A Suite of Tools to Study Wheeler Graph Recognizing Problem

Kuan-Hao Chao (kchao10@jhu.edu)[1,2], Eduardo Aguila (eaguila6@jhu.edu)[1]

[1] Department of Computer Science, Johns Hopkins University, Baltimore, 21211, Maryland, USA,
[2] Center for Computational Biology, Johns Hopkins University, Baltimore, 21211, Maryland, USA

Abstract

A Wheeler graph (WG) is a framework that provides a new way to compactly and efficiently process strings, and it links various BWT variants up by two monotonicity properties shared by their state diagrams. Recognizing if a given directed graph is a valid Wheeler graph is important since it might be a prior step before applying BWT-based compression scheme; however, this problem has been proven NP-complete if the given graph has more than 2 distinct edge label alphabets. This is a new field that emerged in 2017 yet to be fully explored, and it is important to come up with a computationally practical and intractable Wheeler graph recognizing algorithm. We proposed a permutation-based algorithm and proved that it is faster than Gibney's and Thankachan's state-of-the-art exponential algorithm. Furthermore, we implemented a suite of tools, a generator in Python and a recognizer in C++, and tested on more than 49,000 randomly generated valid Wheeler Graphs. Our recognizer successfully gave at least one correct Wheeler graph node order on all of them. Gibney's and Thankachan's algorithm is also implemented to the point of enumerating three bitarrays with obvious filtrations, whose time complexity is the loose lower bound of the full algorithm, to practically compare with our algorithm. Results show that our algorithm is faster. We further benchmarked our permutation-based recognizer with different graph topologies. This is the first released WG-related software, and it is open-sourced on https://github.com/Kuanhao-Chao/Wheeler_Graph.

Keywords: wheeler graph, generator, recognizer, finite automata, Burrow-Wheeler Transformation
Abbreviation: Wheeler graph (WG), Burrow-Wheeler Transformation (BWT)

I. Background

The idea of BWT was devised in 1983, and it was published on *Communications of the ACM* eleven years later in 1994 **(Burrows 1994)**. Although the value of BWT was underestimated by reviewers in the beginning, it soon proved its own significance. Two years after being published, it was implemented in bzip2 by Julian Seward and now it is still one of the most popular Unix compression utilities **(Seward 1996)**; szip developed by Michael Schindler is another efficient large file compressor **(Schindler 1999)**. Burrow-Wheeler Transform (BWT) is a simple yet powerful tool which is the foundation algorithm of some of the best lossless compression tools. Furthermore, it also largely influences many fields. In computational genomics, it successfully reduces memory footprint and makes aligning sequences on PCs feasible. Bowtie2 **(Langmead and Salzberg 2012)** and BWA **(Li 2013)** are two important BWT-based aligners; in image compression, the pipeline Burrows-Wheeler Transform with an Inversion Encoder (BWIC) is shown to outperform the widely used JPEG-LS and JPEG-2000 **(Preston 2015)**.

In the last ten years, there were a lot of variants proposed to solve graph-, trie-, and alignment-related problems; however, some of the so-called BWT variants lose the two main BWT features which are "lossless compression" and "invertibility". Finally in 2017, Gagie et al. linked those BWT variants up by bringing finite automata into the field **(Gagie 2017)**. Surprisingly, the state diagrams of data structures including multi-string BWT **(Mantaci 2005; Mantaci 2005; Mantaci 2007)**, trie **(Ferragina 2005)**, de Bruijn graph **(Bowe 2012)**, positional BWT (PBWT) **(Durbin 2014)**, wavelet tree **(Grossi 2003)**, etc. are all shown to be Wheeler graphs sharing two monotonicity properties. By applying the two properties, we can further prove another property, path coherence, which makes pattern-matching and graph-traversing on a graph a linear time problem, even if its state diagram is a Nondeterministic Finite Automata (NFA). Wheeler graphs provide a new way to compactly and efficiently process strings. Thanks to Gagie et al.'s new unifying view on BWT variants, we have further advanced our understanding of the BWT.

One of the open questions in the paper is whether we can recognize Wheeler graphs in a reasonable amount of time. The Wheeler graph recognizing problem is crucial since it might be the prior step before applying BWT-based compression schemes **(Gagie 2017)**. Later in 2019, Gibney and Thankachan published a theoretical paper discussing the hardness of this problem. It is proven to be NP-complete and has not yet been well-solved **(Gibney and Thankachan 2019)**. At the end of the paper, an exponential wheeler graph recognizing algorithm in pseudocode using Gagie's three-bitvector Wheeler graph data structure, I, O, and L, was proposed; however, as the node number, edge number and label number grow, it soon becomes computationally intractable. In 2020, Gibney further divided the WG recognizing problem into smaller subproblems. He defined d-NFA as "a directed graph where each edge is labelled with a character and at most d edges with any particular label leave from any vertex" **(Gibney and Thankachan 2020)**. Recognizing a 2-NFA WG can be solved in polynomial time, whereas 5-NFA WG recognition is proven to be NP-complete. And the time complexity of recognizing 3-NFAs and 4-NFAs is still an open question.

