A Suite of Wheeler Graph Tools

Eduardo Aguila (eaguila6@jhu.edu), Shaunak Shah (sshah98@jhu.edu),
Kuan-Hao Chao (kchao10@jhu.edu), Beril Erdogdu (berdogd1@jhmi.edu)

Abstract

Wheeler graph (WG) is a framework that provides a unifying view on the BWT variants in strings, de Bruijn graphs, labeled trees, etc. In this project, we developed a suite of WG tools: a visualizer to visualize WG graphs, a generator to generate a random graph that satisfies all conditions of a WG, a checker to validate whether a graph is a WG or not, a recognizer to determine if a graph can be labelled in such a way that satisfies the WG properties, and a pattern-matcher to check if a substring is encoded in a WG. The highlight of the paper is that we designed a new factorial algorithm to solve the NP-complete WG recognizing problem, and it is faster than the state-of-the-art algorithm proposed in 2019. Our WG tools are open-sourced on GiHub: https://github.com/shaunak215/Genomics-Final-Project

Keywords: wheeler graph, visualizer, checker, generator, recognizer, pattern matcher
Abbreviation: Wheeler graph (WG), Burrow-Wheeler Transformation (BWT)

I. Introduction

Burrow-Wheeler Transformation (BWT) has been studied for a long time and is widely used among several fields. For example, a popular compression tool bzip2 was built based on the BWT algorithm (Seward 1996); Bowtie2 (Langmead and Salzberg 2012) and BWA (Li 2013) are important BWT-based aligner in the computational biology field. 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”. In Gagie’s point of view, we were approaching BWT as blind men approaching elephants. Therefore, in 2017, he proposed a new framework, Wheeler graph (WG), to provide a unifying view on the BWT variants (Gagie, et al. 2017). The notion of a WG can help processing strings; even if the state of a problem is a Nondeterministic Finite Automata (NFA), given it is a WG, by applying the path coherence property, pattern-matching and traversing become a linear time problem. Moreover, a WG can be compactly store into three bitarrays, O, I, and L. Thus, WG is a time and space efficient solution and has the potential to booster research on BWT variants, new indexing data structure, and multi-reference genome.

Later, Gibney and Thankachan published a theoretical paper on arxiv in 2019 discussing a series of WG-related mathematical problems. Out of all of them, the most fundamental and important one is the WG recognizing problem. It was proven to be NP-complete and yet has not been well-solved (Gibney and Thankachan 2019). In 2020, Gibney further divided the WG recognizing problem into smaller subproblems. Defining d-NFA as a directed graph with any vertex having at most d edges with any particular label, a 2-NFA WG can be solved in polynomial time, recognizing a 5-NFA WG is shown to be NP-complete, and the time complexity of recognizing 3-NFAs and 4-NFAs is still an open question (Gibney 2020). WG is a relatively new topic, and it lacks open-source tools and libraries. There is plenty of space for us to explore. Intrigued by the WG recognizing problem, we further asked a series of problems related to WG and decided to build a suite of WG tools for the final project.

We can mathematically define a Wheeler Graph as an edge-labeled directed multigraph, where the nodes can be labeled such that the following three conditions hold:

1. 0 in-degree nodes come before others

For all pairs of edges e = (u,v), e'= (u',v') labeled a, a' respectively, we have:

2. $a \prec a' \Rightarrow v \prec v'$
3. $(a = a') \wedge (u < u') \Rightarrow v \leq v'$

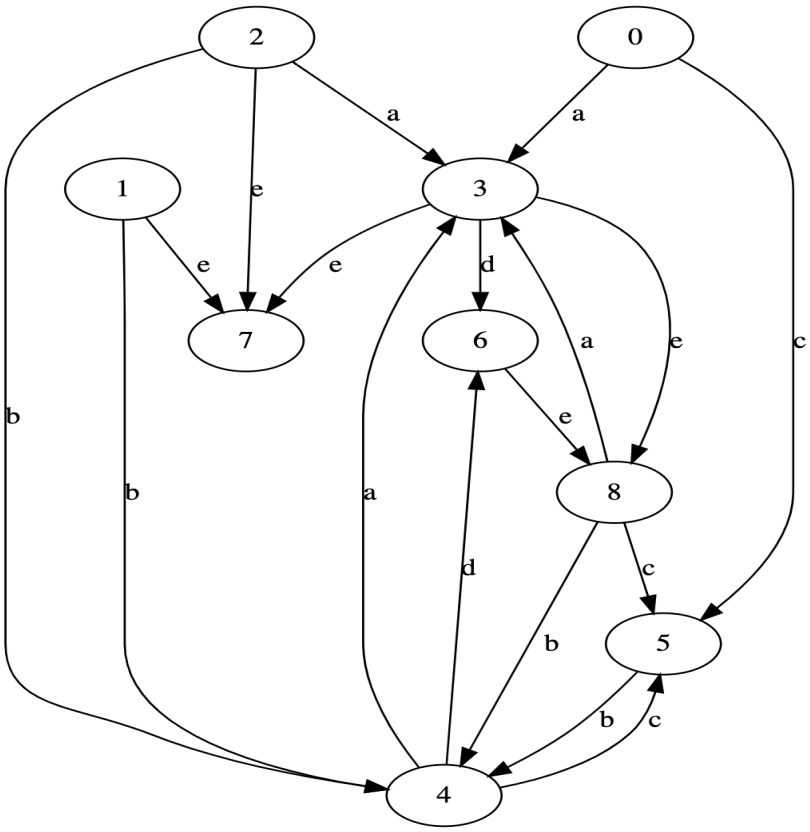


Figure 1 Example of a Wheeler Graph

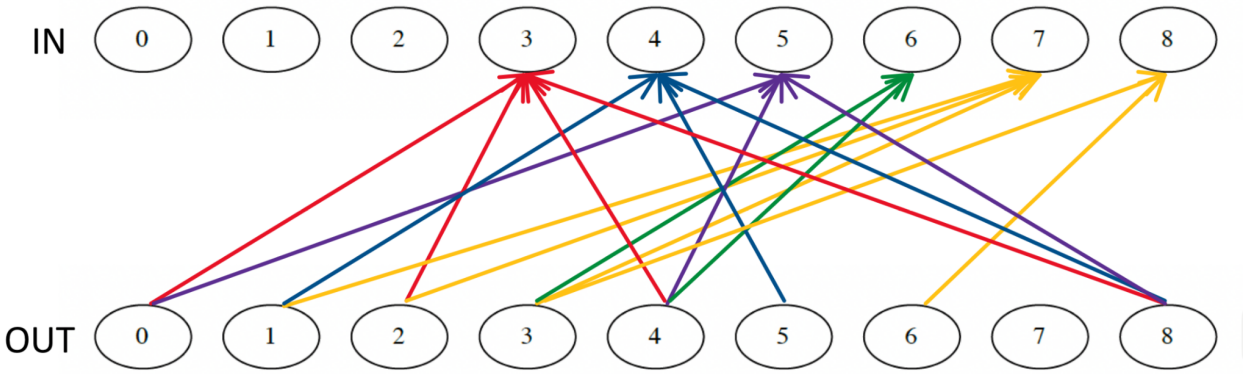


Figure 2 Visualization of Wheeler Graph by splitting the nodes into two tiers, according to their in/out edge node labels (a: red, b: blue, c: purple, d: green, e: yellow)

From the above figure, we can visually validate each of the conditions that we mathematically defined above. In layman's terms, the conditions become the following:

1. All the zero-indegree nodes (in the top row) come before any other nodes
2. There are strict partitions in the top row for each edge label, and they follow increasing lexicographic order (i.e., all a's come before all the b's, etc.)

3. No two edges of the same label (i.e. color) cross each other.

There are then a variety of problems that apply to wheeler graphs, the ones we chose to explore are the following: visualization, generation, checking, recognition, and pattern matching. In the following section, we are going to dive into each tool.

II. Methods and Software (1): Checker

For our checker, we built a python program that was able to determine if a provided graph, supplied in '.dot' format, met all the conditions necessary to be a WG. If the graph is not a WG, the program outputs the conditions that were violated. The program takes a single command line argument, which is the file path to the dot file. Our checker is only compatible with graphs that use integer labels for the nodes. We found this to be the most common data type for node labels, as opposed to nodes being labeled with strings. Additionally, we found it cumbersome to deal with unintuitive lexicographical string comparison in node ordering.

We checked each condition in order, as per the previous definition of a WG. Therefore, we first checked that all zero in-degree nodes come before other nodes in the graph. We utilized the networkx function `` `G.indegree(G.nodes())` ``, which returns a list of tuples that contains all the nodes in the graph, along with their indegree, in the form of [(node, ind), … ,(node, ind)], which we then sorted based on the indegree. We then iterated through the list, and if we found any node with zero in-degree, after a node with non-zero indegree has already been observed, then condition 1 does not hold. Checking the first condition is thus linear on the number of nodes.

Next, we checked the second condition that all node labels for each incoming edge label group are strictly greater than all those in the alphabetically previous edge label group. This was accomplished by first creating a dictionary mapping edge labels to a list of their in/out nodes. We then iterate through the dictionary's keys in alphabetical order and make sure all the in nodes' values for an edge label are strictly greater than the maximum of the previous edge label group. If a node value is less than or equal to the maximum then condition 2 does not hold. Thus, the total work done is $O(ELogE + N)$ where E is the number of edge label groups and N is the number of edges. This is because it takes $O(N)$ time to build the dictionary (which we also use in condition 3), $O(ELogE)$ to sort the dictionary's keys, and $O(N)$ to iterate through all the edge's in/out nodes.

