Wheeler Graph

Emma Gan, Emily Zeng, Lucy Zhang, Ajay Ananthakrishan

November 2021

1 Abstract

The Wheeler Graph is a relatively new idea in the world of text indexing that generalizes the Burrows-Wheeler Transform into trees and graphs. For this project, we accomplished the following goals:

1. Given a graph and ordering, build a visualization that constructs a Wheeler Graph and verifies if a given order is valid.
2. Explore different verification algorithms and find one that has the best time complexity.
3. Extend the verifier to compute a correct ordering given just a graph.

2 Introduction

Why work on this? Why your approach?

With improvements in genomics sequencing, not only are more and more parts of various genomes getting sequenced, but individual genomics reads are also vastly increasing in length. In order to analyze such sequences and make sense of them, there exists an imminent need to efficiently query and do pattern-matching against large genomics sequences. Such querying and pattern-matching of genomics sequences requires the representation of such DNA strings as appropriate data structures. One such data structure is the FM index based on Burrows-Wheeler Transform (BWT) [1].

The BWT is a reversible permutation of characters in a string, constructed from taking the last column of a matrix consisting of all rotations of a string ordered in alphabetical order. It has several advantageous properties that make it suitable for representations of large DNA sequences. First, the BWT has the property of LF (last first) mapping that allows for the reconstruction of the original string. Secondly, the BWT has the powerful property of consecutively; rows are ordered in alphabetical order and thus the BWT ordering is such that adjacent suffixes are in adjacent rows. Consecutively, it can be used to efficiently query the original string against pattern strings. Lastly, the consecutively property of BWT allows for compression; the original matrix of all ordered rotations of a string is represented as a single BWT string.

The BWT is a representation of large genomics sequences that can be efficiently queried and indexed. While the BWT is useful, there exists a need to generalize the BWT in order to account for efficient representations of not just a single string, but multiple strings that have several shared patterns among them (such as shared prefix and suffixes). Such a representation has several useful applications. For example, representations of several different human genomes as one data structure (in which there is significant conserved regions among each of the genomes) can be used to efficiently identify mutations among individuals. Additionally, such a data structure can efficiently represent pan-genomes, which is the entire set of genes within a species. The use of BWT to represent such genomes is quite cumbersome, as each individual genome would require a single Burrows-Wheeler matrix.

The Wheeler Graph is a powerful graph generalization of the BWT that conserves the properties of efficient representation and querying, but allows the representation of multiple genomes in a single graph data structure.

A simple Wheeler graph can be constructed by taking the letters of a string, in their given order, and populating the edges of a graph with such letters. However, the property of efficient querying and indexing must be held, and thus, the immediately preceding nodes of each letter must consist of the row number of the corresponding BWT suffix that is spelled out from following the letter until the end of the string in the graph (dollar sign) is reached. This allows for the property of consecutivity, in which the graph is path coherent. In other words, "for any consecutive range[i,j] of nodes and a string 'a', nodes reached by following edges matching 'a' also form consecutive ranges. The consecutively property is needed for LF-like index matching (of BWT), and the ordering of nodes in a Wheeler graph is necessary for it to be efficiently queried.

An edge-labeled graph is a wheeler graph if there is an ordering of nodes that follows these properties [2]:

1. 0 in-degree nodes come before others
2. For all pairs of edges $e = (u, v), e' = (u', v')$ labeled a, a' respectively,

$$a \prec a' \implies v < v'$$
$$(a = a')^{(}u < u') \implies v \leq v'$$

We use $\prec$ to denote the ordering of the edge labels and $<$ to denote the node ordering. We will refer to these properties as rules 1,2,3.

We are interested in verifying the Wheeler properties of a graph because it allows us to identify graphs with a total Burrows-Wheeler order over all the nodes, so that it could have the consecutivity property, which would allow us to perform string matching in the same way as an FM index. If a graph does not have Wheeler properties, it is possible to fix it by unzipping the forks with ambiguous ordering, but this will not be the focus of this paper. We are also interested in building a visualization tool that allows the user to visualize and track the process of verifying the Wheeler properties of a graph so that the user can clearly detect where the Wheeler properties are not satisfied on the graph.

3 Prior work

What did you read? What did others accomplish before you?

To understand what Wheeler Graphs are and their properties, we referenced the paper written by Travis Gagiea, Giovanni Manzini, and Jouni Sirénd that first defined Wheeler Graphs and showed that these graphs have path coherence [2]. We also referenced Professor Ben Langmead's lecture series on YouTube that went over Wheeler Graphs to get a better understanding of the motivation behind the Wheeler Graphs and go through example graphs that do or do not fulfill the Wheeler properties [3].

