Bringing Pangenomes to Proteomes: Toolkits for Panproteome Construction, Graph Alignments, and Epistasis Detection

Inaugural dissertation

for the attainment of the title of doctor
in the Faculty of Mathematics and Natural Sciences
at the Heinrich Heine University Düsseldorf

presented by

Fawaz Dabbaghie
from Aleppo, Syria

Düsseldorf, February, 2025

From the Institute for Medical Biometry and Bioinformatics
of Heinrich Heine University Düsseldorf

Published by permission of the
Faculty of Mathematics and Natural Sciences of
Heinrich Heine University Düsseldorf

Supervisor: Prof. Dr. Tobias Marschall
Co-supervisor: Prof. Dr. Olga V. Kalinina

Date of the oral defence: 23.06.2025

Statement

I declare under oath that I have produced my thesis independently and without any undue assistance by third parties under consideration of the “Principles for the Safeguarding of Good Scientific Practice at Heinrich Heine University Düsseldorf”.

Düsseldorf, February 27, 2025

Fawaz Dabbaghie

Abstract

Compared to biology, the field of computer science is much younger. However, it has been clear from the beginning that the combination of the two disciplines has had a significant impact. In particular, the use of data structures such as graphs and sequence algorithms has made it possible to analyze large amounts of genome sequencing data. Since the production of the first human reference genome in the early 2000s, many studies have shown the need for a better, more comprehensive representation of a reference genome than the linear reference. This led to the concept of pangenomes, and in particular, graph pangenomes. A graph pangenome is a data structure that is able to represent many linear references and sequences of a given species simultaneously, and is specifically designed to address the challenges and biases that arise when using a linear reference. However, with this advancement came the need to adapt and develop many algorithms and software toolkits to facilitate similar and new analyses that are regularly performed using linear references. In this thesis, we present several software tools that perform different analysis on large genome sequencing data. These tools employ a variety of statistical and algorithmic concepts, with a particular emphasis on bringing pangenomes to proteomes, genome graph manipulation tools, and sequence-to-graph alignment and processing.

The first chapter puts forth the concept of a panproteome in prokaryotes, a pangenome for the protein world. A panproteome, here, is represented as a collection of graphs representing proteins or coding regions. Moreover, we introduce `PanPA`, a software designed for the construction, indexing, and aligning of panproteomes. We assess the efficacy of our software by conducting experiments and providing benchmarks in diverse scenarios, employing multiple real-world datasets. We also show that `PanPA` and panproteomes are useful, especially in capturing sequence alignments that would otherwise be lost in the linear or DNA pangenome world, which further emphasizes the value of moving to the protein world.

The second chapter presents several toolkits pertinent to working and analyzing graphs and pangenomes. We first present `GFASubgraphs`, a simple tool and API for working with genome graphs, aiding users in further downstream analysis of their graphs. Second, we introduce `extgfa`, that employs a similar graph API from the aforementioned tool in this chapter. However, it further explores the concept of external memory representations of graphs, facilitating the analysis of large genome graphs on smaller machines with limited RAM. Third, `gaftools` is introduced, a joint work that introduces crucial functionality to working with genome alignments in the GAF format. It bridges a gap in the alignment pro-

cessing ecosystem, where it implements functionalities that were previously only available in the linear alignment world.

The third chapter describes an ongoing work in analyzing cancer and match normal cell lines using several sequencing platforms. In this work, we try to assemble both cell lines to a high quality, call structural variants using several algorithms and methods, and produce a high confidence set of somatic structural variants. Moreover, we investigate further the use of graphs in aiding in the previous steps, especially to disentangle and differentiate the variants and contigs that represent one cancer subclone from the other. To this end, we develop a graph drawing toolkit called `graphdraw` that assists in visualizing graph components, and extracting various important information from the assembly graphs, which enables us to investigate parts of the graph associated with certain information more efficiently.

The last chapter presents a joint work on a novel statistical method and software for epistasis detection between mutations in proteins is presented, we call our software `EpiPAMPAS`. Subsequently, `EpiPAMPAS` was tested on both simulated and real data, where the results on the simulated data were very promising, and we were able to efficiently detect the epistatic interactions. For the real data, we compared our results to a previously published method and found significant overlap in the epistatic positions detected. Furthermore, we looked into the location of the positions detected in the 3D structure of the corresponding proteins, and investigated the biological significance of some of these positions.

Kurzfassung

Im Vergleich zur Biologie ist die Informatik ein deutlich jüngeres Fachgebiet. Dennoch wurde früh erkannt, dass die Kombination beider Disziplinen große Auswirkungen hat. Besonders der Einsatz von Datenstrukturen wie Graphen und Sequenzalgorithmen hat die Analyse großer Mengen an Genomsequenzierungsdaten ermöglicht. Seit der Veröffentlichung des ersten menschlichen Referenzgenoms in den frühen 2000er Jahren haben zahlreiche Studien gezeigt, dass ein umfassenderes und präziseres Referenzgenom anstelle des herkömmlichen linearen Modells notwendig ist. Dies führte zum Konzept des Pangenome, insbesondere des Graph-Pangenome. Ein Graph-Pangenom ist eine Datenstruktur, die mehrere lineare Referenzen und Sequenzen einer bestimmten Spezies gleichzeitig darstellen kann. Die Entwicklung von Graph-Pangenomen zielte darauf ab, die Probleme zu beheben, die bei der Nutzung linearer Referenzen auftreten. Allerdings führte dies auch zur Notwendigkeit, eine Vielzahl von Algorithmen und Software-Toolkits anzupassen oder neu zu entwickeln, um sowohl bestehende als auch neue Analysen zu ermöglichen, die üblicherweise mit linearen Referenzen durchgeführt werden. In der vorliegenden Arbeit werden mehrere Software-Tools vorgestellt, die verschiedene Analysen großer Genomsequenzierungsdaten ermöglichen. Die Tools basieren auf einer Vielzahl statistischer und algorithmischer Konzepte, wobei ein besonderer Schwerpunkt auf der Verknüpfung von Pangenomen mit Proteomen, der Bereitstellung von Werkzeugen zur Manipulation von Genomgraphen sowie der Entwicklung von Methoden zur Sequenz-zu-Graph Alignierung und -Verarbeitung liegt.

Im ersten Kapitel wird das Konzept eines Panproteoms bei Prokaryoten erörtert, ein Pangenom für die Proteinwelt. Ein Panproteom wird hierbei als eine Sammlung von Graphen dargestellt, die Proteine oder kodierende Regionen repräsentieren.Darüber hinaus wird `PanPA` vorgestellt, eine Software, die für den Aufbau, die Indizierung und das Alignment von Panproteomen entwickelt wurde. Die Leistungsfähigkeit der Software wird durch die Durchführung von Experimenten und die Bereitstellung von Benchmarks in verschiedenen Szenarien bewertet, wobei mehrere reale Datensätze verwendet werden.Es wird gezeigt, dass `PanPA` und Panproteome insbesondere bei der Erfassung von Sequenzalignments nützlich sind, die sonst in der Welt der linearen oder DNA-Pangenome verloren gehen würden. Dies unterstreicht den Wert des Wechsels in die Proteinwelt.

Im zweiten Kapitel werden mehrere Toolkits vorgestellt, die für die Arbeit und Analyse von Graphen und Pangenomen von Relevanz sind. Zunächst wird`GFASubgraphs` präsentiert,

ein Tool und eine API für die Arbeit mit Genomgraphen, die den Benutzern bei der weiteren Analyse ihrer Graphen behilflich ist. Des Weiteren wird `extgfa` vorgestellt, das eine Graphen-API wie das zuvor beschriebene Tool in diesem Kapitel verwendet.`GFASubgraphs` erforscht das Konzept der Darstellung von Graphen in einem externen Speicher, um die Analyse großer Genomgraphen auf kleineren Rechnern mit begrenztem RAM zu erleichtern. `gaftools` ist eine Gemeinschaftsarbeit, die wichtige Funktionen für die Arbeit mit Genom-Alignments im GAF-Format einführt. Es schließt eine Lücke im Ökosystem der Alignment-Verarbeitung, indem es Funktionalitäten implementiert, die bisher nur in der Welt des linearen Alignments verfügbar waren.

Im dritten Kapitel wird eine laufende Arbeit zur Analyse von Krebs- und normalen Zelllinien unter Verwendung verschiedener Sequenzierungsplattformen beschrieben. Ziel dieser Arbeit ist die Auswertung und Assemblierung beider Zelllinien in hoher Qualität, die Bestimmung struktureller Varianten mit verschiedenen Algorithmen und Methoden sowie die Erstellung eines Satzes somatischer Strukturvarianten mit hohem Vertrauen. Darüber hinaus wird die Verwendung von Graphen zur Unterstützung der vorangegangenen Schritte, insbesondere zur Entflechtung und Unterscheidung der Varianten und Contigs, die einen Krebs-Subklon von dem anderen unterscheiden, untersucht. Zu diesem Zweck entwickeln wir ein Toolkit zum Zeichnen von Graphen mit dem Namen `graphdraw`, das bei der Visualisierung von Graphenkomponenten und der Extraktion verschiedener wichtiger Informationen aus den Assemblierunggraphen hilft, wodurch wir Teile des Graphen, die mit bestimmten Informationen verbunden sind, effizienter untersuchen können.

Im letzten Kapitel wird eine neue statistische Methode und Software zur Erkennung von Epistasen zwischen Mutationen in Proteinen vorgestellt, die wir `EpiPAMPAS` nennen.Anschließend wurde `EpiPAMPAS` sowohl an simulierten als auch an realen Daten getestet, wobei die Ergebnisse bei den simulierten Daten sehr vielversprechend waren und wir in der Lage waren, die epistatischen Interaktionen effizient zu erkennen. Bei den realen Daten wurde ein Vergleich mit einer zuvor veröffentlichten Methode vorgenommen, wobei eine signifikante Überschneidung der erkannten epistatischen Positionen festgestellt wurde. Zudem wurde die Lage der entdeckten Positionen in der 3D-Struktur der entsprechenden Proteine untersucht und die biologische Bedeutung einiger dieser Positionen analysiert.

Acknowledgments

I would like to express my deepest gratitude to my supervisors, Prof. Dr. Tobias Marschall and Prof. Dr. Olga Kalinina, for giving me the opportunity to pursue my PhD and for their support, guidance, and mentorship throughout this journey. Their invaluable insights and encouragement have been instrumental in shaping my research and me as a researcher and as a person.

I am also grateful to my colleagues (Hufsah, Hugo, Kai, Mir, Peter, Haniyeh, Sven, and Samarendra) for their support, collaboration, and for making our work atmosphere pleasant, fun, and encouraging. Special thanks to Konstantinn Bonnet for the countless fruitful discussions and encouragements that enriched my work, to Jana Ebler for always being there to answer questions, to Rebecca Serra Mari for her support and sharing the best memes, and to my close friends Sandra, Dania, John, and Kenta for their emotional support.

I would also like to especially thank Rui, Anela, Maartin, Saheli, and Andy for their last minute proofreading of the manuscript.

I cannot express in words my deepest appreciation and gratitude to my parents, whose encouragement and sacrifices have been a constant source of strength throughout my life, and to my brother for his unwavering support.

This journey has been long and sometimes tiring, but I am so grateful for the experience. Since I started my first year at university in 2009, I have known that science and research are my true passion. I am excited to keep learning, sharing what I know, and helping others the way so many people have helped me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

The rapid advances in genome sequencing technologies have undoubtedly revolutionized our understanding of genomes, genomic diversity, genetics, evolution, and other aspects of living organisms [236]. Since the early stages of genome sequencing, it has been clear that computer science and algorithms are needed to process such data [254]. Moreover, with the increased production of data and development of next- and third-generation sequencing, the need for efficient algorithms became even more apparent. The useful algorithms and data structures of computer science allow scientists to disentangle the large amount of sequencing data produced; one of the famous data structures that is widely used is a graph data structure. Graphs and graph theory have been studied in discrete mathematics since Leonhard Euler's paper on the Seven Bridges of Königsberg problem in 1736 [19]. Here, Euler was able to describe the problem more abstractly by representing the parts of the city as the "vertices" of a graph, and the "edges" connecting these vertices as the bridges. Using this representation, Euler was able to prove that the problem of finding a route that crosses each bridge once and only once is impossible. This led to the definition of a graph as a collection of elements called nodes or vertices, and the connections between them called edges. Graphs allow us to represent problems more visually, and help us understand the relationships between different elements. Fast forward to the 20th century, with the birth of computers and computer science, there was a need for a versatile data structure that can represent data and its relationships. To this end, the development of the graph data structures in computer science came about. Since its development in mathematics, graph data structures has been applied to a variety of problems in and outside the field of computer science [219, 62].

Conventional linear reference genomes, which have been widely employed to interpret novel sequences through alignments and to comprehend newly sequenced samples, have shown clear biases in representing the complete spectrum of genomic variation across populations [106, 47, 35]. This limitation has prompted the development of graph pangenomes, which have emerged as a means to circumvent the limitations and biases of linear reference genomes [74]. Pangenomes have the capacity to encode multiple genomes in a graph struc-

ture, allowing for the simultaneous representation of multiple assemblies and their variants. This, in turn, can be used as a more comprehensive reference. However, with this shift from a linear reference to a graph-based reference, came the need to develop new algorithms and software toolkits to perform similar analyses to the ones developed under the linear reference framework; for example, sequence mapping [216, 154] or variant calling [88, 72].

This thesis presents several software toolkits for pangenomes, panproteomes, sequence-to-graph alignment, genome graph manipulation, and epistasis detection and analysis. A detailed outline of the presented chapters can be found in Section 1.5.

1.1 Genomes

A genome is defined as the complete set of genetic material within an organism, encompassing all of its DNA. This genetic material carries the instructions necessary for the growth, development, and functioning of an organism. The concept of the genome was first theorized in the early 20th century, yet it was not until the discovery of the double-helical structure of DNA by James Watson and Francis Crick in 1953 that the molecular underpinnings of heredity became evident [281]. It is also important to acknowledge the seminal contributions of Rosalind Franklin, whose pioneering research on X-ray diffraction provided crucial insights into the structure of DNA [43]. This breakthrough laid the foundation for understanding how genetic information is stored and transmitted. Deoxyribonucleic acid (DNA) is a molecule composed of two complementary strands which are twisted into a double helix. Each strand consists of a sequence of four chemical bases: adenine (A), thymine (T), cytosine (C), and guanine (G). These bases form specific pairs – A with T, and C with G – creating a code that specifies the genetic instructions for building and maintaining an organism. A remarkable property of DNA is its capacity for autonomous replication, enabling it to be transmitted to later generations, thereby ensuring the persistence of genetic information through time.

Eukaryote genomes are usually organized into chromosomes and housed in the nucleus. Moreover, the chromosomes come in sets, called *ploidy*, which refers to the number of complete sets of chromosomes in the cell. Most animals are diploid (e.g., Humans), meaning they have two sets of chromosomes, one inherited from each parent. However, ploidy can vary widely; some organisms are haploid (one set), polyploid (multiple sets, common in plants), or aneuploid (an abnormal number of chromosomes). Ploidy levels influence genetic diversity, evolutionary adaptability, and reproductive strategies, playing a critical role in development, evolution, and species survival. In contrast, the DNA of prokaryotes is circular and located in the cytoplasm within a nucleoid, and lacks histones. Prokaryotic genomes are significantly smaller and consist mostly of coding DNA, with genes often grouped into operons that are transcribed together [29]. However, prokaryote genomes are more diverse within and between species and genera, due to many environmental and genetic reasons,

such as their rapid reproduction, horizontal gene transfer, the adaptation pressure in the environment they live in, higher mutation rates [66, 45].

Genomes contain genes, or coding regions, which are specific sequences in DNA that code for functional products, particularly proteins. Proteins are essential to living organisms because they perform a variety of critical functions, such as catalyzing biochemical reactions as enzymes, facilitating communication between cells, and certain proteins provide structural support to the cell. According to the central dogma of biology, the flow of genetic information is unidirectional. First, the coding regions in the genome are transcribed into messenger RNA (mRNA), then the mRNA is translated into amino acid sequences, these amino acids are linked together to form a polypeptide, which is folded into a functional protein.

Different individuals of the same organism do not share exactly the same genome, but differ slightly, and these difference are driven by various mechanisms of mutation and recombination. These differences or variations are usually called “genetic variants” and encompass a wide spectrum, including single nucleotide polymorphisms (SNPs) insertions, deletions, inversions, duplications, copy-number variations, and translocations. When variants are larger in size, typically, over 50 base pairs, they are then categorized as structural variations. These variants can have profound effects, particularly when impacting coding regions and the resulting amino acid sequence, which can alter the structural and functional stability of the protein. For instance, a single change in the coding region can result in a single amino acid change, called a missense mutation, which can cause premature termination of protein translation. Consequently, these variants can lead to phenotypic effects such as susceptibility to disease, resistance to pathogens, or adaptation to environmental conditions. For example, in humans, a SNP in the CHF gene has been shown to be associated with increased risk of molecular degeneration [30]; and in prokaryotes, some SNPs are associated to microbial antibiotic resistance [235, 80]. Copy number variations, for example, have been associated with Parkinson’s disease [269]. Hunter’s syndrome, a rare disease caused by deficiency of the lysosomal enzyme iduronate-2-sulfatase (I2S), is caused by an inversion of the IDS gene [24]. More complex genomic rearrangement have been associated with the development of cancer cells [256, 15, 147].

