Technical University of Denmark

A Progrssive Alignment Algorithn for Rotation and Orientation of Circular Sequences

Author:
André R. Jørgensen

Supervisors:
Gabriel Renuard
Joshua D. Rubin

for obtaining the Master of Science

in

BIOINFORMATICS AND SYSTEMS BIOLOGY

26nd of August 2024

Acronyms

CSC Circular Sequence Comparison. M

hCED heuristic Cyclic Edit Distance. M, N

MAFFT Multiple Alignment using Fast Fourier Transform. 18, C, L, M

MARS Multiple circular sequence Alignment using Refined Sequences. 18, 22, M

MCSA Multiple Circular Sequence Alignment. iii, M

MSA Multiple Sequence Alignment. 2, 6, 7, 13, 18, 25, L–N

mtDNA mitochondrial DNA. 1, O

vg Variaton Graph toolkit. 10, 25, 26, C, O

Contents

Abstract

Current bioinformatics tools often struggle with circular genomes, primarily due to challenges in linearizing them for analysis. We introduce vgOrient, a a graph-based tool designed to effectively linearize circular genomes to facilitate subsequent analyses. vgOrient detects and corrects misoriented or rotated sequences, arranging them into a consistent linear orientation. In comparisons with leading tools across three distinct datasets, vgOrient showed as or better performance in two. This capability suggests vgOrient as a powerful solution for researchers grappling with circular genomic data.

1 Introduction

1.1 Background

Sequence alignment has been a cornerstone of biological research for decades, serving as a crucial method for comparing two sequences to minimize their differences. This technique is integral to several key areas:

- Understanding the functions of DNA, or the proteins they encode, a focus area in *functional comparative genomics*, which predicts gene function by comparing genetic material across different organisms.
- Exploring evolutionary relationships among organisms, central to the field of *phylogenomics*, where genetic data is utilized to reconstruct evolutionary histories.

Mitochondrial DNA (mtDNA) is particularly significant in phylogenomics due to its unique properties and role in evolutionary studies. [1] Unlike nuclear DNA, mtDNA is highly conserved across generations and does not undergo recombination, allowing it to serve as a clear indicator of phylogenetic evolution. Furthermore, its high mutation rate provides a powerful discriminative feature for studying evolutionary lineages, making it invaluable for tracing genetic history.

Despite its importance, the utility of mtDNA in phylogenetic studies is often compromised by the alignment challenges posed by its circular structure. To support genomic research, the bioinformatics community has developed extensive databases and tools for comparing DNA and other biological sequences. These tools, while invaluable, are primarily designed for linear DNA sequences and do not adequately address the challenges posed by the circular nature of mtDNA.

Typically, circular DNA sequences are artificially linearized by selecting an at times arbitrary start point, which can introduce significant alignment errors. For instance, the linearized versions of human (NC_001807) and chimpanzee (NC_001643) mtDNA sequences do not align at the same genomic location due to these arbitrary cut points. [2] Additionally, some sequences, like the sea cow (NC_040161.1) mtDNA, are stored in reverse complemented formats, further complicating their analysis. [3]

These issues underscore the need for specialized tools that can adeptly manage the unique challenges of circular DNA sequences, thereby enhancing the accuracy and reliability of comparative genomic studies. The next section will discuss how our project aims to develop such a tool, addressing these critical gaps and improving the current state of bioinformatics tools for mitochondrial genomes.

1.2 Project Motivation

We present "vgOrient," a tool designed to address prevalent challenges in handling mitochondrial genomes, particularly issues related to their orientation and rotation as stored in sequence databases. Existing tools often struggle to manage these problems effectively, leading to inaccuracies in downstream analyses.

vgOrient corrects the orientation and rotation of multiple mitochondrial genomes, standardizing their alignment preparation. The tool leverages advanced computational strategies to:

- Ensure uniform orientation across all processed genomes for consistent analysis.
- Rotate each genome to start in the same region, enhancing comparability.
- Output a linearized version of the rotated circular genomes to facilitate standard alignment processes.

With vgOrient, we seek to support more accurate and dependable linear alignment methods such as MSA, which are often undermined by the preprocessing variations of mitochondrial DNA. The application of MSA algorithms to build circular genomic graphs allows for a targeted approach to the specific challenges of mitochondrial genomes. Further discussion on these algorithms will provide insight into the foundational theories driving the development of our tool, with an aim to enhance the efficacy and precision of genomic analysis.

1.3 Theory

This section introduces a series of concepts and algorithms needed to understand and evaluate vgOrient. The first section will present a fundamental concept in evaluating algorithms. The sections following will present progressively more complex problems as well as the algorithms or tools for solving them.

1.3.1 Computational and Space Complexity

Understanding the computational and space complexities of algorithms is crucial in bioinformatics, particularly given the extensive data involved in genomic studies. These algorithms complexities dictate the practicality and performance across various computational scenarios.

Big O Notation Big O notation is used to express the upper limit of an algorithm's time or space complexity as it relates to the size of the input. This notation categorizes algorithms based on how their execution time or space requirements scale with increases in input size.
Figure 1 visually illustrates common complexities, highlighting the impact of different growth rates on computational demands. If multiple complexity terms are present, the term that grows fastest as the input size increases will determine the overall complexity.

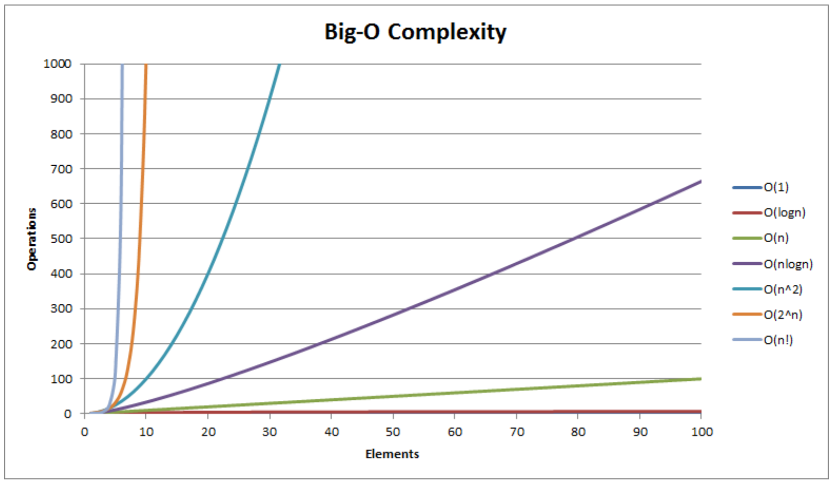


Figure 1: Examples of different computational complexities and how they grow with input size. Figure from [4]

.

Given the constraints imposed by the computational limits, it becomes necessary to consider the trade-offs between accuracy and efficiency, especially for complex problems.

Exact vs. Approximate Algorithms For simpler tasks, optimal solutions are often feasible within reasonable limits of time and space. However, as complexity increases, finding the exact solution can become computationally unmanageable. This necessitates the use of approximate algorithms, which trade a degree of accuracy for a reduction in computational demand. These algorithms often rely on heuristics to provide effective solutions within acceptable time constraints.

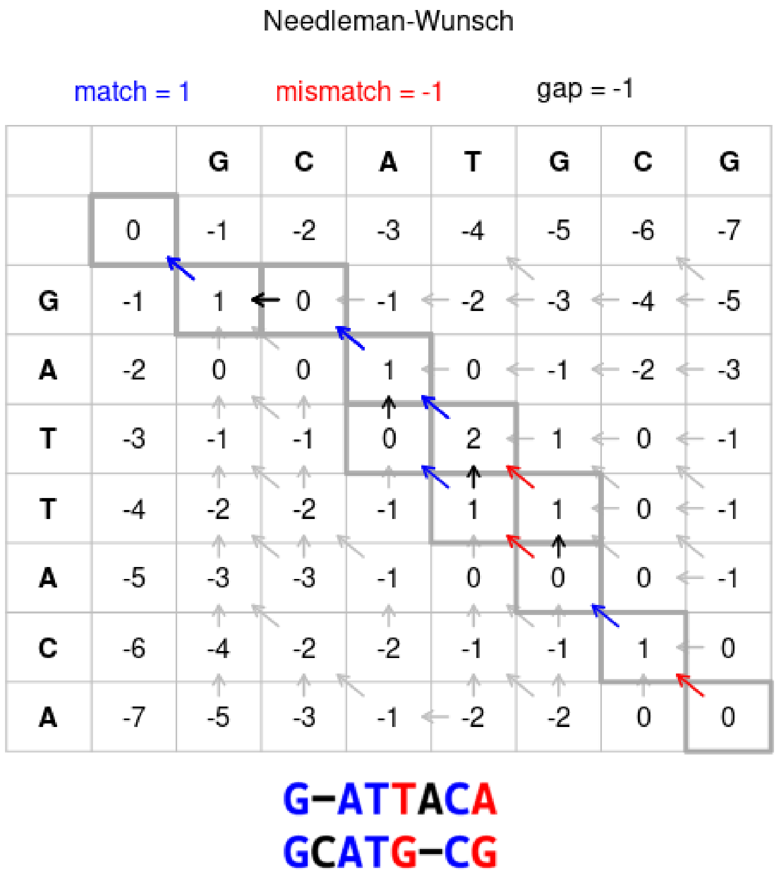
Theoretical Foundations in Bioinformatics Grasping the computational and space complexities, is vital for bioinformatics. This knowledge helps analyze algorithm performance, setting the stage for understanding the problems and algorithms in the coming sections.

1.3.2 Pairwise Alignment

As mentioned in Section 1.1, alignment plays a crucial role in bioinformatics research. There are multiple methods of alignment, each suitable for different tasks.
The simplest type of alignment is pairwise alignment. These algorithms align two linear sequences in a way that minimizes the differences between them. An example of such an algorithm is the Needleman-Wunsch algorithm. This method constructs an $n \times m$ matrix where n and m represent the lengths of the two input sequences. [5]
It employs a scoring system where matches (identical characters in both sequences), mismatches (different characters), and gaps (insertions or deletions) are scored differently to optimize the alignment score. The Needleman-Wunsch algorithm explores possible alignments by populating this matrix based on the scoring of matches, mismatches, and gaps:

- A diagonal move in the matrix corresponds to a match or mismatch, depending on whether the characters in both sequences are the same.
- A horizontal or vertical move corresponds to a gap in one of the sequences.

This process continues until the entire matrix is filled, ensuring all characters from both sequences are inserted, which characterizes this method as a global alignment algorithm. An example of a Needleman-Wunch alignment is presented in Figure 2.



		G	C	A	T	G	C	G
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

Figure 2: A Needleman-Wunsch alignment of $S_1 = \text{GATTACA}$ and $S_2 = \text{GCATGCG}$, yielding a score of 0. Figure adapted from [6]

The computational and space complexity of Needleman-Wunsch is $O(n \cdot m)$, where n and m are the lengths of the two sequences. For the sake of clarity, this can can be rewritten in term of a single variable n, which is the average length of n and m. In simplified terms the complexity is instead $O(n^2)$, indicating that both time and space requirements scale quadratically with the size of the input sequences.

As mentioned, most sequence alignment algorithms cannot easily be used to align circular sequences. The Needleman-Wunsch algorithm is no exception for this. Nevertheless it is crucial to understand Needleman-Wunch, since many more complex algorithms invoke it.

