An Interactive Approach to Wheeler Graph Educational Materials.

Joanna Bi, Collin Hughes, Julia Ross, and Brandon Stride

Abstract

This project is a comprehensive introductory educational tool for Wheeler Graphs. Through the creation of a web application which includes definitions, interactive demonstrations, and functioning applications of Wheeler Graphs, interested persons can gain a thorough understanding of what Wheeler Graphs are and why they are a valuable data structure. The application is composed of four main sections: the Tutorial Page, the Visualize Page, the Pattern Matching Page, and the Find Ordering Page. The Tutorial Page includes the background information necessary to understand what Wheeler Graphs are and the motivation behind them. The Visualize Page allows a user to generate, view, and download Wheeler Graphs using their own string inputs. The Pattern Matching Page traces the pattern matching process for a given pattern P using bit vector inputs. And finally, the Find Ordering Page computes a valid wheeler ordering for an unordered graph if doing so is possible based on the graph structure and size.

Introduction

Wheeler Graphs are a valuable data structure similar to the established FM index which offer compact storage and pattern matching capability. A primary motivation for the use of Wheeler Graphs is in their application to genomics. When working with large genomes, it is difficult to compare entire genomes for variation. The standard approach currently employs a reference genome which is compared to alternative sequences. Doing so creates bias towards the genetic composition of the reference genome (Garrison, E., Sirén, et al., 2018). Wheeler Graphs present a solution to this problem. Since multiple sequences can be represented by one graph, a Wheeler Graph can compactly represent a diverse body of genetic information (Alanko, J., D'Agostino, G., Policriti, A., & Prezza, N., 2021). This prevents bias while occupying a reasonable amount of space.

With these motivations in mind, our project is aimed towards furthering the understanding of Wheeler Graphs by the genomics community as well as other computer scientists with related data compression goals.

Prior Work

There are few educational materials online about Wheeler Graphs–only a handful of university lectures. Professor Langmead's lectures are the most accessible and are often referenced and used in other institutions' lectures (Langmead, n.d.). There is a clear void in the quantity and variety of educational materials available. We were unable to find any educational materials that included interactive features like those in this project.

Pattern matching using Wheeler Graphs has been explored in the literature. Gagie 2017 summarizes the existing approaches to pattern matching for Wheeler Graphs (Gagie, T., Manzini, G., & Sirén, J., 2017). These include utilizing the L and C (string and indices of sorted string) data structures to query or newer approaches which use circular pattern search (Gagie, T., Manzini, G., & Sirén, J., 2017).

This project builds on prior work on the recognition of Wheeler Graphs in practical amounts of time. While this is an NP-complete problem, various papers have attempted to cut time in practice. Gibney and Thankachan optimized this computation to be polynomial-time solvable for an identified class of graphs (Gibney, D., & Thankachan, S. V., 2022).

Methods and Software

Our application is a React App with routing to distinguish between the five usable tabs. The graph components used for visualizing wheeler graphs throughout the project utilize the React Flow library. Python is used to implement most computational work and is connected to the application through Python Flask. Additionally, the Find Ordering Page links to an alternative implementation utilizing OCaml, which provides exciting time savings. However, the OCaml implementation for the Find Ordering tab is not currently connected to the Wheeler Graph Application. We'll now detail the Interactive Graphs found on the tutorial page, as well as the Visualize, Pattern Matching, and Find Ordering tabs' software and algorithms, as well as analyze their accuracy and space/time complexity performances. We focus on these tabs because they required significant computational work and algorithms relating to computational genomics.

Interactive Graphs

On the tutorial page of the Wheeler Graph Application are two separate Interactive Graph Builders (distinguished by the fact that the second one has some additional capability not described until later in the tutorial) that allow users to become more familiar with the definition and rules that make graphs "Wheeler" or not. It is easy and intuitive to use the graph builders, and they provide live feedback as to whether the user has built graphs with the Wheeler property. Importantly, as the user adds new nodes to the graph, they are always added in increasing order (ie, the first node is labeled 0, the next is 1, and so on…) This innate ordering makes the computational task of determining whether the graph is Wheeler much faster than if the nodes came unordered (see the Find Ordering Section for why). There are two simple algorithms used by the Interactive Graph Builders:

1. **Check if a graph is Wheeler or not.**

This algorithm simply checks that a graph satisfies the definition of a Wheeler graph defined by Gagie in section 2 definition 1 of this paper. We do so by following these steps: first check that if a node has indegree 0, it is earlier than any other nodes in the ordering which have positive indegree. This step requires linear time and space in the number of nodes and edges. Next, we check each pair of edges in the graph and ensure they satisfy the rules of Wheeler graphs, returning true if only if each pair satisfies the definition. This step takes quadratic time in the number of edges and linear space.For the small graphs built in the Interactive Graph builder, this algorithm is easily fast enough to provide immediate results whenever the user changes the graph, and there are no latency problems.