Finally, we checked the third condition that given all nodes contained in an edge label group and the out nodes are sorted, the in nodes should be sorted as well. We re-utilized the dictionary created in step 2 above to get a mapping of edge labels to the in/out nodes. We then get all the in/out nodes for each edge label and sort the pairs of in/out nodes based on the out node's values. From there we iterate through all the in node's values and ensure they are in sorted order. Overall the total work is $O(E(NLogN))$ where E is the number of edge label groups and N is the number of edges. This is because for each of the E edge label groups we sort the node pairs inside and iterate through those.

At this point if all three conditions passed we output that the graph is indeed a Wheeler Graph. However, if any of the conditions fail we output which of those conditions fail with an error message describing the condition. Overall, building the checker was relatively straightforward. However, networkx was reading in the nodes as strings and not ints. This caused many problems so we first attempted mapping the old string

labels to int labels and using networkx's `relabel_nodes()` function which still did not work. Eventually we just iterated through all the nodes and created a new list with the nodes as ints instead.

III. Methods and software (2): Generator

We then proceeded to build the generator, which is another python program that takes two command line arguments: the number of samples to be generated, and the approximate number of nodes in the graph(s). Rather than trying to take on the random graph generation portion, we opted to leverage existing networkx functions. We decided that while not impossible to randomly create nodes/edges, it would be significantly easier to leverage existing networkx functions for random graph generation, and then convert it into a WG by modifying it based on the criteria for WGs. Therefore, our program begins by randomly generating an Erdős-Rényi graph/ binomial graph. We utilized this networkx graph generator with three parameters: the number of nodes (determined by the command line argument), the edge probability (we used .2), and a boolean for if the graph is directed (True).

After generating a graph, we began by checking the first condition. In a similar series of steps as our checker, we created a list of tuples containing all the nodes along with their indegree. We then iterated through this list and created two new lists, based on the in-degree of the nodes. Thus, at this point we have two lists: one containing all the nodes with zero-indegree, and one containing all the other nodes. Note that at this point, if there are no nodes with indegree zero, then nothing needs to be done and we can move on to the next condition. The next step is to relabel all the nodes, essentially swapping them such that any node that has indegree zero is first. The networkx function that we utilized is `relabel_nodes`, which takes a dictionary as its parameter, where the key is the old node label, and the value is the new node label. In order to create this dictionary, we iterate through all the zero indegree nodes in the zero_node list, and enumerate them from 0 to the length of this list. This ensures that every node that has indegree zero comes before any other node. We then need to remap the remaining non-zero nodes, to ensure they come after the zero indegree nodes. We then iterate through all of these nodes, and enumerate them from the length of zero list to N, where N is the number of nodes.

The steps required to satisfy conditions 2 and 3 were somewhat more complicated. At this point, the graph contains only edges with no labels. Therefore, we decided that we would assign edge labels to the graph such that they necessarily satisfied condition 2, and we would only need to validate the final condition. Before assigning edge labels, however, we need to determine the size of the alphabet. After completion of the generator and some testing, we empirically determined that an alphabet size equal to `# of edges / 4` provided a sufficient alphabet. In order to label all the edges, we first need to obtain a list of all the edges, which is accomplished through the networkx `G.edges()` function, which returns a list of all edges as tuples in the form of (from_node, to_node). Given the definition of the second condition, we can loosely interpret it as: if we label the successors of edges according to their edge label, the nodes should be in ascending order. More concretely, an example could be the following dictionary: {a: [1,2,3]; b: [4], c: [6,7]}. Therefore, our first step was to sort the list of edges according to the node they point to. At this point, we then split the list into roughly equal partitions corresponding to the size of the alphabet. For example with 10 edges and an alphabet size of 4, we would end up with a new list containing [2,2,2,4] edges, corresponding to the edge labels "a,b,c, and d" respectively. However, we found that simply splitting the edges into these partitions does not necessarily guarantee condition 2. For example, in the case where the partitions hold edges [[(2,1), (7,2)], [(1,3)], [(3,3)]], corresponding to an alphabet size of 3, with edge labels 'a', 'b' and 'c'. Even though the edges are sorted according to the node they point to, there is not a strict border between the partitions for the edge labels and we see that the to_node for b and c is both 3, which is a

clear violation of condition 2. In order to remedy the above problem we had to move certain edge pairs around. For each partition we got the highest to_node. We then iterated through the rest of the list and looked for any tuples with that same to_node. If any matched, we moved that edge pair into the partition containing the original highest to_node. With this process we ensured that each edge label had unique to_nodes that passed all of condition 2.

To guarantee the graph passed condition 3 we decided to remove any violating edges. At this point if we had relabeled any edges we potentially could be ruining condition 2 which would then need fixing. If we went down that path, there potentially could be an endless loop where it becomes impossible to label the edges such that they pass both condition 2 and 3. In order to do this we created a similar series of steps as the checker for condition 3. First we created a mapping of the edge labels to a list of the respective in/out nodes for that edge. That dictionary was then iterated through by edge labels to get a list of each of the in/out nodes for that edge. That list was then sorted based on the out nodes. Now, we iterated through the list and used networkx's `G.remove_edge()` function to remove any violating edge where the in nodes were not in sorted order.

Since we are pruning the graph by removing edges, it is possible to create new indegree 0 nodes which potentially violate condition 1. In order to do this we first create a sorted list of all the nodes based on their indegree using networkx's `G.in_degree()` function. From there we iterate this list and create a new list with all the nodes that have in-degree 0 and keep track of the highest indegree 0 node that occurs consecutively starting from node 0. We then filter the list to get all nodes that are greater than that highest indegree 0 node and delete those from the graph using `G.remove_node()`. However, since these nodes can be connected to other nodes it is possible that we created new indegree 0 nodes. To remedy this we keep looping through the above steps until there are no new nodes to delete from the previous iteration.
Given our approach, it's clear that we prune away both nodes and edges from the initial graph (for removing edges that violate condition 3, and for removing new node violations of condition 1). Therefore, we performed some data analysis in order to provide a more accurate node count, as specified when the user calls the program with a command line argument. We compared the number of the nodes in the function call to the original graph to the number of nodes that were present in the final graph that met the criteria for a WG. We generated 1000 samples from our generator, with an initial function call ranging from 5 to 100 nodes, in increments of 5, and compared them to the final node count present in the graph. We obtained the following data:

Nodes	5	10	15	20	25	30	35	40	45	50
Actual	4.766	7.808	12.233	17.815	22.376	26.228	29.763	32.688	35.377	37.879

Nodes	55	60	65	70	75	80	85	90	95	100
Actual	40.117	42.327	44.22	46.054	48.021	49.721	51.423	53.144	54.915	56.633

After plotting the data, we obtain the following Graph:

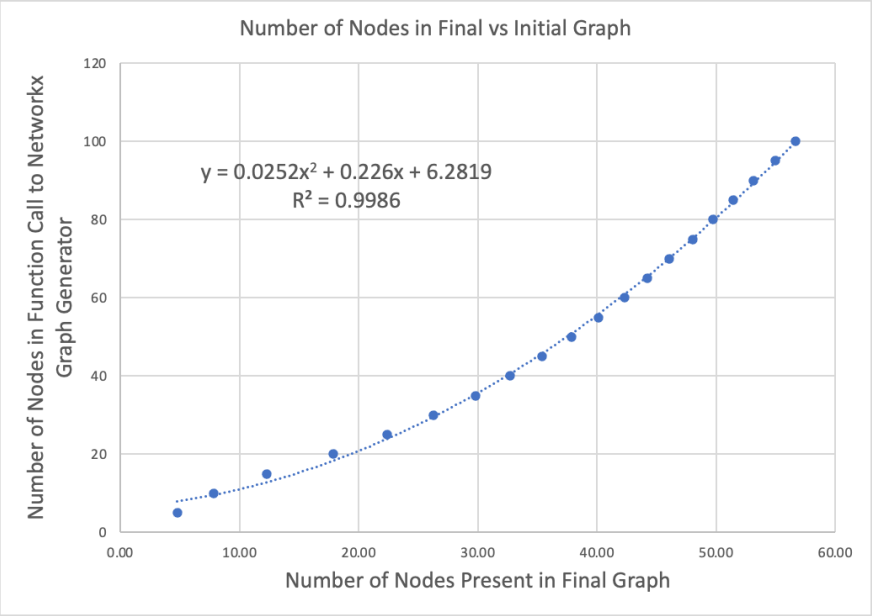


Figure 3 Visualization of number of nodes in final graph vs initial graph.

We then added a degree 2 polynomial trendline, and obtained the following equation:

$$y = 0.0252x^2 + 0.226x + 6.2819$$

Given the r-squared value of .9986, we determined that this was a good fit for the data. We thus modified our generator such that it would use the above formula to determine how many nodes the program should call the networkx generator with, such that the number of nodes in the final graph would be approximately equal to the number of nodes the user entered on the command line.