WG is a relatively new topic, and it lacks open-source tools and libraries. In this research, we proposed a new permutation-based wheeler graph recognizing algorithm and developed it in C++. We also implemented Gibney's and Thankachan's algorithm to the point of enumerating three bitvectors with obvious filtrations, whose time complexity is the loose lower bound of the full algorithm. Last, a valid and invalid random Wheeler graph generators are implemented in Python to systematically benchmark both algorithms.

II. Explanation

Before diving into the topic, we first define the Wheeler graph checking and recognizing problem:

Problem Definition: Wheeler graph checking problem

Given a directed edge labelled and node labelled graph $G = (V, E)$, answer 'YES' if G is a Wheeler graph and 'NO' otherwise.

Problem Definition: Wheeler graph recognizing problem

Given a directed edge labelled graph $G = (V, E)$ with random node labels, answer 'YES' if G is a Wheeler graph and 'NO' otherwise.

The Wheeler graph checking problem is easy. After sorting the edges first by edge labels and then by head labels, three wheeler graph properties can be checked in linear time. As for the recognizing problem, it is fundamentally more complex, and this is the problem that we are solving in this study.

III. Improvements: Wheeler Graph Recognizing Problem

In 2019, Gibney and Thankachan proved that the Wheeler graph recognizing problem is NP-complete, and there are no efficient and elegant ways to solve it **(Gibney and Thankachan 2019)**. In the same paper, they further proposed a state-of-the-art exponential algorithm. Given a directed graph G, and the length of three bitvectors, I, O, and L, which are the Wheeler graph data structure proposed by Gagie et al. **(Gagie 2017)**, the main idea is to **(1)** enumerate three bitvectors, **(2)** check if any of the encoding is a valid Wheeler graph G, **(3)** convert the valid wheeler graphs (G') and the given graph (G) into undirected graphs, $\alpha(G')$ and $\alpha(G)$, and **(4)** check the isomorphism. If any of the valid Wheeler graph encoding is isomorphic to the given graph, then the answer is yes; otherwise no. Its time complexity is claimed $2^{E\ log\sigma + O(N+E)}$.

Although their algorithm is exponential, it is not exponential on N but on $E\ log\ log\ \sigma\ + O(N+E)$. We have proved that their algorithm is in exponential disguise. Only if the graph is sparse, its performance will be close to the exponential algorithm. If we take the relationship between nodes and edges into consideration, in normal cases, their algorithm is slower than trying all node orders. The proofs are in our final deliverable in *Computational Genomics: Sequences*, Fall 2021 **(Aguila, Shah, Chao and Erdogdu 2021)**.

A naïve way to solve the Wheeler graph recognizing problem is to test all permutations of the vertex set V. Let $|V| = n$, the time complexity of the algorithm is $O(n!)$. As n grows, the algorithm quickly becomes computationally intractable. However, instead of permuting first and then checking three Wheeler graph properties later, a better approach is to apply the rules before the permutation and reduce the trial and error. For instance, if we know a certain set of ***a*** nodes that are smaller than the remaining nodes, we can split ***a*** nodes into ***x*** and ***y*** two sub-groups, and the number of permutations is reduced from ***a!*** to $\boldsymbol{x! * y!}$. This is the core concept of our Wheeler graph recognizing algorithm. ***Lemma 1*** and ***Lemma 2*** below are proposed to optimize the factorial algorithm.

Following are self-defined terminologies. We define "**edge group**" as the set of all edges with the same edge label. For example, if the edge labels of edges, e_1, e_2, e_3, are all A, they are in the A edge group; "**nodes in A edge group**" means taking the non-repeat set of all the tails of edges in A edge group; "**in-node list**" is defined as all the nodes that go into the target node in the sorted order. For example, if node n_1 is the tail of edge e_1 ($f \rightarrow n_1$), e_2 ($d \rightarrow n_1$), e_3 ($c \rightarrow n_1$), then the in-node list of n_1 is cdf.

Lemma 1:
For any two edge groups, all the nodes in the edge group with the smaller edge label are smaller than all the nodes in the edge group with the larger edge label.

Lemma 2:
For any two different nodes, n_1 and n_2, in the same edge group in a valid WG, if the in-node list of n_1 is lexicographically larger than the in-node list of n_2 and the smallest in-node of n_1 is larger than the largest in-node of n_2, then the label of n_1 is larger than the label of n_2.

By ***Lemma 1***, we can first split nodes by the edge group into subgroups; by ***Lemma 2***, each subgroup can be further split again into sub-subgroups. Therefore, the total node orders can be largely reduced to permuting in each sub-subgroup. The proofs of ***Lemma 1*** and ***Lemma 2*** are in our final deliverable in *Computational Genomics: Sequences*, Fall 2021 **(Aguila, Shah, Chao and Erdogdu 2021)**.