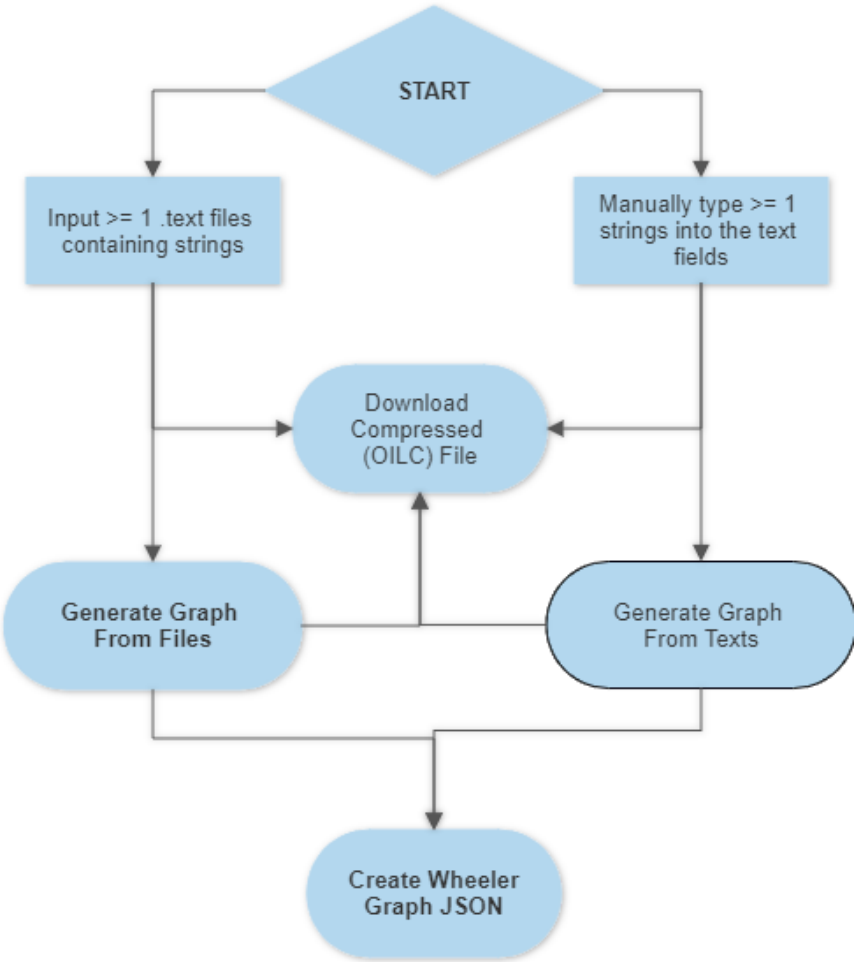
2. **Generate the OILC representation of a Wheeler graph (only used in the second Interactive Graph Builder)**

This algorithm is described in the next section (see the third algorithm for the visualize tab). For small graphs built in the interactive graph builder, this feature also works quickly enough to show live results as the user changes their graph and doesn't cause latency problems.

Overall, the Interactive Graph Builders on the tutorial page are (to our knowledge) fully accurate, and always fast enough to show live results as it is used. They are a good feature for teaching and visualizing Wheeler graphs and can definitely be used to help interested persons gain strong understandings of the properties of Wheeler graphs. An additional success of these components is that the graphs users build can be downloaded as JSON files and used as input for the Find Ordering Tab. For example, if a user is confused as to why the graph they have built is not Wheeler, they can attempt to find a valid Wheeler ordering for it in the Find Ordering Tab.