IV. Methods and Software (3): Recognizer

Before diving into the recognizer, we first define the WG recognizing problem:

Problem Definition: Wheeler graph recognizing

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

IV-I. Comparing with the state-of-the-art algorithm

To solve the WG recognizing problem, an intuitive idea is to check permutations of all node labels, and its time complexity is $O(N!)$, wherse N is the number of nodes in a given graph; however, this factorial algorithm is slow and not scalable. In 2019, Gibney and Thankachan proved that the WG recognizing problem is NP-complete (Gibney and Thankachan 2019). It turns out that there are no elegant and efficient ways to solve this problem. In the same paper, Gibney and Thankachan further proposed a state-of-the-art exponential pseudocode algorithm to solve the recognizing problem. Their recognizer pseudocode takes the WG data structure proposed by Gagie in 2017 (Gagie, et al. 2017), and its main idea is to enumerate three bitarrays of given length and test if the original graph, G, and the new permutation graph, G', are isomorphic. If yes, then it is a wheeler graph; otherwise, no. If the input graph $G = (V, E)$ with $N = |V|$ (number of nodes), $E = |E|$ (number of edges), and σ is the size of each edge label alphabet, the recognizing problem can be solved in time $2^{E\ log\sigma + O(N+E)}$.

At first glance, it seems that they have successfully reduced the problem complexity from factorial to exponential; however, thinking twice, the graphs that factorial algorithm checks are the subset of the graphs checked by Gibney and Thankachan's algorithm since their algorithm includes checking not only the same graph structure with different node labels, but also completely different graph structures that should be avoided. Then, here comes the question: Is their exponential algorithm slower than the factorial algorithm?

To simplify the question, let us just look at one bitarray at a time. Take O bitarray for example. Its length is $N + E$, and the number of checking times is 2^{N+E}. In contrast, without removing repeat check, the number of checking times for the permutation factorial algorithm is $N!$. Comparing both algorithms, we can see that their exponential algorithm depends on two variables, N and E, whereas the factorial algorithm only depends on N. Thus, we can further ask if we also take the relationship between N and E into consideration and do the change of variables, will the time complexity of the two algorithms match our intuitive observation? Following is the proof.

Proof:

Given a graph G. The number of nodes is N, and the number of edges is E. First, we define G_{O_i} as a group of nodes with the same outdegree. Then, we divide O bitarray nodes into groups by their outdegrees. Let us say that we can divide all the nodes into G_{O_0}, G_{O_1}, G_{O_2}, …, G_{O_k}, k groups, and each of them has at least one node. For each group, the size boundary is $1 \leq |G_{O_i}| \leq N$, and

$$\sum_{i=1}^{k} G_{O_i} = N$$

(E- 1)

If we try all possible node labels, the number of times that it takes is

$$\frac{N!}{\prod_{i=1}^{i=k} |G_{O_i}|!} \qquad (E\text{-}2)$$

The main idea to show factorial algorithm is faster than the exponential algorithm is to select the worst case for the factorial algorithm, make it the best scenario for the exponential algorithm, and then compare their time complexities. Back to the factorial algorithm, the worst case for **(E- 2)** is when nodes are equally distributed in every G_{O_i} group; in other words, every group has the same node number, $|G_{O_0}| = |G_{O_1}| = \cdots = |G_{O_k}|$. We assume the number of nodes in each group is a constant variable a. Therefore, the number of nodes N can be expressed as

$$N = k \times a \qquad (E\text{-}3)$$

Next, we are going to find the best scenario for the exponential algorithm under the condition given above. The number of checking times for exponential algorithm is 2^{N+E}, and to minimize it, we need to make E as small as possible. Since nodes in the same G_{O_i} have the same outdegree, which equals to the number of edges that each node connects to, the total number of edges E can be calculated by summing up all the indegrees of every node in every group.

To calculate E, we are going to assign indegree for each node group first. Since nodes in different groups have different in-degree, the best case to minimize E is to assign an arithmetic sequence with common difference 1 starting from 0, and it is showed in the red row in **Figure 4**. Clearly, this is not the only indegree assignment, and we can randomly assign any strictly increasing natural number list, which is showed in the green row in **Figure 4**. However, the total number of edges calculated in green case is always larger than the red case.

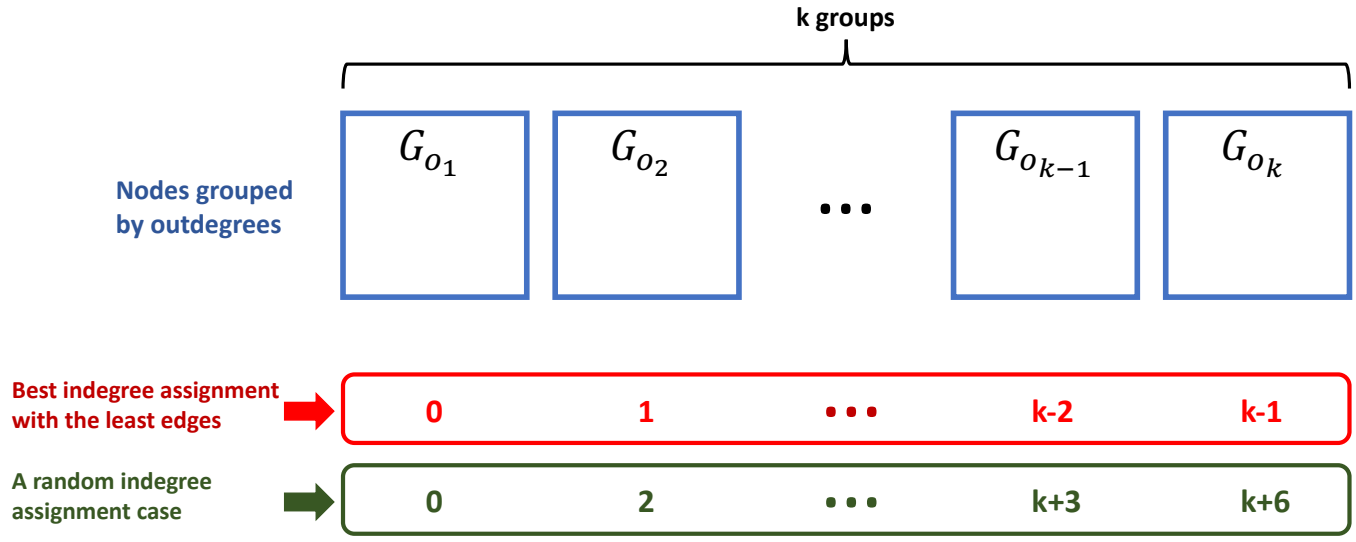


Figure 4. Indegree assignment to each node group with the same indegree.

Therefore, we can find the lower bound of the number of edges E, which is

$$E \geq \sum_{i=1}^{i=k} a \times (i-1)$$

(E- 4)

So far, we have found the equations for both N **(E- 3)** and E **(E- 4)**. Thus, we can further do the change of variables and find their relationship.

From **(E- 3)**, we can get

$$k = \frac{N}{a}$$

(E- 5)

And from**(E- 4)** and **(E- 5)**, we can get

$$E \geq \frac{N^2}{2a} - \frac{N}{2}$$

(E- 6)

The last step is to bring **(E- 6)**, the lower bound of E, back to the exponential algorithm equation, 2^{N+E}. We can get:

$$2^{N+E} \geq 2^{\frac{N^2}{2a}+\frac{N}{2}}$$

(E- 7)

And the factorial algorithm equation from **(E- 2)** can be re-expressed as:

$$\frac{N!}{\prod_{i=1}^{i=k} |G_{O_i}|!} = \frac{N!}{a!^k} = \frac{N!}{a!^{\frac{N}{a}}}$$

(E- 8)

Now, we can compare the time complexity of these two algorithms by visualizing them on the online Desmos calculator. In **Figure 5**, the red function represents the exponential algorithm, and the blue function represents the factorial algorithm. By trying different constant variable a from 1 to 1000, the

blue function is consistently underneath the red function, which shows that the factorial algorithm is faster than the exponential algorithm.