1.2 Genome Sequencing

Genome sequencing can be defined as a procedure or a method to determine the order of the nucleotides in the genome of an organism. The various procedures that have been developed thus far are incapable of determining the complete order of the genome sequence at once; rather, they can only determine smaller parts of the genome, called “reads”, the size and accuracy of which vary depending on the technology used. One of the first methods developed, Sanger sequencing, was a pioneering technology developed back in

1977 [230]. It offered accuracy, however, was costly and had low throughput. Since then, sequencing technologies have evolved substantially, particularly with the emergence of Next-Generation Sequencing (NGS) in the mid-2000s, which revolutionized genomics by enabling high-throughput, cost-effective sequencing [236]. Key NGS platforms include Illumina (sequencing-by-synthesis) [129], Roche 454 (pyrosequencing) [225], Ion Torrent (pH-based detection) [226], and SOLiD (Sequencing by Oligonucleotide Ligation and Detection) [231]. These methods excel in throughput and low cost but are limited by short read lengths and amplification-induced biases[204].

Third-generation sequencing technologies, such as PacBio SMRT sequencing [192] and Oxford Nanopore Long Read sequencing [278], have been developed to address these limitations by enabling single-molecule, real-time sequencing with ultra-long reads. While these platforms reduce amplification biases and simplify complex genome assembly, they tend to exhibit higher error rates and require specialized data analysis [126].

The Circular Consensus Sequencing (CCS) protocol, a feature of PacBio's Single Molecule, Real-Time (SMRT) sequencing technology, was presented in 2019 to solve the problem with the error-prone long reads [284]. This protocol allows the generation of long high-fidelity (HiFi) reads with a median predicted accuracy of 99.9%, a mean of 99.8%, and an average length of 13.5 kilobases [284, 261]. The generation of CCS reads involves the repeated sequencing of a circularized DNA molecule, resulting in multiple passes (subreads) that are then combined to produce a highly accurate consensus sequence [195]. The development of CCS reads represents substantial advancement in the field as it combines the benefits of long-read sequencing and the accuracy of short reads. This combination of features positions CCS as a highly versatile tool for a wide range of applications, including *de novo* genome assembly [37], structural variant detection [284], pangenome analysis [103], and the study of complex regions of the genome that are challenging for short-read technologies [122].

1.2.1 Sequence Alignment

Sequence alignment algorithms are a primary application of algorithms developed to understand the sequencing data produced. Here, algorithms are used to compare DNA, RNA, and protein sequences, with the goal of identifying the similarities and differences between sequences. One of the major application of sequence alignment is read mapping to references, which is usually a precursor for many genomic analysis methods and pipelines [238]. Moreover, sequence alignment and mapping is important in facilitating comprehension of evolutionary relationships, the identification of conserved genes across species, and detecting the location of mutations associated with diseases [5]. Furthermore, the ability to compare the differences between sequences is crucial in building phylogenetic trees and understand evolution better [11].

We can categorize sequence alignment algorithms into two main groups: *Pairwise Alignments* and *Multiple Sequence Alignments*. Pairwise alignments involve the alignment of two sequences against each other, whereas Multiple Sequence Alignments (MSAs) perform the

alignment of three or more sequences simultaneously.

1.2.1.1 Pairwise Sequence Alignments

Pairwise alignment algorithms include both *local* and *global* alignments. The former is capable of identifying regions of similarity within the sequences being aligned, i.e., in a subsequence of the sequence. One of the best known local alignment algorithms is the Smith-Waterman algorithm [251]. In contrast, global alignment algorithms attempt to align the entire length of the sequences. A classic example of global alignment is the Needleman-Wunsch algorithm, which was introduced in 1970 [183]. Pairwise alignment methods can be further divided into the following three categories:

(1) **Dot matrix or "the diagram"** method as it was called in the original paper [93], is a method that allows for a comprehensive visualization of the alignment landscape between two sequences. In this method, we have a matrix of dimensions $n \times m$, where n and m represent the length of the two aligned sequences. Furthermore, the nucleotides of one sequence are associated with the rows of the matrix, while the nucleotides of the other are associated with the columns. Then, for each cell in the matrix, if the characters corresponding to the row and column of that cell match, a dot is drawn; otherwise, the cell remains empty. The dots that connect to form a diagonal correspond to areas of similarity between the two sequences. In addition, the presence of distinct diagonals (e.g., perpendicular or X-shaped) can serve as indicators of specific features associated with the degree of similarity between the two sequences, including inversions, repeats, palindromes, frame shifts, and other features. However, due to the limited size of the sequence alphabet, drawing a dot for every character match across the two sequences can result in a noisy figure, especially for longer sequences. This can be mitigated by implementing a sliding window filter, where a window of size n consecutive characters is used, and a dot is only placed if a threshold σ is met, where the threshold represents the number of matches in the window. An example of this method is shown in Figure 1.1.

(2) **Dynamic programming (DP)** alignment methods uses dynamic programming, which is a known algorithmic technique for solving optimization problems in computer science. This technique breaks a bigger problem into smaller and simpler problems to solve. In this context, a dynamic programming matrix is used to find the optimal alignment between two sequences. This method is optimal as it explores all possible alignments between the two sequences before reporting the best one, moreover, DP algorithms have been shown to be mathematically optimal [94]. Famous aforementioned algorithms like Smith-Waterman or Needleman-Wunsch are based on dynamic programming. Briefly, this algorithm starts by constructing a dynamic programming matrix of dimensions $(n+1) \times (m+1)$, where n and m correspond to the lengths of the two sequences being aligned. Similar to the dot matrix method, one sequence corresponds to rows and the other to columns, and the extra row and column are used for initialization, where this initialization differs depending on the

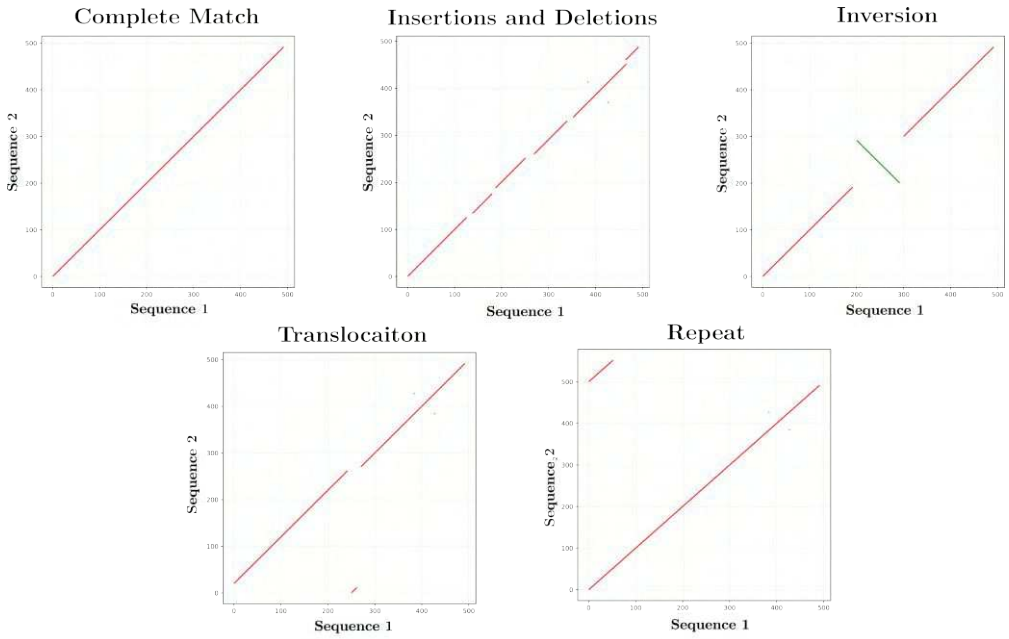


Figure 1.1: Examples of dot matrix alignments between two sequences, five examples are shown that represent different characteristics of the alignment that correspond to structural changes or variants between the sequences. Generated with `https://en.vectorbuilder.com/tool/sequence-dot-plot.html`

algorithm used. The cells of the DP matrix are then subsequently filled one by one using a recurrence relationship to the scores of neighboring cells, depending on the algorithm used. The objective of the DP table then is to systematically compute the optimal alignment of two sequences, by breaking the alignment into smaller problems (substrings), a cell (i, j) in the table then, represent the alignment score or cost for up to position $(i-1)$ in one sequence and $(j-1)$ in the other sequence. The value of a cell represents the score—or cost—of an alignment that ends at the corresponding characters to that cell. Moreover, there are two ways to generate the scores and evaluate the results in the DP matrix, which are either by looking at the highest score (*score scheme*), or lowest cost (*cost scheme*). For example, Equation 1.1 shows the recurrence relation for calculating the scores of the cells with a constant gap penalty, where the score for a cell at position (i, j) depends on the scores of adjacent cells. In this equation, Δ is the gap penalty, and function $sub(c_1, c_2)$ takes two characters and returns a value that depends on whether the two characters match or mismatch. Furthermore, the substitution value can be based on binary value (match or mismatch), e.g., edit or Levenshtein distance [146], or something more complex like using a substitution matrix with some biological basis such as BLOSUM [116].

$$Score(i,j) = \max \begin{cases} Score(i-1, j-1) + sub(seq[i], seq[j]) \\ Score(i-1, j) + \Delta \\ Score(i, j-1) + \Delta \end{cases} \tag{1.1}$$

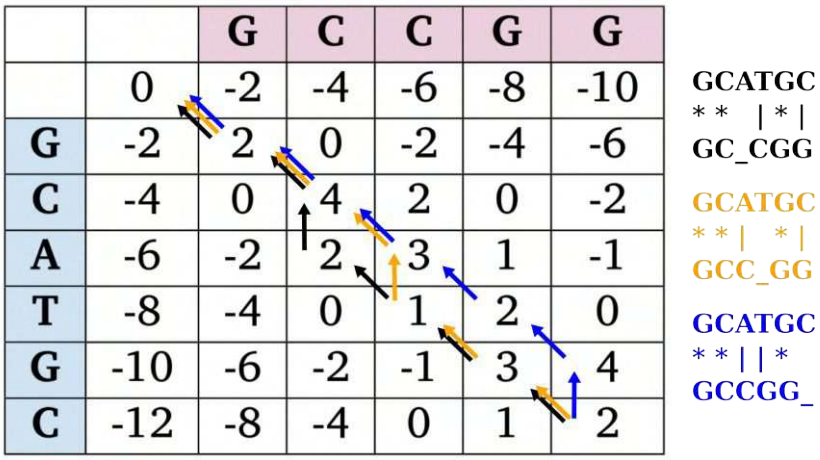


Figure 1.2: Example of Needleman-Wunsch global alignment algorithm matrix with a match score of 2, mismatch score of -1, and a gap penalty of -2. The gap penalty here is constant, i.e., opening and extending the gap have the same score. In this algorithm, the first row and column are initialized with the gap penalty. The colored arrows correspond to the different tracebacks available from the last cell, this occurs when there is more than one potential maximum score following Equation1.1, i.e., following one or the other path both results in an alignment with the same score. On the right side, we see the three potential alignments between the two sequences that have the same score, the colors here match the color of the traceback. In this alignment representation, the "*" represents a match, the "|" represents a mismatch, and the "_" represent a gap.

To find the optimal alignment after the matrix has been filled, we need to find the highest scoring or lowest costing cell, depending on the scheme used to calculate the values. Once we locate this cell, we need to follow back the path that led to the value for that cell, or what is called a *traceback*. An example of the Needleman-Wunsch global alignment algorithm is shown in Figure 1.2, with scores calculated using Equation 1.1.
Since the development of the first DP-based algorithms for sequence alignment back in the 1970s, this algorithm has been widely used and is still further improved. For example, using different gap penalties to resemble more biological mechanisms such as using affine gap penalty [99], or logarithmic gap penalty [33]. Furthermore, optimizations were done on the alignment algorithm and the calculations of the scores in the DP matrix, such as banded alignment [270], bit-vector algorithm [178], single instruction multiple data (SIMD) alignment speedups [60, 78], and more recently, the Wavefront algorithm [164].

(3) **Word or k-tuple** methods, which are heuristic methods that do not guarantee to find the optimal alignment. However, they are more efficient compared to the DP-based methods. They are usually used to find alignments against a large database of sequences, where a large proportion of the sequences in the database will not match the query sequence. One of the most famous algorithms and tools using this method is the Basic Local Alignment Search Tool (BLAST) [6], which has revolutionized genomics by providing a fast and accurate way to compare sequences against large databases, aiding in gene annotation and the discovery of novel genetic elements. This method works by cutting the sequence into substrings or

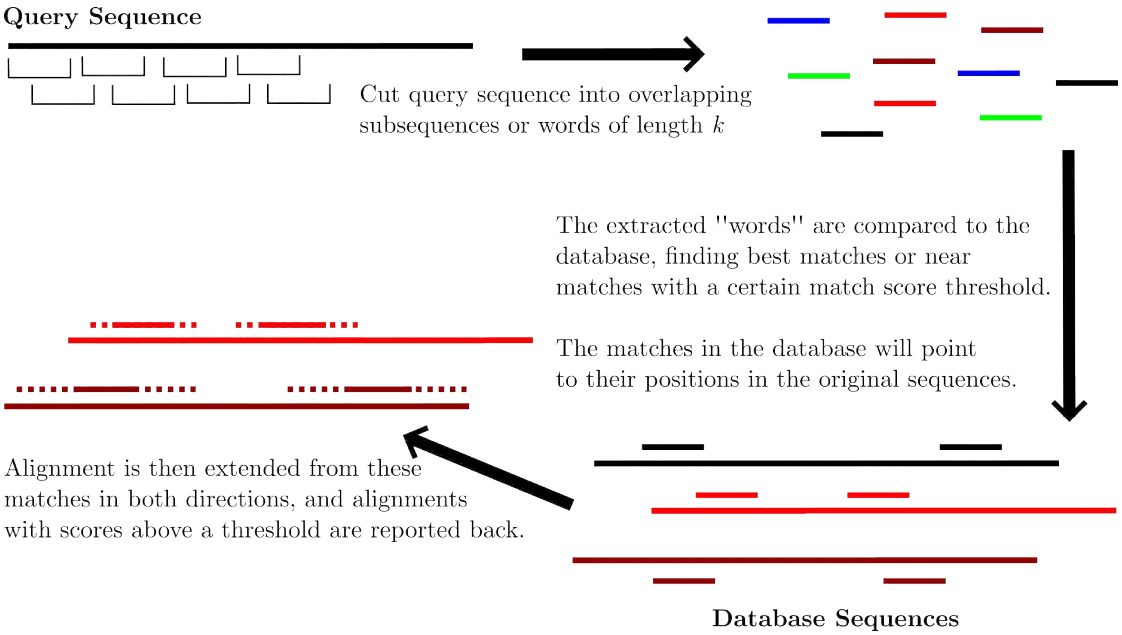


Figure 1.3: Simple schematic showing how database aligners such as BLAST work. First, the sequences are cut into equally sized "words" of length k, then these words are matched to words in the database to find exact or near exact matches. Once the matches are found and their locations in the squences in the database, the word match is then extended to both directions on the target sequence and alignments above a certain threshold are reported back.

words of length k, and then comparing these words against the database to find matches or near matches. Once matches are found, the algorithm finds these words' location on the sequences in the database. From here, the algorithm can perform a DP-based alignment against these potential sequences in the database that have matching words, or it can extend the initial word found and join the various matches found on the sequence. For example, BLAST uses what is known as a seed-and-extend algorithm, which extends the comparison between the two sequences beyond the word matches to the left and right without allowing insertions or deletions. BLAST then uses a modified version of a Smith-Waterman algorithm on the top candidates to account for insertions and deletions. Figure 1.3 shows a rough outline of the steps BLAST takes to return alignments to databases.

1.2.1.2 Multiple Sequence Alignments

MSA algorithms can align three sequences or more simultaneously, which facilitates capturing certain aspects such as conserved regions, functional domains, motifs, and phylogenetic relationships across a family of sequences that might not be otherwise detectable with pairwise alignments. Computationally, producing MSAs is a much more complex and computationally-heavy procedure compared to pairwise alignment [293], and producing an optimal solution has been shown to be an NP-complete problem [277]. However, obtaining high quality MSAs is crucial for many applications, especially ones related to de-

tecting secondary and tertiary protein structures. Most notably, AlphaFold2 that was developed in recent years, which revolutionized the protein 3D prediction field. It relies on high quality MSAs, as they are essential for it to produce an accurate prediction of protein structures [135]. Due to the complexity of the problem and the need for high-quality alignments, many algorithms and heuristics have been developed. The following briefly summarizes some of the common methods used to perform multiple sequence alignments:

(1) **Dynamic Programming** methods, sometimes called exact methods, where instead of building a 2-dimensional table, similar to pairwise alignment, one needs to build an n-dimensional table, for n sequences and find the optimal traceback. This is, of course, impractical. However, some practical exact methods have been produced such as the on in [159] where they greatly reduce the computational demands of dynamic programming. More recently, an exact solution in polynomial time was suggested by [125].

(2) **Progressive** methods are among some of the most widely used heuristic methods, they are more efficient for aligning thousands of sequences, however, do not guarantee a global optimum alignment. They were first introduced back in 1987 [79]. The vast majority of progressive aligners use a dynamic programming approach internally. The first step is usually to construct a tree called a *guide tree*, which represents the distance relationship between the sequences. This is followed by performing pairwise alignments starting from the most similar sequences to start the MSA, then extending the MSA one sequence at a time following the guide tree. Examples of software tools based on this method are ClustlW [268], MAFFT [137], and T-coffee [187].