Visualize Tab

The Visualize tab allows users to input their own string data and visualize it as a Wheeler graph. The user can either input .txt files or type in their own strings. In addition, to make our tabs work together well, the user can then download either a large (json) version or a small (OILC) version of their Wheeler graph to be used in the find ordering or pattern matching tabs respectively. While our primary objective with the overall application is to provide an interactive teaching and visualization tool, the visualize tab also allows the user to forego the visualization of a large Wheeler graph (say for example the Wheeler graph made from a small genome) and instead immediately download the compact OILC representation of the data. For clarity, here is a graphic of the flow one might follow in using the visualize tab:



There are 3 important algorithms at work when the visualize tab is used:

1. **Create a Wheeler Graph from a single string.**

This algorithm follows the methodology described in section 3 of this paper (Gagie et. al 2017). We first break a string into the set of all its prefixes and create nodes for each prefix. Then, we connect each prefix to the next longest prefix with an edge containing the next letter. For example, in the string ‘dog’, the nodes would be the prefixes including a ‘$’ for the empty string: ‘$’, ‘d’, ‘do’, and ‘dog’. There would be an edge from ‘$’ to ‘d’ labeled with ‘d’, an edge from ‘d’ to ‘do’ containing ‘o’, and so on. After the nodes and edges are created, we order the

nodes from 1 to n where n = (the length of the string + 1) by lexicographical ordering of the reverse of the prefixes. Thus, in our example the four nodes would be ordered 1 to 4 in this order: '$', 'd', 'dog', 'do'. With each node ordered and connected by labeled and directed edges, the result is then passed to React Flow to be visualized.
Splitting up a string into its prefixes (and creating nodes and edges) can be done in O(n) time over the length of the string. However, since we need to store each prefix of the string in order to sort them reverse lexicographically, the space and time usage of this algorithm is actually O(n^2).

2. **Create a Wheeler Graph from multiple strings.**

This is an extremely similar process to the previous algorithm, though in this case, any number of strings can be provided as input, and a trie will be built out of the strings before the prefixes are ordered. Ordering is still done by lexicographical ordering of the reverse of prefixes. Importantly, tries are guaranteed to have unique Wheeler orderings no matter what the strings are because all prefixes in a trie are *uniquely orderable* by reverse lexicographical ordering. The time and space requirements for this algorithm are equivalent to the previous as well, only now n represents the sum of lengths of all input strings.

3. **Generate the OILC file for a string or multiple strings.**

The OILC method is perhaps the most problematic in our application, since its algorithm follows this method:

a) Find the nodes and edges and order them (using one of the previous two algorithms).
b) Use the information obtained from the nodes and edges to generate O, I, L, and C per the specifications in section 2 of this paper.
c) Compress L.

Step (a) unfortunately takes quadratic time and space. Timewise, Python can still sort prefixes for node ordering decently quickly even for somewhat large datasets, but this space bound poses a big problem for large datasets since the deployed server can only handle requests that use <4GB of storage. For example, trying to generate the OILC for any string(s) with length ~ >90,000 will not work due to large storage requirements of step (a). Steps (b) and (c) on the other hand will only take linear time in the length of the string(s). We expect there is a more space and time efficient way of generating O, I, L, and C that doesn't require step (a) and allows the creation of a Wheeler graph compressed index to be completable in linear time, but we were unable to implement something like this.

Overall, the visualize tab is best suited for visualizing small Wheeler graphs but can also be used to generate OILC files for up to medium sized datasets (please don't try to create the OILC file for a human genome with our app 🙂). Another success from the visualize tab is that it can also be used to generate the input data for the pattern matching tab. The user can take a downloaded OILC from their data and use it in the pattern matching tab to try to find desired substrings in their text, which we describe next.

