessing */
1 Arbitrarily order delimiters.
2 Build $BWT(T) = BWT(T_0S_0 \ldots T_{d-1}S_{d-1})$ maintaining index, $idx$, of characters in $T$.
/* Construct tuples and record blocks */
4 $S \leftarrow$ new set({$BWT[1]$})
5 $tuples \leftarrow$ empty list
6 for $i \leftarrow 2$ to $n$ do
7  $idx = BWT[i].idx$
8  $\ell \leftarrow lce(idx,BWT[i-1].idx)$
9  if $\ell > 0$ and $T[idx + \ell + 1]$ and $T[BWT[i-1].idx + \ell + 1]$ are delimiters then
10    $S \leftarrow S \cup \{T[idx]\}$ ; /* Still in block, add to tuple set */
11    if $i = n$ then
12      $tuples \leftarrow tuples.append(S)$ ; /* Finish tuple */
13    end
14  else
15  $tuples \leftarrow tuples.append(S)$; /* Block finished, add tuple */
16  $S \leftarrow$ new set({$T[idx]$}); /* Start new tuple */
17  end
19  $Block[idx] \leftarrow S$; /* Record which block $idx$ is in */
20 end
/* Create tree structure for blocks */
21 for $i \leftarrow 1$ to $n$ do
22  $idx \leftarrow BWT[i].idx$
23  if $T[idx + 1 \mod n]$ is delimiter then
24    $Block[idx].parent \leftarrow$ NULL ; /* Mark as root block */
25  end
26  else
27    $Block[idx].parent \leftarrow Block[idx + 1]$ ; /* Store block’s parent */
28  end
29  if $T[idx - 1 \mod n]$ is delimiter then
30    $Block[idx].children \leftarrow \emptyset$ ; /* Mark as leaf block */
31  end
32  else
33    $Block[idx].children \leftarrow Block[idx].children \cup Block[idx - 1 \mod n]$ ; /* store child */
34  end
35 end
/* Obtained optimally ordered tuples */
36 $tuples \leftarrow OrderTuples(tuples)$
/* Use ordering tuples to obtain delimiter ordering */
37 Order the tree structure using the child order induced by each vertex’s tuple order.
38 Obtained delimiter ordering by performing a linear scan of the tree structure’s leaves.
Algorithm 2: Ordering Tuples

1 Function OrderTuples(tuples):
   /* Construct tuple graph */
2     V ← ∅
3     E ← ∅
4     for ti in tuples do
5         Li ← {ℓi j : j ∈ ti} ; /* new nodes created */
6         Ri ← {ri j : j ∈ ti} ; /* new nodes created */
7         V ← V ∪ Li ∪ Ri
8         E ← E ∪ {(ℓi j, ri j) : ℓi j ∈ Li, ri j ∈ Ri, j ≠ k} ; /* add consistency edges */
9     end
10    for i ← 1 to |tuples| − 1 do
11       E ← E ∪ {(ri j, ℓi j+1) : j ∈ tuples[i] ∩ tuples[i + 1]} ; /* add adj. edges */
12    end
13    G ← (V, E)
14    /* Find set of paths maximizing adjacency edges */
15    P ← ∅
16    len ← 0
17    while len < 2|tuples| − 1 do
18        p ← LONGESTPATHFROMLEVEL(len + 1, G) ; /* get next longest path */
19        len ← len + |p|
20        P ← P ∪ {p}
21    end
22    /* Extract Ordering */
23    for consistency edge (ℓi j, ri k) in edge set of P do
24        Order tuple[i] as (j,...,k), with internal values ordered arbitrarily.
25    end
26    return tuples

Problem 7 (Arbitrary Delimiter Ordering (ADO)). Given a set of strings $T_0,\ldots,T_{d-1}$ all of length at most $n$, and all appended with the new symbol $\$ i (0 ≤ i ≤ d − 1), find a ordering $\pi$ on the symbols $\$ i such that $\pi$ is compatible with the natural ordering given on 0,...,σ − 1, and the number of runs in $BWT(T_0\$0T_1\$1,\ldots,T_{d-1}\$d-1)$ is minimized.

The constraints placed on the symbol ordering in this problem lie somewhere between the constraints in Alphabet Ordering and Delimiter Ordering. Letting $\rho(T_{\pi^{*}_{ADO}})$ be the number of runs in $T = T_0\$0 \ldots T_{d-1}\$d-1$ in an optimal delimiter ordering, $\rho(T_{\pi^{*}_{ADO}})$ the number of runs in an optimal arbitrary delimiter ordering, and $\rho(T_{\pi^{*}_{ADO}})$ the number of runs in an optimal alphabet ordering, we have $\rho(T_{\pi^{*}_{ADO}}) ≤ \rho(T_{\pi^{*}_{ADO}}) ≤ \rho(T_{\pi^{*}_{ADO}})$. However, the Alphabet Ordering Problem is NP-hard while the Delimiter Ordering Problem is solvable in polynomial time. The key element in the reductions used for $AO_k$ is having multiple strings containing the same symbol $a$, where that $a$’s order needs to be determined in the alphabet. This no longer holds in ADO, but still no obvious polynomial time algorithm is evident.