IV. Implementation: Generator, Recognizer

i. Generator

We developed a suite of generators that produce Wheeler Graphs to aid in testing our recognizer, as well as to perform timing analysis. While there are multiple variations on the generator, they all function from the same core program. Rather than randomly generating a WG from scratch, we leverage existing network **(Hagberg 2008)** graph generation to produce an Erdős Rényi graph / binomial graph. The networkx graph generator takes three parameters: the number of nodes, edge probability, and a boolean for if the graph is directed. From this graph, we rearranged node labels and assigned edge labels such that they would satisfy all necessary conditions to be a WG. The process is outlined in detail in our previous work **(Aguila, Shah, Chao and Erdogdu 2021)**. Since this process meant removing some nodes from the initial graph, we further determined an empirical relationship between the number of nodes in the original graph, and the final WG our generator produced, such that we could produce graphs with the desired final node count.

From this core program, we then built multiple variations to help us in understanding and timing the recognizer problem. The most straightforward generator takes five command line arguments: folder for samples, number of samples, start node, stop node, and step. This will produce samples for node counts from start node to stop node, incremented by the step value. For example, running with `` `../test 20 5 15 5` `` will produce samples for each of 5, 10, and 15 nodes, and place those 20 within subdirectories of the `../test` folder.

There is also another generator that functions similar to this, but instead of producing valid WG with different node counts, produces valid WG at one specified node count, varying the edge probabilities as specified by the command line arguments.

The final generator we created produces invalid Wheeler Graphs, where the violations are made at specific edge labels within the graph. We used this generator to generally test that our recognizer could correctly identify invalid Wheeler Graphs, as well as to perform some timing analysis on how the recognizer would perform when violations at earlier edge labels vs. edge labels that come later in the WG. This invalid generator begins with the same WG generator as described earlier, and produces a valid WG. The program then takes two parameters: partition to mutate, and total number of partitions. These are both in respect to the alphabet of edge labels. For example, with an alphabet of $[a, c, d, e, f, h, i, j, k, m]$, if we were to divide this alphabet into five partitions, the program would create a list of $[[a, c], [d, e], [f, h], [i, j], [k, m]]$. The user could then specify which partition the violation should occur in and produce the appropriate samples. For our purposes, we wrote a generator that takes the number of nodes, number of samples, and total number of partitions as parameters.

ii. Recognizer

We implemented our permutation-based algorithm discussed in the previous ***Improvement*** section in C++. Following are the steps of the algorithm: **(1)** A graph is created, and the in-node list of each node is initialized. **(2)** Edges in the graph are split by edge groups, and nodes in each edge group are relabelled to the maximum possible label. **(3)** Within each edge group, nodes are sorted by their in-node list, and relabelled accordingly. **(4)** For different nodes having the same labels, their orders are recursively permutated and checked. **(5)** For every permutation on nodes with the same initialized label, if any of the three Wheeler graph properties are violated, the downstream recursive call for those nodes is halted, and the recognizer moves on to their next permutation. The input to the program is a DOT file, and the recognizer will recognize if the given graph is a valid Wheeler graph. If yes, then five files which are *I.txt*, *O.txt*, *L.txt*, *node.txt*, and *graph.dot*. Among them, *I.txt*, *O.txt*, and *L.txt* are three bitvectors WG data structure proposed by Gagie **(Gagie 2017)**, *node.dot* stores the mapping from old node labels to new node labels, and *graph.dot* is the correct and sorted dot file with new node labels. The total numbers of node orders that are tried and runtime are outputted.

iii. Gibney & Thankachan's Exponential Recognizing Algorithm

In the Supplementary C.1 Proof of Theorem 20 section of Gibney's & Thankachan's paper, a pseudocode of their exponential algorithm was proposed **(Gibney and Thankachan 2019)**. The core of their algorithm is to essentially enumerate three bitvectors of a given length. Details are in the previous ***Improvement*** section. To further compare our permutation-based algorithm to Gibney's and Thankachan's exponential algorithm, we also implemented the first part of their algorithm, which is to enumerate three bitvectors, I, O, and L, in C++. Although they did not mention any filtration criteria and only list the upper bound of the set of all possible Wheeler graph encodings to be checked, which is $2^{E\ log\sigma\ +\ 2(N+E)}$, there are a few obvious cases that we can skip to accelerate the algorithm. For example, I and O bit arrays must have $|E|$ 0s, their last bits must be 1, etc; therefore, simple filtration criteria are implemented as well to reduce some unnecessary permutations. The second part of their algorithm that is not implemented includes **(1)** validating the iterated Wheeler graph encoding, **(2)** transforming graphs into undirected graph, **(3)** checking if graphs are isomorphic, which is $n^{O(1)}\, 2^{(E\ log\sigma\ +\ 2(N+E))^{1/2}}$. Combining the first and second part of the algorithm, the total time complexity is $2^{E\ log\sigma\ +\ O(N+E)}$. Therefore, the second part of their algorithm does not add on the overall time complexity; the complexity of this problem is determined by the first enumerating part, which is the loose lower bound of the full algorithm.