1.3.3 Multiple Sequence Alignment

The challenge of processing hundreds or thousands of sequences in bioinformatics research necessitates efficient computational approaches. Extending pairwise alignment methods to multiple sequences rapidly increases their complexity, making such methods impractical for large datasets. To address these limitations, alternative heuristic methods like progressive alignment have been developed. [7]

Heuristic MSA: Progressive Alignment Progressive alignment is a widely adopted heuristic method that offers a practical solution to the challenges posed by traditional multiple sequence alignment approaches. The process involves the following steps:

1. Computing pairwise distances between all sequences to construct a distance matrix.
2. Applying a clustering algorithm to the distance matrix to build a guide tree.
3. Using the guide tree to construct a series of pairwise alignments, each yielding a profile sequence that represents the alignment.
4. Continuously aligning sequences and profile sequences until a final profile sequence is produced.

This method is depicted in Figure 3, and further details on its motivation and computational complexity are provided in Appendix 6.7.

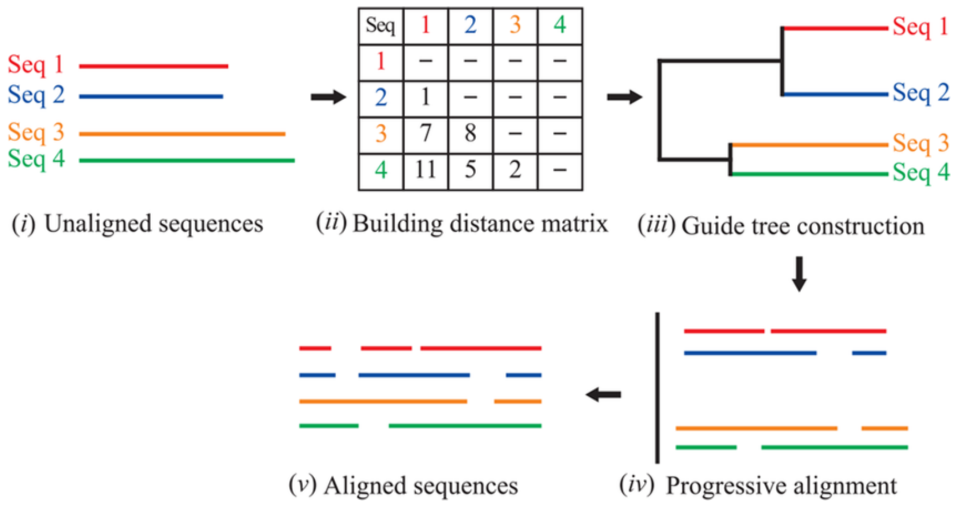


Figure 3: The process of progressive alignment with each of the individual steps. Figure from [8]

Progressive Alignment Steps In-depth The initial step in progressive alignment involves conducting pairwise comparisons across all sequences to calculate distances, typically using the Needleman-Wunsch algorithm to compute scores for insertion into the matrix.

Following this, a clustering algorithm, often the neighbor-joining method, is applied to the distance matrix to build a guide tree. [9] This tree dictates the order of sequence alignments, ensuring that the most closely related sequences are aligned first, which minimizes the overall alignment error as the process progresses.

The final phase of progressive alignment involves a series of $k - 1$ pairwise alignments for k sequences, guided by the structure of the tree. This series of alignments gradually builds up a comprehensive profile that includes all sequences, with each step integrating a new sequence or aligned pair into the existing profile. This iterative approach effectively reduces the complexity of aligning multiple sequences simultaneously.

Creating Profile Sequences in Progressive Alignment The profile plays an essential role in understanding progressive alignment. It is meant to be a representation of the alignment that encapsulates both consensus and variations. An essential part of using profile sequences is that they themselves can be used in pairwise alignments to yield even larger profile sequences. This way a final MSA can be reached by progressively aligning sequences or profiles, each time creating a new composite profile until all sequences are amalgamated into a single profile sequence. Figure 4 illustrates an example of how profiles are constructed.

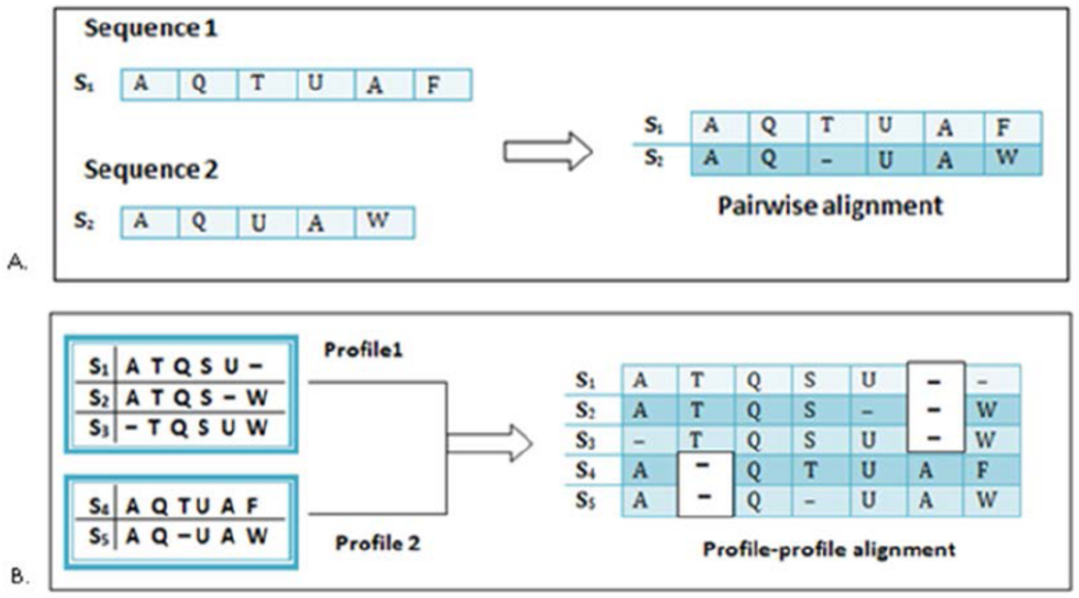


Figure 4: Illustration of progressive alignment starting from pairwise alignment of protein sequences and leading to the creation of composite profiles. The figure shows how individual alignments (A) and profiles (B) are progressively integrated. Figure from [10]

While progressive alignment provides a scalable approach for linear genomes, its adaptation to circular genomes remains challenging. To integrate the method with circular genomes a more flexible data structure is needed.

1.4 Genome Variation Graphs

This issue can be addressed through the use of graphs. A graph is a data structure represented as a graph $G = (N, E)$, where N denotes nodes, and E represents the edges connecting these nodes. This structure is commonly used in computer science to model problems and derive algorithmic solutions.

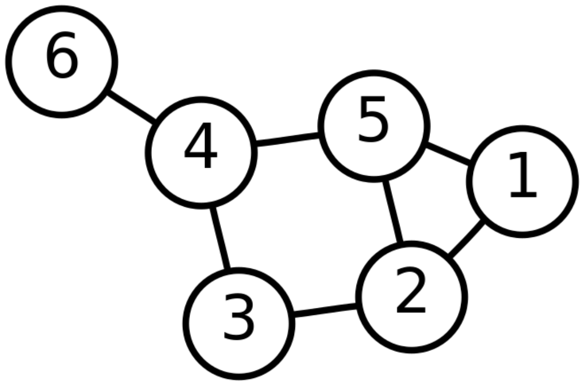


Figure 5: An example of a simple graph with 6 nodes and 7 edges. Figure from [11]

Graphs can be extended to more complex forms to accommodate the needs of genomic data. For constructing genome variation graphs, the following adaptations are made:

- **Paths (P)**: These are sequences of edges $p = (e_1, e_2, \ldots, e_{n-1})$ that connect multiple nodes, representing an aligned genome in the graph.
- **Node Content**: Each node encapsulates or represents a genetic sequence constructed from a nucleotide alphabet $A = \{A, C, G, T\}$.
- **Bidirectional Edges (e_{ij})**: Nodes are connected in one of two directions, where $e_{ij} = (n_i, n_j)$ denotes sequence of i followed by the the sequence of j and e^*_{ji} denotes the sequence of i followed by the reverse complement of j.

This structure allows for representing pangenomic alignments effectively. By following paths within the graph, one can reconstruct genomes from their alignments by appending the genetic sequences represented by each node. Figure 6 shows an example of a genome graph. In this example, the genomes can be reconstructed from the paths: p_1 : ATCCCCTA and p_2 : ATGTCTA.

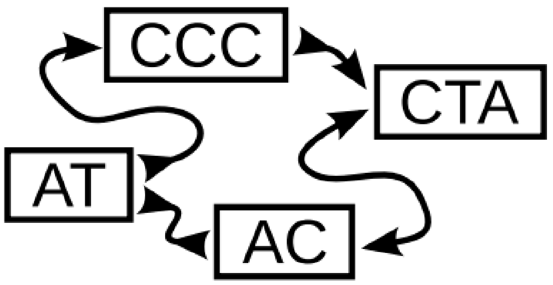


Figure 6: A bidirectional genome graph. The nodes can be identified as $N = n_1(AT), n_2 : (CCC), n_3 : (CTA), n_4 : (AC)$ and paths are $P = p_1 : (e_{12}, e_{23}), p_2 : (e^*_{14}, e_{43})$ Figure from [12]

Moreover, since graphs can be circular, genome variation graphs has the capacity to model circular genomes. To utilize these circular genome variation graphs, appropriate tools for alignment are necessary.

1.5 Aligning Sequences to Graphs

The Variaton Graph toolkit (vg) offers a series of tools for constructing, manipulating, and aligning sequences to genome variation graphs. [13] This subsection highlights the workflow centered around three critical tools: `vg map`, `vg prune`, and `vg augment`. A more comprehensive description of each relevant vg tool is available in Appendix 6.2.

`vg map` serves as a cornerstone of the workflow, designed to align DNA sequences against genome variation graphs. For `vg map` to operate effectively in complex genomic regions, the graph must be simplified. This simplification is facilitated by `vg prune`, which removes complex regions from the graph, thereby making it more manageable for `vg map` to process. [3] [14] Following the alignment, `vg augment` is employed to integrate the sequence and alignment data back into the original graph. By utilizing this workflow, it is possible to construct a circular pangenomic graph that accurately represents a set of circular alignments.

Despite its popularity in genomic studies, the documentation for vg is notably sparse, posing challenges for users who wish to fully leverage its capabilities. Nonetheless, it remains a powerful tool capable of handling circular graphs.

The following section will build on this foundation by delving into the specific methods behind implementing vgOrient, showcasing how the principles and tools discussed are applied in practice.

2 Methodology

2.1 Implementation

To enhance the efficiency of the genomic analysis, we developed a series of scripts. These scripts are improving a previous project's tool designed for orienting, cutting, and rotating DNA sequences. The initial tool also employed `vg map` (see 6.6.3) and demonstrated potential, but was computationally intensive and had issues with reproducibility.

In this project, we aimed to streamline the process by implementing a variant of progressive alignment (see Section 1.3.3). The individual steps of vgOrient are:

1. **Calculating a distance metric** for each sequence pair to build a matrix of pairwise distances.
2. **Running a simple ordering algorithm** to optimize the sequence processing order.
3. **Conducting a series of alignments** using `vg map`, aligning each sequence against a graph to iteratively build a comprehensive graph that embeds these alignments.
4. **Parsing the graph** to identify the longest node shared by all paths within the graph. The FASTA files are then reconstructed starting from this node.