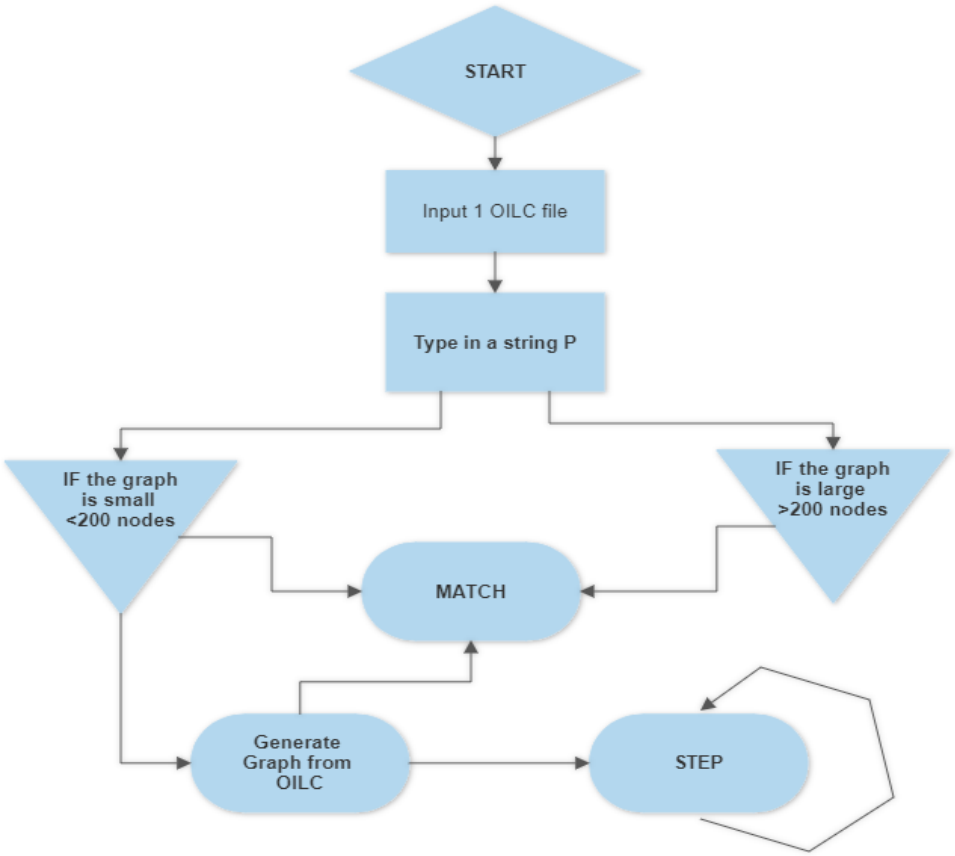
Pattern Matching Tab

Like the visualize tab, the pattern matching tab is primarily meant for visualization and interactive learning about the pattern matching process with Wheeler graphs. To use the tab, the user first inputs an OILC file for a Wheeler graph (such as one they downloaded from the visualize tab) and manually types a string P they would like to match.

If the inputted Wheeler graph OILC is sufficiently small (< 200 nodes), then the user can generate the Wheeler graph visualization and step through the matching process letter by letter

until P terminates. At each step, nodes corresponding to the current letter in P will be highlighted green. When they have finished stepping through the entire string, the total number of times the string P occurs in the Wheeler graph will be displayed, and all nodes which correspond to the termination of P will be highlighted green. For example, if a Wheeler graph for the following strings is created: ‘intern’, ‘internet’, and ‘internal’, and the user inputs a string P: ‘inter’, then they can step through the pattern matching process letter by letter, and when it concludes, the app will display that P was found three times. The user could have also foregone the letter by letter stepping, and clicked a different button to skip to the end of the matching process and they would still see the same result that P was found three times.

If, however, the user inputs an OILC file that is large (>200 nodes), they can only click the ‘match’ button to see if P occurs in the string. They can’t step through P and visualize the process along the way because the graph is just too big to show on our screen. The match button will not always provide the exact number of occurrences of P for multi string Wheeler graphs, but it will provide the minimum number of occurrences of P (more on that later). Here is a visual of the flow of actions a user can take when using the pattern matching tab:



The pattern matching tab uses two primary algorithms:

1. **Create a graph from an OILC file.**

This algorithm uses information from O, L, and C to create a list of ordered nodes and a list of directed, labeled edges. **Please Note** that since we are not using I, (the 'I' part of OILC), our application is currently unable to convert OILC files to Wheeler graphs IF any nodes from the OILC file have an indegree > 1. This is never the case for string and trie Wheeler graphs, which is the primary focus of our application. However, a general Wheeler graph can absolutely have nodes with indegree >1, but we don't have that implemented yet for pattern matching. The algorithm for creating a Wheeler graph from an OILC file is O(n) in time and space complexity, where n is the number of nodes in the Wheeler graph. Notably, here we don't need to concern ourselves with lexicographic sorting since we already have node ordering explicitly from the O data structure in OILC. That's why this graph generator algorithm is linear instead of quadratic like the generator algorithms from the visualize tab.