V. Results

i. Correctness Unit test

To test the correctness of our recognizer, we randomly generated 49,000 graphs, with node numbers ranging from 4 to 100 with the common difference of 2. For all of them, at least one correct wheeler graph node order can be found. All 49,000 graphs are online at https://drive.google.com/drive/folders/1wRWQSdfZE4D9d720_DrGkfuAihlxDZhh?usp=sharing, and the result is available online: https://github.com/Kuanhao-Chao/Wheeler_Graph/tree/main/results/TEST_correctness.

ii. Algorithm Benchmarks

All benchmarks in this section were run on a 2.9 GHz Quad-Core Intel Core i7 processor with 4 cores, 16 GB 2133 MHz LPDDR3, Apple Macbook Pro.

1. Valid Wheeler Graph Benchmarks

As stated previously we did not implement the full exponential algorithm but rather just the first part, which is to enumerate the three bitvectors for I, O, and L. Therefore, we only performed a timing analysis for this section of the algorithm. We found that the exponential recognizer vastly underperformed when compared to the permutation recognizer. As such, for the exponential algorithm, we were only able to get timing data for three samples of Wheeler graphs with five nodes. On average, the exponential recognizer took 604.98 seconds to enumerate the three bitvectors for Wheeler graphs with five nodes. We also attempted to get data for testing the exponential recognizer on a WGs with 10 nodes, but to run just one sample took over 30 hours (at which point we terminated the program), since it was clear at this point the exponential algorithm could not perform well in practice.

For the permutation algorithm, the default option, finding the first valid Wheeler graph and stop, was applied to the following benchmarks. We first collected data on how the execution time would vary as a function of the number of nodes in the Wheeler graph, for a set edge probability of 0.2, with the results shown in ***Figure 1***.

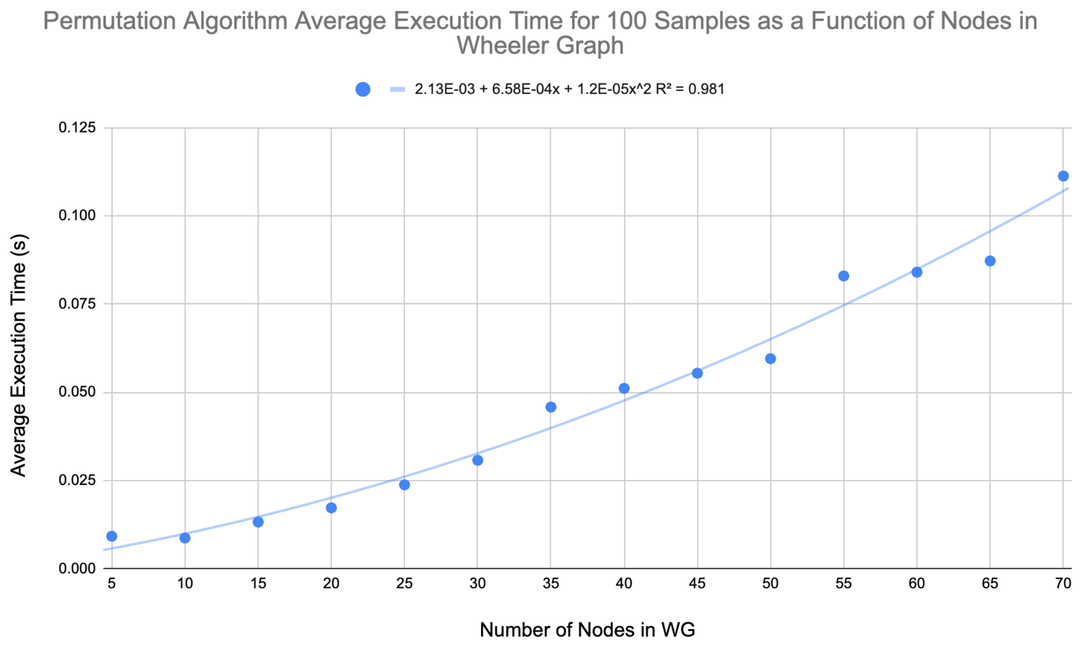

Figure 1 *Permutation Algorithm Average Execution Time for 100 Samples as a Function of Nodes in Wheeler Graph.*

The figure demonstrates the data collected, fitted with a polynomial trendline and an R-squared value of .981. We found that in practice, for nearly every sample, the recognizer performed very close to average within this range (5 to 70 nodes). We also tested the permutation recognizer for up to 100 nodes, and we were able to obtain some results. However, for Wheeler graphs with more than 70 nodes, the worst-case time was significantly worse. For example, for testing a Wheeler graph with 80 nodes, we can partition the 100 samples into those that took <2 seconds, and those that took >= 2 seconds. There were 90 samples that took less than 2 seconds, with an average execution time of 0.284 seconds. The remaining 10 samples took longer than 2 seconds, with an average execution time of 22.02 seconds. We hypothesize that the samples took longer time have less "informative" edges which impose requirements to break ties and reduce the permutations, and thus increases the trial-and-error times. It is our future goal to systematically analyze the characteristics of those outliers.