(3) **Iterative** methods are similar to progressive methods. However, they attempt to improve the quality of the alignments by repeatedly aligning the original sequences before adding new sequences to the growing MSA, which refines the final alignments. Examples of software tools based on this method MUSCLE [73] and PRRN, which is based on the algorithm introduced here [100].

(4) **Consensus** methods try to find a better, or consensus alignments for the sequences by combining several MSAs produced by other methods. Tools such as MergeAlign [46] and M-Coffee [275] are an example of consensus alignments method.

(5) **Probabilistic** methods that use statistical models to infer the alignments. One of the best known tools is HMMER [81, 266], which uses a Hidden Markov Model (HMM). It first builds a probabilistic model (usually called a profile HMM) of the most common sequences, finding the likelihood of amino acids, insertions, and deletions at each position. Using this HMM, HMMER can then align new sequences. HMMER is also very useful and efficient for aligning distant homologous sequences.

Figure 1.4 shows an example of an MSA of the ATP Synthase alpha/beta protein family, this MSA was taken from the Pfam database [172]. The figure shows how an MSA can visualize and emphasize conserved regions between the different sequences.

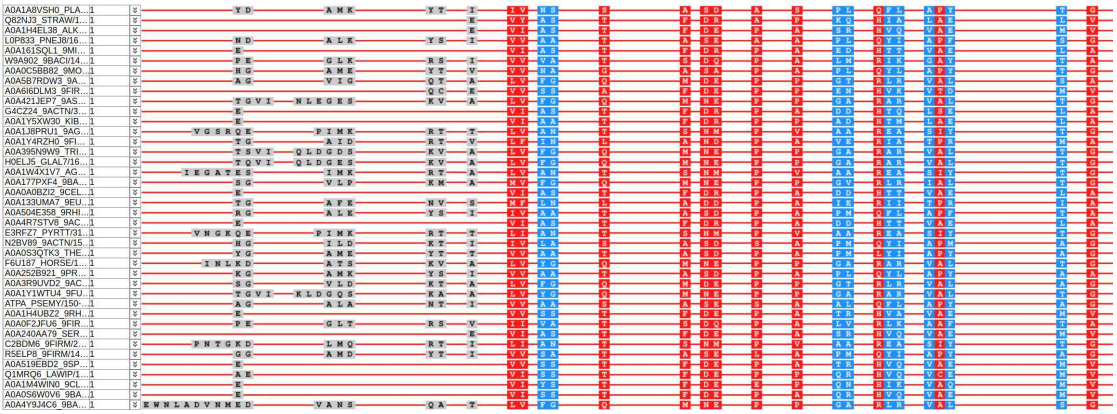


Figure 1.4: This figure shows a subset of the ATP Synthase alpha/beta protein family multiple sequence alignment. This was retreived from the Pfam database with the accession number PF00006. The different colors of certain positions in the MSA correspond to the relative conservation in that position in the alignment. This figure was created using the NCBI Multiple Sequence Alignment Viewer `https://www.ncbi.nlm.nih.gov/projects/msaviewer/`.

1.2.2 Genome Assembly

As stated in Section 1.2, sequencers are unable to read the complete genome at once; rather, they read overlapping fragments of the genome repeatedly. Consequently, *genome assembly* can be defined as the process of reconstructing the original genome sequence from a collection of reads, while taking into account various hurdles such as the different orientations of the reads, errors introduced to the reads by the sequencers, the highly repetitive regions in the genomes that the reads do not cover completely, and the gaps in the coverage of the genome due to the random sampling of the sequencers [179]. A *de novo* genome assembly is the process of assembling a genome without using a reference to guide the assembly. This circumvents any biases that might be introduced by the reference genome. Where a genome reference is defined as the representative sequence of an organism, it is usually made of several samples of that organism, and is a mosaic of these samples. Therefore, it might not contain all the genes or important region of the genome of a sample from that organism. However, we map new sequences against it to find variations.

Since the production of Sanger sequencing, it has been recognized that with the more sequencing data produced, computers and algorithms were needed to solve the assembly problem [254]. Since then, several breakthroughs happened in terms of *de novo* genome assembly. Most notably, the first draft of the human genome that was published in 2001 by the Human Genome Project [272]. Over the years, the human genome reference has been continuously updated and improved. Nevertheless, approximately 8% of the human genome remained unassembled until recently, with the publication of the first complete human genome assembly by the Telomere-to-Telomere (T2T) Consortium [188]. Despite advances in sequencing technology and assembly algorithms, the problem is not yet completely solved and efforts are underway to produce better assemblies. For example, `Verkko` is

a pipeline that attempts to produce high-quality and gapless assemblies for individual samples [218], and `hifiasm` produces high-quality haplotype-resolved assemblies from long accurate reads. Algorithmically, we can roughly organize the assembly methods into the three following categories:

(1) **The greedy** method is considered the simplest and most naïve way to assemble a genome. It simply connects reads that overlap the best with each other to build a longer contiguous segment, usually called *contig*. It iterates this process until no more reads can be overlapped to continue building the contig [170]. It is called greedy because it only focuses on the local optimum, which leads to misassemblies, especially in repeat regions [209]. This method was used in the Sanger sequencing era, with software tools such as the `TIGR` assembler [258] and `SSAKE` [280].

(2) **Overlap-Layout-Consensus (OLC)** methods divide the assembly problem into three steps; these steps help to perform a more global alignment, avoiding the inherent locality of the greedy method [209]. The first step (overlap) is similar to the greedy methods, where an all-against-all read comparison or alignment is performed to find common overlaps. The overlaps do not have to match perfectly and errors are allowed. Usually, the error rate of the sequencing technology is taken into consideration. The second step (layout), uses the overlaps identified in the first step to construct a graph. In this graph, each node represents a read, and each edge between two nodes indicates that the reads have an overlap above a certain threshold. In theory, the best-case scenario is to find a path in the graph that traverses each node once. Concatenating the sequences along the path should spell out the complete genome from which these reads originated; such a path is called a Hamiltonian path and is computationally challenging to find, falling in the NP-hard category. In practice, however, algorithms and heuristics are used to find the longest possible contigs in the graph. The final step (consensus) involves generating a consensus sequence of the paths identified in the previous step, where a vote is taken among the overlapping regions of the reads to determine the sequence. Early automated assembly methods that followed a similar approach can be traced back to the 1980s [199]. Later, well-known assemblers such as `Celera` [177] and `Velvet` [118] were developed that use the same strategy. More recently, with advances in sequencing technologies, and the production of long, highly accurate reads, newer assemblers have returned to a similar strategy, such as Hifiasm [37], where its first steps include an all-against-all alignment and overlap detection to build a graph that is used for the assembly.

(3) **De Bruijn Graph (DBG)** methods use internally the so-called k-mer graphs, where instead of representing each read as a node, consecutive subsequences of length k are extracted from each read, where subsequence overlap by $k-1$ base-pairs. The nodes of the graph represent the k-mers and the edges represent the $k-1$ overlap. This construction is advantageous for several reasons. It avoids the need for an all-against-all comparison between the reads to find the overlaps, it scales better in terms of memory usage as the algorithm

only stores the unique k-mers instead of the complete reads, it is able to compress highly repetitive regions, and it helps to avoid sequencing errors, as error will cause low-frequency k-mers to appear that can be filtered out [48]. This idea was first presented for genome assembly in [128], based on an earlier proposal in [201]. Similar to the OLC approach, in theory, if the de Bruijn graph is generated from k-mers extracted from error-free reads that completely cover the genome, then there exists a Eulerian path in the graph, that would spell out the underlying genome. A Eulerian path is a path that traverses each edge exactly once and can be found in polynomial time. However, as pointed out in [167], in both of the previous problems the goal is not to find a perfect Hamiltonian or Eulerian path. However, in general, the assembly algorithms are more focused on finding the longest possible contig path, i.e., long segments in the graph that can be inferred to be part of the original genome without ambiguity.

Figure 1.5 illustrates how the three different assembly methods work. The Figure shows how the greedy method can results in shorter contigs due to misassemblies from repeat regions in the genomes, it also illustrates how both OLC and de Bruijn graph methods can handle errors in the reads. For the OLC method, this is done in the consensus step, and for de Bruijn graphs assemblies, this is done be eliminating low-frequency k-mers.

1.2.3 Variant Calling

As mentioned in Section 1.1, genome variants are very important, and understanding their mechanisms and effects is crucial in understanding diversity, evolution, and diseases. The advancement in sequencing technologies, genome assembly, and sequence mapping algorithms (which were discussed in Sections 1.2, 1.2.2, and 1.2.1), facilitated the development of methods to identify variants in organisms, which are usually termed "variant calling". Methods for calling variants are always under development to match the new advancement in sequencing technology and the large amount data produced. We will not give an extensive view on all the methods here, however, will mention some of techniques used.

Starting with molecular-based techniques, such as karyotyping, where the chromosomes are stained the bands are visualized to detect large structural variant events [291]. This is still used in cancer genomics to visualize large events effecting the genome [276]. DAN microarray methods, such as Comparative Genomic Hybridization (array CGH) [205] methods specialized in detecting copy-number variations, and SNP arrays methods [237] for detecting certain targeted SNPs in samples. Optical Genome Mapping (OGM) methods, which uses restriction enzymes to cut the DNA into fragments, then these fragments are assembled back into a consensus genomic optical map, and the location and order of the restriction enzymes can be then used to detect large-scale structural variation [240].

Sequencing-based techniques are very common and diverse, many bioinformatics tools and algorithms have been developed for variants calling using sequencing data. For example, tools such as `GATK` [166], `FreeBayes` [87], and `DeepVariant` [210] are commonly used

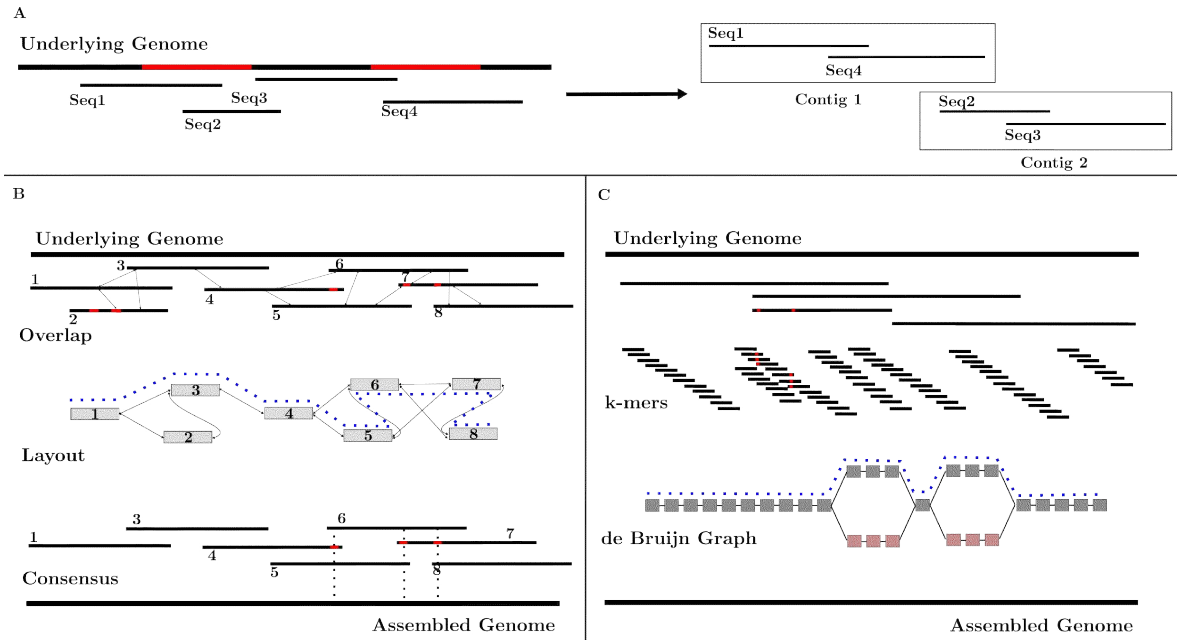


Figure 1.5: Sketch of the three different assembly methods: (A) Shows the greedy method, where the underlying genome has a repeat (in red), causing the greedy method to erroneously produce two contigs instead of one, where Seq1 and Seq4 are assembled first, preventing further extension of the contig. (B) Shows the OLC method, first, all the overlaps between the reads are identified, then a graph is built where each node represents a read with edges representing the overlaps, the dotted line represents the best path taken to build the contig, where for example, read 2 was excluded from the path as its sequence is already covered by reads 1 and 3. In the consensus step, a majority vote is taken over the overlapped region, eliminating the errors in the reads (marked in red). (C) Shows the de Bruijn graph method, where k-mers are extracted from the reads, and errors in the reads lead to erroneous k-mers, when the graph is built from the k-mers, the divergent paths emerge from the errors. The dotted line in the graph represents the path taken to build the contig, and the erroneous nodes (in red) are skipped as k-mers resulting from errors tend to have a low frequency.

for SNPs and small insertions and deletions (indels) detection from short and accurate reads. Other tools such as `Sniffles` [252], `PBSV` [193], and `Delly` [215] that use long-reads (such as the ones produced by PacBio or ONT) to detect structural variants. Tools such as `PAV` [70] and `SVIM-asm` [114] are able to call structural variants using assemblies.

1.3 Pangenomes

This section reuses some materials from the Introduction section of [57], of which I am the first author.

So far, we have talked about the importance of genomes, sequencing and assembling genomes, mapping sequencing reads to references, and calling variants. We have also defined that a

reference genome is a linear representative DNA sequence of an organism, it is built from the sequences of one or more samples of that organism, and that the reference provides a framework or a background against which other sequences can be compared [98, 290]. In Section 1.2.3 we touched upon how sequencing reads from a new sample are aligned directly against the reference genome of the organism in question to gain all kinds of insight into the sample and understand its variants. However, relying on this single mosaic linear reference is not always a good representation of the genome of the organism, as it might not be able to represent the complete diversity properly, which leads to this reference bias. For example, if the query sequence contains a non-reference allele, this could lead to incorrect or missing alignments [36], or biases in genotyping highly variable regions such as the human leukocyte antigen (HLA) genes [28], and biases in analyzing and interpreting ancient genomes [106]. This bias is even more pronounced in highly variable organisms such as prokaryotes, as they evolve more rapidly and exhibit events such as horizontal gene transfers, which results in their reference genomes to be less representative [47].

In an effort to find ways to better describe this genomic variability, the terms *core* and *accessory* genes were first coined by [265], where the "core" genes refer to essential genes (e.g., housekeeping genes) that are present in all or nearly all isolates, and the "accessory" genes (sometimes called "dispensable" genes) refer to the genes that are not present in every genome or isolate sequenced. The term *pangenome* was first introduced by [246] to describe a database of tumor genome and transcriptome alterations, as well as relevant normal cells. There have also been other ways to represent the variability in an organism's genome and reduce the reference bias, such as using alternative alleles along with the reference genome [40], representing the pangenome as a collection of genomes with an index that can report matches across the collection of genomes [61], extending the Burrows-Wheeler transform (BWT) algorithm to a graph with paths representing the different sequences [247], or using haplotype panels using a matrix with columns as variant sites and rows as haplotypes and a positional BWT (PBWT) for sequence matching [67].

We mentioned in Section 1.2.2 how graph data structures are very useful in representing large amounts of sequencing and its relationships. In [49] they define more broadly what characteristics of a pangenome data structure, and what functionality it needs to offer to be useful and be able to replace linear references. For example, we should be able to construct this data structure dynamically from different and independent sources, such as raw sequencing reads, complete genome assemblies, haplotype panels, or variants. This data structure should have some coordinate system, so we can identify loci without ambiguity. It should also contain other important features such as ability to visualize it, search it, annotate it, and compare different pangenomes. With such features, the move toward a graph-based pangenomes was clear, and graph representations of pangenomes have become more widespread, providing a more complete picture of pangenomes than a simple distinction into core and accessory genes. In graph-based models of pangenomes or "graph-

ical pangenomes", one represents the genomic variability of a population using a graph data structure, where nodes are labeled with sequences and edges connect nodes representing sequences that are adjacent to each other in one or more genomes in a population [74]. Most importantly, the graph contains several paths, where the concatenation of the sequence of a path retrieves the full sequence of one sample or haplotype that was added to the graph. One can then use these graph data structures as a reference instead of using a linear reference to reduce reference biases [198]. However, moving from a linear reference to a graphical one came with its own hurdles, such as graph pangenome construction. Until now, there is not one agreed upon way to construct a pangenome, and several methods have been developed. For example, variation-graph based models with tools such as `minigraph` [154], `minigraph-cactus` [120], and `PGGB` [89]; or de Bruijn graph-based methods for pangenomes such as `Bifrost` [121] and `mdbg` [75]. In this study [9], the authors reviewed the different methods for constructing human pangenomes graphs, and they highlighted weaknesses and strengths of the current methods, especially the problem with their computational efficiency.

Even though the pangenome field is still young, but it is developing rapidly, especially with the publication of the first draft of a human pangenome by the Human Pangenome Consortium (HPRC) [157]. Moreover, graph pangenomes have already shown their usefulness over linear reference in different context, for example, in terms of variant and structural variant analysis in rare diseases [103], better genotyping a wide range of variants, especially for complicated and repetitive regions [72], its application in biodiversity and conservation genomics [241], construction of personalized pangenome graph by sampling haplotypes close to the sample analyzed, giving a subgraph of the original graph to use instead [250, 39]. Moreover, while the adoption of graph pangenomes methods has been more evident in humans, it has also been shown their importance in other organisms. For example, capturing missing heritability in plants [294], annotation and variant calling in bacteria [131, 47], and improving sequence alignments in viral pangenomes [65].