These enhancements not only improved the computational efficiency but also streamlined the usability of the tool, facilitating more rapid and consistent genomic analysis. These steps are explored in depth below.

2.1.1 Distance Matrix

Getting a distance metric As mentioned in 1.3.3, the first step of progressive alignment is to calculate the pairwise distances between the sequences. One of the tasks of our project is to handle reverse complemented sequences, so we need a way to account for the orientation of the sequences. We decided to use q-grams, as they are a good alignment-free method that can easily be adjusted for circular inputs.

q-grams are subsequences of length q derived from a larger sequence and are useful in approximating the similarity between sequences in an efficient manner. [15] They allow for a straightforward adjustment to circular sequences by wrapping the sequence to include the first $q-1$ bases at the end of the sequence during q-gram construction.

In order to account for orientation, the script generates two sets: one for each sequence and one for its reverse complement. This process yields four sets of q-grams for each pair of sequences that the script can then use for building distance matrices.

We decided to use the Jaccard distance over the q-gram sets. The Jaccard distance is defined as:

$$J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

where A and B are two sets. [16] It ranges from 0 to 1 where 1 indicates that the sets are fully disjoint.

As mentioned, this process results in four sets of q-grams. Only *three* of these sets are actually needed. To understand why that is, we need to look at how reverse complementation works. This operation consist of two operations, the reversal of the and the complementing of the string. Both of these operations maps a unique string to another unique string. As a result, the reverse complementing also has this mapping property.

This mapping can also be applied over a set. In that case, the reverse complement of the set (e.g. $r(A)$) is the reverse complement applied to all of the elements in the set. Since the inputs and output are unique and set operations above only care about identity, we can conclude that $J(A, B)$ and $J(r(A), r(B))$ are identical. Similarly, $J(r(A), B)$ and $J(A, r(B))$ are also identical. This means that each pairwise comparison yields two values: Cis Jaccard, which is $J(A, B)$ or $J(r(A), r(B))$ and trans Jaccard which is $J(r(A), B)$ or $J(A, r(B))$.

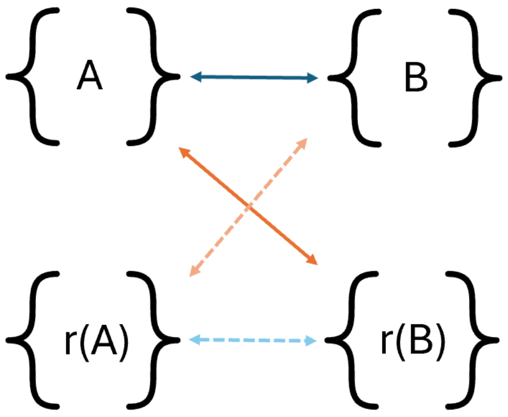


Figure 7: The Jaccard distance between two sets and their reverse complements. The Jaccard distances are pictured as arrows. Cis Jaccard is in blue, trans Jaccard is in orange. Only two Jaccard calculations need to be made.

Orienting the Sequences Usually in MSA a $k \times k$ distance matrix is built, where k is the number of sequences to be rotated. In the case of our algorithm, we have two values for each pair so we have two $k \times k$ matrices instead.
To figure out the correct orientation for all the sequences, we want to optimize for the minimal sum of Jaccard distances. This is trivial to do when $k = 2$. If trans Jaccard is lower than cis Jaccard one of the sequences should be reverse complemented.

In the case of $k > 2$, finding the optimal solution
becomes much harder. When flipping a sequence, each metric in the distance matrix related to that sequence should be flipped from cis to trans or vice versa. In this case, that means flipping all cells that share a col or a row with that sequence. This means that flipping a single sequence changes $(k - 1) \cdot 2$ cells. Of those cells $k - 1$ share a row and the rest share a col.

To parameterize the problem for n sequences we have $(k - 1)^2$ unique orientations of the sequences. These orientations indicate which of the two metrics to use, but since inverting the orientation of each sequences results in the same set of metrics, we can reduce the number of orientations by a factor of two. This essentially means that we can fix the first sequence to always be oriented forwards. Each of these unique orientations result in a single unique matrix $k \times k$ which can then be summed. Flipping a single sequence only flips $(k-1) \cdot 2$ cells, so we can avoid rebuilding the entire matrix with each flip and instead only do $O(k)$ operations to update the sum. Since there are $(k-1)^2$ orientations to check and each of these checks take $O(k)$ time an exact solution to this problem can be found in $O(k^3)$ time.

While the exact solution is feasible for smaller datasets, it becomes impractical for large k. Therefore, we also implemented a heuristic approach, which is more efficient for larger datasets. It should be noted that vgOrient has two ways to determine orientation of the input sequences. Minimizing Jaccard distance is one of them, the other is described in Section 2.1.4.

After determining the optimal orientation using either the exact or heuristic method, the distance matrix is reconstructed and used in the sequence ordering algorithm.

2.1.2 Ordering

Issues with previous version The previous version of the tool did not order the sequences, but instead chose one at random to be the starting sequence. It then proceeded to do sequence-to-graph alignments on all of the remaining sequences. After doing all the alignments with the starting sequence, the one with the highest identity score is kept, while the others are discarded. This step is iterated upon, each time aligning the remaining of the sequences then keeping the highest identity score.

We found numerous issues with this method of alignment. Firstly, the choice of the first sequence is fully at random. If a sequence with high distance to the remaining sequences is chosen, the produced alignments will be very bad. Secondly, it takes a lot of time. This is a series of $\frac{k(k-1)}{2}$ comparisons, so it has the time complexity of $O(k^2)$ times the runtime of `vg map`, where k is the number of sequences. The time complexity `vg map` is not described in its paper, but testing shows that it scales badly (see Appendix 6.3). We therefore decided on an alternative to reduce processing times.

Divergence from Progressive Alignment Since vgOrient has built a distance matrix, we can now determine an order based on distance. This ordering algorithm

is quite different than the clustering algorithm usually used in progressive alignment. This is because the usual alignment methods support not only aligning sequences with sequences but also aligning sequences with profiles or profiles with profiles. As mentioned in Section 1.5, `vg map` only supports aligning sequences with graphs.
This means that the usual guide-tree structures cannot be followed, since the only possible operation is to join leaves (sequences) to the single node (graph) that is built. Thus, the guide-tree is instead always sequential.

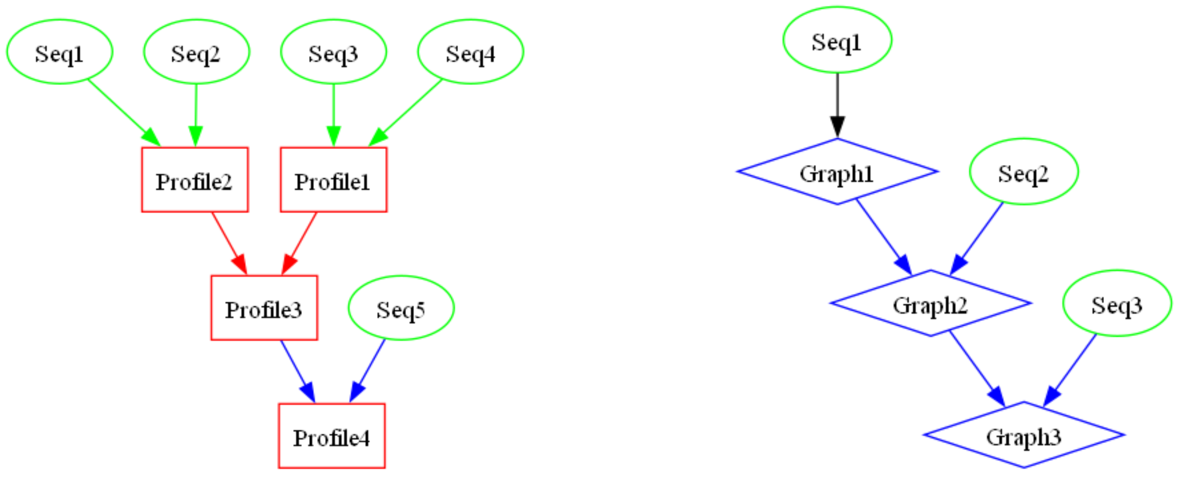


Figure 8: Comparative view of guide trees. The left figure shows an example guide tree in progressive alignment. The three possible types of alignment are shown by the different arrows. The right figure shows the only possible guide tree topology with vg map. The illustrations were rendered with GraphViz.

As a consequence, we did not need to implement a clustering algorithm. We just needed to order the sequences.

Ordering Implementation The script finds the selects sequence as the one which has the lowest lowest average distance to the other sequences. It then builds the order by selecting the sequence with lowest average distance to the current selection. This step takes in total $O(k^3)$ time. Once all sequences are selected, the script outputs the order the files should be processed by. After this step, the sequences are ready to be aligned.

2.1.3 Progressive Graph Alignment

The purpose of the progressive alignment is to convert a series of sequences into a single circular pangenomic graph, following the order determined in the last step. This alignment is implemented through a series of vg commands executed using the Python subprocess module, and it involves the following phases:

Building the Initial Graph Initially, the script builds the graph from the first sequence, which sets the foundation for subsequent alignments. The first sequence is converted into a genome variation graph, thus creating a path that represents a single genome. This path is then circularized to represent a singular mitogenome, which prepares it for iterative alignment.

Iterative Sequence-to-graph Alignment For each of the remaining $k-1$ sequences, the script follows the alignment process outlined in Section 1.5. It begins by aligning a sequence to the existing graph using `vg map`. After each alignment, the script runs `vg augment` to yield a new graph that includes the alignment. The script then prunes the graph to prepare it for the next iteration. This pruning is crucial for the subsequent `vg map` runs. This iterative process gradually transforms the initial singular genome representation into a comprehensive pangenomic graph. By processing this graph, we can finally rotate our genomes.

2.1.4 Graph Parsing and FASTA Rebuilding

In final step of the tool, we want to use the graph to find the longest subsequence that is shared among all the sequences. By finding the subsequence, we can then rotate the sequences so they start at that point. We also want to make sure all output sequences have the same orientation. As mentioned in Section 1.4, a genome variation graph contains nodes, edges and paths. Our script gets the set of nodes that is shared among all paths by taking the intersection of all nodes across all paths. The script can then look through them and find the longest one. By using the longest node, we had two options for producing a rotated FASTA file.

Rotation by FASTA search The first option is to read each FASTA, then search for the sequence embedded in the longest shared node. This lets us rotate the FASTA as long as the longest shared node is unique in the FASTA.

Rotation by Rebuilding from Path Another option to rotate the paths in the graph so that the longest node is the first entry. We can then write a new FASTA by reading the path and appending embedded sequences in each node. This also is an easy way to handle reverse complemented sequences. Since sequence orientation is stored in the edges, the sequences in the nodes have the same orientation. The only thing the script needs to do is to invert the order of any region of the path that has reverse edges. This solution does not rely on the shared node to be unique, but instead relies on the graph to correctly represent each sequence. Since the graph gets pruned between each alignment, this not guaranteed.

2.1.5 Multiple Implementations

In our discussion of the methods above, we highlighted two different approaches for orienting sequences and another two for writing rotated FASTAs. We chose to explore these multiple implementations with a specific purpose: to determine the most effective method. This approach is crucial, especially given the complexities surrounding vg.

vg is central to the functionality of vgOrient, but it comes with limited documentation and is inherently complex. As a result, it is unwise to make assumptions about its behavior.