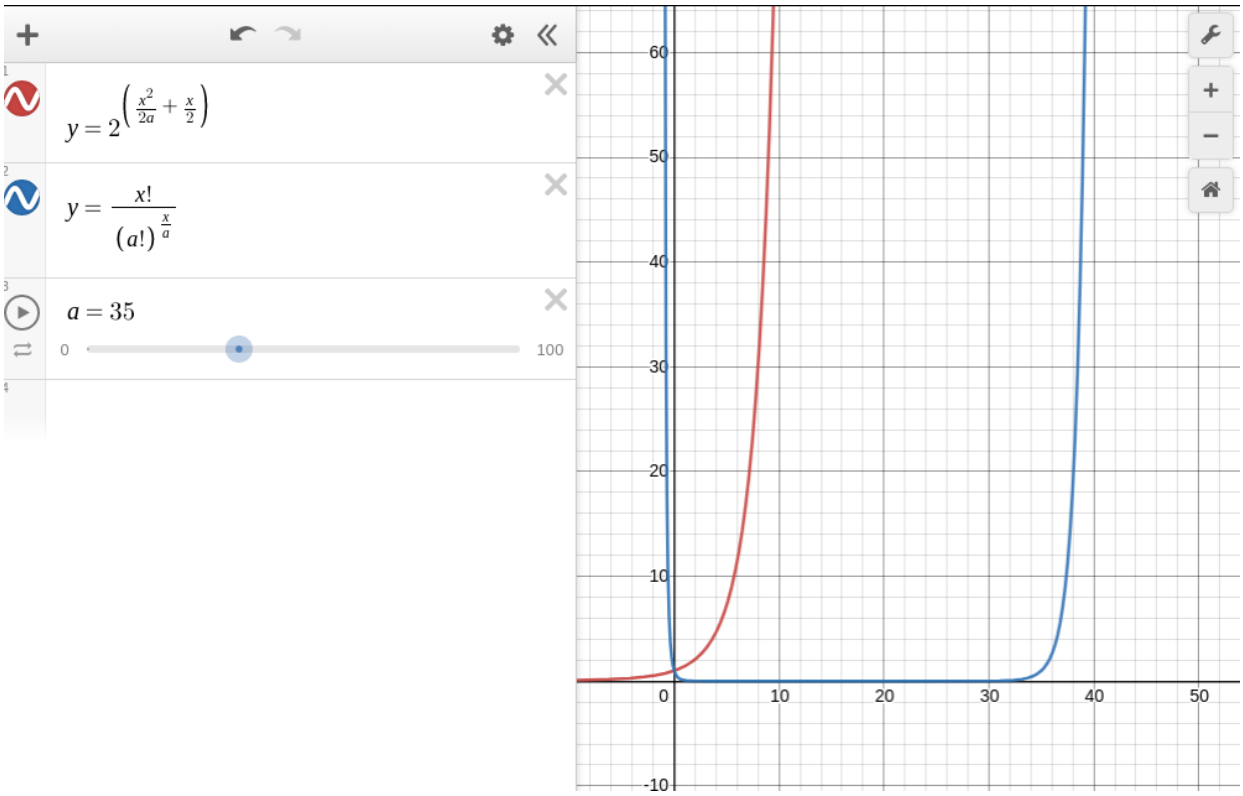


Figure 5 Visualizing the time complexity equations of the exponential and factorial algorithms on the Desmos calculator.

In conclusion, after taking the relationship between N and E into consideration, we have shown that Gibney and Thankachan's exponential algorithm on N and E is actually exponential on N^2, which is much slower than the factorial algorithm. Moreover, this result matches our intuitive observation that factorial algorithm is faster because the graphs that it checks are the subset of the graphs checked by the exponential algorithm.

One thing to emphasize here is that the result shown above is on one bitarray O only. However, we can extend this to three bitarrays and get the same conclusion.

IV-II. Recognizer Overview

The recognizer is implemented in C++ as a command-line tool. There are three main reasons why C++ is favored over other high-level languages: it is faster, it is more memory efficient, and most important of all, it has the pointer feature. The pointer feature allows us to easily permutate different new node labels. When referring to the same node, we can define different pointers in the edge and the digraph instances pointing to the same address. Thus, all we need to do is to make sure that pointers are pointing to the correct node and do not need to update each instance individually.

In terms of input and outputs, the recognizer takes a DOT file as an input and checks whether the given graph without node labels or randomly assigned node labels is a wheeler graph. If it is a wheeler graph, users can choose whether to output all possible sets of node labels or just the first set. Once a correct set of node labels are found, the recognizer outputs 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 bitarrays WG data structure proposed by Gagie (Gagie, et al.

2017); they are the inputs of the pattern matcher in the next session. Following are descriptions of five output files.

1. *I.txt*: it stores the I bit (indegree) array in the new node label order.
2. *O.txt*: it stores the O bit (outdegree) array in the new node label order.
3. *L.txt*: it stores the L array. It's the concatenation of the corresponding edges labels of nodes in O bit array.
4. *node.dot*: it stores the mapping from old node labels to new node labels.
5. *graph.dot*: it is the new, correct, and sorted dot file with new node labels.

In the recognizer, two classes, `edge` and `digraph`, are implemented. The edge class stores an edge label, original node labels, and pointers to the head and tail. As for the digraph class, it defines three unordered_maps (`_node_2_ptr_address, _node_2_innodes, _node_2_edgelabel_2_outnodes`), two maps (`_edge_label_2_edge, _edge_label_2_next_edge_label`), and member functions related to the core recognizing algorithm.

IV-III. Design Concepts & Lemma Proofs

To find all sets of correct wheeler graph node labels, the most intuitive way is to try all the permutations and check each of them whether the three WG properties hold. For example, if there are n nodes, the number of times to try is $n!$. However, it is not an efficient way to go with because the larger permutation base number makes the program slow.

Instead of permutating first and then check three WG properties, a better approach is to apply the WG property restrictions and split the base number first before doing the node label permutation. For example, if we know a certain set of nodes that are smaller than the remaining nodes, we can split n nodes into a and b two sub-groups, and the number of permutations is reduced from $n!$ to $a! * b!$. By doing so, some invalid WG labels can be skipped, and this is the main concept of how we faster the factorial algorithm. Back to the recognizer, our main algorithm applies two main approaches to split nodes into smaller groups and break down the large permutation base number. The two approaches are defined as ***Lemma 1*** and ***Lemma 2*** below.

Before starting on explaining ***Lemma 1***, we first 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. Moreover, we further define "**nodes in *A* edge group**" as taking the non-repeat set of all the tails of edges in A edge group.

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.

Proof:
Suppose G is a valid wheeler graph with edges labeled as A and B. Let $e_{A_i} = (n_x^A,\ n_y^A)$ be any edge in edge group A, and $e_{B_j} = (n_w^B,\ n_z^B)$ be any edge in edge group B. Since edge label $A \prec B$, by the second WG property, $n_y^A < n_z^B$. Thus, all the nodes in the edge group A are smaller than all the nodes in the edge group B.

Therefore, by applying ***Lemma 1***, if a valid WG G has n edge groups, we can divide nodes into n sub-groups and do the node label permutation within each edge group. But it is not good enough. To make the program even faster, we further apply ***Lemma 2*** to split each n sub-group into smaller sub-sub-group. Before starting on ***Lemma 2***, we first define "**in-node list**" 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 2-1:
For any two different nodes, n_1 and n_2, in the same edge group in a valid WG. If the label of n_1 is larger than the label of n_2, then the in-node list of $n_1$is lexicographical 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.

Proof:
Suppose G is a valid wheeler graph with nodes n_i and n_j. Given that they are in the same edge group, and the label of n_1 is larger than the label of n_2.
To meet the WG third property, each in-node of n_1 must be larger than each in-node of n_2.
Thus, "the smallest in-node of n_1 is larger than the largest in-node of n_2" holds.
Moreover, by concatenating all in-nodes of n_1 and n_2 into two in-node lists, the in-node list of n_1 is lexicographical larger than the in-node list of n_2.
Thus, ***Lemma 2-1*** holds.

Lemma 2-1 is straight forward. It simply applies WG third property and our "**in-node list**" definition; however, it cannot be directly used in our recognizer algorithm because we only know which edge groups they are in, and thus we only know their node label ranges. We do not have their precise node labels. To make it useful, we can slightly modify ***Lemma 2-1*** to ***Lemma 2-2*** by swapping its premises and arguments.

Lemma 2-2:
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.

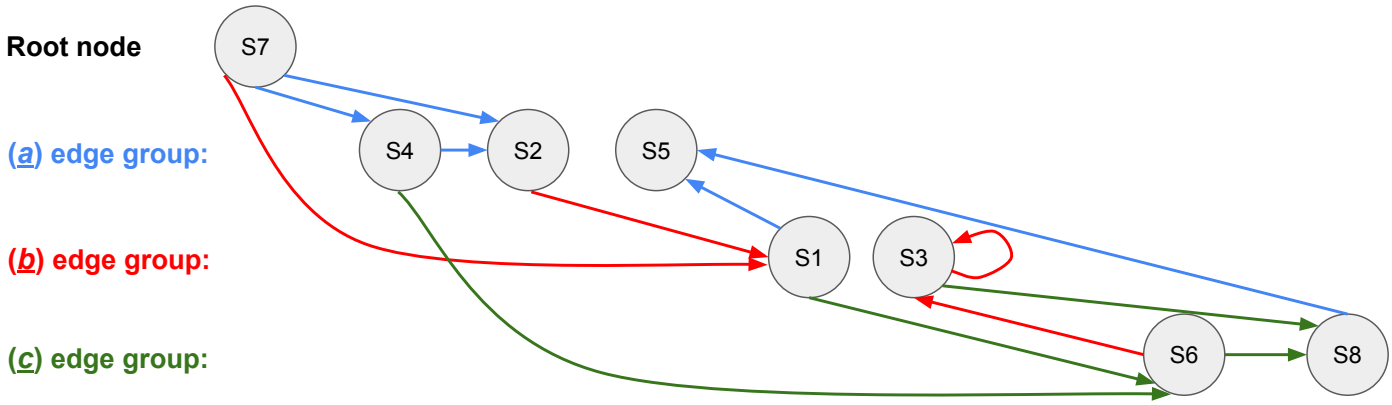
Proof:
For this proof, we are going to use the Proof by Contradiction technique.

Assume the label of $n_1$is smaller than the label of n_2 (they cannot be equal since they are different nodes). Suppose G is a valid wheeler graph with nodes n_i and n_j. Let the in-node list of n_i be $n_1^i n_2^i \dots n_x^i$, which implies that G has edges ($n_1^i \rightarrow n_i$), ($n_2^i \rightarrow n_i$), …, and ($n_x^i \rightarrow n_i$). Similarly, let the in-node list of n_j be $n_1^j n_2^j \dots n_y^j$, and it implies that G has edges ($n_1^j \rightarrow n_j$), ($n_2^j \rightarrow n_j$), …, and ($n_y^j \rightarrow n_j$). Given the known condition that n_1 and n_2 are in the same edge group; thus, all the edges above have the same edge label. And we also know that the in-node list of n_1, $n_1^i n_2^i \dots n_x^i$, is lexicographically larger than the in-node list of n_2, $n_1^j n_2^j \dots n_y^j$. Moreover, the smallest in-node of n_1, n_1^i, is larger than the largest in-node of n_2, n_y^j; thus, we can conclude that each and every in-nodes of n_1 (n_1^i, n_2^i, ..., and n_x^i) is larger than each and every in-nodes of n_2 (n_1^j, n_2^j, ..., and n_y^j).