1.3.1 Sequence-to-Graph Alignment

This section reuses materials from the Introduction section of [57], of which I am the first author.

As described in Section 1.2.1, sequence alignment has a very important role in several applications. Therefore, when graph pangenomes were shown to have major advantages as a new reference, came the need to adapt the alignment algorithms to align to graphs instead. Sequence alignment and pattern matching to a string graph are not new problems; they were described almost three decades ago. Pioneering studies include [163] where pattern matching on hypertext was described, and [4] where an algorithm for exact pattern matching to hypertext on a tree structure was developed. In 1995 [197] described regular pattern matching on a directed acyclic graph (DAG). Later on, a simpler algorithm with similar complexity to previous ones was developed that does pattern matching on *any* hypertext graph, that was also extended for *approximate* pattern matching [8], then [181] improved

both time and space complexity for pattern matching on a string graph.

In 2002, an algorithm similar to the previous one was independently developed specifically for biological data by [144]. Their algorithm, the Partial Order Alignment algorithm, was used for generating an MSA in a graph representation, the algorithm allows the alignment of a sequence against a DAG. In essence, it is a modified version of the common sequence alignment with dynamic programming algorithms (Section 1.2.1). This algorithm takes into account the incoming edges to nodes when calculating the scores then the traceback in the DP matrix. We use this algorithm later in Chapter 2 with some modification to work on both DNA and amino acid sequences. In recent years, several other tools have been introduced that perform sequence-to-graph alignments with better speeds and accuracy. Moreover, most of the current tools are not only restricted to DAGs, but can align against graphs with cycles, such as `GraphAligner` [216], `Giraffe` [249], `AStarix` [130], and `minigraph` [154].

1.4 File Formats

Here, we introduce the file formats that have been used throughout the thesis. Namely, FASTA/FASTQ formats to represent sequencing data, SAM (Sequence Alignment Map)/BAM (Binary Alignment Map) for representing linear sequence alignments, GFA (Graphical Fragment Assembly)/rGFA (reference Graphical Fragment Assembly) for representing genome graphs, and GAF (Graph Alignment Format) for representing sequence alignments on a graph.

1.4.1 FASTA and FASTQ Formats

The FASTA file format was first developed by David J. Lipman and William R. Pearson in their paper on protein similarity search [158]. The FASTA format is a text-based format, and it represents different genome sequence data, whether it was DNA, RNA, or amino acid sequences. Each sequence record consists starts with a single-line representing the name or identifier of the sequence, this line must start with "$>$"; this is then followed by one or more lines representing the sequence [180].
Sequences in the FASTA format are expected to follow the IUB/IUPAC amino acid and nucleic acid codes, with the following exceptions: lower-case letters, a dash or hyphen which represent gaps in the sequence, and in amino-acid sequences, U and * are accepted as well [180].

The FASTQ format was developed at the Wellcome Trust Sanger Institute [44]. FASTQ is also a text-based format and is similar to FASTA, however, it also includes information related to the sequence quality, and it is generally used to represent data coming from genome sequencers [189]. Each record in a FASTQ file consists of four lines, (1) starts with "@" until the first white space character, which represents the sequence identifier, (2) the sequence, (3) is a separation line with "+", it used to be followed by the sequence identifier,

but not anymore; finally, (4) a string with the same length of the sequence and represent the quality score (Phred scale) as described in [44].

1.4.2 The SAM and BAM Alignment formats

The Sequence Alignment Map format, or SAM, is a text-based and tab-delimited format for representing sequence alignments against other sequences. The file starts with a header which is optional; only the header lines start with "@". Alignment lines have 11 mandatory fields describing an alignment; it includes information such as position of the alignment in the query sequence, alignment position in the target sequence, length of alignment, the CIGAR string, and so on. The Concise Idiosyncratic Gapped Alignment Report or CIGAR string, is a string that describes the alignment in terms of matches, deletions, insertion, and mismatches [153]. Additional tags can be used, and should follow the form of "TAG:TYPE:VALUE" where the tag is made of two characters, and the type must be one of the following: "A" for a character, "B" for an array, "f" for a real number, "H" for a hexadecimal, and "Z" for a string [267].
The BAM format is the binary or compressed version of the SAM format.

1.4.3 Graphical Fragment Assembly Format

The GFA or Graphical Fragment Assembly format is used to represent genome graphs. The following information focuses on GFA1 format and is adapted from [91]. GFA2 is a superset of GFA1, and everything that can be encoded in GFA1 can be encoded with GFA2 with the ability to represent extra information; however, in this thesis, we only use the GFA1 format, which we will denote as GFA from here on out.
GFA files are text-based and tab-delimited, they use UTF-8 encoding but should not contain codepoint values higher than 127. The first column or field of each line in the GFA file is a one-letter line identifier, and the first line of the file is a header. Table 1.1 shows the different line types. In this thesis, we only used the following four line types:

1. File header, with "H" in the first field, followed by an optional value describing the version number.

2. Segment or node, with "S" in the first field, followed by two mandatory fields, the segment name, and the sequence. Several optional and user-defined fields can be added that represent different tags and follow the SAM format tag specifications [267].

3. Links or an edge that links two segments, the link line starts with an "L" in the first field, followed by five mandatory fields, name of the first segment, orientation of the first segment, name of the second segment, orientation of the second segment, overlap.

4. Path lines, with "P" in the first field, followed by two mandatory fields, the first comma separated segment names and their orientations, the second is a comma separated overlaps between the segments in the path.

Type	Description
#	Comment
H	Header
S	Segment
L	Link
J	Jump (since v1.2)
C	Containment
P	Path
W	Walk (since v1.1)

Table 1.1: Types of records in the GFA. Each line in the GFA file must start with one of these types. Adapted from [91]

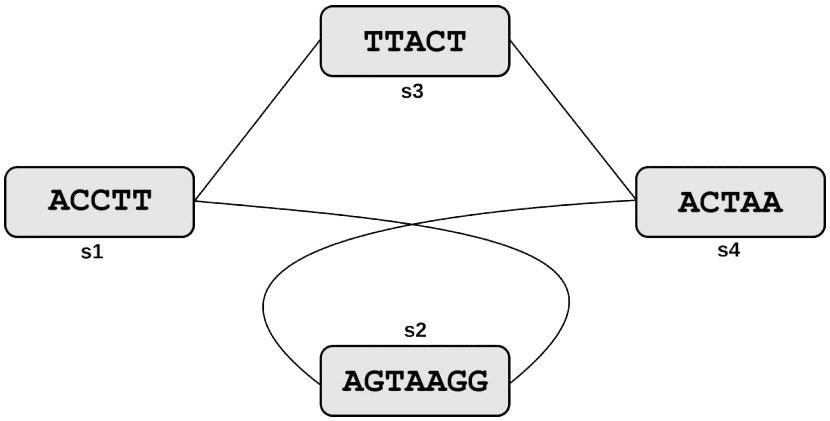
Figure 1.6 shows an example of a GFA file with four segments and its representation as a graph.

1.4.3.1 Reference GFA (rGFA)

The reference GFA or rGFA format was designed to address the issue of indexing the sequences in the segments, as certain base can be indexed by keeping a record of the segment ID and the offset from the beginning of the segment, this coordinate is called the *segment coordinate*, and this is unstable, as any changes happen to a segment, for example, splitting the segment into one or more segments will break this index. In applications like pangenomics, it is important to have a more stable coordinate system that can be used similar to how we use the coordinates in the linear reference, i.e., "chromosome:offset" instead of "segmentID:offset". In this format, each segment needs to have three obligatory tags: (1) "SN:Z:Value" where the value here is the name of the stable sequence that this segment originated from; (2) "SO:i:value" where the value is an integer representing the offset in the stable sequence that this segment originated from; (3) "SR:i:value" where the value is an integer representing the rank, this rank is 0 if the segment originates from the linear reference, an integer bigger than 0 otherwise. This information has been adapted from [150]. Figure 1.7 shows an example of an rGFA containing three segments, the segments "s1" and "s3" originate from chromosome 10 of a linear reference, therefore, they both have rank 0, and the "SO" tag indicates where these segments are located on that chromosome. "s2" is a segment that originates from some other sample and not a linear reference; therefore, it has rank 1.

1.4.4 Graph Alignment Format (GAF)

The Graph Alignment Format (GAF) is a tab-delimited and text-based format for representing sequence alignments against a genome graph. It comprises 12 mandatory fields. GAF format is a superset of the Pairwise mApping Format (PAF), with the addition of having an alignment path, which is a path in the graph where the sequence is aligned against [149].




```
H       VN:Z:1.0
S       s1      ACCTT
S       s2      AGTAAGG
S       s3      TTACT
S       s4      ACTAA
L       s1      +       s2      -       4M
L       s1      +       s3      +       2M
L       s3      +       s4      +       3M
L       s2      -       s4      +       3M
P       path1   s1+,s2-,s4+     4M,3M
P       path2   s1+,s3+,s4+     2M,3M
```

Figure 1.6: The top part of this figure shows a graph of four segments, the bottom part of the figure shows how this graph is represented in the GFA file. In the GFA file, we see that the link going from "s1" to "s2", the "s2" segment is in reverse complement, and that is why in the visualization, the edge goes from the end of "s1" to the end of "s2", which signifies that we read "s1" in the forward direction, i.e., "ACCTT" and we read "s2" in the reverse direction, i.e., "CCTTACT", we see then that this reverse complement sequence satisfy the overlap requirement of 4 characters, here, the four characters overlap are "CCTT"

A path can be represented with stable or unstable coordinates similar to what is described in Section 1.4.3.1. In the case of unstable coordinates, the path string is formed from segment IDs with the orientation of each segment represented as a prefix with the character "$>$" for the forward direction, and "$<$" for the reverse direction (reverse complement). For the stable coordinates, instead of using the segment IDs, the value in the "SN" tag is used. Figure 1.8 shows an example of two alignments against the rGFA in Figure 1.7, each alignment is represented twice, once with unstable coordinates and once with stable coordinates. The first sequence is "TCAGAATGCCCA" and aligns to the last four characters of "s1", all of "s2", and the first four characters of "s3". The second sequence is "TCAGCCA", it aligns to the last four characters of "s1" and the first four characters of "s3" with a deletion.

```
H   VN:Z:1.0
S   s1   CGGGGCTCAG   SN:Z:chr10   SO:i:10   SR:i:0
S   s2   AATG   SN:Z:sample1   SO:i:5   SR:i:1
S   s3   CCCAGTGA   SN:Z:chr10   SO:i:20   SR:i:0
L   s1   +   s2   +   0M
L   s3   +   s3   +   0M
L   s1   +   s3   +   0M
```

Figure 1.7: This is an example of an rGFA file with three nodes or segments, we see that segments "s1" and "s3" have rank 0 specified by the "SR" tag, both segments belong to the linear reference, specifically, chromosome 10 specified by the "SN" tag. "s2" has rank 1 and belongs to another reference or sample used to build the graph. The "SO" tag shows the offset of the sequence in reference.

```
sequence1_u 12 0 12 + >s1>s2>s3 22 6 18 12 12 60 cg:Z:12M
sequence1_s 12 0 12 + >chr10:10-20>sample1:5-9>chr10:20-27 22 6 18
   12 12 60 cg:Z:12M

sequence2_u 7 0 7 + >s1>s3 18 6 14 7 8 60 cg:Z:5M1D2M
sequence2_s 7 0 7 + chr1 18 6 14 7 8 60 cg:Z:5M1D2M
```

Figure 1.8: This shows the GAF alignments of two sequences against the rGFA shown in Figure 1.7 in both unstable and stable coordinates. In the stable coordinates, if the sequence aligns to segments that belong to the linear reference, the path string then represents the chromosome in the "SN" tag, and the following fields describe the alignment coordinates, e.g., "sequence2_s". However, if the alignment aligns through segments that do not belong to the linear reference, then each "SN" tag is used for each segment touched by the alignment, e.g., "sequence1_s".

1.5 Outline

This thesis presents several methods and software toolkits developed to address various problems related to genome graphs, pangenomes, proteomes, and panproteomes. These methods and toolkits include new concepts such as bringing pangenomes to the prokaryotic protein world the development of panproteomes and sequence-to-panproteomes alignments; moreover, efficient tools for processing and manipulating graphs and alignments, and epistasis detection between variants in proteins. Each chapter starts with a brief summary introducing the chapter, the software it presents, and key results. If the chapter's contents reuse materials from papers, this will be indicated. Furthermore, if certain materials were not my personal contribution, this will also be stated at the beginning of the chapter. Supplementary Chapter F also states the author list, contribution, and license of each paper underlying this thesis.

In Chapter 2 we present `PanPA`, a software toolkit for building, indexing, and aligning panproteomes. Here, we define the panproteome as a collection of graphs, where each graph represents different sequences of a protein or a coding region. In this chapter, we show that adapting the graph pangenome to the amino acid world is particularly advantageous in prokaryotes, and that the panproteome of distantly related organisms can be used as a reference that is able to capture many alignments that would otherwise be lost in the DNA space.

In Chapter 3 we introduce several tools related to working with graphs in the GFA format, and graph sequence alignments in the GAF format. First, we showcase `GFASubgraphs`, and its internal GFA API. Following, we present `extgfa`, a proof-of-concept implementation of a GFA library that can utilize ideas from video games to have a lower RAM footprint and keeps most of the graph on the disk (external memory). Finally, we present `gaftools`, a tool for downstream processing and analysis of GAF alignments to pangenomes. The aim of `gaftools` is to bridge a gap between linear and graph alignment processing by implementing several useful functionalities such as alignment ordering, indexing, viewing (subsetting), and other functions.

In Chapter 4 we present an unpublished collaborative effort in cancer genomics and structural variant detection. Here, we use multi-platform sequencing data for the cancer cell line NCI-H2087 and the matched normal cell line NCI-BL2087. We first assemble the genomes of both cell lines using the `PGAS` pipeline, which utilizes both long HiFi reads and single-cell strand sequencing technology. We then call structural variants using 5 different callers using both the long HiFi reads and the assemblies. Subsequently, we intersect the calls to generate a set of somatic structural variants for the cancer cell line. In addition, we investigate and discuss the role of assembly graphs, graph visualization, and graph alignments in disentangling and potentially producing higher quality assemblies and somatic variants for the cancer cell line.

In Chapter 5 `EpiPAMPAS` is introduced, a tool for detecting epistatic interactions between

mutations in Multiple Sequence Alignments (MSA), mainly focusing on protein sequences. `EpiPAMPAS` employs a hierarchical clustering dendrogram instead of a phylogenetic tree to understand, reconstruct, and analyze the most-likely ancestry of the sequences. Subsequently, uses a mutation counting method on the dendrogram to detect potential ecstatically interacting variants.

The publications underlying this thesis are listed below, information about authors contributions and license for each publication is described in Supplementary Materials F. Shared first authorship are indicated with a *:

- **F. Dabbaghie**, S. K. Srikakulam, T. Marschall, and O. V. Kalinina. PanPA: generation and alignment of panproteome graphs. *Bioinformatics Advances*, 3(1), Jan. 2023.
- **F. Dabbaghie**. extgfa: A low-memory on-disk representation of genome graphs. *bioRxiv*, Dec. 2024.
- **F. Dabbaghie***, K. Thedinga*, G. A. Bazykin, T. Marschall, and O. V. Kalinina. EpiPAMPAS: Rapid detection of intra-protein epistasis via parsimonious ancestral state reconstruction and counting mutations. *bioRxiv*, Dec. 2024.
- S. Pani, **F. Dabbaghie**, T. Marschall, and A. Soylev. gaftools: a toolkit for analyzing and manipulating pangenome alignments. *bioRxiv*, Dec. 2024.
- **F. Dabbaghie**, J. Ebler, and T. Marschall. BubbleGun: enumerating bubbles and superbubbles in genome graphs. *Bioinformatics*, 38(17):4217–4219, Sept. 2022. [1]

[1] The main algorithm and code was developed during my Master's final project under Prof. Tobias Marschall. However, further code optimization, experiments, and paper writing were done during my PhD time.

Chapter 2

PanPA: PanProteome Graph Builder and Aligner

This chapter introduces PanPA, a tool for generating graphs from Multiple Sequence Alignments (MSAs) representing individual genes or proteins, also indexes these graphs and is able to align both amino acid and DNA sequences back to these graphs. We show that the idea of generating graphs from protein sequences is very helpful when aligning phylogenetically distant organisms that might not have a proper reference against graph made from another organism. We also show that we were able to increase the number of aligned DNA sequences that would be unaligned otherwise when using a linear reference.

This chapter is based on a publication in Bioinformatics Advances [57], of which I am the first author. Materials reused from the publication will be indicated.

2.1 Introduction

This section reuses material from [57] of which I am the first author

Prokaryotes have evolved rapidly for billions of years, and due to geochemical changes on the planet, bacteria needed to adapt in order to survive, which was the major contributor to their vast genetic diversity [66]. When stable environments are studied, such as garden soil, lakes, or costal seawater, which are ecosystems that do not experience extreme environmental changes, we still observe a vast diversity of prokaryotic organisms. It is expected that not more than 1% of the bacteria in these samples can be cultivated in the lab [7]; this suggests that the true diversity is even much larger than estimated. The number of prokaryotic cells on earth has been estimated to be around $4 - 6 \times 10^{30}$ and their cellular carbon amount is $3.5 - 5.5 \times 10^{14}$ kg [288]. Both the advancement in sequencing technologies (Section 1.2), and sequence alignment algorithms (Section 1.2.1) highlighted the similarities, differences, divergence, and variability between the different prokaryotic genomes [200, 127].