By experimenting with various methods, we do not only identify the most efficient technique but also deepen our understanding of vg's behavior. This dual benefit underscores the value of our exploratory approach in software development.

These vgOrient variants will be elaborated further in Section 3.3, where we present the benchmarks and results of our experiments.

3 Results

3.1 Tools

In order to accurately and consistently evaluate the performance of vgOrient, it is compared alongside the state-of-the-art rotation tool MARS. [17] Both tools were assessed using identical methods to ensure a fair comparison. This evaluation involved performing multiple sequence alignments with the MSA tool MAFFT [18] on the oriented and rotated genomes from both vgOrient and MARS. The quality of these alignments was then evaluated using a weighted version of Shannon's entropy, known as Wentropy. [19] Wentropy quantifies the conservation within an MSA on a scale from 0 (indicating perfect conservation) to 1 (denoting a fully random alignment). For in-depth descriptions of Wentropy, MARS, and MAFFT, please refer to Appendix 6.6.

Additionally, to ensure the reproducibility of our results, we have documented each command used throughout this project, along with the versions of all tools employed. This documentation, including a list of all major and minor tools used, is accessible in Appendix 6.2.

3.2 Datasets

This project made use of three different datasets for benchmarking, denoted Easy, Medium and Hard. Easy and Medium are smaller datasets with low or somewhat low divergence, respectively. The Hard dataset is larger and more divergent. The Easy dataset consists only of the Suina suborder. Both the Medium and Hard datasets that contain apes, but the Hard dataset has been expanded to also contain more distantly related species such as whales and pigs. The accession IDs and species names of all the datasets are listed in Appendix 6.1.

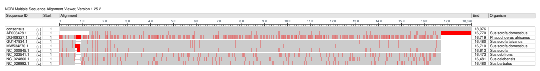
The benchmarking datasets were transformed to create two datasets each. The transformation consisted of either

- rotating each sequence randomly
- reverse complementing the sequence with a 50% chance in addition to random

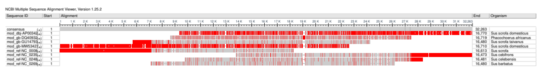
rotation

The transformation was repeated 50 times to yield 100 variants of each dataset, 50 of them flipped as well as 50 non-flipped. These variants are the ones we use for our benchmarking.

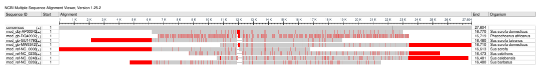
Visualizing Transformed Data Below is a set of visual representation of the MAFFT MSA applied on the Easy dataset with different methods applied. These provide a good visual aid to understanding the consequences of the transformations as well as potential issues with the dataset. From Figure 9a, it is clear that the first sequence (AP003438.1) is not rotated correctly with respect to the rest. It can also be seen how a rotation tool can restore any of the other datasets to a much better rotation.



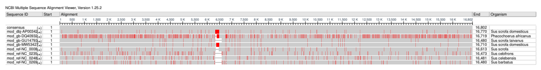
(a) The Easy dataset alignment from the original non-transformed dataset. Wentropy = 0.0678



(b) The Easy dataset alignment from the flip dataset without rotation tool (run 1). Wentropy = 0.367



(c) The Easy dataset alignment from the noflip dataset without rotation tool (run 1). Wentropy = 0.326



(d) The Easy dataset after rotation and orientation by vgOrient. Wentropy = 0.039

Figure 9: Comprehensive view of the Easy dataset alignments. Mismatches are highlighted in red, gaps are white and matches are grey.

3.3 Benchmarks

3.3.1 vgOrient Variants

Method of Variants We tested three distinct variants of vgOrient, each employing different methods for orienting sequences and generating rotated FASTA files. vgOrientGraph utilizes the paths in the graph produced by progressive vg alignments, orienting each sequence by reconstructing the FASTA from sequences in each path node. vgOrientMatrix, on the other hand, employs the q-gram based distance matrix from step 1 of the progressive alignment (see Section 2.1) for orientation, rebuilding the FASTAs in the same way as vgOrientGraph. Lastly, vgOrientSearch also uses the q-gram distance matrix for orientation but diverges in its approach by searching each FASTA for the longest shared node in the graph, using found indices to rotate the FASTAs instead of reconstructing them. A comparative table of each variant's characteristics is presented in Table 1.

Variant	**vgOrientGraph**	**vgOrientMatrix**	**vgOrientSearch**
Orientation Method	Uses paths in graph from step 3 to orient sequences by reading edges in path nodes.	Uses q-gram based distance matrix from step 1 of the progressive alignment for orientation.	Uses q-gram based distance matrix from step 1 of the progressive alignment for orientation
FASTA Writing	Rebuilds the FASTAs in by reading sequences in each path node.	Rebuilds the FASTAs in by reading sequences in each path node.	Searches each FASTA for the longest shared node and uses indices to rotate FASTAs instead of rebuilding them.

Table 1: Comparison of vgOrient variants. The steps mentioned are from Section 2.1
.

Wentropy of Variants The primary way we want to compare the variants is by how well they rotate our dataset. Table 2 presents the Wentropies of each variant. Each cell in the table represents the average of up to 50 replicates. If a run on a replicate failed, it is not included in the average. From the table, we can be see that vgOrientSearch performs considerably better than the other two variants. vgOrientGraph and vgOrientMatrix have very similar Wentropy scores, with vgOreintGraph having slightly lower scores overall.

Table 2: Performance by Method, Difficulty Level, and Flip State

Flip	Method	Easy	Medium	Hard
✓	vgOGraph	0.0433	0.0822	NA
✓	vgOMatrix	0.0435	0.0808	0.2151
✓	vgOSearch	**0.0397**	**0.0748**	**0.1955**
✗	vgOGraph	0.0440	0.0842	0.2172
✗	vgOMatrix	0.0441	0.0829	0.2244
✗	vgOSearch	**0.0397**	**0.0748**	**0.1956**

Errors of Variants This is not the only basis for comparison. The variants also differ in how prone they are to errors. Figure 10 shows the error distributions from each variant. The variant least prone to error is vgOrientMatrix, which succeded 33 out 50 runs on the flipped Hard dataset. vgOrientSearch is more error prone with 17 successful runs out of 50. vgOrientGraph did not have any successful runs, which is why it has a missing Wentropy score in Table 2.

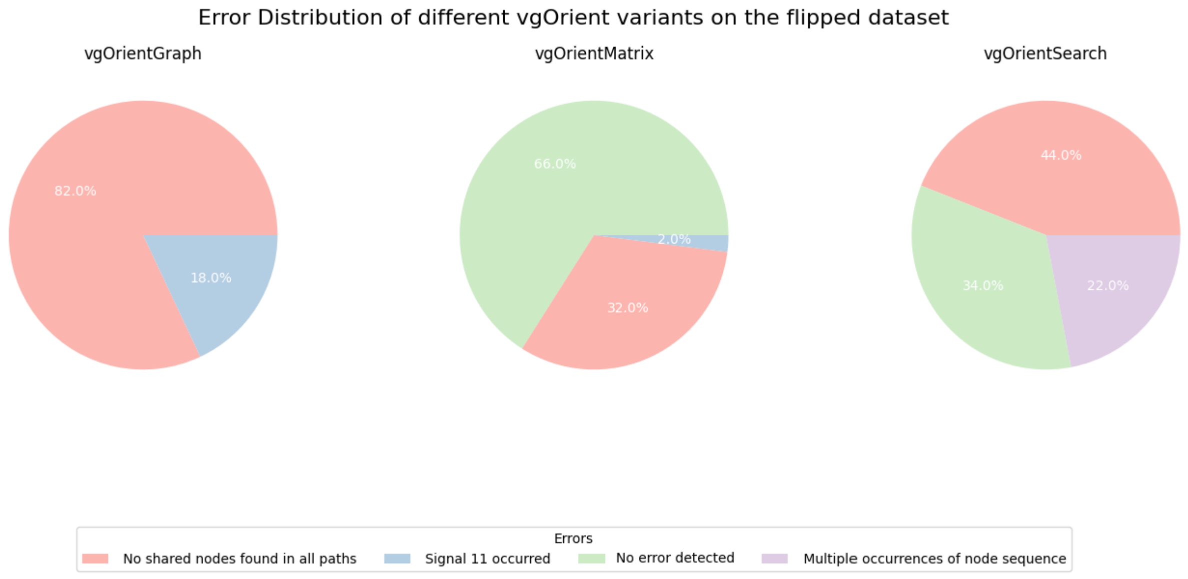


Figure 10: Distribution of errors with vgOrient on the flipped Hard dataset

The tests produced three different types of errors: Signal 11 is a vg error that happens when augmenting a graph (see Section 6.6.3). The "no shared nodes found in all paths" error happens when the progressive alignment is done, but the produced graph does not have a region that is shared among all the sequences. The final error only happens in vgOrientSearch when the search yields multiple hits.

The error counts of the rest of the datasets are presented in Table 3. From it, it is clear that all variants run mostly fine on the Easy and Medium datasets. vgOrientGraph has issues specifically with the flipped Hard dataset. Also, it seems that vgOrientSearch is considerably more error-prone than vgOrientMatrix.

Table 3: Error Count by vgOrient Variant, Difficulty Level, and Flip State

Flip	**Method**	**Easy**	**Medium**	**Hard**
✓	vgOGraph	1	0	50
✓	vgOMatrix	0	0	17
✓	vgOSearch	0	1	33
✗	vgOGraph	0	0	23
✗	vgOMatrix	0	0	23
✗	vgOSearch	0	0	31

Runtime and Memory Usage of Variaints The variants are also compared by runtime and memory usage. Here they perform very similarly. The runtime and memory usages can be found in Appendix 6.4. Overall, vgOrientSearch seems to perform the best even though it is more error-prone than vgOrientMatrix. Therefore, it will be the chosen variant to use when we compare performance with MARS.

3.3.2 Comparing vgOrient with MARS

Wentropy Across Methods The Wentropies of vgOrient is compared to the performance of MARS as well as a "control" treatment in which neither rotation algorithm is used before MAFFT. To provide an additional point of reference, we also aligned and evaluated the "original" non-transformed linearized genomes supplied by the databases (NCBI or Ensembl).

vgOrient has much lower Wentropies than MARS on the flipped datasets. On the non-flipped datasets the performances are very similar. MARS performs worse than vgOrient on the Easy dataset, but almost identically on the Medium and Hard datasets. The original rotations have slightly worse performances to MARS and vgOrient, though it is considerably worse on the Easy dataset.

Table 4: Performance by Method, Difficulty Level, and Flip State

Flip	**Method**	**Easy**	**Medium**	**Hard**
✓	Control	0.3648	0.3435	0.3652
✓	MARS	0.3339	0.3118	0.3553
✓	vgOrient	**0.0397**	**0.0748**	**0.1955**
✗	Control	0.2967	0.2778	0.3330
✗	MARS	0.0404	0.0748	0.1957
✗	vgOrient	**0.0397**	**0.0748**	**0.1956**
NA	Original	0.0678	0.0849	0.2031

Run Time and Max Memory Usage The differences in run time between vgOrient and MARS vary by dataset. In the Easy dataset, vgOrient outperforms MARS in both run time and max memory usage. In the Hard dataset, vgOrient takes about 3 times as much time to run as MARS.