The small scale examples in the papers described above were mostly motivating examples. However, there are certain classes of graphs that are always Wheeler. These were particularly of interest so that we could expand our dataset.

Wheeler graphs were found to have a succinct representation using bitvectors [2]. It unified a large family of compressed data structures that revolved around BWT [1]. Examples include FM-index [4], the eXtended Burrows Wheeler Transform (XBWT) [5], and the BOSS respresentation of de Bruijn graphs [6]. We have already discussed the FM-index extensively in class. Thus, we focused on the other two classes: tries and de Bruijn graphs [2].

The XBWT succiently describes a trie. Prezza (2021) showed that run-length encoded BWTs could be compressed into tries [7]. In fact, these tries were equivalent to Wheeler deterministic finite automata and are deeply related to the tunneling technique for Wheeler graphs described by [1]. Thus, any trie can be described in the compressed XBWT form and is more generally a Wheeler graph.

Similarly, the de Bruijn graph was found to have a compressed structure in the BOSS representation [6]. A natural LF correspondence was exploited in this graph representation, which supported fast navigation. Within this BOSS representation, nodes are sorted according to the colexicographic order of the corresponding k-mer. This naturally embedded sorting is precisely the order used to check the Wheeler graph properties [1]. Since any de Bruijn graph can be represented with this BOSS form, all de Bruijn graphs must be Wheeler graphs as well.

Other Wheeler graphs in disguise that Gagie et al. found were multi-string BWT with the permuterm index, which generalizes the XBWT trie representation, and Generalized Compressed Suf

In addition to expanding our dataset, we looked at existing approaches to verifying the ordering of a Wheeler Graph and computing the ordering given a Wheeler Graph [8] [9] before coming up with our own approaches.

4 Methods and software

What did you implement? Why? How?

Our project was split into four main parts: generating data, creating a

verifier with ordering, extending the verifier to work without a given ordering, and a visualization of our verification process.

4.1 Dataset Generation

4.1.1 Format

Each of the graphs that we used for evaluation have the following format:

V
[list of vertex names, separated by space]

E
[list of directed edges (v_1, v_2, l) which represents a directed edge from v_1 to v_2 with an edge label l]

Ordering
[list of vertex names, separated by space in increasing order (if applicable)]

Source
[file/link name (if applicable)]

4.1.2 Manual Generation

At the start of this project, many of the Wheeler Graph examples that we used to test our verification algorithm were pulled from existing papers and Github repositories. Most of these Wheeler graph examples were extremely small with no more than 5 nodes and edges. In order to get more examples, we transcribed a couple of our own.

4.1.3 Tries

Because Tries are always Wheeler Graphs, we used them to generate larger Wheeler Graphs to test on. A representation of a Trie can be seen in figure 1.

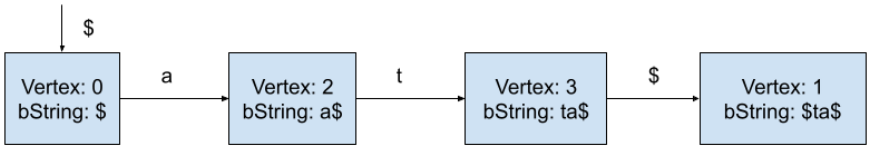

Figure 1: Small Trie Example

Every vertex node has an additional parameter: backwardString. The main purpose of the backwardString field for each node was to label the nodes in

the correct order so that the trie fulfills properties of a Wheeler Graph. In the getOrdering() method of the Trie class, the nodes were sorted in order alphabetically by their backwardString using Python's sorted() function, and then sequentially labeled from 0 to $n-1$ where n is the total number of nodes. Because of this, the root node has an incoming \$ edge with the label; thus, the backwardString value of the root node is always \$ which means that it will always have the vertex label of 0.

Once we had our Trie representation, we wrote a script trie_to_graph.py to convert the Tries to our desired graph format. To generate the vertices and the ordering, we just listed the numbers sequentially from 0 to $n-1$ where n is the total number of nodes. In order to generate the edges, we performed a BFS on the Trie.

For easy creation of large Tries, we wrote a script that generated input files that contained randomly generated DNA sequences given the desired number of lines for the input and the desired number of characters per line.

4.1.4 Bitvectors

Originally, we planned on converting our data set to use bitvectors in order to test our algorithm on graphs of immense size. However, since we found that we had issues running our algorithms on graphs that held more than 400 characters (20 strings, each with 20 characters), we decided that our current graph representation was sufficient for the files that we were using.