By the third WG property, when edge labels are same and the heads are larger, the tails have to be larger or be the same. Thus, picking one edge from n_1 and one edge from n_2, each and every pair of edges leads to the conclusion that the label of n_1 is larger or equal to the label of n_2.
Thus, by the Proof by Contradiction, ***Lemma 2-2*** holds.

You might be confused why ***Lemma 2-2*** is useful because we still do not have precise node labels. How are we going to sort nodes by their in-node list? The technique here is that we do not need precise node labels. Having the range of node labels is enough for us to sort nodes and get their orders. **Figure 6, Figure 7** below is an example showing how ***Lemma 2-2*** works.

Figure 6 visualizes the example graph in Gagie's paper in 2017. It has eight nodes and three edge labels, a, b, and c. The node labels of this graph are shuffled. For each node in the graph, its in-node list is written out in the bottom of the image.

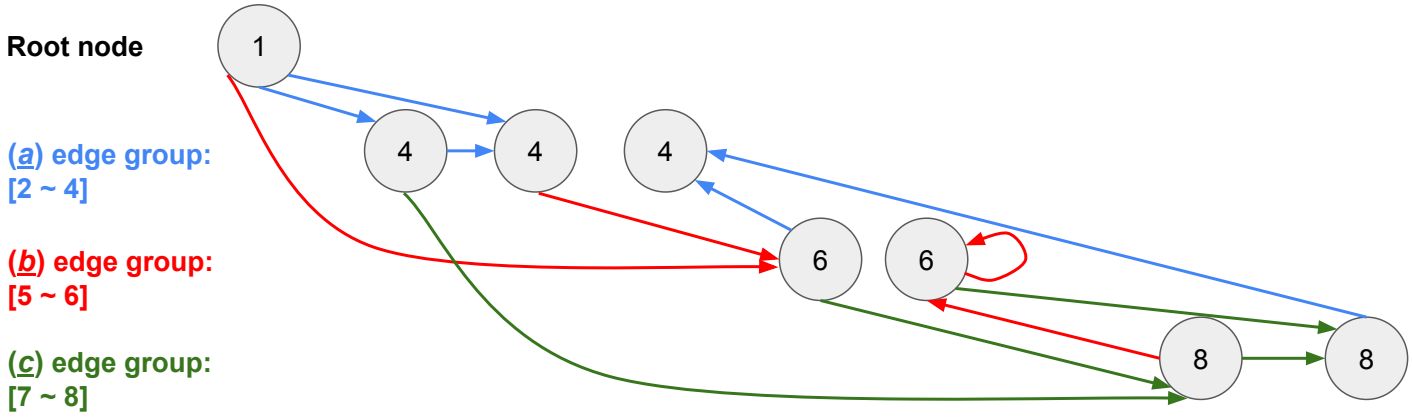


In-node list:

Root node	(a) edge level	(b) edge level	(c) edge level
S7: ⌀	S4: S7	S1: S7, S2	S6: S4, S1
	S2: S7, S4	S3: S3, S6	S8: S3, S6
	S5: S1, S8		

Figure 6 A graph with edge labels and random node labels. The in-node list of each node is written out in the bottom of the image.

Since all edge labels are given, we know the edge group where each node belongs. Our recognizer first relabels nodes with the **largest** possible value in each of their edge group. **Figure 7** shows the result of the graph after relabeling. For each node, its in-node list is also updated and written out in the bottom of the image.



In-node list:

Root node	(a) edge level	(b) edge level	(c) edge level
1: ⌀	4: 1	6: 1, 4	8: 4, 6
	4: 1, 4	6: 6, 8	8: 6, 8
	4: 6, 8		

Figure 7 The result of relabelling nodes with the largest possible value in each of their edge group. The in-node list of each node is written out in the bottom of the image.

Let us take a deeper look at the updated in-node lists. For *a* edge group, the three in-node lists of three different nodes are "1", "14", and "68" respectively. By applying ***Lemma 2-2***, we can sort three nodes and give them new labels which are "2", "3", and "4" according to the sorting order. Repeating the same technique in *b* edge group and *c* edge group, we can get the new node labels for this valid WG.

In sum, this example shows that ***Lemma 2-2*** does not require precise node labels and can further split nodes in the edge group into smaller subgroups. What we have achieved in this example is that we even skip the permutation step and get the new and correct set of node labels simply by sorting. Although it is not always the case, and we probably will encounter ties while sorting in-node lists; however, by applying ***Lemma 1 and Lemma 2-2***, our permutation-based recognizer algorithm can split nodes into smaller subsets and only do the permutation within each subset. It largely improves the speed of the factorial algorithm. And this is the core concept of our WG recognizer.

IV-IV. Core Recognizer Algorithm in Pseudocode

There are four main steps for the recognizer algorithm which are 'graph initialization', 'node labels initialization', 'in-node list sorting & relabeling', and 'repeated node labels permutation'.

Step 1: Graph Initialization

```
// digraph constructor
digraph g = digraph()
g.add_edges();

def digraph:
    locate a chunck of continue memory for all nodes.
    for node in nodes
        create the hash table from node name to address (_node_2_ptr_address)
        initialize node value
```

```
    end for


def add_edges:
    for each edge:
        find the out-node list for each head (_node_2_edgelabel_2_outnodes)
        find the in-node list for each tail (_node_2_innodes)
        // edge constructor
        edge e = edge()
        create the hash table from edge label to edges (_edge_label_2_edge)
    end for
    find root nodes (nodes with indegree 0)
    create the sorted edge label hash table (_edge_label_2_next_edge_label)
```

Step 2: Node Labels Initialization

```
def relabel_initialization:
    // Relabel root nodes with the **larget** possible value.
    for each root in roots:
        relabel_by_node_name(root_node, roots.size())
    end for

    // Relabel all edge group nodes with the **largest** possible values.
    accum_label= roots.size();
    for (edge_label, edges) in _edge_label_2_edge:
        // create a set to remove repeat nodes
        for edge in edges:
            edgenodes_set.insert(edge.get_tail());
        end for
        accum_label += edgenodes_set.size();
        for edge in edges:
            relabel_by_node_name(edge.get_tail(), accum_label)
        end for
    end for

    // After initialization, check whether it is a WG.
    WG_valid = WG_checker()
    if (not WG_valid):
        Terminate the program
    end if
```

Step 3: In-node List Sorting & Relabeling

```
def innodelist_sort_relabel:
    for (edge_label, edges) in _edge_label_2_edge:
        //sort nodes by in-node list
        in_edge_group_sort()

        // get the new pre-label list.
        // If there is a tie, relabel nodes with the **smallest** possible value.
        in_edge_group_pre_label()
    end for

    // After in-node list sorting & relabeling, check whether it is a WG.
    WG_valid = WG_checker()
    if (not WG_valid):
        Terminate the program.
    end if
```

Step 4-1: Repeated Node Labels Permutation (permutation_start)

```
def permutation_start:
    // Permutate the root labels.
    while (there are more root node label permutations):
        // Try the permutation
        relabel_forward_root();
        // Start the first edge group node permutation.
        permutation_4_edge_group(get_first_edge_label());
        // Relabel the node labels back to the original labels.
        relabel_reverse_root();
    end while
```

Step 4-2: Repeated Node Labels Permutation (permutation_4_edge_group)

```
def permutation_4_edge_group(edge_label):
    for edge in _edge_label_2_edge[edge_label]:
        all_sets = find all sets of different nodes with same node label.
        // Start the permutation of the first set of nodes with the same label in the same edge group.
        permutation_4_sub_edge_group(edge_label, all_sets)
    end for
```

Step 4-3: Repeated Node Labels Permutation (permutation_4_sub_edge_group)

```
def permutation_4_sub_edge_group(edge_label, all_sets):
    if (all_sets is not empty):
        while (there are more sub-edge group node label permutations):
            // Try the permutation
            relabel_forward();
            WG_valid = WG_checker_in_edge_group();
            if (WG_valid):
                // Start the permutation of the next set of nodes with the same label in the same edge group.
                permutation_4_sub_edge_group(edge_label, all_sets.next());
            end if
            // Relabel the node labels back to the original label
            relabel_reverse();
        end while
    else
        if (edge_label is not the last edge_label):
            // Start the permutation of the next edge group.
            permutation_4_edge_group(get_next_edge_label(edge_label))
        else
            WG_valid = WG_checker()
            if (WG_valid):
                valid_wheeler_graph()
            else
                invalid_wheeler_graph()
            end if
        end if
    end if
```

IV-V. Recognizer Usage & One Example:

Following is the usage of the recognizer. For the current version, users need to input a DOT file and have three options:

```
Usage:

   recognizer  <in.dot>  [wg_recognizer_method]  [stop_condition]  [print_invalid]
```

One option to mention here is `wg_recognizer_method`. We implemented two algorithms, and users can select either `m1` or `m2`. `m1` is the permutation algorithm described in the previous section; as for `m2`, it is a second algorithm that is still under development and is our future goal. The main idea of `m2` is to further sort the nodes by their "**out-node lists**" before permutating node labels, which can further improve the speed.