We discussed how the concept of graph pangenomes came about (Section 1.3, and their importance in better capturing the diversity of genomes and ability to circumvent the linear reference bias. However, if we look at the various pangenomes that have been constructed in recent years. For example, in bacteria for *E. coli* [47], in plants for *Cucumis sativus* [155], and in humans [154, 74], including the work of the Human Pangenome Reference Consortium (HPRC) [157]; we see that they all build a pangenome for a single species in the DNA space, which may still not be able to capture the high diversity in prokaryotes and may not be able to produce good alignments. This problem is even more exacerbated in highly diverse and less-studied clades, such as *Actinomycetes* or *Myxobacteria*, which are an important source of natural products that can be used in drug discovery [90]. The diversity of these clades is much higher than that described, due to limitations in cultivation and in-lab growth [173]. Moreover, some of their species even lack a reference genome. In these cases, however, sequence similarity can still be traced by looking at protein alignments instead, i.e., looking only at the amino acid sequences in the coding regions, as these alignments will have a higher identity compared to DNA sequence alignments. There are still several reasons for this: 1) Amino acid sequences are evolutionarily more conserved compared to the DAN sequence of the complete genome [134]. 2) The amino acid alphabet is larger, the "signal-to-noise ratio" is better, e.g., two random DNA sequences will exhibit around 25% similarity over the entire sequence and can go over 50% in certain local stretches [285]. 3) The same amino acid can be encoded by several codons, hence, a part of the mutations in DNA are not visible at the amino acid level. These mutations are called silent mutations [34]. 4) Some of the errors introduced during sequencing can cause a frameshift during alignment, which can be avoided by using amino acids [244]. 5) In amino acid sequence alignment, we usually use a substitution matrix instead of just edit distance in DNA sequence alignment, better capturing biological reality [20]. 6) In prokaryotes, the proportion of non-coding regions in the genome can range from 5 to 50%. However, for the vast majority, the fraction is less than 18% [223], further motivating a focus on coding sequences.

Here, we propose a new tool, we call `PanPA` (**PanP**roteome **A**ligner) to conduct pangenomic analyzes that considers amino acid, or protein sequences. `PanPA` builds directed acyclic graphs for each individual protein or protein cluster in order to represent a pangenome as a collection of these graphs. Computing alignments in amino acid space can give a big advantage in terms of finding more sequence similarity, and being able to align more phylogenetically distant organisms against each other while losing relatively little genome information. In [286] they showcased how aligning in protein space introduces significant improvements in alignment accuracy and functional profiling in a metagenome scenario. The idea of having many graphs representing a pangenome instead of one large graph was presented in [47]; in their tool `Pandora`, the authors define a pangenome as a collection of *local graphs* where each local graph represent some locations in the genome that can be

pre-defined by the user. PANPA combines the two ideas of 1) having a pangenome consisting of many smaller graphs, where each graph represents a protein or a protein cluster, and 2) working in amino acid space rather than nucleotide sequences to support pangenomic analyzes over larger evolutionary distances. We call such a collection of graphs a *panproteome*.

2.2 Methods

With PanPA, we aim to build a graphical panproteome and be able to align new DNA or amino acid sequences back to it. We define the panproteome here as a collection of graphs representing proteins sequences or protein clusters, i.e., each protein or protein cluster is represented as a separate graph.
PanPA consists of three main steps, as follows:

1. Building an index from the input MSAs (described in Section 2.2.1).
2. Constructing a directed graph from each MSA (described in Section 2.2.2).
3. Aligning query sequences to these constructed graphs with the help of the index constructed from these MSAs (described in Section 2.2.3).

2.2.1 Building Seed Index from MSAs

This section reuses material from [57] of which I am the first author

For each sequence N of length m, we define a substring $s = N[i, j]$, where $0 \leq i \leq j \leq m-1$, as a substring of N starting at position i and ending at position j with a length of $j - i + 1$ for the substring. A k-mer from a string N is then defined as a substring of length k. We also define a function min(S) that takes the set $S = \{s_1, s_2, \ldots, s_w\}$ of size w containing w equal-length strings and returns the lexicographically smallest string in this set; we call this function a *minimizer*.

Two types of indexes are implemented in PanPA based on the types of seeds, a *k-mer-based* seed index and a *(w, k)-minimizer-based* seed index. The *minimizer-based* index was originally developed by [239] and was first used in bioinformatics to reduce storage requirements for sequencing data by [220]. Where a minimizer is usually the alphabetically smallest substring in a collection of substrings representing a sequence, or the smallest substring of a window of w equally-sized substrings from a sequence. In both cases, the index stores a key-value map, where the keys are the k-mers or (w, k)-minimizers seeds extracted from each sequence in the input MSAs, and the value for each key is an ordered array of MSAs where that seed was found. This ordering is based on the number of times that seed was seen in that certain MSA. More details on the indexing implementation is provided in Section 2.3.

To construct a k-mer-based seed index of the MSAs collection, for every string N, we extract all the k-mers from a string, where S_{seeds} is the collection of the consecutive k-mer from string N, and defined as $S_{seeds} = \{N[0, 0+k-1], N[1, 1+k-1], \ldots, N[i, i+k-1]\}; \forall i \in \{0, \ldots, (m-k)\}$, where each string N of length m will contain $(m-k+1)$ k-mers.

As for the (w,k)-*minimizers*, for each sequence N, instead of taking the set of all consecutive k-mers, we take the set S_{seeds} that contains the minimizer of every consecutive window of w k-mers, i.e., we take the smallest seed in a set of seeds for each consecutive window of seeds. $S_{seeds} = \{\text{min}(S_{0,w}), \text{min}(S_{1,w}), \ldots, \text{min}(S_{i,w})\}; \forall i \in \{0, \ldots, (m-w-k+1)\}$, where $S_{i,w}$ is a set of w consecutive k-mers starting at position i in the string N.

2.2.2 Generating a Directed Acyclic Graph from a MSA

For this step, we developed a simple algorithm to convert each MSA into a corresponding graph in the GFA format (see Section 1.4.3 for GFA format definition), where each original sequence from the MSA is represented in the GFA file as a path, i.e., we can reconstruct the original sequences by concatenating the strings in each node of the path. This algorithm runs in $\mathcal{O}(n \times m)$ time, where n is the number of sequences in the MSA and m is the length of the alignment, and has two steps, graph generation and graph compaction.

2.2.2.1 Graph Generation

This section reuses material from [57] of which I am the first author

We define an alphabet A as the amino acid alphabet, and a matrix $M = (a_{i,j}) \in \{A \cup -\}^{m \times n}$. Each column in matrix M is a vector $\{A \cup -\}^n$ and each row is a vector $\{A \cup -\}^m$. In a nutshell, the algorithm loops through each column vector at position j where $0 \leq j \leq m-1$, and for each vector, it constructs a new node $node_j(c)$ for each unique character $c \in A$ encountered. Subsequently, edges are added between two nodes $node_{j_1}(c_1) \rightarrow node_{j_2}(c_2)$ (where $j_1 < j_2$) if and only if the characters c_1 and c_2 were consecutive in one of the row vectors in matrix M after ignoring the character $\{-\}$. Algorithm 1 is a pseudocode for this algorithm.

As a simple example, consider an MSA with three sequences as shown in Figure 2.1. In this figure, the columns marked yellow are the *current* column in the loop, and the column in red is the *previous* column. The algorithm loops through the columns of the MSA, and at each column, it scans the character in that columns, if the character is new, then a new node is initialized for this character (lines 18–22 in Algorithm 1). Otherwise, if the character is not new, i.e., a node was already been constructed for that letter at that column, we assign the character a corresponding node identifier. After building nodes for a column j, i.e., the *current* column in the loop, we synchronize with the previous column $j-1$ (if it exists) (lines 2–10 in Algorithm 1). For the synchronization, we go through each row i in both columns, and for every row i we have three choices:

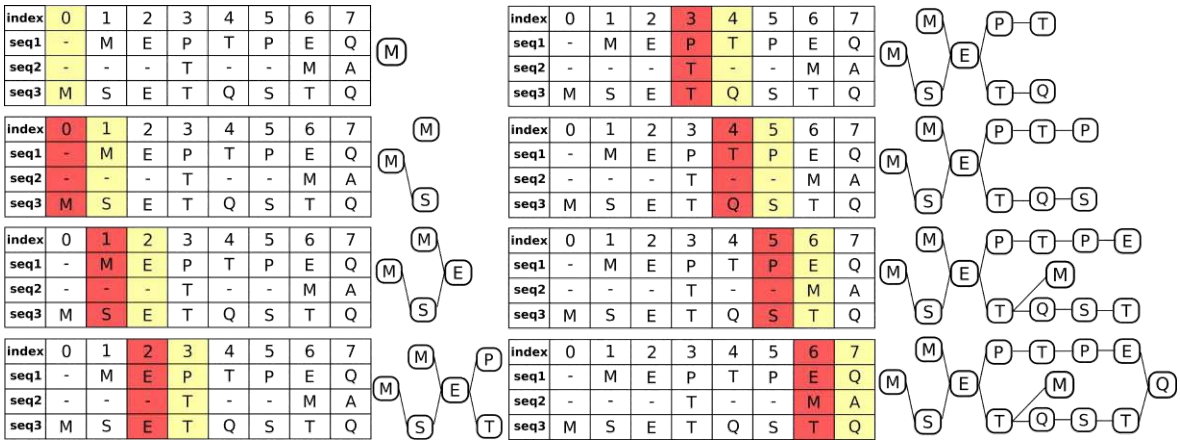


Figure 2.1: MSA to GFA: turning an MSA into a graph. The MSA in this example contains three sequences, -$MEPTPEQ$, ---T-MA, and $MSETQSTQ$; and the step-by-step graph construction is shown on the panels from top to bottom. At every step, the yellow column is the current position and the red column is the previous one. *Figure adapted from [57].*

1. if $c_{i,j}, c_{i,j+1} \in \{-\}$ then there is nothing to be done. For example, the first two gaps in the second sequence in Figure 2.1)

2. if $c_{i,j}, c_{i,j+1} \in A$, then we need to draw an edge between $node_j(c_{i,j})$ and $node_{j+1}(c_{i,j+1})$;

3. if $c_{i,j} \in A$ and $c_{i,j+1} \in \{-\}$ then we need to keep the character $c_{i,j}$ "saved" and continue going through the MSA until we reach a column $j + x$ where $c_{i,j+x} \in A$ and $x > 1$, then we can draw an edge between $node_j(c_{i,j})$ and $node_{j+x}(c_{i,j+x})$. An example of this final case in Figure 2.1 is the second sequence, where columns 4 and 5 have gaps, but column 3 has a T; we then keep track of this until we reach the character M in column 6, where we construct a node for it. Now, we can draw an edge between the nodes representing T of column 3 and M of column 6.

Since we iterate through the MSA from left to right and only draw edges between consecutive nodes between the current nodes and the previous ones, we guarantee that resulting graph is directed and acyclic (DAG).

2.2.2.2 Graph Compaction

This section reuses material from [57] of which I am the first author

Linear stretches of nodes can arise while generating a graph from an MSA. A set of consecutive nodes $\{node_{j_1}(c_1), node_{j_2}(c_2), \ldots, node_{j_n}(c_n)\}$ is a linear stretch, if and only if each node in the set has an in-degree and out-degree of one, with an exception that the first node $node_{j_1}(c_1)$ can have a higher in-degree and the last node $node_{j_n}(c_n)$ can have a higher out-degree. We can, therefore, compact these nodes into one node and concatenate their

Algorithm 1 Constructing a DAG from MSA. *Taken from [57]*

Matrix M {Matrix of dimensions $m \times n$}
Map $nodes$ {A map of node IDs: array of children IDs}
Array $previous$ {Empty array of length n}
Array $current$ {Empty array of length n}
Int n {Integer starting with 0}

for $j \in \{0 \dots m\}$ **do**
  for $i \in \{0 \dots n\}$ **do**
    if $(current[i] \equiv None)$&&$(previous[i] \neq None)$ **then**
      $current[i] \leftarrow previous[i]$
    else if $(current[i] \neq None)$&&$(previous[i] \neq None)$ **then**
      $nodes[previous[i]].append(nodes[current[i]]))$
    else
      pass
    end if
  end for

  $previous \leftarrow current$
  Array $current$ {Empty array of length m}
  Array $column \leftarrow M[j]$ {characters in column j}
  Map $seen$ {empty map} {character:node ID}
  for $i \in \{0 \dots m\}$ **do**
    if $column[i] \in seen$ **then**
      $current[i] \leftarrow seen[column[i]]$
    else
      $n \leftarrow n + 1$
      $nodes[n] \leftarrow []$
      $current[i] \leftarrow n$
      $seen[column[i]] \leftarrow n$
    end if
  end for
end for

sequences. For example, in Figure 2.1 at the last step of constructing the graph, the stretch of nodes $P \rightarrow T \rightarrow P \rightarrow E$ can be compacted into one node.

There is, however, one special case one needs to take into consideration where not all nodes in a linear stretch of nodes can be compacted. This happen because paths in the P line in the GFA are represented as a list of nodes that their concatenation generates the original sequence. Therefore, the path only considers the complete sequence that the node encodes, and it cannot represent a part of the node.

As an example to this, looking at Figure 2.2, we have an MSA of three sequences each of length 4, $MTQT$, `--`QT, and MT`--`. When we generate a DAG from this MSA, we end up with a line graph of four nodes (M, T, Q, and T). However, we cannot merge all of the nodes together into one node, then we will not be able to represent *Seq2* or *Seq3* individually

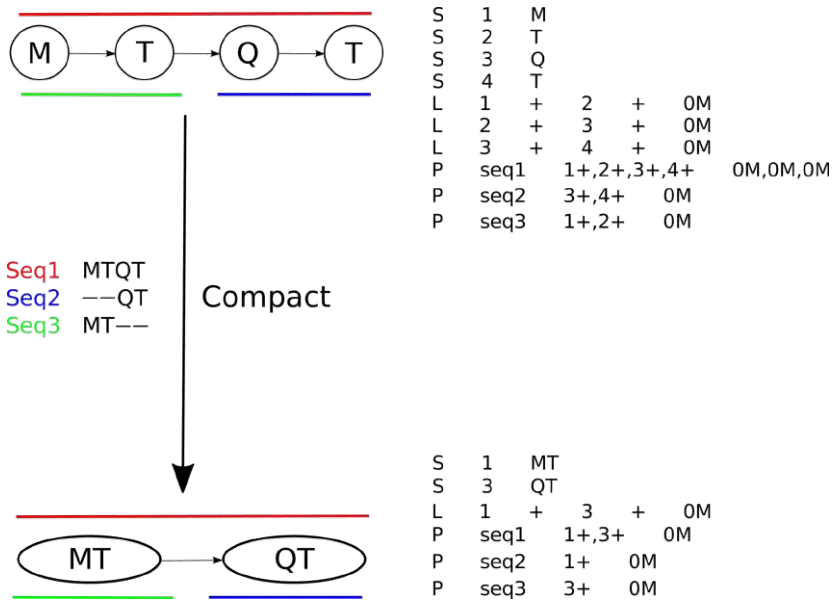


Figure 2.2: This figure shows an example of how we cannot always compact linear stretches of nodes into onde node. Here, we have three sequences of length four each, and when we build the graph, we get four nodes, but we can only compact the first two nodes and the second two nodes, and now we can represent the three sequences as a node path in the output GFA file. *Figure taken from [57].*

in a P line. Therefore, we can only merge the first two nodes together and the second two nodes together. Now, we can represent all three sequences as separate P lines as shown in the Figure.

2.2.3 Aligning Query Sequences

2.2.3.1 Amino Acid Query Alignment

This section reuses material from [57] of which I am the first author

`PanPA` uses a modified version of the Smith-Waterman algorithm for local alignments [251] known as partial-order alignment [144], this algorithm is capable of performing alignment on graphs instead of a linear sequence. In brief, the algorithm concatenates all the sequences in the nodes into one long sequence, the target sequence. Subsequently, it builds a Dynamic Programming (DP) table between the query and the target sequences, similar to normal Smith-Waterman. However, in this table, we cannot simply look at the left or diagonal cell when calculating the score for the current cell, because the previous column does not necessarily correspond to the previous character topologically in the graph, however, we need to make jumps through the table, following the edges in the graph. Moreover, if there is more than one incoming edge, we need to follow each edge, calculate the scores, then decide which score is best. As the graphs constructed from the MSAs are DAGs, the graph can be topologically sorted generating a list of ordered vertices. The concatenation of the

sequences of the ordered vertices is the target sequence to align against (Figure 2.3).