4.1.5 De Bruijn Graphs

The Wheeler graph can be viewed as a loose generalization of BWT-based data structures, and there has been previous work done on generalizing de Bruijn graphs to Wheeler Graphs. Thus, given certain de Bruijn graphs, a Wheeler graph can be constructed. Specifically, a special case of the de Bruijn graph, called the BOSS representation, can be represented as a Wheeler graph [9]. The BOSS representation requires all nodes of a de Bruijn graph to be ordered lexicographically, and requires a bitvector representation of such nodes. Since we chose not to pursue a bit vector representation (as explained in the previous section), we chose not to create Wheeler graphs from de Bruijn graphs (although it is a means to construct large Wheeler graphs).

4.2 Verifier with ordering

For checking rule 1, we iterated through all the nodes to find the node(s) with in-degree 0. We then checked if those nodes come first in the provided ordering. The time complexity of this step, given a graph with m nodes and n edges, is O(m). For checking rule 2 and 3, we had two approaches: the naive approach and the partition approach. Both of our approaches would break the verifier program upon seeing an edge pair that violates the rules.

4.2.1 Naive Approach

For the naive approach, we iterated through each edge pair. If edge labels were equal, and one source node u came before the other source node u', we checked if that edge's destination node v's order was less than or equal to the other destination v' (rule 3). If one edge label a was smaller than the other edge label a', we checked if that edge's destination node v came before the other destination node v' (rule 2). The time complexity, given m nodes and n edges, is $O(n^2)$ since we iterated through all the edge pairs.

4.2.2 Partition Approach

For the partition approach, we first grouped the edges with the same labels together in a map, where the keys are labels and the values are sets of edges with the same labels. Then, for edges that share the same labels, we generated all combinations of edge pairs.

For each generated edge pair we checked if one source node u came before the other source node u' and that edge's destination node v's order was less than or equal to the other destination v' (rule 3).

To check rule 2, we generated all combinations of the different edge labels and check if one edge label a was smaller than the other edge label a' and that edge's destination node v came before the other destination node v'.

The time complexity, given m nodes and n edges, is still $O(n^2)$ since in the worse case if all edge labels were different, we still have to iterate through all edge pairs. However, we had broken down the problem into iterating through the edge pairs with same labels and different labels respectively instead of iterating through all edge pairs. Also, by generating combinations instead of permutations of the edge pairs, we avoided iterations of the same edge pair twice. Thus, the total number of iterations should decrease in most cases compared to the naive approach.

4.3 Verifier without ordering

To verify the wheeler properties of a graph without ordering, we iterated through the permutations of all the nodes and checked if the ordering satisfied the rules using either the naive approach or the partition approach. Given a graph with m nodes and n edges, iterating through the permutations would have a time complexity of $O(m!)$ and verifying the rules in each iteration would take $O(n^2)$ time. Thus, the overall time complexity of our verifier without ordering is $O(m!n^2)$.

Our approach to verify graphs without ordering is very naive and slow. One way to eliminate some iterations is to first order the nodes with respect to the edge labels of their incoming edges. According to rule 2, nodes with a smaller incoming edge label should precede notes with larger incoming edge label. Thus, now we only have to check permutations for nodes with the same edge labels of incoming edges instead of checking permutations with all the nodes naively.

4.4 Visualization

The visualization for a Wheeler and non-Wheeler graph is shown in Figure 2.

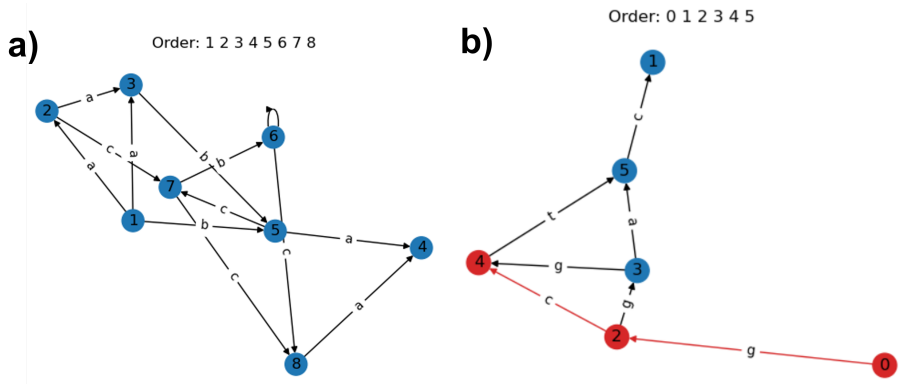

Figure 2: Visualization of a) Wheeler graph and b) non-Wheeler graph

The default coloring is blue nodes and black edges. For an incorrect ordering, the first pair of edges or nodes that do not satisfy the Wheeler properties are colored red.