Figure 8 below shows the results of a simple example. Given a seven-node graph with random node labels, our recognizer successfully found a new set of node labels fulfilling three WG properties and wrote out five output files.

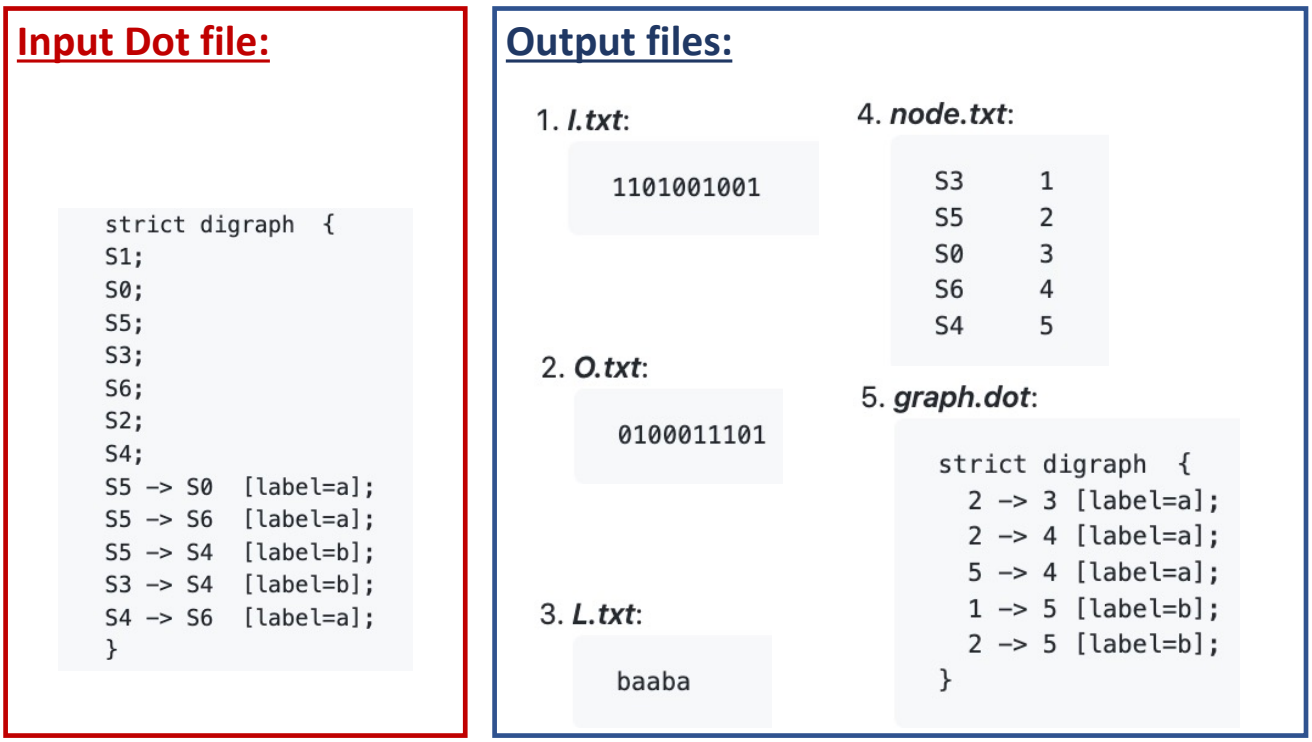


Figure 8 The input and outputs of an example graph with random node labels.

IV-VI. Recognizer Time Complexity

Although we split N nodes by ***Lemma 1*** and ***Lemma 2*** before permutation, our recognizer is still a factorial algorithm. In **Figure 9** below, we demonstrate a worst-case example of our recognizer. Given a graph G with two edge labels, a and b, we can group edges into two edge groups. Assume there are X nodes in the a edge group and for each node in the a edge group, it connects to Y nodes in the b edge group. In total, there are $X \cdot Y + 1$ nodes in G. In this case, the time complexity of our recognizer is $O(X! \cdot (Y!)^X)$. Although our algorithm is faster than permutating without WG properties restriction, which is $O\big((X \cdot Y + 1)!\big)$, it is not scalable.

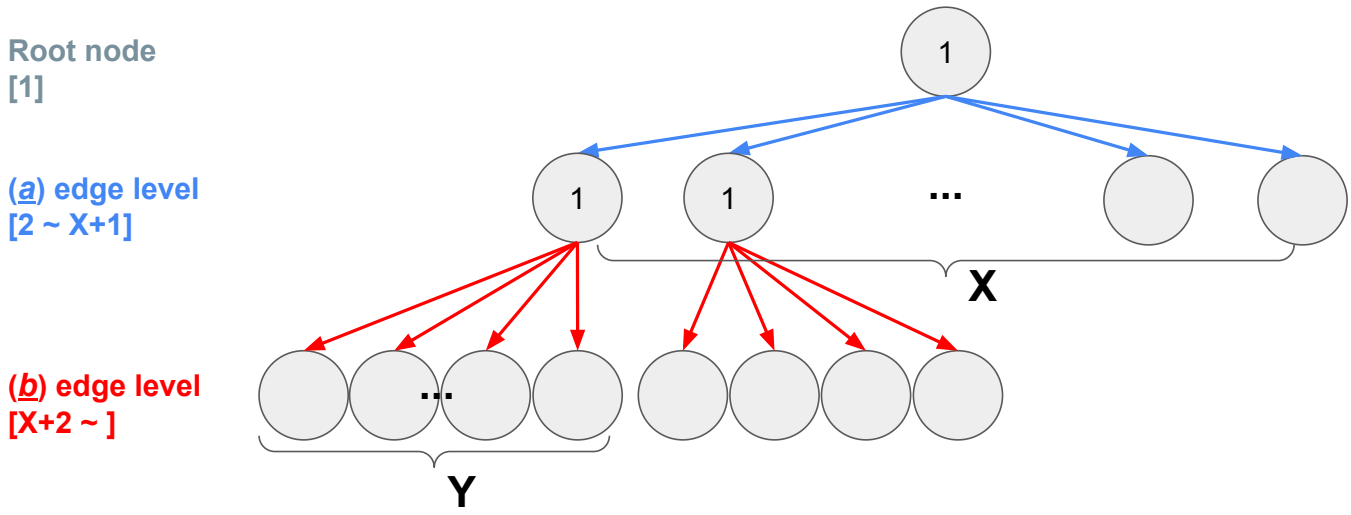


Figure 9 The worst-case analysis of our recognizer

Luckily, in real world WG applications, it is uncommon to have large number of nodes in a tie that need to be permutated. Therefore, in our opinions, our algorithm can solve the WG recognizing problem in a reason amount of time on real world cases. Take a step further, upon our observation, swapping node labels of some nodes do not violate three WG properties, and we call those nodes "interchangeable". Checking again the worst-case example above, we realized that the main reason why it requires lots of checking times is that all nodes in the a edge group are interchangeable. Therefore, we are asking the following question:

"Is there a way to apply three WG properties and leave those interchangeable nodes in the same label without doing permutation?"

If that is the case, then we can solve the WG recognizing problem without any permutations.

IV-VII. Space Complexity

For the edge class, it stores an edge label (`string _label`), two pointers (`int* _head, _tail`), and two original node labels (`int _head_name, _tail_name`). Therefore, for each edge, it requires 16 bytes plus the size of one edge label string. Notice that although the type of original node labels is integer, it can take any strings. The recognizer converts the string into ASCII codes and concatenates them into integers. The reason why integer type is preferred over string is because we want to avoid the Small String Optimization (SSO) overhead.

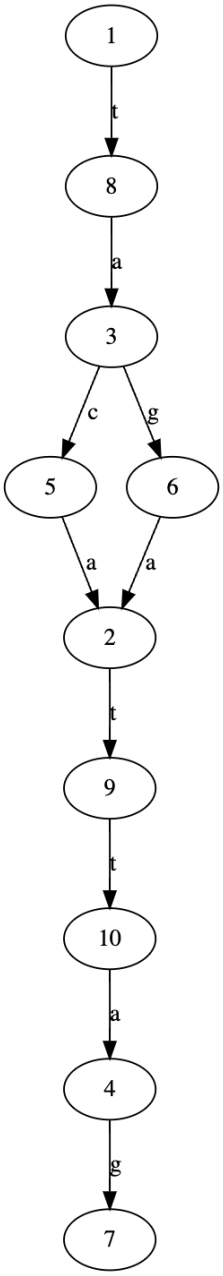
For the digraph class, there are three unordered_maps (`_node_2_ptr_address, _node_2_innodes, _node_2_edgelabel_2_outnodes`), and two maps (`_edge_label_2_edge, _edge_label_2_next_edge_label`). Say that a given graph G has N nodes, E edges, and q different edge labels, the space complexity of `_node_2_ptr_address` is $O(N)$. The total number of in-nodes of every node is exactly the number of edges since each edge contribute to one in-node to the corresponding tail. Therefore, `_node_2_innodes` is $O(N+E)$. Similarly, the total number of out-nodes of every node also equals to the number of edges, and `_node_2_edgelabel_2_outnodes` is $O(N+E)$ as well. As for `_edge_label_2_edge`, it stores edges corresponding to their edge labels and is $O(E)$. Finally, `_edge_label_2_next_edge_label` is bounded by the number of edge label alphabets, and the space complexity is $O(q)$. In sum, since q is relatively small compare N and E, the total space complexity of the digraph class is $O(N+E)$.