Here, the dynamic programming matrix is defined as $H = (a_{i,j}) \in \mathbb{R}^{(n+1)\times(m+1)}$ where m is the size of the query sequence M, and n is the size of the concatenated sequences N from the topologically-ordered vertices. We add one extra row and column filled with 0 as the initializing row and column. Similar to the Smith-Waterman algorithm, for each cell, we are dependent on the scores of adjacent cells describing the alignment operation (Match/Mismatch, Insertion, or Deletion). As some characters or columns correspond to the first character of a node in the graph, instead of looking at the previous column to get the previous character, we need to calculate the score of that cell based on all possible previous characters following all incoming edges to that node. Therefore, for calculating the score of cell i, j, we take the max of all scores calculated considering all characters from the incoming edges p_l, where p_l is the column index that represent the previous character after following the incoming edge. Equation 2.1 describes that the score for cell (i, j) is based on the max score of all possible incoming edges:

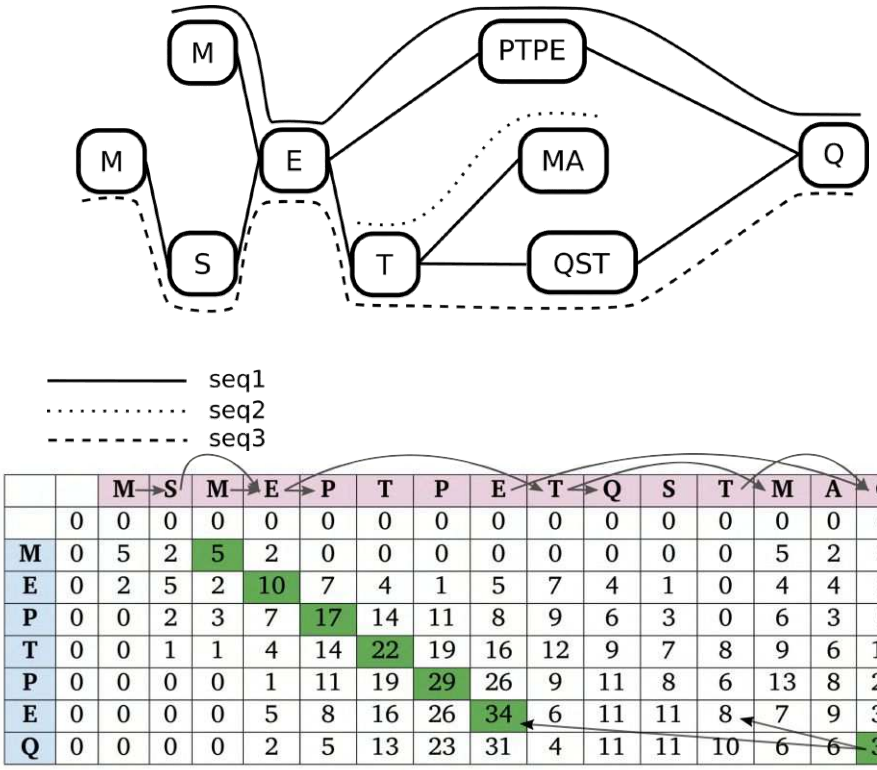
$$H_{i,j} = \max_{\forall l: p_l \in P_{in}} (score(i, j, p_l)). \tag{2.1}$$

To calculate a single score for one previous character following an incoming edge, we have three possible choices: a match/mismatch, an insertion, or a deletion, this is shown in Equation 2.2:

$$score(i, j, p_l) = \max \begin{cases} H_{i-1,p_l} + sub(N[p_l - 1], M[i - 1]) \\ H_{i-1,j} + \Delta \\ H_{i,p_l} + \Delta \\ 0 \end{cases} \tag{2.2}$$

where Δ is the gap score, and $sub(c_1, c_2)$ is a function that takes two characters and returns the substitution score, e.g., Blosum62 [116]. Since our graphs are compacted, one node can have several characters. Therefore, if we are calculating the score for some $H_{i,j}$ and the column j does not correspond to the first character in the node, then we can simply use Equation 2.2 with p_l being $j - 1$.

For tracing back the alignment, we use the same approach as in the classical Smith-Waterman algorithm, checking where the score of the cell came from to know which path our query sequence aligned to. For example, in Figure 2.3, the bottom right corner cell has the highest score in the table. When tracing back from there, we see two incoming edges, one leading to the character E and the other to the character T, the traceback then jumps to where the score 39 came from (the maximum between the two after adding the match score). This corresponds to the character E at $j = 9$. Now starting the traceback at $j = 9$, we see that there are no incoming edges, so we only need to look at $j = 8$, and so on. Once we finish the traceback and stop, we conclude that our query sequence $MEPTPEQ$ matches



		M	S	M	E	P	T	P	E	T	Q	S	T	M	A	Q
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
M	0	5	2	5	2	0	0	0	0	0	0	0	0	5	2	0
E	0	2	5	2	10	7	4	1	5	7	4	1	0	4	4	2
P	0	0	2	3	7	17	14	11	8	9	6	3	0	6	3	5
T	0	0	1	1	4	14	22	19	16	12	9	7	8	9	6	13
P	0	0	0	0	1	11	19	29	26	9	11	8	6	13	8	23
E	0	0	0	0	5	8	16	26	34	6	11	11	8	7	9	31
Q	0	0	0	0	2	5	13	23	31	4	11	11	10	6	6	39

Figure 2.3: Alignment of a sequence to a protein graph. Top: example protein graph, which is also the compacted version of the graph made in Figure 2.1. bottom: the corresponding DP table. The ordered graph vertices are in the columns, and the query sequence is in the rows. Arrows between columns correspond to the graph edges. Arrows in the DP table correspond to potential previous cells in the DP process. *Figure taken from [57].*

the nodes $M, E, PTPE, Q$, forming the path that represents "seq1" in Figure 2.3.

2.2.3.2 Frameshift-Aware DNA Query Alignment

This section reuses material from [57] of which I am the first author

When translating DNA sequences to proteins, indels can cause frameshifts, which would cause the alignment in the amino acid space to stop short due to sequence divergence. To account for this, we developed another alignment algorithm that is inspired by the method in [244]. This alignment algorithm takes into account indels and the frameshifts they cause in the DNA query sequence.

In this method, we do not translate the DNA query sequence, but align the DNA sequence directly to the target amino acids graph unchanged. Here, when we are looping through the rows in the DP table, every row (or character in the DNA sequence) belongs to one of the three possible reading frames. For example, the first character/row represents the first amino acid of the first frame, the second character/row represents the first amino acid in the second frame, and the third character/row represents the first amino acid in the third frame.

Now, the next character/row would represent the *second* amino acid in the first frame, and so on. Moreover, we modified the scoring equation to also account for frameshift errors, which then allows the traceback to switch frames if the scores justifies the switch.

Figure 2.4 is an example of a DNA sequence with an insertion of one nucleotide that causes a frameshift. The inserted nucleotide is marked in red. We see that our algorithm was able to account for this frameshift, and when we follow the green cells that represent the traceback, we see that it follows the complete DNA sequence, i.e., the alignment will be reported fully as aligning against the amino acid sequence with an insertion.
To explain in more details, when filling the cells in the DP table, we start from the third nucleotide C which represents the amino acid of the codon ACC, i.e., the first amino acid in the first frame. When we move to the next row, we are then considering the amino acid of the codon CCT, which the first amino acid of the second frame, and so on. Looking at Equation 2.3, we see that to get the score of a cell (i, j) in the table, we need to take into consideration the in-frame insertion and deletion (in the equation represented with the first three terms). However, to account for the out-of-frame scores, we introduce two more terms to the equation. These terms represents the jump between the frames:

1. $i-4, j-1$ jump, which describes an insertion frameshift, when the DNA sequence has an extra nucleotide that introduced a frameshift. This moves the current alignment to the previous frame;

2. $i-2, j-1$ jump, which describes a deletion frameshift, where the DNA sequence has one nucleotide deleted, which moves the current alignment to the next frame.

For the $i-4, j-1$ and $i-2, j-1$ jumps, we also introduce a frameshift penalty σ.

$$score(i,j) = \max \begin{cases} H_{i-3,j-1} + sub(trans(N[i-2,i]), M[j-1]) \\ H_{i-3,j} + \Delta \\ H_{i,j-1} + \Delta \\ H_{i-4,j-1} + \sigma \\ H_{i-2,j-1} + \sigma \\ 0 \end{cases} \tag{2.3}$$

where N is the DNA sequence, M is the amino acid sequence, the function $trans(codon)$ takes a codon and returns the equivalent amino acid, and the function $sub(c_1, c_2)$ takes two amino acids and returns the substitution score between them.

2.3 Implementation

`PanPA` was built using `Python` with the only dependency being `Cython`. `Cython` was used mainly to optimize the core alignment algorithm. Each of the three steps mentioned in

		P	P	T	H	Q
	0	0	0	0	0	0
	0	0	0	0	0	0
A	0	0	0	0	0	0
C	0	0	0	0	0	0
C	0	2	2	1	0	0
T	0	0	0	0	0	0
C	0	0	1	1	0	0
T	0	2	4	3	2	1
G	0	0	1	1	0	0
A	0	0	1	3	2	1
C	0	1	3	3	2	1
C	0	0	1	3	2	1
C	0	2	2	2	2	1
A	0	2	3	2	2	1
C	0	0	1	2	5	4
C	0	1	1	4	3	2
A	0	2	4	3	2	4
A	0	0	1	2	4	7

Figure 2.4: Frameshift aware alignment. The scores here are as following: match = 2, mismatch = framshift = gap = -1. We have the DNA sequence $ACCTCTGACCCACCAA$ aligning against the amino acid sequence $PPTHQ$, if we remove the G from the DNA sequence, we actually get a perfect match. Looking at the table, we see in the traceback, that we were able to account for the insertion and still able to align the DNA sequence against this amin aicd sequence completely. *Figure taken from [57].*

Section 2.2 is implemented as a separate subcommand. The subcommands are `build_index`, `build_gfa`, and `align`.

Figure 2.5 shows the pipeline of `PanPA`. It starts with MSA files, where each MSA represents one protein or a protein cluster. This input is required by both `build_index` and `build_gfa` modules. The subcommand `align` takes a FASTA file with query sequences, the graphs generated from the MSAs, and the index file, to perform the alignments. The output alignments are in GAF (Graph Alignment Format) format with unstable coordinates (explained in Section 1.4.4).

2.3.1 Indexing

This section reuses material from [57] of which I am the first author

As explained in Section 2.2.1, the index is a key-value map, where the keys are the unique seeds and the values are arrays of the MSAs (equivalently, graphs) where the seed belong to. In our implementation, the value array is ordered based on the number of times that seed showed up in an MSA, normalized by the number of sequences in that MSA. With this ordering, users can choose a cutoff limit on how many graphs one seed can belong to, as

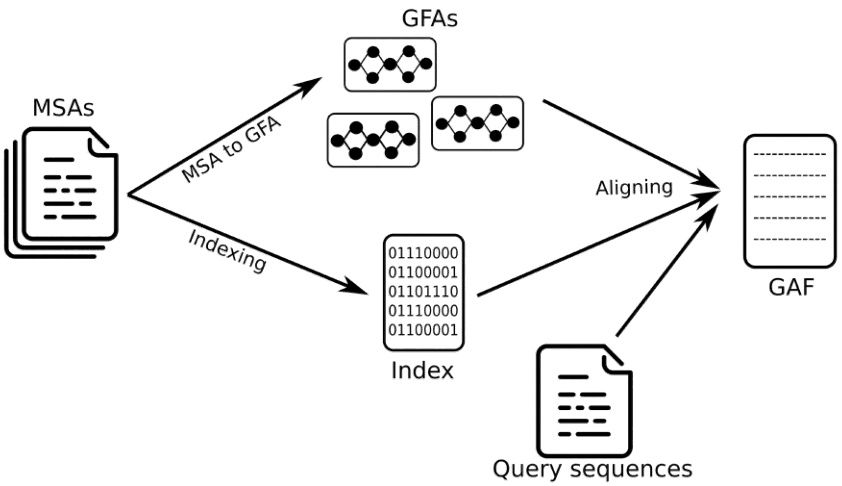


Figure 2.5: Here, we show the general pipline of PanPA and its subcommands. Each subcommand can be also run separately or more than once with different parameters. *Figure taken from [57].*

some seeds can be promiscuous, especially if a small value for k is used.

For example, if we have three MSAs m_1, m_2, and m_3 containing 10, 7, and 3 sequences respectively. Seed s_1 was found in m_1 twice, found in m_2 four times, and found in m_3 three times (with the normalized counts 0.2, 0.57, and 1, respectively), and the user cutoff is set to 2, then in the resulting index, the seed s_1 will point to a list containing $[m_3, m_2]$.

Extracting (w, k)-minimizers can be time consuming, as we need to find the lexicographically smallest k-mer in the window. Therefore, we used the Sliding Window Minimum algorithm [32], which has a time complexity $\mathcal{O}(n)$ where n is the size of the input sequence. This algorithm is described in more detail in Algorithm 1 in [132].
Supplementary Section A.5 further shows the tradeoff between time and index size when using different parameters.

2.3.2 Generating Graphs

This section reuses material from [57] of which I am the first author

The implementation for converting an MSA to a GFA is based on a previous implementation of a standalone command-line tool called `msa_to_gfa` [55]. More information about this standalone tool can be found in Supplementary Section A.2.

In PanPA, each MSA is converted into a DAG in GFA format, i.e., there is a 1 to 1 correspondence between the MSAs and GFAs. Therefore, when a seed in the index points to one MSA, we can align the query sequence to the graph that corresponds to that MSA. This subcommand is also parallelized, so the user can provide more cores to convert MSAs to GFAs simultaneously.

2.3.3 Aligning

This section reuses material from [57] of which I am the first author

Given a query sequence, we count all the seed hits from the query to the MSAs using the index. We use the index to generate a list of MSAs (equivalently, graphs) to align against. This list is sorted based on the number of hits. For example, if the query sequence had five seeds, where four of them pointed to m_1, and one pointed to m_3, our list of matches will be $[m_1, m_3]$. The user can also specify to how many potential MSAs/graphs can one query be aligned against, or choose to align to all matches. If, for example, the limit of matches was set to 1, our query sequence will only be aligned to m_1. Moreover, the user can filter the alignments based on a minimum alignment identity score. PanPA uses a linear gap penalty and the user can choose one of many substitution matrices available, such as benner [17], BLOSUM [116], Point Accepted Mutation (PAM) [119], and many others. The user also has control over other scores such as gap penalty and frameshift penalty.
This step is also parallelized and the user can provide more CPU to speed up the process. More on runtime is provided in Section 2.4.3.

2.4 Validation of PanPA

This section reuses material from [57] of which I am the first author

Here, we want to confirm that PanPA is working as intended, that it is able to use the index to find the correct graphs to align to, able to align to the correct paths in the graph, and that it is able to handle errors in the sequences. To this end, in Section 2.4.1 we first construct a panproteome of *E. coli* using assemblies from a public repository. Then, in Section 2.4.2, we use sequences from the same panproteome to align back and investigate whether PanPA was able to find the correct graph using the index and align to the correct path. We also tested the effects of different indexing parameters on the alignment accuracy. In Section 2.4.3 we expand about the runtime using different parameters. Finally, in Section 2.4.4, we experiment with the robustness of PanPA and its alignments, and how well it handle mismatches on the query sequences.

2.4.1 Building an *E. coli* Panproteome

This section reuses material from [57] of which I am the first author

First, we want to validate that PanPA is able to build, index, and find correct alignments of a panproteome. To that end, we downloaded 1,351 *E. coli* assemblies that were marked as “Complete Genome” from RefSeq [190]. We extracted every amino acid sequence corresponding to a coding region from the annotations provided in RefSeq and clustered them

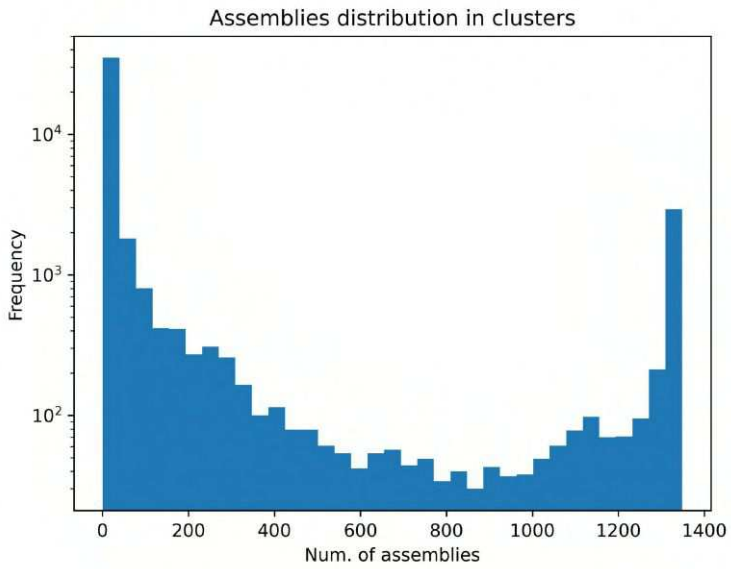


Figure 2.6: Plotting the distribution of samples in the clusters. As expected, this plot displays a characteristic U-shape. This shape emerges when looking at core and accessory genes in a collection of samples in a species. Here the peak to the left at 1 basically represents the unique clusters where only one sample is represented (accessory genes), the peak to the far right represents the clusters where all the samples were represented (core genes). *Figure taken from [57].*

using `mmseq2` [112] with default parameters, resulting in 44,204 protein clusters. The distribution of the number of strains per cluster shown in Figure 2.6 demonstrates the characteristic U-like shape, which evidences the presence of both core genes that are present in nearly all assemblies (right peak of the plot) and accessory genes that are mostly unique to one assembly or present in only a few (left peak of the plot). Now that we have similar proteins clustered together, `mafft` [137] was used on each cluster to produce a corresponding MSA. Subsequently, we converted each MSA into a corresponding DAG in GFA format, this took `PanPA` about 6 minutes with 10 cores. This collections of graphs, then constitutes our panproteome.