2. **Pattern match a string P using the OILC representation of a Wheeler graph.**

This algorithm is used when the user clicks the 'MATCH' button only (see the diagram above). The algorithm takes in O, L, C, and P, and uses information from these data structures to trace P through the graph. Note that this algorithm doesn't translate O and L to nodes and edges before tracing P; it just uses them as given to be as optimal as possible. That said, the algorithm is *not* as optimal as possible since we didn't fully implement a rank-select data structure to augment O, L, and C. Such data structures are sublinear in space and allow constant time queries (see Jacobson's rank and Clark's select algorithms). It's hard to give an exact big-O time bound to the algorithm because of the specific implementation, but it's certainly better than quadratic in the number of total nodes in the graph. The overall idea of the algorithm is very similar to the pattern matching process for an FM index but is not totally optimized for speed. For example, if you input a small covid-sized genome and a string P with |P| < 10, then the algorithm will succeed in a couple seconds maximum. This seems to scale up linearly with the size of the input genome.

A final note on the number of occurrences of P. When the user chooses to use the STEP function to find P in their Wheeler graph, the algorithms involved actually use the nodes and edges generated from the OILC, not the OILC itself. This is because of simplicity and the fact that these graphs are guaranteed to be <200 nodes, and latency will never result from this. As a result, even for trie-based Wheeler graphs, the exact number of occurrences of P is calculated by using DFS on the nodes when P is fully stepped through (we discussed this algorithm very early in the semester for counting occurrences of P in a trie or suffix trie). However, when the MATCH function is used, DFS is not because, as stated in the previous paragraph, the match function actually uses the OILC representation for speed purposes. We did **not** implement the DFS in that algorithm (because that was a stretch idea we didn't get to). Therefore, when the MATCH function is used, it provides a minimum for the number of times P occurs instead. If the graph is from a single string, that will be the exact number of occurrences, but if the Wheeler graph is from a trie, the count of P is not guaranteed to be exact.

Overall, the pattern matching page is relatively complete for string and multi string Wheeler graphs, but it could be improved in the future by allowing for general Wheeler graphs to be traced as well. The visualization of tracing the pattern matching process is nice for teaching about pattern matching because it shows visually how the range of correct nodes gets progressively smaller as the user traverses P letter by letter. The match function is a nice alternative to demonstrate the practicality of Wheeler graphs for pattern matching because you can input a small genome OILC file and get pattern matching results reasonably quickly.

Find Ordering Tab

The final tab in the Wheeler Graph Application is the Find Ordering Tab. This tab allows users to input normal graph JSON files for *unordered* graphs and try to find a Wheeler ordering for a sufficiently small graph. Correctly structured input files can be obtained by downloading graphs from either the Interactive Graph Builders on the tutorial page or the visualize tab, though graphs from the visualize tab already come with valid Wheeler orderings guaranteed. (One might use the findOrdering tab to test the visualize tab's correctness if they don't trust its results!) Usage of the findOrdering tab is self explanatory. Now we'll describe the software and strategies it uses to find Wheeler orderings as quickly as possible.

To find a potential ordering on a graph, we first rule out graphs that will always violate the Wheeler criteria. This can be done in polynomial time. First, if there is a node with incoming edges with at least two distinct labels, then no ordering can satisfy rule two. Further, if a graph contains a cycle using exactly one edge label, then there is no possible ordering such that rule three is satisfied. Once these graphs are ruled out, we try orderings on the graph.

Not all orderings on a graph need to be tried. To limit the orderings, we consider that incoming edge labels partition the nodes, creating equivalence classes. Nodes in these equivalence classes are constrained to a compact subset in the ordering, and the equivalence classes are ordered with respect to each other and occupy disjoint ranges in the ordering. Thus, there are *suborderings* (i.e. orderings within the equivalence classes) of a Wheeler graph, and these can be found quickly if the equivalence classes are not large. It is fast to try all suborderings for small equivalence classes. For example, if the equivalence class has size no greater than 8, then the number of suborderings for the given class is not greater than 8! = 40320, a relatively small number. Any suborderings that violate rules two or three do not need to be considered in the total orderings of the nodes. Once all valid suborderings are found, we try all possible combinations of the suborderings. This can be done with brute force or by continuing to rule out failing sub-orderings as they are combined.