We used matplotlib to draw the NetworkX graphs. The built-in NetworkX attributes were extremely helpful. There were three important attributes

1. **Label**: All nodes and edges had a "label" attribute attached during the file parsing step. (We assumed that the labels have a partial order, like lexicographical or numeric.)

2. **Color**: The first offending pair of edges or node that don't satisfy the Wheeler properties was marked with a "red" color attribute during the verification step. The rest of the nodes and edges were initialized with "blue" and "black", respectively.

3. **Order**: The current ordering being checked was a graph attribute. This is set either in the file parsing step (ordering given) or when we check all possible orderings (no ordering given).

The matplotlib animation library was used to display each ordering attempted. There were three components of the animation:

1. **Animation wrapper:** The created animation was stored in a variable that persists as long as the animation runs for. The animation ran once through all frames, with each frame corresponding to an ordering of the nodes. The animation could be saved as a GIF file.

2. **Per-frame drawing update:** This function housed all the drawing-related steps for a single ordering (i.e. clearing figure axes, drawing nodes

and edges, etc.). For visualizations given an ordering, we called this function a single time. For an animation of multiple orderings, this was given as an argument of `FuncAnimation` and is called for every frame/ordering.

3. **Generator:** The generator contained all the side effects of checking the Wheeler properties. It updated the graph coloring attributes for every iteration.

5 Results

How well did your method work compared to others?

Graph˙name	V	E
nonwheeler2	12	13
nonwheeler3	6	7
nonwheeler4	6	6
nonwheeler5	10	9
nonwheeler6	11	11
nonwheelersmall	3	3
wheeler	8	13
wheeeler2	3	3
wheeler3	5	4
wheeler4	6	5
wheeler5	10	10
wheeler6	12	12
wheeler7	6	5
trie	7	6
trie1	397	396
trie2	7	6
trie3	397	396
trie4	104	103

Table 1: Description of size of data used in our experiments. Size measured by number of edges and nodes in manually-annotated graphs and tries.

	naive-order	partition-order	naive	partition
nonwheeler2	0.0022213	0.0021558		
nonwheeler3	0.0017405	0.0014881	0.645436	0.659197
nonwheeler4	0.0017999	0.0018	0.586502	0.604752
nonwheeler5	0.0043477	0.0020467	0.0508088	0.0375882
nonwheeler6	0.0030612	0.0024195		
nonwheelersmall	0.0017304	0.0024145	0.0240926	0.0246926
wheeler	0.0078355	0.0039801	0.0282888	0.0252572
wheeler2	0.0011645	0.0008835	0.0190121	0.0182313
wheeler3	0.0025024	0.0017003	0.0220339	0.0190194
wheeler4	0.0021076	0.001274	0.0229047	0.0197489
wheeler5	0.0058254	0.0041707	0.0265375	0.0273503
wheeler6	0.0091654	0.0029075		
wheeler7	0.002469	0.0016189	0.0205313	0.0215812
trie			0.814613	0.704957
trie2	0.0027095	0.0014147	0.0225339	0.0200372
trie3	7.47678	0.37176	7.52905	0.52294
trie4	0.47951	0.0339867	0.634947	0.0705263

Table 2: Benchmarking for manually-annotated graphs and tries. 'naive-order' and 'partition-order' refer to verifying given an ordering, whereas 'naive' and 'partition' compute an ordering.

We benchmarked the naive and partition approaches with our dataset of manually-annotated graphs and tries. We timed each verification method with and without ordering 10 times and took the average. We omitted results for long-running computations with extremely large graphs with greater than 11 nodes (manual) and trie1 (trie).