2.4.2 Validating Alignments on a Panproteome of *E. coli*

This section reuses material from [57] of which I am the first author

To validate whether `PanPA` aligns sequences correctly, we randomly selected 32,289 protein sequences from our panproteome. The random selection was done by, first, randomly selecting 10% of all the MSAs representing the protein clusters, then for each MSA chosen, we randomly selected 5% of sequences in that MSA. More details on the random selection can be found in Supplementary Section A.3. This random sample of sequences is then considered as “ground truth”, because we know to which cluster, hence, which graph each sequences

should align against. Moreover, we know the path in the graph that the alignment should follow. We expect, of course, that PanPA aligns each of these sequences to the correct corresponding graph. We constructed a pipeline using Snakemake [174] to run the indexing and alignments steps with a combination of parameters to demonstrate the effect of different parameters on the correctness and accuracy of the alignments.

We define a "mismatched alignment" here, as the highest-scoring alignment of a sequence, but to a different graph than the one the sequence originated from. Figure 2.7 plots the percentage of mismatched sequences against the different indexing combinations. When $w = 1$, this is equivalent to a k-mer index, because if the window size is one, then we are taking every k-mer in the sequence. We see that when $k = 3$, we get a relatively high number of mismatched alignments, unless the index stores all the seed hits (unlimited in the seed hits limit). Whereas higher k values produce very few wrong alignments regardless which cutoff was used for the index. From these results, we can recommend a k value larger than 3 when aligning against closely related species, and a cutoff of 5 on the index can be used without losing too many alignments. For full sensitivity, we recommend using a small k and not limiting the index to keep all seed hits. However, this will result in a longer alignment time as many more alignments need to be performed. Supplementary Figure A.3 shows the index size with different seed parameters, and we see that the number of graphs allowed per seed does not have a major impact on the index size, compared to the choice of the seed, i.e., whether we choose k-mer or (w, k)-minimizer.

2.4.3 Runtime for the *E. coli* Panproteome

This section reuses material from [57] of which I am the first author

Figure 2.8 shows the system time in seconds measured for aligning the previously mentioned sample of 32,289 sequences aligned against the *E. coli* panproteome. We see in the figure that the time differs tremendously depending on the k and w values chosen for the index, and the limit of the index seeds and the alignment hits limit. Most notably, we see that using a small k value, results in much higher alignment times due to the many hits generated, as a small k will results in too many hits to many graphs. However, when we increase the k value, we will have more unique seed matches that would point PanPA to align to the relevant graph. For this figure, the time hits a maximum of around 23,000 seconds for using $k = 3$ and $w = 1$ with unlimited seed index, and 30 alignment hits limit, this also results in over 99% of the sequences being aligned correctly. On the other hand, we can also get over 99% of the sequences aligned to the correct graph with only around 200 seconds when using an index with $k = 9$ and $w = 5$ and an unlimited seed index. Of course, this might not be realistic when aligning sequences across the phylogenetic tree, where longer seeds will be too specific to find hits and one needs to use smaller seeds to find matches.

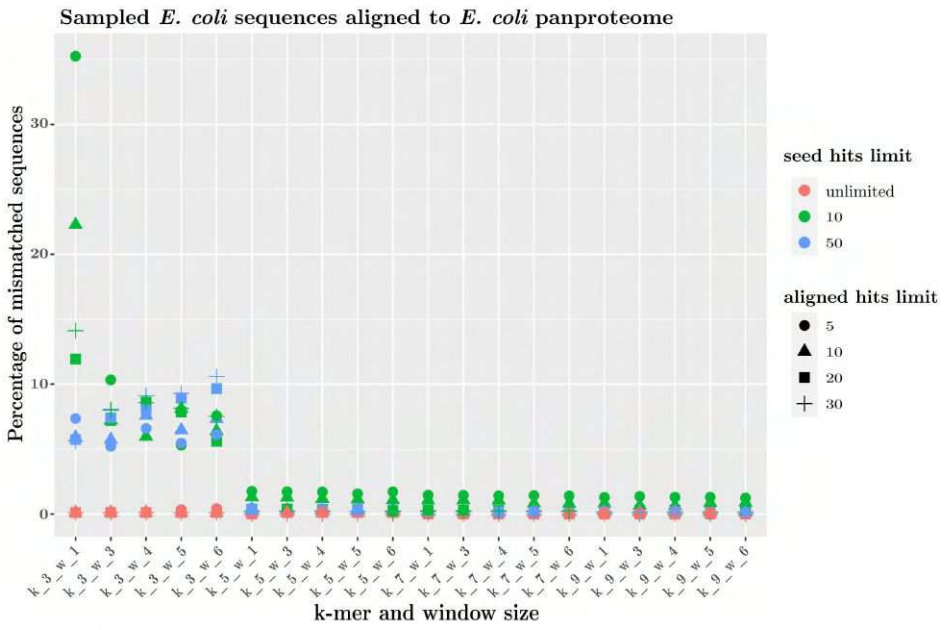


Figure 2.7: Effect of the different parameters on the fraction of mismach alignments, where sequences aligned to the wrong graph. Each point is colored with respect to the seed hits limit (the limit of how many hits can each seed point to), and shapes correspond to the aligned hits limit (the limit of how many graphs can one sequence align to). We see that for a small k values, a high number of wrong alignments is produced, unless the index size is limited. We also notice that the align seed limit has a relatively small effect on the percentage of wrong alignments. *Figure taken from [57].*

2.4.4 Alignment Robustness Validation

This section reuses material from [57] of which I am the first author

To further test the robustness of PanPA's alignments, and how well does it handle mismatches and indels in the query sequence, we used the graph that represent the MSA of the protein GyrA used in later Results Section 2.5.4. We then aligned the 1,392 protein sequences of GyrA that contains both antibiotic resistant and susceptible strains back to the graph. With each alignment iteration, we randomly added errors to the sequences at a rate of 5%, i.e., We replaced some amino acids at random positions with some other random amino acid. The first iteration had 0% error rate and we did 10 iteration to reach a 50% introduced error rate.

For each alignment run, we compared the alignment's path in the graph to the path of the original sequence, moreover, we also looked at the alignment identity. Looking at Table 2.1, we see that when there are no errors, all alignments match exactly the original paths of that sequence, with an alignment identity of 1 (i.e., 100% aligned positions), as expected. The more errors we added, the more the alignment diverged from the original path. However, we see that the average alignment identity is consistent with the percentage of errors introduced. Strikingly though, the average path coverage is less susceptible to the

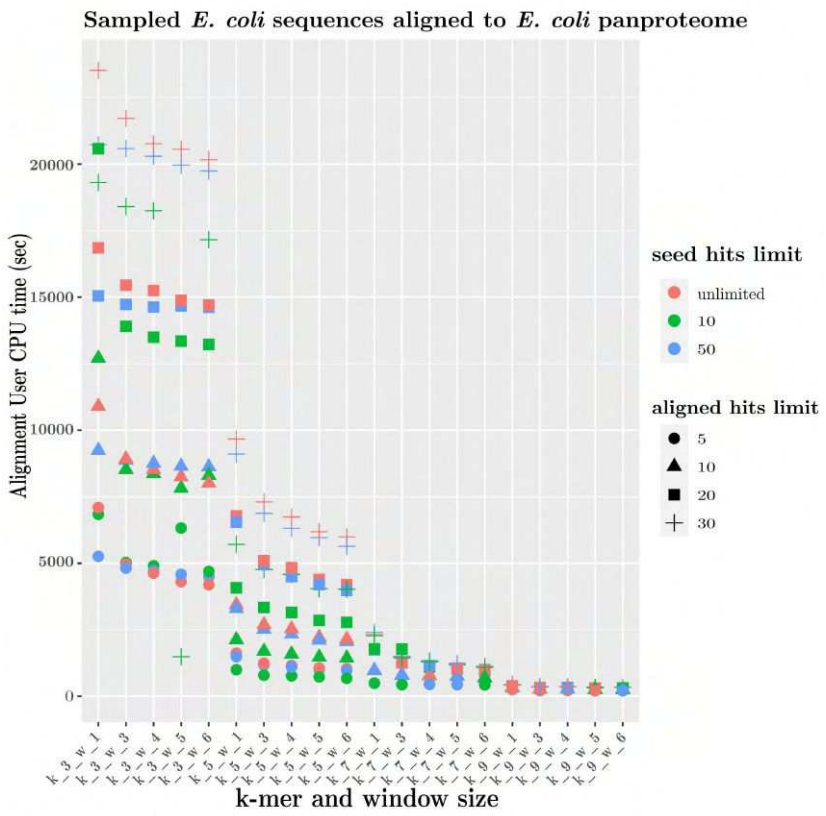


Figure 2.8: The effect of the different k and w value combinations on alignment's User CPU time on the sampled sequences. We see that small values of k results in much more time, due to the fact that smaller k values produce more promiscuous seeds to match to many graphs, so `PanPA` needs to spend more time aligning to these graphs then filtering out the alignments with low scores. However, we can still get close to 100% correct alignments when using unlimited seed hits, but then the time increases dramatically. On the other hand, when using a bigger k value, the seeds will have a more unique hit to the correct graphs and `PanPA` doesn't need to spend too much time aligning. *Figure taken from [57].*

errors. This, probably, stems from the fact that in the MSA there were many stretches of conserved sequences which results in one node after compacting, and when errors are introduced in the sequence that would align to that node, it would still align to the node but with mismatches.

2.5 Results

2.5.1 Aligning Unseen Sequences from *E. coli*

This section reuses material from [57] of which I am the first author

Using the same panproteome constructed in Section 2.4.1, we further downloaded 80 *E. coli* assemblies from RefSeq that were not used in building the panproteome as they were not

Number of sequences	Perc. of error introduced	Matching paths	Mismatching paths	Average alignment identity	Average path coverage
1392	0	1392	0	1	1
1392	5	265	1127	0.952	0.988
1392	10	50	1342	0.905	0.98
1392	15	12	1380	0.857	0.971
1392	20	2	1390	0.809	0.962
1392	25	0	1392	0.762	0.952
1392	30	0	1392	0.715	0.94
1392	35	0	1392	0.667	0.931
1392	40	0	1392	0.62	0.919
1392	45	0	1392	0.575	0.901
1392	50	0	1392	0.529	0.889

Table 2.1: Inserting random errors to the Gyra sequences before aligning back to the graph constructed from the MSA of the same query sequences. We see that the Average alignment identity follows properly the percentage of errors introduced, which further indicates that PanPA is aligning the sequences properly. Moreover, we see that when there are no errors, the alignment path matches the correct path of the sequences in the graph. *Table taken from [57].*

marked as complete assemblies, and extracted the protein sequences from the corresponding annotation files. After removing redundant sequences, we were left with 92,196 sequences. We used the same Snakemake pipeline as in the previous experiment to align these sequences against the panproteome with the different parameter combinations. To consider an alignment correct, we require that its alignment identity be above 90%. After aligning, we got an average alignment identity of about 99.8%.

We observe again that for small values of k, the majority of sequences (between 50% for $k = 3$ and $w = 6$ and 99% for $k = 3$ and $w = 1$) did not produce an alignment (Figure 2.9). These results emphasize the conclusion from the previous experiment, that choosing a very small size for the seeds (e.g., $k = 3$) and limiting the index hits size will result in a high number of false positive index hits that; in turn; will result in alignments with a low identity that will be filtered out due to low identity scores. When the index hits size is unlimited, PanPA is then able to produce better alignments. Keeping in mind, that an unlimited index will result in a much longer alignment time as there is a need to align to more sequences. For example, for $k = 3, w = 1$, and unlimited index, it takes PanPA over 80,000 seconds of User CPU time to finish alignments compared to slightly over 1,000 seconds with $k = 9$ and $w = 1$, all combinations are shown in Figure 2.10.

2.5.2 Comparison of PanPA, BWA and GraphAligner Using *S. enterica* Sequences

This section reuses material from [57] of which I am the first author

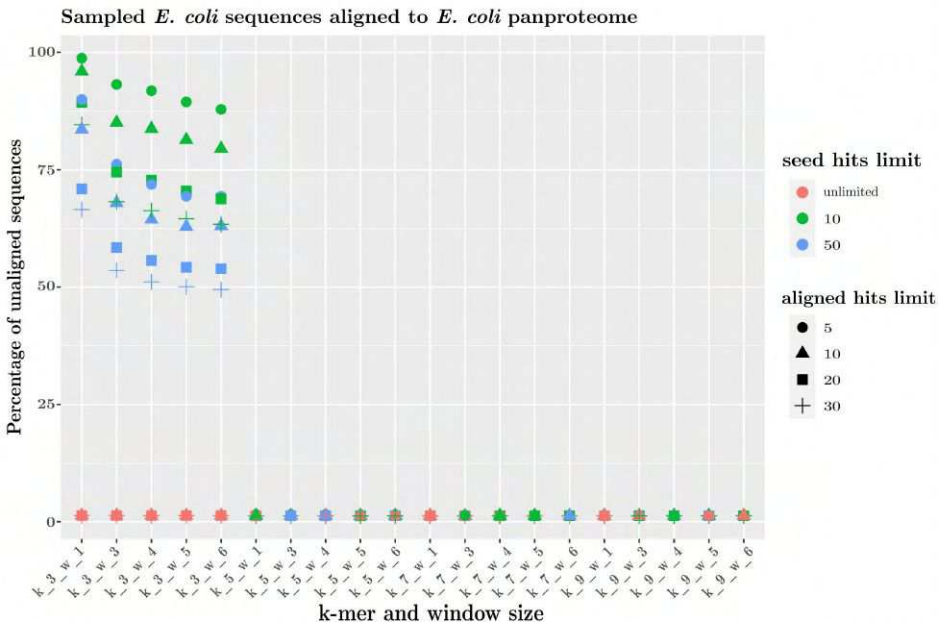


Figure 2.9: Effect of the different parameters on the number of unaligned sequences when aligning 92,196 unseen *E. coli* sequences. For small k values, the majority of sequences were not aligned unless a limit for the index hits size is set (the red marks); if the index hits size is not limited, over 99% of sequences produce an alignment. *Figure taken from [57].*

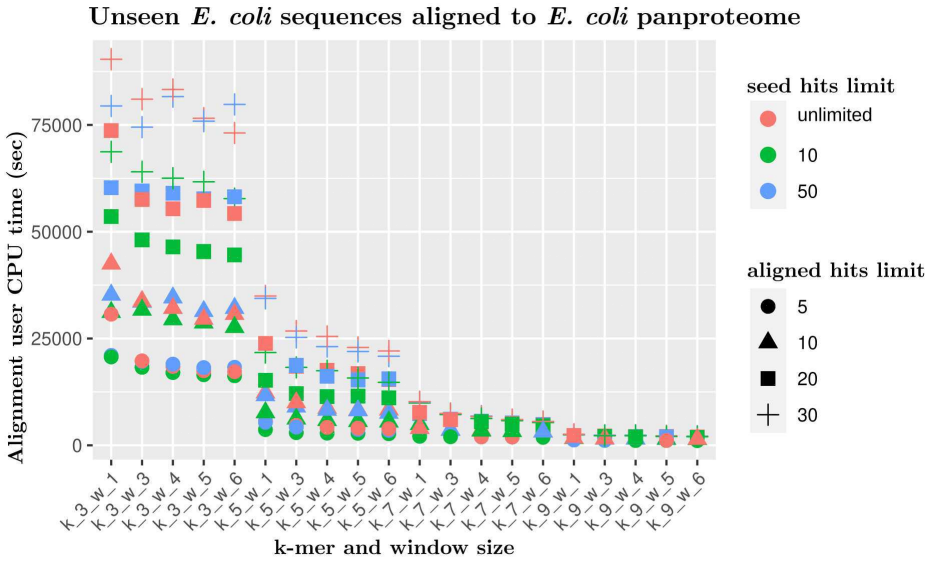


Figure 2.10: Unseen sequences alignment speed with the different indexing parameters. We can clearly see that for small seeds, the alignment time increases dramatically, due to the fact that smaller seeds are very promiscuous and can have hits to too many graphs, resulting in performing many alignments that ultimately result in low identity scores and be filtered out. *Figure taken from [57].*

One major advantage of moving from DNA to the amino acids space, is the ability to have better alignments between more phylogenetically distant organisms. Both *E. coli* and *S. enterica* belong to the same family *Enterobacteriaceae*, but to different genera, hence, are expected to be far enough apart from each other evolutionary to make a good test case for PanPA.

In order to compare DNA and protein alignments, we downloaded 1,078 annotated assemblies of *S. enterica* from RefSeq, and extracted all DNA sequences of coding regions and their corresponding amino acid sequences from the *S. enterica* annotations, obtaining 4,839,981 sequences. We compared three types/methods of alignments here:

1. DNA sequence alignments against the *E. coli* linear reference genome (strain K-12 substrain MG1655) using BWA [152].

2. DNA sequence alignments using against the *E. coli* pangenome GraphAligner [216]. The was graph built from all 1,351 assemblies and was constructed using minigraph [154]

3. Amino acid sequence alignments using PanPA against the *E. coli* panproteome constructed from the same assemblies.

Both BWA and GraphAligner were run with their default parameters, and PanPA was given an index with $k = 5$, $w = 5$, an index limit of 10, and only aligning each sequence to the top 10 graph hits. The alignments were then filtered based on alignment length and alignment identity. Only alignments with a length of over 50% of the original sequence length and alignment identity of at least 50% were kept.