Table 5: Runtime in mm:ss by Method, Difficulty Level, and Flip State

Flip	**Method**	**Easy**	**Medium**	**Hard**
✓	MARS	01:31	01:21	06:32
✓	vgOSearch	00:37	01:07	17:51
✗	MARS	01:15	01:16	05:32
✗	vgOSearch	00:37	01:06	16:41

The max memory usages tell a similar story. On the Easy and Medium dataset the vgOrient has lower average max memory usage than MARS.
On the Hard dataset, memory usages are erratic, with vgOrient peforming better in the non-flipped dataset even though we would expect the opposite. We wanted to explore this further, so we used a box plot to visualize the distribution of max memory usages.

Table 6: Max Memory Usage (KB) by Method, Difficulty Level, and Flip State

Flip	Method	Easy	Medium	Hard
✓	MARS	1587566	1454789	1542590
✓	vgOrient	100165	95453	962410
✗	MARS	1115661	1089557	1194390
✗	vgOrient	100444	95168	1633485

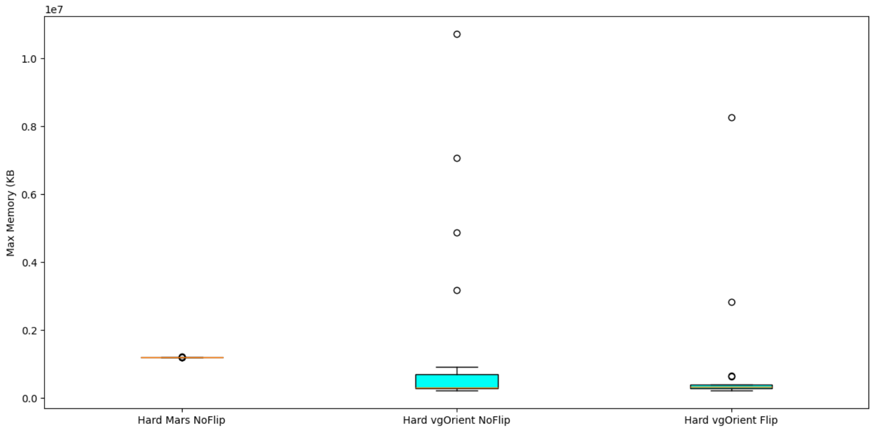


Figure 11: Box plot of max memory usages in the Hard dataset

From the boxplot (Figure 11) it is easy to understand what is going on with vgOrient. There are a couple of runs that inflate the max memory usages by a lot. This exemplifies pattern that can be seen in all the datasets. Compared to MARS, vgOrient has far less consistent memory usages and runtimes. More examples can be seen in Appendix 6.5.

4 Discussion

4.1 Interpretations of Results

vgOrient is a unique rotation tool that also has the capacity to reorient the input sequences if they are reverse complemented by mistake. This is why it has much lower Wentropies than MARS in the flipped datasets. In this context vgOrient is clearly the better tool, but even without this ability vgOrient is still a promising alternative to MARS.

In the non-flipped dataset vgOrient has as good as or better Wentropies than MARS. On smaller datasets it also runs faster than MARS. While this is impressive, vgOrient also has its drawbacks. On larger, more divergent datasets it is way slower than MARS. Furthermore, vgOrient cannot consistently run without error on these datasets. Overall, vgOrient is an effective tool that works the best on smaller, less divergent datasets.

Both MARS and vgOrient produced better rotations than the ones provided from the databases, but the original rotations were quite good compared to the control orientations. This indicates that most rotations from the databases are not at all random. The transformations we performed on the datasets were far more extreme than can be expected from any set of sequences from a database. Additionally, it also indicates the importance of rotation tools. Each dataset could be improved by rotation and some of them contained sequences that had clear rotation issues (Figure 9a).

4.2 Limitations

4.2.1 Limitation of Experimental Methods

The experiments we conducted could have been more thorough. Specifically, the flipped dataset could have been created by randomly flipping sequences from the non-flipped dataset, which would make it easy to discern the effects of flipping from those of rotation. Furthermore, employing a series of metrics, rather than just Wentropy to assess MSA quality would provide a more comprehensive evaluation.

4.2.2 Limitations of vgOrient Implementation

Variant-Specific Behavior of vg In our results, we presented three variants of vgOrient, each relying on vg to different extents. This section explores the performance differences among these variants and what these differences reveal about vg.

Firstly, `vg map` requires graph pruning in each iteration, which means the graph produced by vgOrient might not be intact. We observed that vgOrientSearch performed better than the other vgOrient variants. The higher reliance on the graph by the other variants could explain this performance discrepancy.

Moreover, it is notable that vgOrientGraph consistently fails on the flipped Hard dataset, whereas vgOrientMatrix does not. This failure cannot be solely attributed to the pruning process. Given that `vg map` is designed to identify the orientation of nodes from reverse complemented sequences, we initially expected similar behavior on both flipped and non-flipped datasets. The differing results suggest a specific issue with how vg handles combinations of sequences with different orientation, likely related to sequence alignment challenges posed by `vg map`.

General Behavior of vg Further complicating our understanding of vg's functionality, the vg workflow exhibits erratic behavior overall. While parts of vg support circular graphs and bidirectional edges, it is not well-integrated with the entire vg-toolkit. This is evidenced by the inconsistent outputs from transformed dataset. The variability is particularly pronounced with the Hard dataset, where some rotations cause vg to crash and others result in graphs without any shared nodes. Much of this unpredictable behavior may also stem from the graph pruning process.

Limitations of the Other vgOrient Steps While vgOrient consist of other steps that do not directly rely on vg, the methods used for graph parsing and sequence ordering are limited by what vg is capable of. The limitation on graph parsing is that the final graph is unreliable. Similarly, implementing a more complex clustering algorithm is not necessary since vg cannot follow more complex guide trees. The only step that has no clear restrictions imposed by vg is the first step: Building a distance matrix. With the current implementation, the distance matrix serves two purposes. To calculate distances and to identify reverse complements. There could well be a more accurate way to do both of these that still accounts for the circularity of the inputs. However, without the need to do clustering it is not clear how much this would improve the performance of vgOrient.

4.3 Future Work

This project lays a solid the foundation for graph-based rotation and orientation of multiple sequences. While there are likely many ways to improve the progressive alignment workflow implemented here, we suggest to focus on either:

- **Improving the progressive alignment step** by finding a better graph alignment algorithm than `vg map` that supports circular graphs and sequences. Ideally such algorithm would not need pruning. It could also support other types of alignments such as graph-graph or sequence-sequence. This would allow us to use other methods than we do with `vg map`. We could rely more on the output graphs and also implement a clustering algorithm to build a guide tree. Overall, such a tool would allow us to leverage the strengths of using progressive alignment and variation graphs to a larger extent. Sadly, we are not aware that a tool with this combination of features exists.
- **Improving the FASTA writing step** to accommodate the use of `vg map`. Since the graph embedded sequences cannot reliably be used to rebuild FASTAs, it should be used as minimally as possible. A simple solution would be to extend the vgOrientSearch implementation. The current implementation fails if the longest node occurs multiple times in the sequence. An easy improvement is to expand the search to include neighbouring nodes in such cases. While such a solution is simple, it leverage graphs to a lesser extent. This means that extensions that would otherwise be easy like reorienting genomes with reverse complemented parts cannot be easily implemented. Improving vgOrientSearch is a more feasible improvement, but will still be constrained by the issues with `vg map`.

These options provide both a short and a long-term plan for further developing this tool and increasing its efficiency.

5 Conclusion

In this paper we introduced a new tool to orient and rotate multiple sequences using graphs. vgOrient is heuristic tool based on progressive alignment. It takes a number of circular sequences, which are then reoriented and rotated so they can be aligned by the preffered MSA algorithm.

We used experimental studies with three transformed mtDNA datsets to compare vgOrient with the current state-of-the-art rotation tool MARS. From the experiments it was clear that vgOrient performed as good or better than MARS in the two smaller, less divergent datasets. vgOrient surpassed MARS Especially in terms of runtime and memory. With the larger more divergent dataset, vgOrient was still competitive with MARS in terms of rotation quality, but was error-prone and slower.

During this project a number of issues with the graph aligner used vgOrient appeared. Further work will seek to reduce reliance on this aligner as well as reduce the number of errors on more divergent datasets.

References

[1] P. D. N. Hebert, A. Cywinska, S. L. Ball, and J. R. deWaard. Biological identifications through dna barcodes. *Proceedings of the Royal Society B: Biological Sciences*, 270:313–321, 2003.

[2] F. Fernandes, L. Pereira, and A. T. Freitas. Csa: an efficient algorithm to improve circular dna multiple alignment. *BMC Bioinformatics*, 10:1–13, 2009.

[3] Alexandra Weronika Balicki. Algorithms to find appropriate rotations and orientations for circular sequences. Master's thesis, DTU, July 2022. Available at: `https://findit.dtu.dk/en/catalog/630d553e6076c725114ac90d` (Accessed: 2024-08-22), 32.5 ECTS.

[4] Michael Hoang. Simplifying big-o expressions, 2019. Available at: `https://medium.com/swlh/simplifying-big-o-expressions-4f7c6059d3d5` (Accessed: 2024-08-22).

[5] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[6] Kamil Slowikowski. Needleman-wunsch illustration, 2010. Available at: `https://gist.github.com/slowkow/508393` (Accessed: 2024-08-22, Modified in paint).

[7] D. F. Feng and R. F. Doolittle. Progressive alignment of amino acid sequences and construction of phylogenetic trees from them. *Methods in Enzymology*, 266:368–382, 1996.

[8] Soniya Lalwani, Harish Sharma, Abhay Verma, and Rajesh Kumar. An efficient discrete firefly algorithm for ctrie based caching of multiple sequence alignment on optimally scheduled parallel machines. *CAAI Transactions on Intelligence Technology*, 4, 2019.

[9] J. D. Thompson, D. G. Higgins, and T. J. Gibson. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting,

position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680, 1994.

[10] Umar Manzoor, Sarosh Shahid, and Bassam Zafar. A comparative analysis of multiple sequence alignments for biological data. *Bio-medical materials and engineering*, 26:1781–1789, 2015.

[11] Wikimedia simple graph, 2024. Available at: `https://commons.wikimedia.org/wiki/File:6n-graf.svg`
(Accessed: 2024-08-22).

[12] Novak A. M. Eizenga J. M. & Garrison E. Paten, B. Genome graphs and the evolution of genome inference. *Genome Research*, 27(5):665–676, 2017.

[13] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, and R. Durbin. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 2018.

[14] VG Team. Working with a whole genome variation graph, 2024. Available at: `https://github.com/vgteam/vg/wiki/Working-with-a-whole-genome-variation-graph`
(Accessed: 2024-08-22).

[15] K. Katoh, J. Rozewicki, and K. D. Yamada. Mafft online service: multiple sequence alignment, interactive sequence choice and visualization. *Briefings in Bioinformatics*, 20(4):1160–1166, 2019.

[16] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et du jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37(142), 1901.

[17] L. A. K. Ayad and S. P. Pissis. Mars: improving multiple circular sequence alignment using refined sequences. *BMC Genomics*, 18(1):86, 2017.

[18] K. Katoh, K. Misawa, K. Kuma, and T. Miyata. Mafft: a novel method for rapid multiple sequence alignment based on fast fourier transform. *Nucleic Acids Research*, 30(14):3059–3066, 2002.

[19] William S. J. Valdar. Scoring residue conservation. *Proteins: Structure, Function and Genetics*, 48:227–241, 2002.