We then tested how the execution time would vary as a function of the edge probability, which controls how likely the edges are created. With higher edge probability, we are more likely to get a high-density multigraph (in other words, the fatter / wider graph). We arbitrarily fixed the Wheeler graph to have 60 nodes, and varied the edge probability from .05 to .65, as seen in ***Figure 2***.

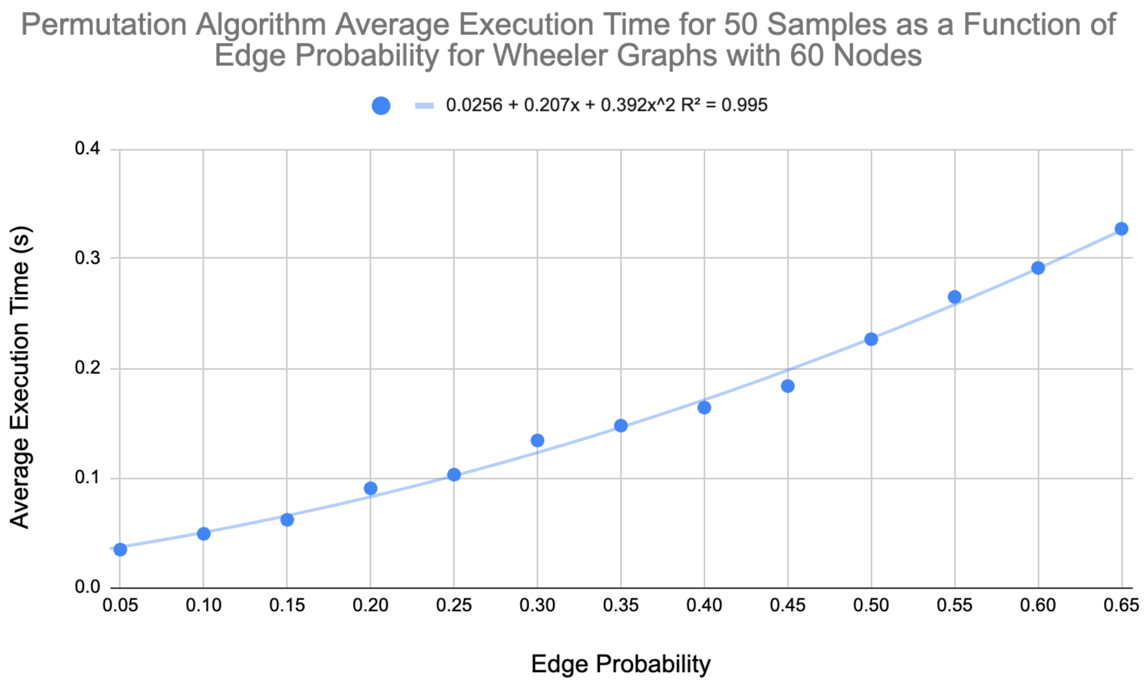

Figure 2 *Permutation Algorithm Average Execution Time for 50 Samples as a Function of Edge Probability for Wheeler Graphs with 60 Nodes.*

The figure demonstrates the data collected, fitted with a polynomial trendline and an R-squared value of .995. Similar to the previous analysis, we found that in practice many of the samples performed very close to average, however there were some extreme outliers which we removed before plotting. These outliers were concentrated in the samples for especially dense Wheeler graphs (those with edge probability >= .5). In total we removed 12 samples from the 650 Wheeler graphs (50 samples for each edge probability tested), with 10 of those being from edge probabilities greater than or equal to .5. The result shows that under the same node number, the edge number is another important factor that quadratically affects the runtime.

2. Invalid Wheeler Graph Benchmarks

We then moved on to testing how the recognizer would perform for invalid Wheeler Graphs. First, we tested how the recognizer would perform for invalid WGs for a fixed edge probability, and edge violations occurring at a fixed level within the alphabet, varying the number of nodes in the WG. We used the bad generator to create invalid WGs and produced graphs such that there were seven partitions and the violation occurred at the fourth partition. The results are shown in ***Figure 3***.