We conjecture that running Algorithm 1 without modification for the ADO problem gives an
approximation which is some constant times optimal, this is largely based on Lemma 6 and the idea that paths can be shifted into blocks which have multiple characters in block’s string at most $d - 1$ times. Each of these shifts only adds finitely many runs to the BWT. An obstacle to this argument arises when one considers, among other things, that there can be up to $nd$ blocks with strings containing only a single character.

Another question unresolved here is the approximability of the optimization problems for AO and SO where the cost function is defined as only the number of runs in the BWT, or as $\rho(BWT(T)) - \delta\sigma$ for all $\delta > 0$. For $\sigma = \Omega(n^{\epsilon})$, if we had defined the cost function for the Alphabet Ordering Problem as simply $\rho(BWT(T))$, rather that $\rho(BWT(T)) - (1+\delta)\sigma$, then any random ordering of the alphabet gives an $n^{1-\epsilon}$-approximation algorithm. Similarly, if $\sigma = \Omega(n)$ and the cost function of Alphabet Ordering were just $\rho(BWT(T))$, we would immediately have a constant approximation algorithm. It seems plausible that solutions which are within some constant or logarithmic factor of $\sigma$ can be found in polynomial time since $\sigma$ needs to be of reasonable size relative to $n$ for our hardness results to hold. Theorem 3 leaves open the possibility for an $n^{1/2}$-approximation for AO$_{1+\delta}$ as well.

Lastly, the problem of ordering the alphabet so as to maximize the value $\rho(BWT(T))$ is also an interesting problem. It’s direct applicability to compression is less obvious, but understanding this problem would likely help in understanding the complexity of approximation for the problems presented in this paper. Indeed, the above statements regarding the approximability of solutions rely on the fact that the number of runs is trivially bounded above by $n$. Having stronger results on the upper bound of $\rho(BWT(T))$ for a string $T$ will allow for stronger statements, perhaps in terms of $\sigma$ or other parameters of $T$.

Acknowledgments This research is supported in part by the U.S. National Science Foundation under the grant CCF-1703489.

References


7 Appendix

A Wheeler Graphs

Recently, a new line of research into a class of graphs with properties related to the LF-mapping has been initiated. These graphs were introduced by Gagie et al. since they can easily model the BWT, de-Brujin graphs, variation graphs, etc. Wheeler graphs admit a space efficient index which allows for matching patterns in optimal time. Since their introduction, the research on Wheeler graphs has continued to be expanded upon and combined with other techniques in [1, 2, 14]. We first define a Wheeler graph. Let \((u, v, k)\) denote the directed edge from \(u\) to \(v\) with label \(k\).
**Definition:** A Wheeler graph is a directed graph with edge labels where there exists an ordering \( \phi \) on the vertices such that for any two edges \((u, v, k)\) and \((u', v', k')\):

1. \( k < k' \implies v \prec_\phi v' \);
2. \((k = k') \land (u <_\phi u') \implies v \preceq_\phi v' \)

and vertices with in-degree zero must be placed first in the ordering. We consider an order which satisfies Conditions 1 and 2 to be a **proper ordering**.

When the vertices are in a proper ordering, if we lay the vertices of a Wheeler graph \( G \) out as though they were in the LF-mapping (see Figure 4) we would see that no edges with the same label cross. Also, we would see that all edges with the same label get mapped into the same portion of \( F \) in the LF-mapping. The differences between Wheeler graphs and previous case lies in the fact that now vertices can map edges to multiple places simultaneously.

Using the natural definition of \( BWT(G) \) presented in Section 4, the more general properties of Wheeler graphs makes the reduction from \( \text{CO}^\dagger \) easier. Instead of using extra symbols to bind together paths induced by the same column of modified incidence matrix, we simply bind them together at the same source. There is no more loop-back necessary as well. To be more precise, given the modified matrix \( M \) the input to the problem \( \text{CO}^\dagger \) construct a graph as follows. For \( 1 \leq j \leq n \), \( 0 \leq i \leq m' + 1 \), if \( i = 0 \) simply create a new source vertex \( s_j \), if \( 1 \leq i < m' + 1 \) and \( M_{i,j} = 1 \) we construct a directed path starting at \( s_j \) with edge labels \( 0^i + 1 \), and if \( i = m' + 1 \) we construct the directed path with labels \( 0^{i+1} \) again rooted at \( s_j \). See Figure 7 for an illustration. Note that this graph is a forest and hence a Wheeler graph.

![Figure 7: Reduction from CO$^\dagger$ to SO. We use sources to bind together paths needed for each column.](image-url)