[20] vgorient github, 2024. Available at: `https://github.com/whelixw/mitographs`.

[21] Robert C. Edgar and Serafim Batzoglou. Multiple sequence alignment. *Current Opinion in Structural Biology*, 16(3):368–373, 2006. Nucleic acids/Sequences and topology.

[22] Kazutaka Katoh and Hiroyuki Toh. Recent developments in the mafft multiple sequence alignment program. *Briefings in Bioinformatics*, 9(4):286–298, July 2008.

[23] Mars github, 2024. Available at: `https://github.com/lorrainea/MARS` (Accessed: 2024-08-22).

[24] R. Grossi, C.S. Iliopoulos, R. Mercas, et al. Circular sequence comparison: algorithms and applications. *Algorithms for Molecular Biology*, 11:12, 2016.

[25] vg github, 2024. Available at: `https://github.com/vgteam/vg` (Accessed: 2024-08-22).

6 Appendices

6.1 Datasets

Table 7: Easy Dataset

Accession number	Organism
DQ409327.1	Phacochoerus africanus (common warthog)
MW534270.1	Sus scrofa domesticus (Gannan Tibetan domestic pig)
NC_000845.1	Sus scrofa (pig)
NC_026992.1	Sus barbatus (bearded pig)
NC_024860.1	Sus celebensis (Celebes wild boar)
NC_023541.1	Sus cebifrons (Visayan warty pig)
AP003428.1	Sus scrofa domesticus (domestic pig)
GU147934.1	Sus scrofa taiwanus (Taiwan pig)

Table 8: Medium Dataset

Accession number	Organism
GU189665.1	Chimp
JN191188.1	Bonobo
KF914214.1	Gorilla
KU308539.1	Chimp
NC_002083.1	Orangutan
NC_011120.1	Gorilla

Table 9: Hard Dataset

Accession number	**Organism**
AJ421451.1	Lemur
EF545592.1	Pig
GU189665.1	Chimp
JN191188.1	Bonobo
JN632613.1	Bovid
KC572708.1	Whale
KF914214.1	Gorilla
KU308539.1	Chimp
LC726588.1	Human
MW242801.1	Gibbon
MW538294.1	Pig
MW733805.1	Marmoset
NC_002083.1	Orangutan
NC_011120.1	Gorilla
NC_012771.1	Lemur
NC_014047.1	Gibbon
NC_020690.1	Bovid
NC_050682.1	Marmoset
OQ199689.1	Rhesus Macaque
OQ199690.1	Rhesus Macaque
NC_002765.1	Slow Loris
NC_040292.1	Slow Loris

6.2 Tools

This section details the tools and their respective commands used in the project to ensure reproducibility. The version information and specific commands are also listed here. All project work was done on node 7 of the DTU healthtech cluster.

6.2.1 vg

Version 1.55.0 "Bernolda" of the Variaton Graph toolkit was utilized for the progressive alignment step of vgOrient. Commands are detailed in the project github. [20] The software was configured to run on up to 10 threads.

6.2.2 MARS

MARS, which manipulates circular sequences, was compared against vgOrient. As MARS only has a single release, no version numbering is applicable. The command used was:

```
mars -a DNA -i input -o output -m 0 -T 10
```

The command specifies the fast approximate mode (-m 0) and allows up to 10 threads, aligning with the configuration used for vg.

6.2.3 MSA

MAFFT v7.505 was selected for multiple sequence alignment. The commands executed were:

```
mafft --retree 2 --maxiterate 0 input > output
```

6.2.4 Runtime Testing

Runtime and memory usage were recorded using /usr/bin/time, alongside stdout and stderr redirection for logging purposes. The command used was:

```
/usr/bin/time -v script 2>&1 | tee time_file
```

6.3 Progressive vg Alignment Times

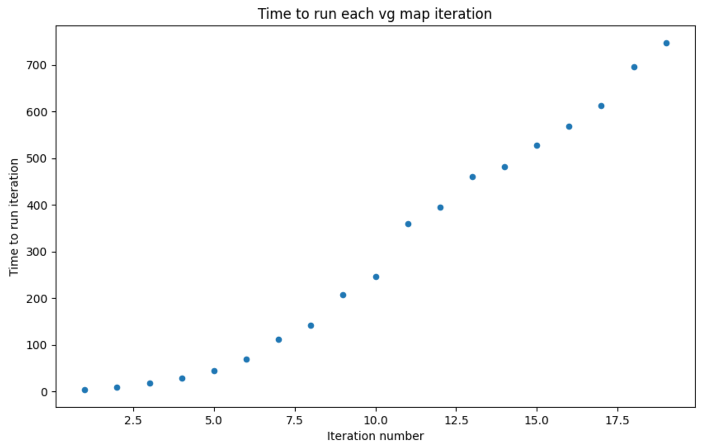


Figure 12: Time to run each iteration of the progressive alignment on the hard dataset

6.4 vgOrient Variants Performance Analysis

6.4.1 Wentropy Performance

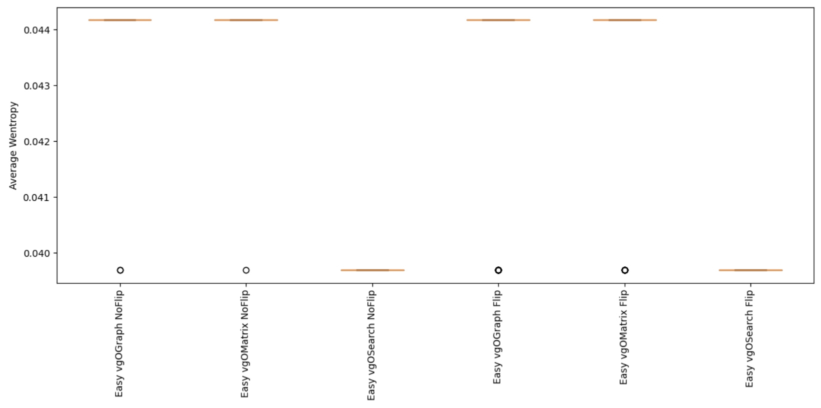


Figure 13: Boxplot of Wentropy performance for vgOrient variants on easy difficulty datasets.

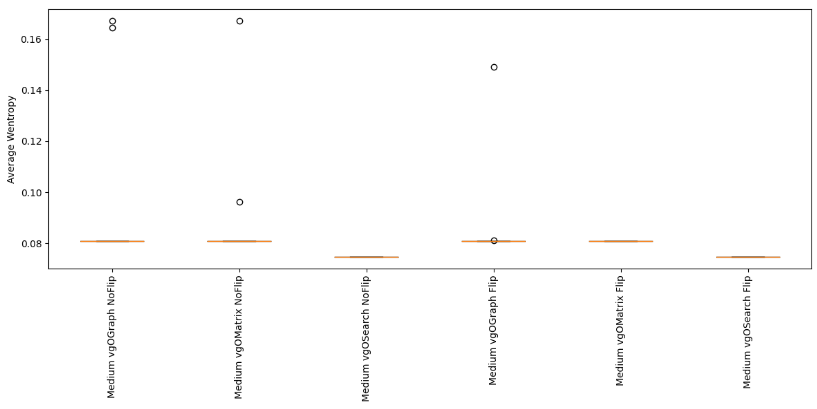


Figure 14: Boxplot of Wentropy performance for vgOrient variants on medium difficulty datasets.

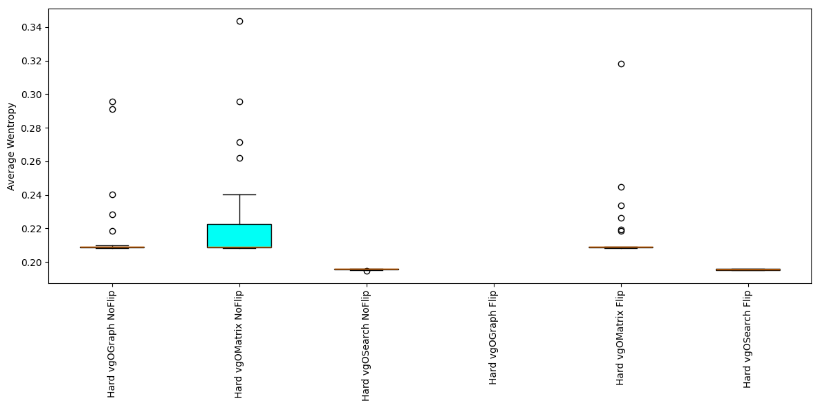


Figure 15: Boxplot of Wentropy performance for vgOrient variants on hard difficulty datasets.

6.4.2 Memory Usage

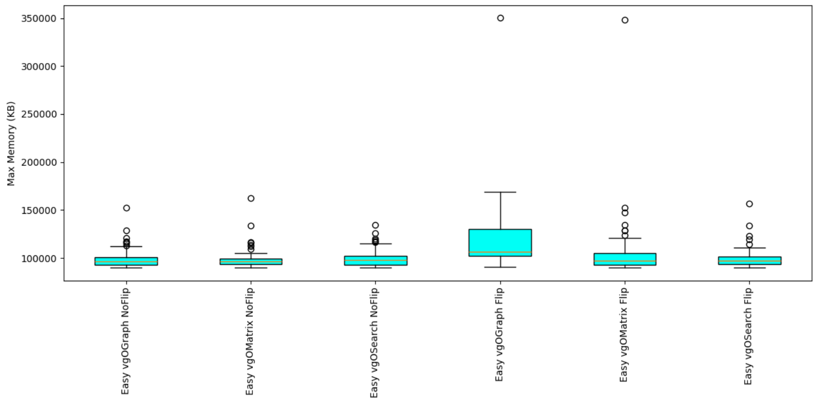


Figure 16: Boxplot of memory usage for vgOrient variants on easy difficulty datasets.

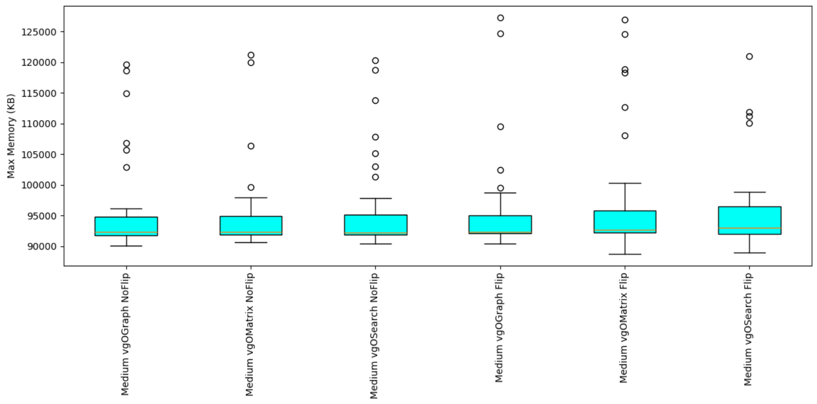


Figure 17: Boxplot of memory usage for vgOrient variants on medium difficulty datasets.