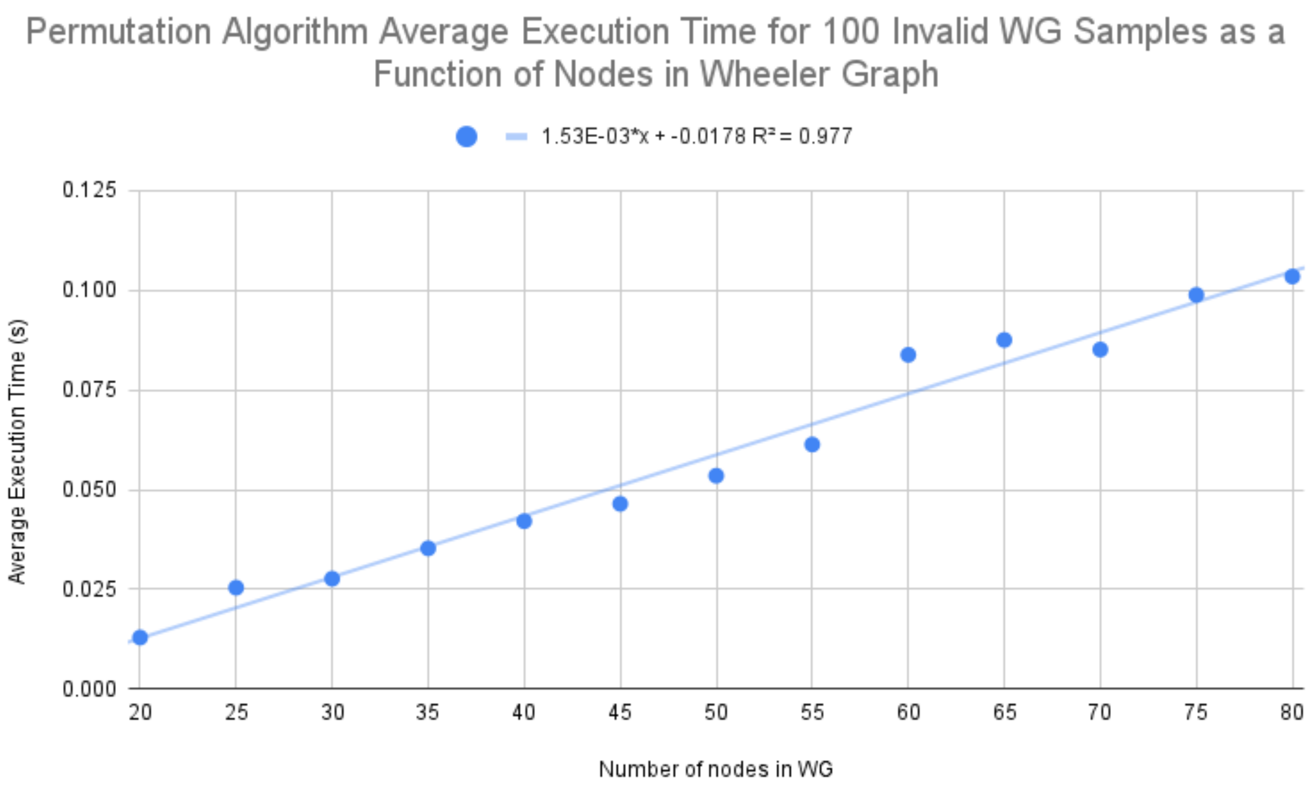

Figure 3 *Permutation Algorithm Average Execution Time for 100 Invalid WG Samples as a Function of Nodes in Wheeler Graph*

The figure above demonstrates the data collected, fitted with a linear trendline and an R-squared value of .977. We can observe that the time complexity to determine invalid Wheeler Graphs (independent of where the violation occurs) is $O(n)$. This is because after initializing a graph and relabelling nodes within each edge group and each in-node list tie sub-group, if the relabelled node labels violate the Wheeler graph properties, they are rejected directly. We call those "obvious violations", and they are a subset of invalid Wheeler graphs that can be detected in linear time. From the plot, we observe that when the node number is under 80, the obvious violation dominates most of the cases. There are a few outliers starting to occur when the node number is larger than 80, and their runtime are expected to be on the same order as for determining valid Wheeler Graphs and largely affected by the depth of error that occurs during the permutation.

Finally, we evaluated how the permutation algorithm would perform with WGs for a fixed node size and edge probability, only varying where the violation occurs with respect to the edge labels (using the invalid generator described in the Implementation: Generator section). We tested invalid WGs with 70 nodes and edge probability of 0.2, and partitioned the alphabet in 10, with the results shown in ***Figure 4***.