V. Methods and software (4): Pattern Matcher

V-I. Pattern Matcher Overview

Upon receiving a directed edge labeled graph that can be a proper Wheeler Graph, the Wheeler Graph recognizer described above outputs updated *O, I* and *L* vectors with appropriate node labels. Given the Wheeler Graph representation of a genome or any other sequence, one of the most important tasks we would like to accomplish is to be able to efficiently match a pattern to it. The main challenge of the Wheeler Graph Pattern Matcher is to find a way of "translating" and utilizing the three bit vectors *O, I,* and *L* for querying the graph to find occurrences of a given pattern *p*.

In order to legally navigate between the *O, I,* and *L* vectors and map the path taken onto the actual graphical representation of the Wheeler Graph, we must first revisit the definitions of these bit vectors. Let us consider the following example demonstrated in the lecture series by Prof. Langmead:



O = 01010010101011010101

I = 10010101010101010101

L = $t_1t_2c_1g_1g_2a_1a_2a_3t_3a_4$

Recall that the outdegree bit vector *O* represents nodes and the corresponding edges that leave those nodes in order, and that order is preserved in all these vectors. Exploiting this parallelism between the vectors we can easily map the edges represented in the bit vector *O* with their labels in the vector *L*. Below is a table that maps the 0 and 1 bits in outdegree bit vector *O* to the edges (E) and nodes (N) of the graphical representation on the left:

O	0	1	0	1	0	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1
E	t_1		t_2		c_1	g_1		g_2		a_1		a_2			a_3		t_3		a_4	
N		1		2			3		4		5		6	7		8		9		10

Similarly, we can translate the information in the indegree *I* bit vector as such:

I	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
E		a_1	a_2		a_3		a_4		c_1		g_1		g_2		t_1		t_2		t_3	
N	1			2		3		4		5		6		7		8		9		10

V-II. Implementing Helper Functions and Data Structures

The very first step of the pattern matching problem is to find a node and/or edge of interest in the parallel information sequences visualized above. Let us say that we are looking for **Node 3**. To access the offset in the *O* and *I* vectors we need to implement helper functions that can find the corresponding '1' bits in these vectors. The two functions that we implement for this task are **rank_1** and **select_1:**

rank_1(V, i):

Given a bit vector V, returns the number of 1s up to but not including offset i:

$$\text{rank\_1}(V, i) = \sum_{j=0}^{i-1} V[j = 1]$$

select_1(V, i):

Given a bit vector V, returns the offset of the i^{th} 1-bit:

$$\text{select\_1}(V, i) = \max \{ j \mid \text{rank}_{1(V,j)} = i \}$$

Then to find Node 3 in the *O* bit vector, we would call **select_1(O, 2)** (because the nodes are 1-indexed Node 3 is of rank 2).

Correspondingly, we need to implement the rank_0 and select_0 helper functions to access the edges of interest in the *O* and *I* vectors:

rank_0(V, i):

Given a bit vector V, returns the number of 0s up to but not including offset i:

$$\text{rank\_0}(V, i) = \sum_{j=0}^{i-1} V[j = 0]$$

select_0(V, i):

Given a bit vector V, returns the offset of the i^{th} 0-bit:

$$\text{select\_0}(V, i) = \max \{ j \mid \text{rank}_{0(V,j)} = i \}$$

To find Edge 2 in bit vector *I* for instance, we would call select_0(I, 2).

Now that we can access Edge 2 in either of the bit vectors, we need to determine which character that edge corresponds to. In order to answer this question, we implement rank_c which follows the same logic as the rank functions above for any string character 'c':

rank_c(V, i):

Given a bit vector V, returns the number of occurrences of character 'c' up to but not including offset i:

$$\text{rank}_c(V, i) = \sum_{j=0}^{i-1} V[j = V[i]]$$

Finally, to rank all the characters in our *L* vector we constructed a "skip-dictionary" which follows a very similar logic to the "skip-array" of FM-index. This dictionary-type data structure *C* stores the start and end positions of each character in the sorted *L* vector.

Note that these helper functions are implemented in their naïve versions where each time they are called the input vector *V* is scanned linearly resulting in $O(|V|)$. For future improvements we would like to implement Jacobson's rank and Clark's select for higher efficiency.

V-III. Pattern-Matching Logic and Pseudocode

With the help of the functions and data structures we defined above, we can in fact complete a full iteration of the pattern matching algorithm. Let us use the same example graph above and try to find $p = aga$:

The pattern matcher will look for any given *p* starting from right to left, and so the first character we are interested in locating in the graph is character 'a' of rank 1:

p = aga (= a_0, g_0, a_1)

1. First, we will define the search range in terms of the sorted *L* vector (which would correspond to the *F* column of a *BWT* matrix if *L* were its last column):

   F = aaaacggttt

   Now when we search for character a in our skip-dictionary *C* we get the range [0, 3]:

```
C [a] = [0, 3]
search_range_start = 0, search_range_end = 3
```

2. The first search in the bit vectors will be to find the edges that correspond to character 'a' in the *I* bit vector. Recall that the *I* bit vector represents the nodes and the edges that are incoming to those nodes in the same order as *F*. And so each of the 0 bits will correspond to a character in *F* in the offset of its rank_0. The corresponding edges in *I* to the search range [0,3] will be:

   F = aaaacggttt
   I = 1001010101010101010101

   We find this by simply calling:

```
incoming_start_edge_in_I = select_0(search_range_start, I)
incoming_end_edge_in_I = select_0(search_range_end, I)
```

Notice that in each of these steps (as well as in the following steps), we only call these functions only twice to find the start and end points of our current search rather than calling it on each of the characters separately.

3. Now that we have found the edges that correspond to the character that we were searching for, we will want to "follow" the edges to the nodes that they are leading to.
   In other words we need to find the rank of the next 1 bit in the *I* bit vector (because that node *receives* that *incoming edge* by definition of the *I* bit vector:

```
receiving_strt_node_I = rank_1(incoming_strt_edge_I, I)
receiving_end_node_in_I = rank_1(incoming_end_edge_in_I, I)
```

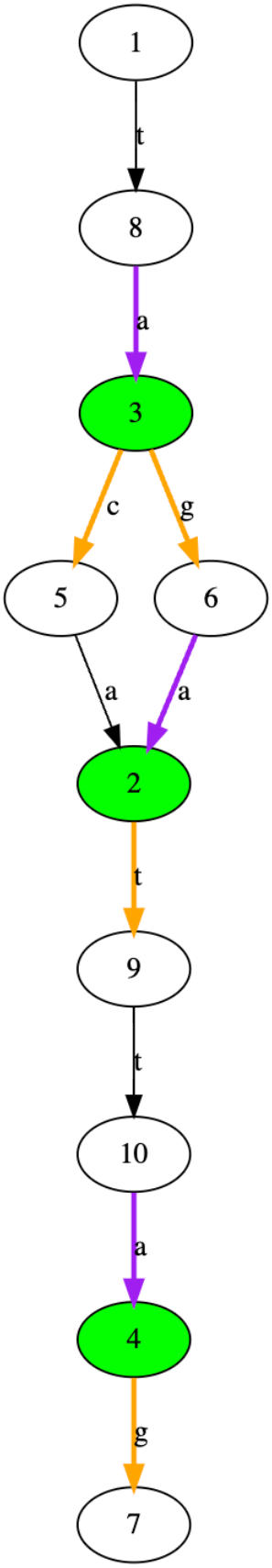
4. The next step is to find the edges that leave these nodes in the bit vector *O*. Using the fact that the node order is preserved in both bit vectors, we simply call:

```
corresponding_start_node_in_O =
     select_1(receiving_start_node_in_I, O)
corresponding_end_node_in_O =
     select_1(receiving_end_node_in_I, O)
```

5. Having found the new range of edges that we will potentially follow, we find the offset for their character labels in *L* by:

```
new_edge_range =
[rank_0(select_1(rank_1(corresponding_start_node_in_O, O)-1, O),
O), rank_0(corresponding_end_node_in_O, O)-1]
```

Let us take a moment to visualize where we are in the Wheeler Graph:



p = aga

Steps 1 through 3:

Starting with the edges labeled with character "a" (highlighted here in purple), we have followed them to the nodes that they are leading into in the Wheeler graph which are filled in green.

Steps 4 & 5:

We have identified the edges that leave those nodes which are highlighted here in orange. Notice that not all of these edges match to the next character in our pattern *p*, and we will certainly not want to follow the ones that do not.

6. To check for this criterion, we will loop through the new_edge_range that we defined in step 5 and check the corresponding character labels in vector *L* as such:

```
for n in new_edge_range[0]:
      if(L[n] == next_c_in_p):
          matching_chars.append(n)
```

7. As the final step of our iteration, we check if there are any matching characters in the leaving edges that we identified to the next character in *p*, and set the new search range to the offset of those characters in *F*. If not, we "fall off" and conclude that there are no occurrences of *p* in the inputted Wheeler Graph.

```
if(len(matching_chars) > 0):

     search_range_start =
     C[L[min(matching_chars)]][0] +
     rank_c(min(matching_chars), L)

     search_range_end =
     C[L[max(matching_chars)]][0] +
     rank_c(max(matching_chars), L)

else:
     return -1
```

V-IV. Runtime and Space

The logic outlined above to match one character of the pattern *p* to an input Wheeler Graph is repeated at worst |*p*| times, in which case *p* does occur in the Wheeler Graph. One advantage of this iterative process is that there is no extra cost even if *p* occurs multiple times in the Wheeler Graph.