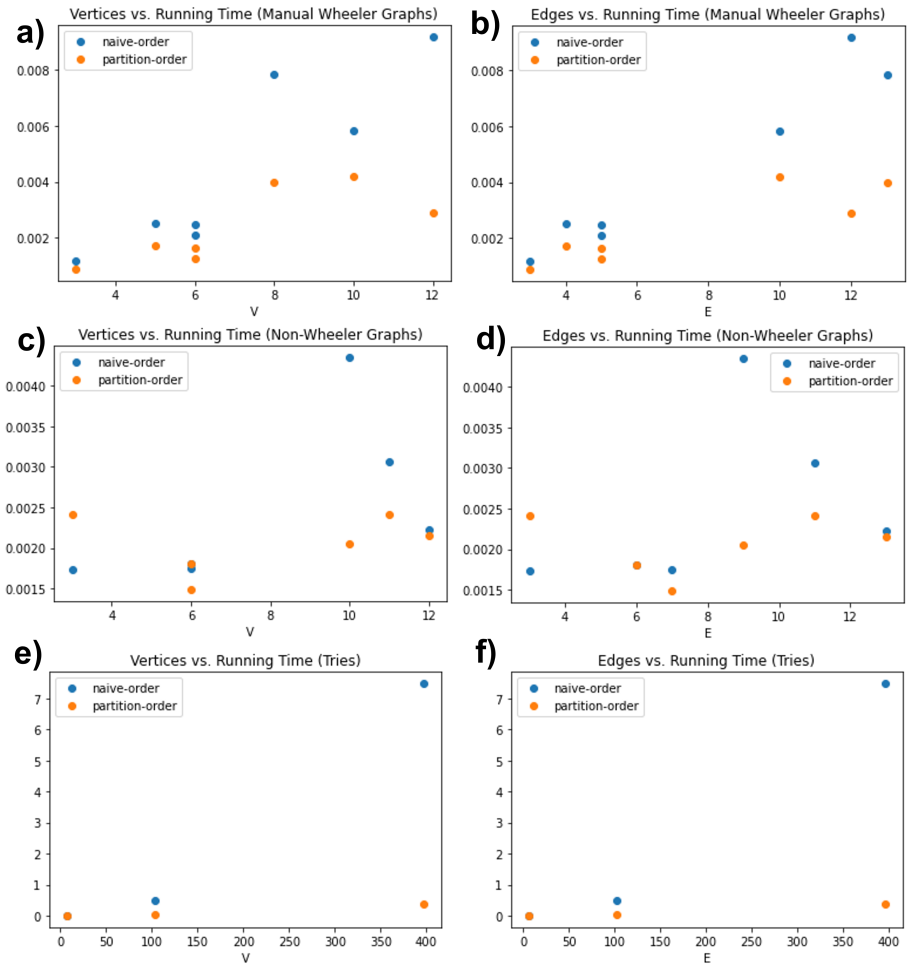

Figure 3: Runtimes compared to number of vertices (a,c,e) and edges (b,d,f) for a-b) manual graphs, c-d) non-wheeler, and e-f) tries

Looking at Figures 3, we observed that when comparing the running times of the naive and partition algorithms with respect to the number of vertices and edges for the different types of graphs (manually-transcribed wheeler, tries, non-wheeler graphs), we see that the partition approach performs slightly better than the naive algorithm in most cases but not by much.

6 Conclusions

What did you learn? What should we come away with?

The Wheeler graph is a useful data structure because it allows for efficient processing of multiple strings with shared motifs while maintaining compact representations of said strings. With increasingly large genomics data, Wheeler graphs can be beneficial for representing sequences, as well as analyzing differences in sequences.

We built a visualizer that takes in graphs as text input representation and represents them graphically. Our algorithm also includes a verifier component; it checks node and edge relationships to make sure they abide by the Wheeler property.

Lastly, we also take in graph representations (as text input) without ordering, and create an ordering of nodes (if such a ordering exists). We used two algorithms to do this; a naive approach that checks every permutation of edges, and a partition that groups edges with same labels together before checking against other edges. While our partition approach generally seems to perform better, our method did not complete computing a Wheeler ordering for some of our input graphs.

Whether a graph is Wheeler or not is important, and thus, further areas of exploration are to implementing a more efficient verifier algorithm.

References

[1] Jarno Alanko, Travis Gagie, Gonzalo Navarro, and Louisa Seelbach Benkner. Tunneling on wheeler graphs. *CoRR*, abs/1811.02457, 2018.

[2] Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for bwt-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. Algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo).

[3] Ben Langmead. Wheeler graphs, part 1.

[4] Paolo Ferragina and Rossano Venturini. *Indexing Compressed Text*, pages 1861–1868. Springer New York, New York, NY, 2018.

[5] Paolo Ferragina and Rossano Venturini. The compressed permuterm index. volume 7, pages 535–542, 01 2007.

[6] Alex Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. pages 225–235, 09 2012.

[7] Nicola Prezza. *On Locating Paths in Compressed Tries*, pages 744–760. 01 2021.

[8] Giovanni Manzini. The burrows-wheeler transform: Theory and practice. 09 1999.

[9] Lavinia Egidi, Felipe Alves Louza, and Giovanni Manzini. Space efficient merging of de bruijn graphs and wheeler graphs. *ArXiv*, abs/2009.03675, 2021.