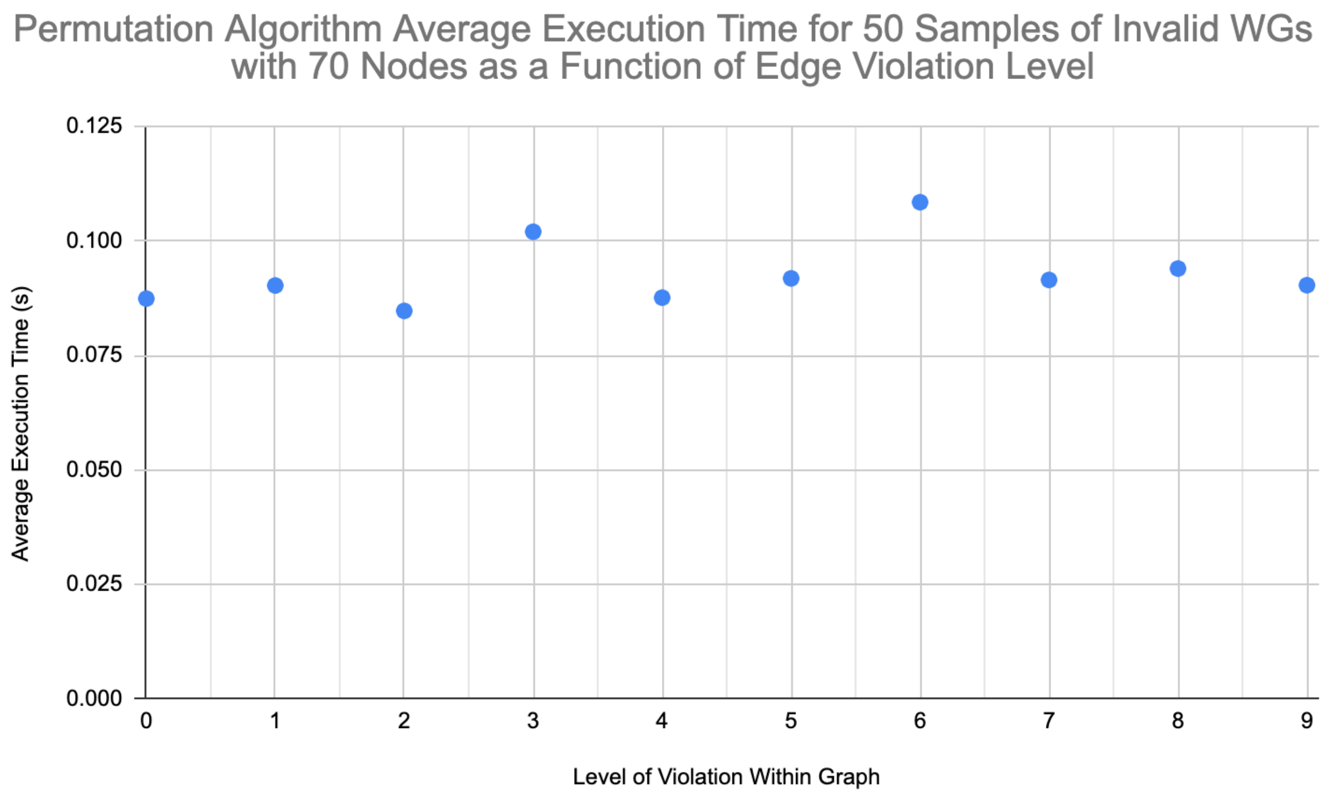

Figure 4 *Permutation Algorithm Average Execution Time for 50 Samples of Invalid WGs with 70 Nodes as a Function of Edge Violation Level*

As seen in the figure above, we found little correlation between where the violation occurred at, and the time needed for the permutation recognizer to determine that it was an invalid WG. This is also because of the obvious violations. They can be rejected after being relabelled, and the program does not need to enter the permutation checking step; therefore, when node number is 70, the obvious violation dominates, and it is roughly constant time to determine invalid WG for a fixed node size, independent of where the violation occurs within the graph. Again, when the node number gets larger, more outliers are expected to occur, and their runtimes are expected to fit the trend for determining valid Wheeler Graphs.

VI. Discussion

The generator can randomly generate valid Wheeler graphs with node labels before and after being shuffled. We also implemented a bad generator which generates directed graphs with node labels violating Wheeler graph properties. One thing to note is that although existing node order violates the Wheeler graph properties, they are not necessarily invalid; it is the recognizer's job to answer if it is possible to find another set of node labels that fulfill three Wheeler graph properties. It is like a chicken-and-egg problem. Although most of the graphs generated by the bad generator are invalid wheeler graphs, the invalidness cannot be guaranteed before the recognizing step.

We also observed a subset of invalid Wheeler graphs that can be detected in linear time, and we call those obvious violations. Last, for the recognizer, users can choose either to find all Wheeler graph node orders or to just find one and halt the program. The outputted three Wheeler graph bitvectors, *I.txt*, *O.txt*, *L.txt*, can be applied to pattern matching in the future.