When the graph is small and an ordering can be found in a few seconds, there is not a consistent best method to combine suborderings. Simply trying all combinations of orderings and checking if the Wheeler properties are satisfied is quite fast. Another method considers two equivalence classes at a time and merges them, ruling out any orderings that fail rules two or three after the merge. After adjacent equivalence classes are merged, the process repeats until all are merged into one total ordering. Then, rule one is checked on the remaining orderings. This method appears to be slower on average in practical uses.

We implement the above methods in Python. However, Python is a notoriously slow language, and while this problem is NP-complete, and the worst-case grows exponentially, we still strive for every bit of optimization. For this reason, we also chose to implement the methods in OCaml, a functional programming language that is much faster than Python. It is difficult to get Python to communicate with OCaml, and OCaml is slow to build, so we compile the OCaml solution into an executable. When this executable is run as a subprocess from Python, it is slower than the original Python code, but standalone, it is a fast command-line executable for discovering orderings on Wheeler graphs. The executable is found at algorithms/caml/order.exe, and the command-line arguments are described in algorithms/caml/order.ml.

We quickly find orderings on graphs for which a brute-force algorithm would take far too long. Consider the following results, where N = number of nodes, E = number of edges, L = number of distinct edge labels, and the test files are in algorithms/unit_test_files/'Find Ordering Test Files'/wheeler/.

Test file	N	E	L	Brute force orderings	Orderings tried
complex_orderable1.txt	8	13	4	40320	24
complex_orderable5.txt	15	16	5	1.3 * 10^12	336
complex_unorderable1.txt	12	15	5	4.8 * 10^8	0
complex_unorderable2.txt	12	14	5	4.8 * 10^8	72
large_orderable1.txt	21	26	5	5.1 * 10^19	128,000
large_orderable3.txt	24	26	5	6.2 * 10^23	32
large_orderable4.txt	33	33	17	8.7 * 10^36	3,359,232
large_unorderable1.txt	21	27	5	5.1 * 10^19	38,400

These results show considerable improvements over a brute-force algorithm with no optimization. Graphs up to ~24 nodes are solvable in a few seconds, and the largest graph with 33 nodes is solved in a few minutes. In the worst case scenario, a graph with 33 nodes and 33 edges on 10 billion different computers each processing 10^10 iterations over graph elements per second would take 3 trillion years to try all orderings.

Conclusions

Wheeler graphs are highly visual data structures and therefore translate well to visual teaching aids. This project contributes to the current educational resources surrounding Wheeler graphs. Further, this project contributes to the current body of work in computing pattern matching using Wheeler graphs and finding Wheeler orderings as quick as possible.

As we've stated throughout, the main use of our project is as an interactive teaching and visualization tool that could be useful for students and computer scientists with background knowledge in why compressed indexes of genomes (or other large datasets) are useful. We think the best use case is for students who have an existing knowledge of FM indexes, as Wheeler Graphs are best understood as alternatives to FM indexes but with perhaps more generality to other non-genomics applications as well as, of course, great visualization capability.

Overall, we learned a lot about Wheeler graphs and indexes both from the building of the Wheeler Graph Application with its associated algorithms and in testing it. We think future students could access the algorithms behind our application and build on them, continuing the quest towards optimal data processing for Wheeler graphs.

We also hope our application is used by someone in the future to help them learn about Wheeler graphs or acquire Wheeler graph data!

Literature Cited

1. Alanko, J., D'Agostino, G., Policriti, A., & Prezza, N. (2021). Wheeler languages. *Information and Computation*, *281*, 104820. https://doi.org/10.1016/j.ic.2021.104820
2. Gagie, T., Manzini, G., & Sirén, J. (2017). Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, *698*, 67–78. https://doi.org/10.1016/j.tcs.2017.06.016
3. Garrison, E., Sirén, J., Novak, A. M., Hickey, G., Eizenga, J. M., Dawson, E. T., Jones, W., Garg, S., Markello, C., Lin, M. F., Paten, B., & Durbin, R. (2018). Variation Graph Toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, *36*(9), 875–879. https://doi.org/10.1038/nbt.4227
4. Gibney, D., & Thankachan, S. V. (2022). On the complexity of recognizing Wheeler graphs. *Algorithmica*, *84*(3), 784–814. https://doi.org/10.1007/s00453-021-00917-5
5. Langmead, B. (n.d.). *Wheeler graphs, part 2*. Retrieved October 8, 2022, from https://www.cs.jhu.edu/~langmea/resources/lecture_notes/260_wheeler_graph2_pub.pdf