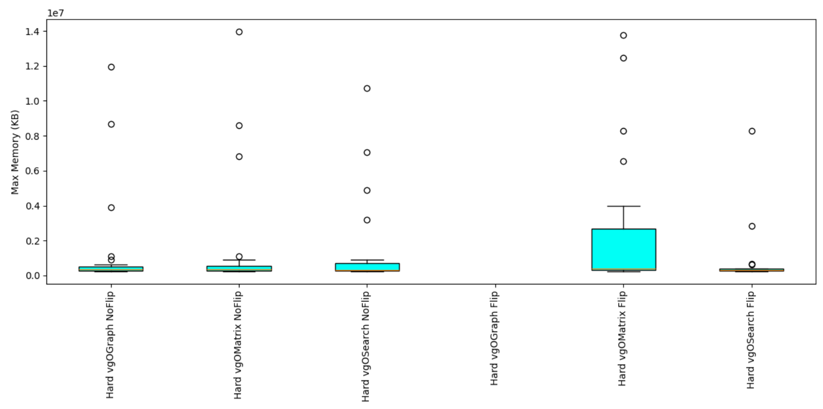


Figure 18: Boxplot of memory usage for vgOrient variants on hard difficulty datasets.

6.4.3 Runtime Performance

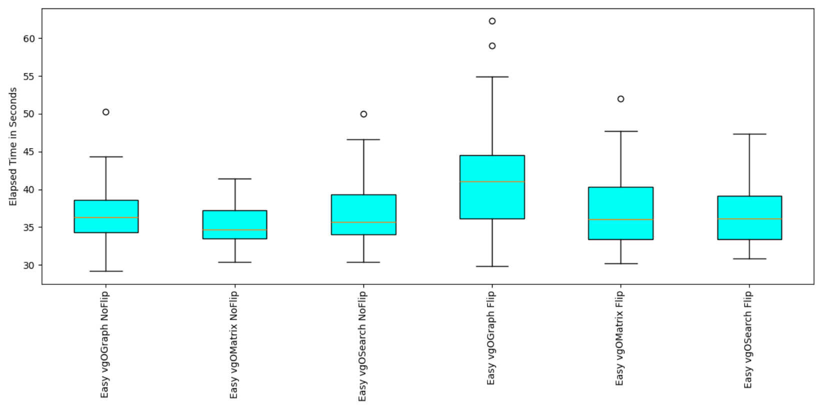


Figure 19: Boxplot of runtime performance for vgOrient variants on easy difficulty datasets.

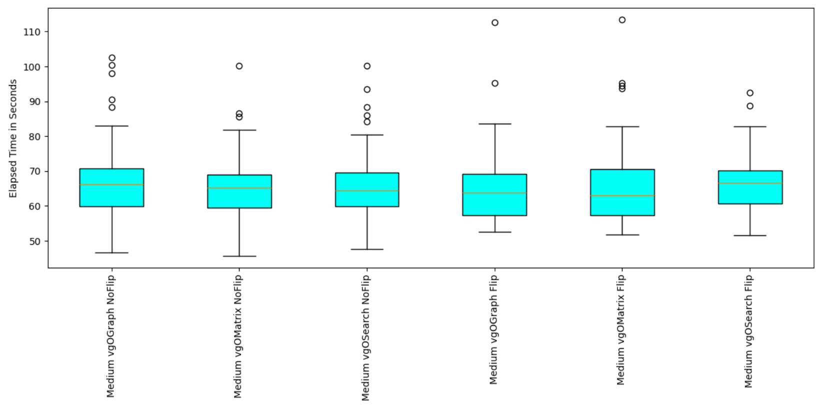


Figure 20: Boxplot of runtime performance for vgOrient variants on medium difficulty datasets.

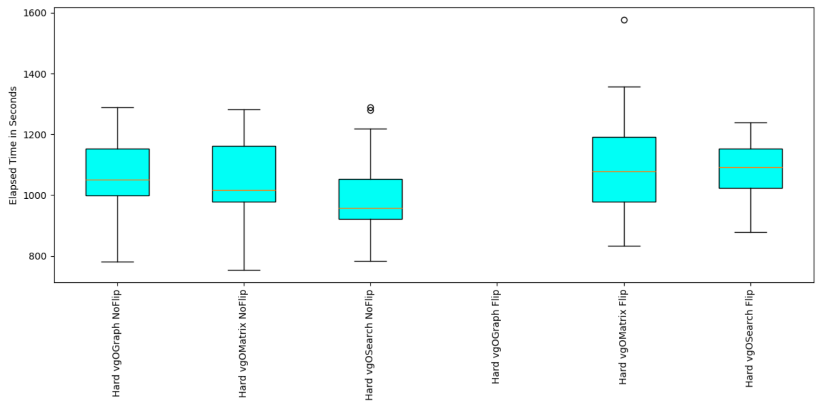


Figure 21: Boxplot of runtime performance for vgOrient variants on hard difficulty datasets.

6.5 Performance Analysis vs. MARS

6.5.1 Overall Runtime Across All Datasets

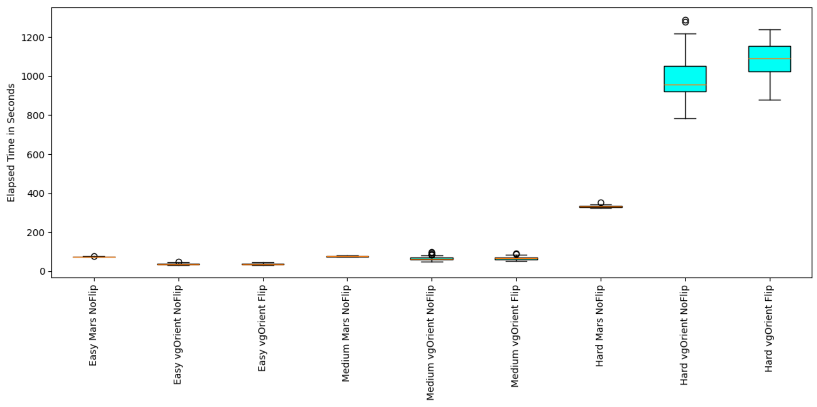


Figure 22: Boxplot comparing overall runtime across all datasets including Mars comparison.

6.5.2 Runtime by Dataset Difficulty

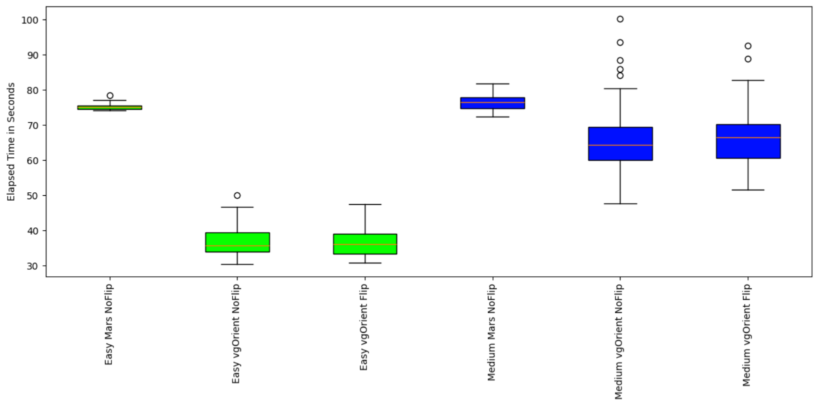


Figure 23: Boxplot of runtime on easy and medium difficulty datasets.

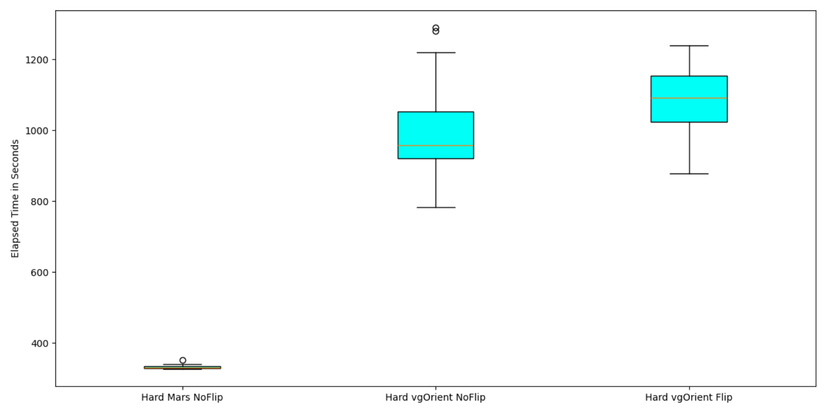


Figure 24: Boxplot of runtime on hard difficulty datasets.

6.5.3 Overall Memory Usage Across All Datasets

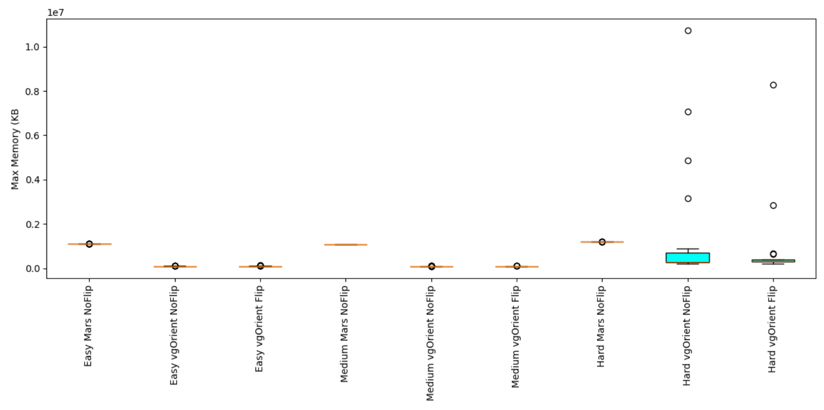


Figure 25: Boxplot comparing overall memory usage across all datasets including Mars comparison.

6.5.4 Memory Usage by Dataset Difficulty

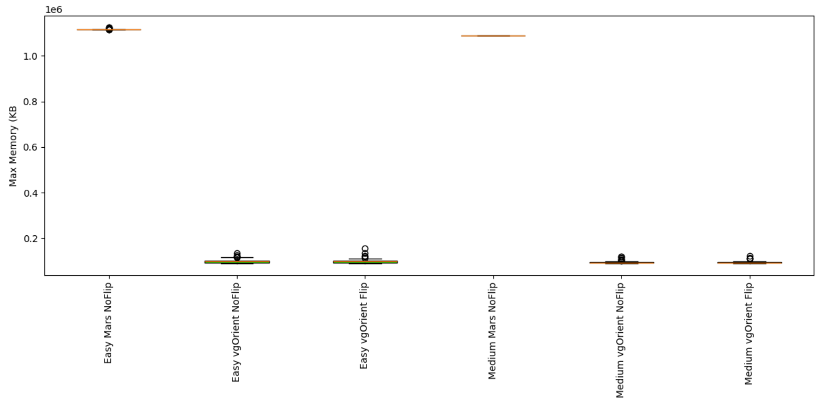


Figure 26: Boxplot of memory usage on easy and medium difficulty datasets.

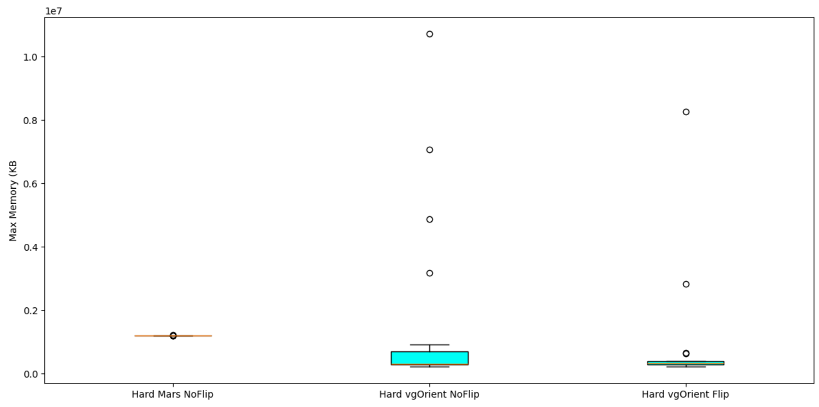


Figure 27: Boxplot of memory usage on hard difficulty datasets.