For each iteration (character match) of the above algorithm, we have the following operations:

1) Look up in skip-dictionary *C*
2) rank_0 in bit vectors *I* and *O*
3) rank_1 in bit vectors *I* and *O*
4) select_0 in bit vectors *I* and *O*
5) select_1 in bit vectors *I* and *O*
6) rank_c in vector *L*
7) character comparisons of the new edge labels to the next character in *p*

Any look up in the skip dictionary *C* will take constant time. However, since we have implemented the select and rank queries in their naïve forms instead of using wavelet trees or more efficient approaches like Jacobson's rank and Clark's select, items through 2 to 6 are all linearly dependent on the length of the input vectors.
Both the *O* bit vector and the *I* bit vector will have one 0 bit per edge and one 1 bit per node, making their sizes $|E| + |N|$, E being the edges in the Wheeler Graph and N being the nodes. The *L* vector, however, has the same number of characters as the number of edges in the graph, making its size $|E|$. Therefore, linear iterations for these look-ups become:

2) rank_0 in bit vectors *I* and *O* $= O(|E| + |N|)$
3) rank_1 in bit vectors *I* and *O* $= O(|E| + |N|)$
4) select_0 in bit vectors *I* and *O* $= O(|E| + |N|)$
5) select_1 in bit vectors *I* and *O* $= O(|E| + |N|)$
6) rank_c in vector *L* $= O(|E|)$

In the worst-case scenario (e.g. *L = {aaaa}, p = {aa}*) character comparisons of the new edge labels to the next character in pattern *p* will take $|E|$ iterations, making item 7) = $O(|E|)$.

In the inner loop, all of these operations are called a constant number of times, making the worst case $big - O$ runtime $O(|E| + |N|)$. Since this process is repeated at most |*p*| times in the outer loop (in which case *p* is found in the graph), the overall runtime becomes $O\big(|p| * (|E| + |N|)\big)$. As mentioned earlier, this runtime can be significantly improved by better representation and implementations of the select and rank queries. There is no significant space cost of the pattern matcher independently from the Wheeler Graph itself. The only new data structure is the skip-dictionary *C* which takes approximately $\sigma \log n$ bits (same as in BWT).

V-V. Visualizing the Matched Pattern

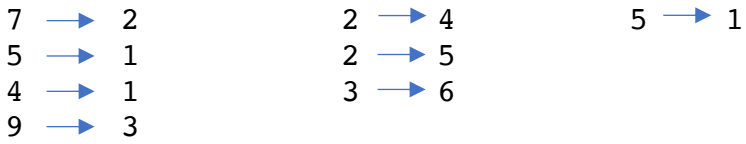
The Wheeler Graph Pattern Matcher we implemented takes as an optional argument the original graph structure -in .dot format- outputted by the Wheeler Graph Recognizer. If this option is taken and the inputted pattern *p* is found in the Wheeler graph, the pattern matcher will output the same Wheeler Graph with all the occurrences of *p* in the graph being highlighted.

1. To keep record of the path taken in the search, we populate an independent array visited_node_range at step 5 of the above searching algorithm by simply appending the range of all nodes we are visiting:

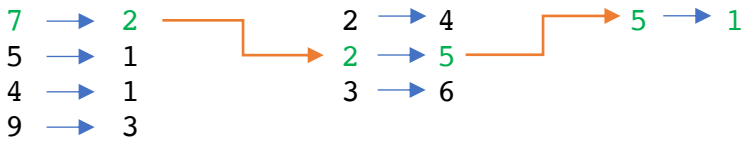
```
visited_node_range.append([rank_1(corresponding_start_node_in_O, O)+1,
rank_1(corresponding_end_node_in_O, O)+1])
```

Note that the indices are incremented by 1 because the Wheeler Graph recognizer outputs nodes starting with 1 (no nodes are labeled 0).

2. Once the pattern is found in the Wheeler graph, we first distribute the visited_node_range by adding all edges within the range to the path. We call this visited_nodes. Then, we construct the path by **only keeping the nodes that receive edges from the previous step and have outgoing edges to the next**. If we were to visualize this using the above example, we would initially have populated the following visited_nodes:

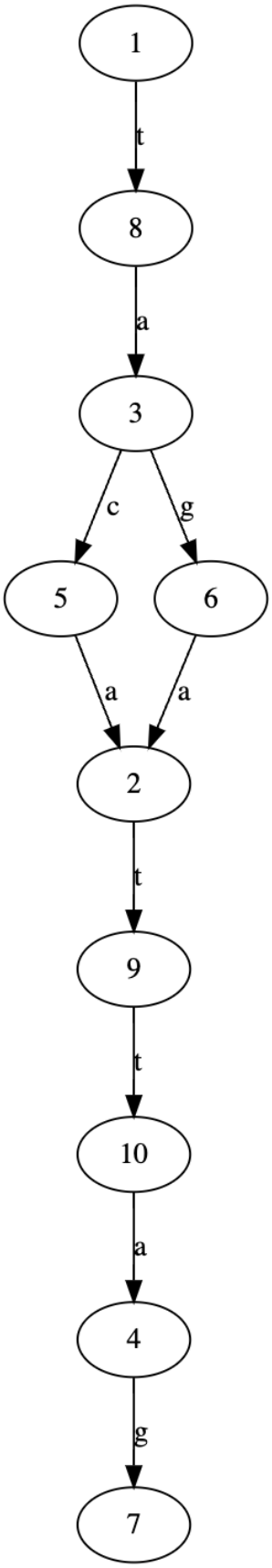


Then we would only keep the nodes that were the "receiving" nodes in the previous step **and** are the "sending" nodes in the next as below:

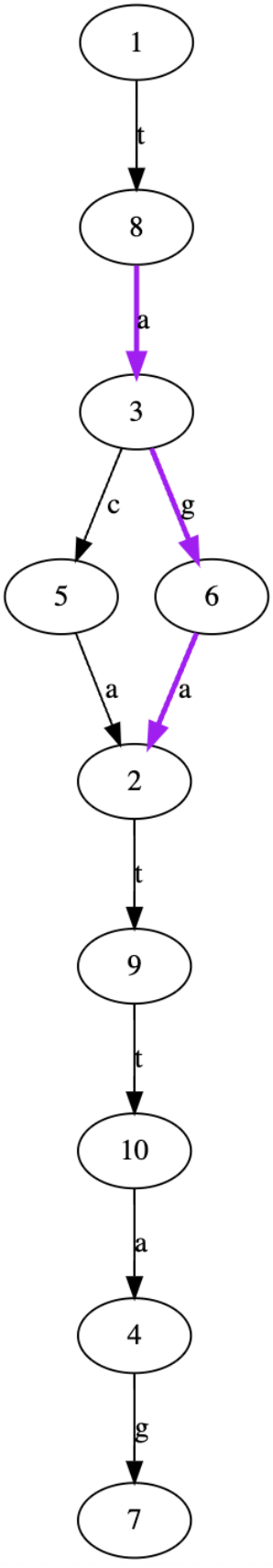


Please find the example graphical output of the pattern matcher for the example we have worked on throughout this section on the next page.

Input:



Output:



VI. Conclusion

We developed a suite of wheeler graph tools. Starting from visualizing graph (visualizer), validating whether a labeled graph is a WG (checker), and generating random valid WG (generator), we further worked on designing algorithm to rank nodes and generate a new node label mapping (recognizer), and matching patterns with Gagie's *O*, *I*, and *L* WG data structure.

This project is the first step. We have a few questions that are still solving and more ideas that would like to try. For example, we would like to further extend generator to control the shape of valid WG graphs, the depth of edges and nodes that violate WG properties, and the number of interchangeable nodes. For the recognizer, we are implementing the out-node list sorting and further working on the algorithm without premutation.

In sum, we hope our software and new perspective on wheeler graph recognizing problem can push the wheeler graph field forward.

VII. Reference

1. Gagie T, Manzini G, Sirén J 2017. Wheeler graphs: A framework for BWT-based data structures. Theoretical computer science 698: 67-78.
2. Gibney D editor. Proceedings of the Open Problem Session, International Workshop on Combinatorial Algorithms, Pisa, France. 2020.
3. Gibney D, Thankachan SV 2019. On the hardness and inapproximability of recognizing wheeler graphs. arXiv preprint arXiv:1902.01960.
4. Langmead B, Salzberg SL 2012. Fast gapped-read alignment with Bowtie 2. Nature methods 9: 357-359.
5. Li H 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv preprint arXiv:1303.3997.
6. The BZIP2 Homepage [Internet] http://www.bzip.org/. 1996.

VIII. Teamwork

Eduardo Aguila	**Shaunak Shah**	**Kuan-Hao Chao**	**Beril Erdogdu**
Abstract (P1), Introduction (P2), Checker (P3), Generator (P4~ P6), Proofreading		Abstract & Introduction (P1), Recognizer (P7~P18), Conclusion (P27), Finalizing report	Pattern-matcher (P19~P26), Proofreading