VII. Conclusions

We proposed a minimum permutation-based wheeler graph recognizing algorithm and implemented it in C++. Our algorithm is theoretically and practically proven to be faster than the state-of-the-art WG recognizing algorithm proposed by Gibney and Thankachan. The loose lower bound C++ implementation of their algorithm took over 30 hours for one sample on a WGs with 10 nodes, which is impractical for real-world application; as for our algorithm, recognizing directed graphs with less than 100 nodes are computationally tractable on a desktop computer. To test the correctness of our recognizer, more than 49,000 randomly generated valid WGs with node numbers ranging from 4 to 100 are tested and verified. Moreover, we observed a subset of invalid Wheeler graphs that can be detected in linear time and named the subgroup as "obvious violations". They dominate most of the cases when the node number is under 80. Last, a series of benchmarks have been done to show the characteristics of our algorithm in response to different directed graph topologies, for instance, the node number and edge probability. In sum, we hope our new algorithm and tool can bring a new perspective on the wheeler graph recognizing problem and push the wheeler graph field forward.

VIII. Reference

1. Burrows, M., & Wheeler, D. (1994). A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*.
2. Seward, J. (1996). The bzip2 home page. URL http://www. bzip. org.
3. Schindler, M. (1999). The szip home page. URL http://www.compressconsult.com/szip/.
4. Langmead, B., & Salzberg, S. L. (2012). Fast gapped-read alignment with Bowtie 2. *Nature methods*, *9*(4), 357-359.
5. Li, H. (2013). Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*.
6. Preston, C., Arnavut, Z., & Koc, B. (2015, August). Lossless compression of medical images using Burrows-wheeler transformation with inversion coder. In *2015 37th annual international conference of the IEEE engineering in medicine and biology society (EMBC)* (pp. 2956-2959). IEEE.
7. Gagie, T., Manzini, G., & Sirén, J. (2017). Wheeler graphs: A framework for BWT-based data structures. *Theoretical computer science*, *698*, 67-78.
8. Mantaci, S., Restivo, A., Rosone, G., & Sciortino, M. (2005, June). An extension of the Burrows Wheeler transform and applications to sequence comparison and data compression. In *Annual Symposium on Combinatorial Pattern Matching* (pp. 178-189). Springer, Berlin, Heidelberg.
9. Mantaci, S., Restivo, A., & Sciortino, M. (2005, March). An extension of the Burrows Wheeler transform to k words. In *Data Compression Conference* (p. 469). IEEE.
10. Mantaci, S., Restivo, A., Rosone, G., & Sciortino, M. (2007). An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, *387*(3), 298-312.
11. Ferragina, P., Luccio, F., Manzini, G., & Muthukrishnan, S. (2005, October). Structuring labeled trees for optimal succinctness, and beyond. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)* (pp. 184-193). IEEE.
12. Bowe, A., Onodera, T., Sadakane, K., & Shibuya, T. (2012, September). Succinct de Bruijn graphs. In *International workshop on algorithms in bioinformatics* (pp. 225-235). Springer, Berlin, Heidelberg.
13. Durbin, R. (2014). Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics*, *30*(9), 1266-1272.
14. Grossi, R., Gupta, A., & Vitter, J. S. (2003). High-order entropy-compressed text indexes.
15. Gibney, D., & Thankachan, S. V. (2019). On the hardness and inapproximability of recognizing wheeler graphs. *arXiv preprint arXiv:1902.01960*.

16. Gibney, D. (2020, July). Wheeler Graph Recognition on 3-NFAs and 4-NFAs. In *Proceedings of the Open Problem Session, International Workshop on Combinatorial Algorithms, Pisa, France* (pp. 23-25).
17. Aguila, E., Shah, S., Chao, K., & Erdogdu, B. (2021, December). A Suite of Wheeler Graph Tools. In *Computational Genomics: Sequences, Fall 2021*.
18. Hagberg, A., Swart, P., & S Chult, D. (2008). *Exploring network structure, dynamics, and function using NetworkX* (No. LA-UR-08-05495; LA-UR-08-5495). Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

IX. Teamwork

Eduardo Aguila	**Kuan-Hao Chao**
Summary: 1. Developed Generator in Python 2. Benchmarked on both recognizers 3. Final writeup Detailed: From our last course (Computational Genomics) we had already created a generator. I first worked to remove some minor bugs that we had, and then worked to produce a suite of generators, as outlined in the paper. After the implementation of both recognizers were complete, I created multiple scripts to randomly generate different samples and collect the timing data. I then cleaned the data, performed the analysis seen in the paper, and created the figures.	Summary: 1. Permutation-based recognizer 2. Gibney's & Thankachan's exponential recognizer 3. Correctness unit test 4. Final writeup Detailed: We had already developed the permutation-based recognizer in our last course. For this final project, I first did a bunch of tests to confirm its correctness, fixed some bugs for some special cases, and then implemented Gibney's and Thankachan's exponential algorithm to the point of enumerating three bitvectors. Finally, I used generator to generate 49,000 graphs and tested them on my recognizer.