6.5.5 Wentropy by Dataset Difficulty

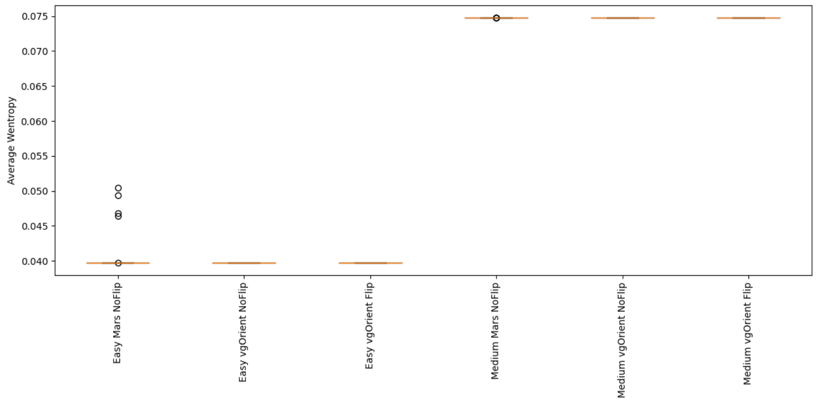


Figure 28: Boxplot of Wentropy on easy and medium difficulty datasets comparing vgOrient vs Mars.

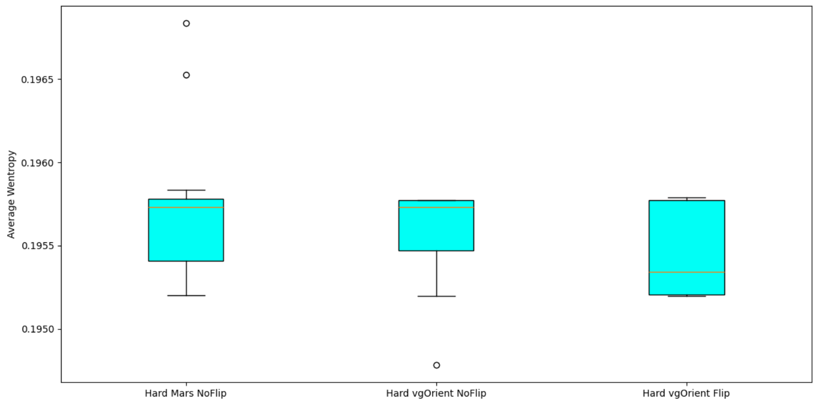


Figure 29: Boxplot of Wentropy on hard difficulty datasets comparing vgOrient vs Mars.

6.6 Tool Details

6.6.1 State-of-the-art MSA: MAFFT

This project will instead make use of a state-of-the-art progressive alignment tool. Multiple Alignment using Fast Fourier Transform (MAFFT) is a MSA tool that is known for its accuracy.[21] It mainly uses a progressive alignment paradigm and therefore follows the steps in Figure 4. The methods used in each step will not be explained in depth, but they result in an overall time complexity of $O(k^2n + kn^2)$. [22]

MAFFT operates under the assumption that the order and blocks of aligned sites are consistent across all sequences in the Multiple Sequence Alignment (MSA) This assumption can be violated by genomic rearrangements such as inversions, translocations, or duplications. This is why MAFFT is unsuitable for whole genome alignment. Mitogenomes on the other hand, are highly conserved and not suceptible to such genomic rearragements. For this reason, the linearized mitogenomes produced by vgOrient are well suited for MAFFT.

6.6.2 State-of-the-art MCSA: MARS

Like MAFFT MARS is also based on the progressive alignment paradigm. The steps are as follows:

1. use a heuristic Cyclic Edit Distance (hCED) algorithm on each pair of sequences. [23] hCED tries find the best rotation and edit distance of that rotation.
2. build a matrix that contains the edit distances for each pair
3. use a neighbour-joining algorithm to cluster the matrix and build a guide-tree
4. apply Needleman-Wunch on the rotated inputs according to the guide-tree (see 1.3.3)

The heuristic Cyclic Edit Distance algorithm is based on the Circular Sequence Comparison (CSC) problem. [24] The problem definitions are as follows:
A *q-gram* is any substring of length q over an alphabet Σ. The collection of all such q-grams is denoted by Σ^q. The q-gram profile of a string x of length m is represented by the vector $G_q(x)$, where $q > 0$, and $G_q(x)[v]$ indicates the total occurrences of q-gram $v \in \Sigma^q$ in x.
For strings x of length m and y of length $n \geq m$, and an integer $q > 0$, the q-gram distance $D_q(x, y)$ is defined as follows:

$$D_q(x, y) = \sum_{v \in \Sigma^q} |G_q(x)[v] - G_q(y)[v]| \tag{1}$$

Given an integer parameter $\beta \geq 1$, a generalization of the q-gram distance can be achieved by dividing x and y into β blocks as evenly as possible, and calculating the q-gram distance between each pair of blocks, one from x and one from y. This method aims to account for local similarities in the overall distance calculation. For strings x of length m and y of length $n \geq m$, and integers $\beta \geq 1$ and $q > 0$, the β-blockwise q-gram distance $D_{\beta,q}(x, y)$ is defined as:

$$\sum_{j=0}^{\beta-1} D_q \left(x \left[\left\lfloor \frac{jm}{\beta} \right\rfloor \dots \left\lfloor \frac{(j+1)m}{\beta} \right\rfloor - 1 \right], y \left[\left\lfloor \frac{jn}{\beta} \right\rfloor \dots \left\lfloor \frac{(j+1)n}{\beta} \right\rfloor - 1 \right] \right) \tag{2}$$

By using the definitions above the hCED algorithm can be described. hCED works by partitioning each sequence into β partitions and counting the occurrence of each q-gram within each of the β partitions. The goal of the algorithm is to find a rotation that minimizes $D_{\beta,q}(x, y)$. It is not clear from [**?**] exactly how the search for the optimal rotation happens.

6.6.3 Scoring Alignments: Wentropy

In bioinformatics, entropy measures such as Shannon's entropy are essential for analyzing MSAs, highlighting the uncertainty in sequences. Traditional entropy metrics, however, do not fully account for the evolutionary importance of different sequences in an alignment. To overcome this, "Wentropy," which incorporates sequence weights into Shannon's entropy, has been introduced. [19]
Wentropy emphasizes sequences critical for understanding evolutionary and functional diversity and downplays redundant sequences. It is calculated using a weighted version of Shannon's entropy formula, adjusting the probabilities of symbols (nucleotides or amino acids) at each alignment position according to sequence weights. These weights reflect each sequence's unique contribution to the alignment's information.
The Wentropy for each position in the alignment formula is given by:

$$W(t) = \lambda_t \sum_{a=1}^{K} p_a \log_2(p_a) \tag{3}$$

where:

- λ_t is a scaling factor for normalization, defined as $\frac{1}{\log_2(\min(N,K))}$,

- p_a denotes the weighted probability of observing symbol a,
- K represents the number of different symbols,
- N is the number of sequences.

Here, $W(t)$ ranges from 0 (complete conservation) to 1 (total randomness). This project will only make use of Wentropy averaged over all positions of the alignment to give a general indicator of MSA quality.

Variation Graphs Toolkit Variaton Graph toolkit (vg) is a series of tools to construct and manipulate genome variation graphs. [13] This project will use vg tools extensively, especially vg map. The most relevant tools will be presented here.

`vg construct` is a simple tool that can convert a number of input sequences into a single graph. When constructing a graph from a single sequence there would be no variation in the graph. The graph can then in theory be reduced to a single node by construct. This can cause issues with other tools within vg, so construct usually produces graphs where all nodes are below a maximum node size.

`vg circularize` simply circularizes a path in a graph. This means that an extra edge is added to the graph that connects the last node in the path with the first node. This is highly relevant for representing circular sequences such as mtDNA correctly.

`vg index` builds an on-disk index of the graph as well as any q-grams less than a given size. This index is used for alignment by `vg map`.

`vg prune` prunes the graph by removing complex regions. This is necessary for `vg map` to work.

`vg map` is a general-purpose alignment algorithm and one of vg's many mappers. [25] It aligns DNA sequences with a variation graph using a q-gram based seeding algorithm. Initially, the tool identifies the longest possible exact matches between a query DNA sequence and the graph. These matches are then clustered based on proximity, an essential step for paired reads to align correctly. The matches are ordered logically within each cluster to prepare them for alignment. In complex genomic regions, vg simplifies alignment by restructuring the genomic graph. Finally, it incorporates the quality of DNA data into the alignment process, enhancing the accuracy of genetic variant identification. This method efficiently scales with long sequences and complex genomic structures.

`vg augment` can take an alignment produced by `vg map` and a graph to produce an augmented graph that includes the alignment.

This is just a summary of what each tool does. vg is not very well documented and as so, details of the exact implementation of each tool are sparse.

6.7 Computational Complexity of MSA Methods

This appendix provides detailed insights into the computational complexities of traditional and progressive alignment methods used in multiple sequence alignment, specifically focusing on the Sum of Pairs scoring system and ClustalW's implementation.

Computational Complexity and Sum-of-Pairs Scoring Traditional dynamic programming approaches to MSA, such as using a sum-of-pairs scoring system, involve substantial computational complexity. In this system, each cell in an alignment matrix interacts with $O(2^k)$ possible neighboring cells, where k is the number of sequences. Calculating the sum-of-pairs score for each cell requires $O(k^2)$ operations, as it involves evaluating every possible pairing of sequences to determine the optimal score.
The sum-of-pairs method aggregates the scores for all pairings of aligned characters across the sequences, adjusting for mismatches and gaps. This scoring aims to optimize the total alignment score by either maximizing matches or minimizing mismatches and gaps.
However, the complexity of managing such an alignment matrix, which expands exponentially as $O(n^k)$ with n being the average sequence length and k the number of sequences, leads to a computational demand of $O(n^k 2^k k^2)$. This super-exponential increase in complexity renders the method impractical for large datasets or a substantial number of sequences, typically beyond 8-10.

Computational Complexity of ClustalW ClustalW's implementation of progressive alignment involves several computationally intensive steps:

- **Distance Matrix Calculation**: Needleman-Wunsch is employed to score the sequence pairs. Each pairwise alignment costs $O(n^2)$, with the total number of pairwise alignments being $O(k^2)$, resulting in a runtime of $O(k^2n^2)$.
- **Clustering**: Neighbour-joining clustering, used to build the guide tree, runs in $O(k^3)$.
- **Iterative Construction of MSA**: Requires $O(k)$ alignments, with each profile-profile alignment costing up to $O(kn + n^2)$, cumulatively $O(k^2n + kn^2)$.
- **Overall Complexity**: Aggregates to $O(k^3+k^2n^2)$, demonstrating how progressive algorithms better handle larger datasets compared to direct dynamic programming approaches like the sum-of-pairs scoring.

Understanding the details of these methods provides a deeper insight into why progressive alignment algorithms are favored for large-scale genomic datasets, managing the complexity of aligning multiple sequences more effectively than traditional methods.