Figure 2.11 is an upset plot showing the intersection between the alignment results of the different aligners. Out of the 4,839,981 sequences, 1,638,936 were successfully aligned by all three aligners, while 1,694,181 could only be aligned by the graph-based methods GraphAligner and PanPA. Strikingly, PanPA could align a further 744,033 sequences that were not aligned by any of the other two aligners.

Furthermore, Figure 2.12 shows the distribution of alignment identity across the three aligners tested. We can see that PanPA's alignments have higher identity scores, which is to be expected as in the amino acid space sequence identity is higher.

For this experiment, PanPA required around 17 minutes to build the index, and around 5 hours to align the sequences, using 2.3 Gb memory. BWA only took around 6 minutes to run and needed around 900 Mb of memory. GraphAligner needed around 20 minutes to run and used around 700 Mb of memory. All of the tools were run with 20 cores. PanPA did take more time to perform the alignment compared to the other tools. However, PanPA was able to align more sequences, and more importantly, due to the fact that PanPA uses a non-linear substitution matrix instead of simple edit distance in the alignment algorithm, certain algorithmic speeding tricks cannot be used by PanPA. We elaborated more on this point in the Section 2.6. Supplementary Table A.2 contains the raw numbers of alignments

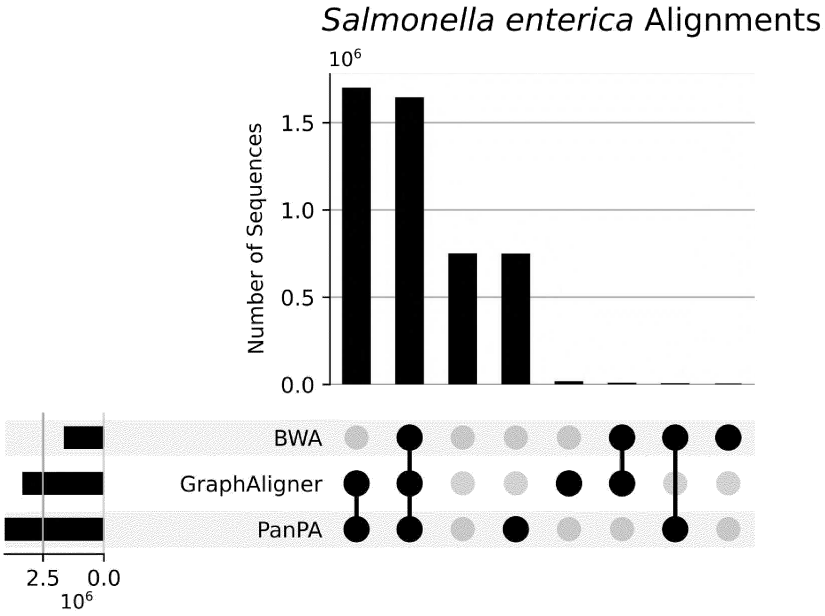


Figure 2.11: Upset plot of the alignments of 4,839,981 sequences from the coding regions of 1,074 *S. enterica* assemblies from RefSeq against *E. coli*. *Figure taken from [57].*

and intersections presented in Figure 2.11, and Supplementary Table A.1 contains the raw alignment numbers of the three aligners used in this experiment before and after filtering.

2.5.3 Aligning *S. enterica* Illumina Short Reads to the *E. coli* genome, pangenome, and panproteome

This section reuses material from [57] of which I am the first author

As explained in Section 2.2.3.2, `PanPA` can perform a frameshift-aware DNA alignment against the amino acid graphs. To find the candidate graphs to align against, similar to amino acid query sequences, k-mer or (w, k)-minimizer seeds are extracted. `PanPA` then uses the index to find to the potential graphs to align against. Since we do not know from which strand the DNA sequences are from, `PanPA` also aligns both the DNA query sequence and its reverse complement as well. If the reverse complement was aligned, this will be reported in the output GAF file with the tags `st:Z:forward` or `st:Z:reverse`. To test this, we downloaded one *S. enterica* Illumina whole genome sequencing (WGS) short reads sample (SRR22756191) from NCBI SRA database [145] containing 1,110,471 sequences, this sample is part of PulseNet USA surveillance for food-borne diseases. We proceeded to align the DNA sequences using `BWA` against the linear reference of *E. coli*, and against the *E. coli* panproteome using `PanPA`. We used an index with $k = 5$, $w = 3$, no seed hits index limit, and a limit of 20 graphs for the alignment. We filtered the output retaining alignments with greater than 50% alignment sequence identity. `BWA` was used with default parameters. As expected, using a distant linear reference has a major disadvantage: around 65% of the

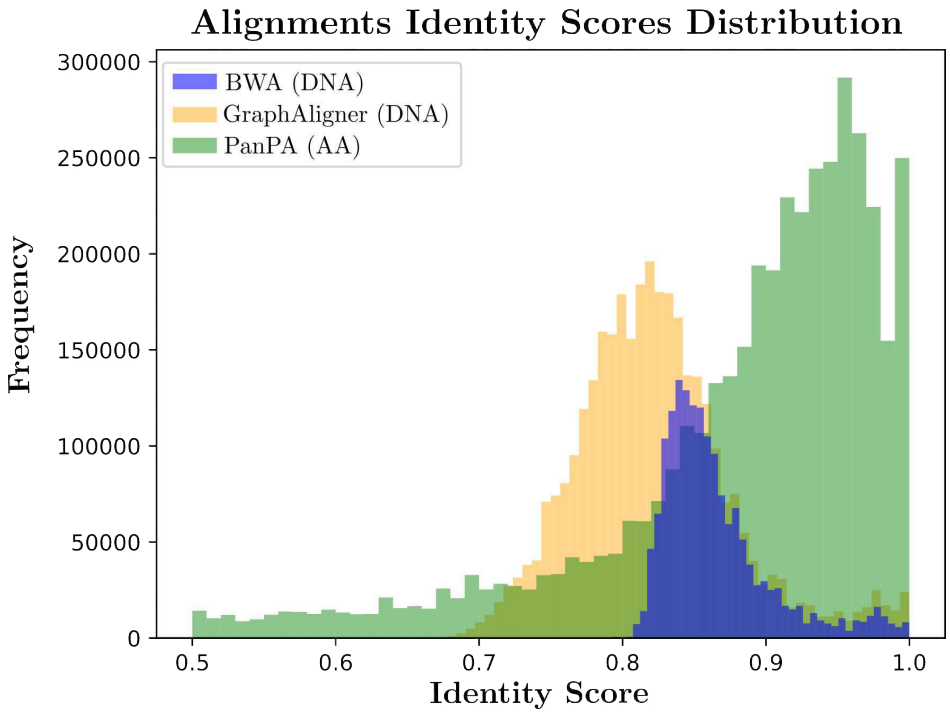


Figure 2.12: Distribution of identity scores between `BWA`, `GraphAligner`, and `PanPA` from aligning the *S. enterica* sequences. The pique for `PanPA` is shifted to the right, meaning higher sequence identity, as amino acid sequences align with higher identity compared to nucleotide sequences. *Figure taken from [57].*

	Identity >50%	**Identity and Length >50%**
BWA	391,041 (35.2%)	48,937 (4.4%)
PanPA	801,389 (72.2%)	755,009 (68%)

Table 2.2: Number of *S. enterica* DNA short reads aligned against *E. coli*'s linear reference with BWA and against its panproteome using `PanPA`. *Table taken from [57].*

reads could not be aligned with `BWA` with identity over 50%; after additional filtering requiring alignment length to be over 50% of the length of the DNA sequence, only 4.4% were reported (Table 2.2). On the other hand, `PanPA` was able to produce alignments for 72% of the reads with identity over 50%, and 68% of sequences could be aligned with over more than 50% of their length. 355,462 sequences were not aligned by either aligners. In this experiment, `PanPA` needed about 6 hours to align the DNA sequences using 10 threads, and used about 1.8 Gb of memory. `BWA` only took 17 seconds to run with 10 threads (CPU time 162 seconds). Even though `PanPA` took significantly more time to run, it was able to retrieve way more alignments than `BWA`. Moreover, the memory requirement was low enough that it can easily be run on a modern personal computer in the background, or given more threads on a high-performance computation cluster to increase the speed.

2.5.4 Using PanPA to Display Phenotypic Traits: a Case of Antimicrobial Resistance in *E. coli*

This section reuses material from [57] of which I am the first author

In prokaryotes, certain mutations can be associated with resistance or susceptibility to antibiotics. This has been a main focus of many researchers, as resistance against antibiotics presents a major threat to public health [59]. We explored the applicability of PanPA to identifying such mutations. To this end, we used the Pathosystems Resource Integration Center (PATRIC) [63] database to downloaded ciprofloxacin-resistant and susceptible strains from *E. coli*. This database contains assemblies and annotations for many antibiotic-resistant and susceptible bacterial strains. The dataset we obtained comprised 556 resistant and 1,295 susceptible genomes. We extracted two genes, *parC* that encodes the A subunit of topoisomerase IV, and *gyrA* that encodes the DNA gyrase subunit A. Mutations on both are associated resistance to quinolones, and particularly ciprofloxacin in *E. coli* [273, 14]. For each of these two proteins, we were able to extract 1,236 susceptible and 309 resistant sequences, and randomly split the sequences into two sets, one containing 10% of the sequences and the other 90% of the sequences. Subsequently, we mixed the 90% sample of both susceptible and resistant together, generated an MSA using mafft, we then generated a graph for each protein using PanPA. Figure 2.13 shows examples of mutations apparent in the graphs, these resistance-associated mutations (S83L, D87N in GyrA [282, 214, 292], S80I in ParC [182]) cause the generation of bubbles in the graph. Besides these canonical resistance-associated variants, we observed other potential variants that are present predominantly in resistant strains: alanine, leucine, and valine at position 83 and alanine, tyrosine, and asparagine at position 87 of GyrA, as well as arginine at position 80 of ParC. We aligned the 10% sequence set aside to the graphs using PanPA. Visualizing the corresponding paths in Figure 2.13, one can see that the vast majority of the sequences extracted from resistant strains are aligned to the nodes that correspond to variants associated with resistance, and susceptible sequences aligned to mostly nodes associated with susceptible variants.

2.5.5 Comparing against HMMER

This section reuses material from [57] of which I am the first author

HMMER is a widely used tool for searching remote homologs in protein databases [81]. HMMER has a high sensitivity, which renders it very useful for aligning sequences that have lower similarity due to their large phylogenetic distance from the target. In brief, HMMER builds a hidden Markov model profile for each MSA given, which is then used for aligning a new sequence against the profile.

To compare PanPA's performance with HMMER, we consider each protein cluster as a separate profile. HMMER can be then used to align new sequences against these profiles and

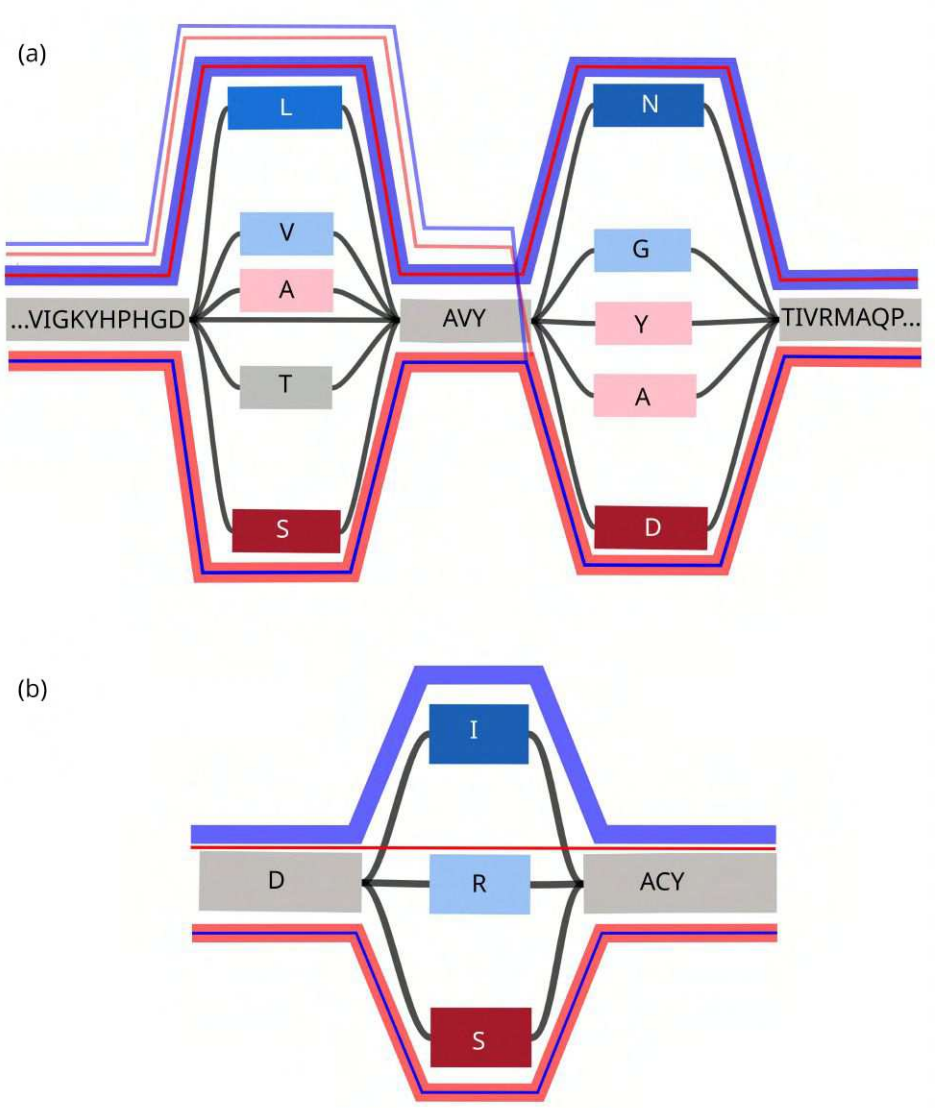


Figure 2.13: Visualization of parts of the protein graphs for (a) GyrA and (b) ParC using `Bandage` [289]. Nodes are colored according to the number of resistant/susceptible strains that pass through them, with blue color representing resistance, and with red representing susceptibility; the color intensity corresponds to the number of strains. Additional colored lines show the paths of the aligned 10% sequence that were set aside (45 resistant and 117 susceptible sequences), the color representing the type, and the thickness representing the number of sequences taking that path. A thick blue line of resistant sequences took the blue path passing through the blue nodes, and vice versa, a thick red line for susceptible sequences took the red path passing through the red nodes. *Figure taken from [57].*

choose the best hits. More formally, we performed two comparative steps between HMMER and PanPA:

1. Building HMM profiles in HMMER, and generating graphs and an index in PanPA, as both are preprocessing steps before doing alignments.

2. HMMER search step and PanPA's alignment step, as HMMER search also produces alignments.

Again, we used the 44,204 protein clusters of the *E. coli* sample we have from previous experiments (Section 2.4.1). For PanPA, we used the (w, k)-minimizer index for the clusters with $k = 5$ and $w = 3$. Building HMM profiles from the same MSAs with `hmmbuild` command of HMMER took 2 hours, 46 minutes, and 18 seconds for all 44,204 clusters. As HMMER runs separately on each MSA, only one thread was used. However, one can use a bash script or a Snakemake pipeline for example to run several MSAs at the same time on different threads.

For aligning, we extracted a random sample of 10,000 protein sequences from the *S. enterica* sample we used in the experiment described in Section 2.5.2, and aligned these sequences to graphs and HMMs, for PanPA and HMMER, respectively.

PanPA needed 20 minutes and 57 seconds to align all 10,000 sequences, with a minimum alignment identity threshold of 10%. Using 10 threads brought the time down to 7 minutes and 25 seconds. PanPA always spends about 5 minutes loading all the graphs into memory before alignments, which means the more sequence are aligned, the smaller this overhead's effect relative to the total runtime. PanPA used 2.2 Gb of memory, and the number of query sequences does not affect the memory profile. As for HMMER, it took 19 minutes and 29 seconds to align all 10,000 sequences with `hmmalign` against the database of HMM profiles constructed previously, and used about 1 Gb of memory and 10 cores.

Comparing the results, we found that 9,813 of the query sequences were aligned to the same target cluster by both tools. 187 query sequences were aligned by PanPA, but not by HMMER. However, these 187 sequences had a very low alignment sequence identity averaging at 25%, which can explain why HMMER might have filtered these out. The major reason for PanPA performing faster compared to HMMER, is the use of the index that guides PanPA on where to align and thus reduces the search space considerably. HMMER, on the other hand, aligns each query sequence to each profile, which makes the runtime linear in the number of clusters. PanPA's ability to run in multiple threads also has a major role in reducing the alignment time. For example, in this alignment experiment, the actual alignment time for PanPA (excluding the graph loading step) was 15 minutes and 47 seconds using 1 thread, but only 2 minutes and 13 seconds when using 10 threads.

In conclusion, for the preparation step, PanPA needed, in total, around 24 minutes to generate both the index and the graphs, HMMER on the other hand needed around 2 hours. For the aligning step, PanPA needed around 7 minutes to align all sequences and HMMER

needed around 19 minutes. More details about time and memory requirements for this experiment are in Supplementary Table A.3.

2.5.6 Gene Order Analysis with PanPA

It has been shown that the order of genes or coding regions in prokaryotic genomes has some significant effects, particularly regarding aspects such as genome organization, gene function prediction, and evolutionary conservation. For example, looking at evolutionary conservation, in [222], they found that although gene order is less conserved compared to the amino acid sequence of genes, specific operons or genes that are transcribed together remain more conserved. These types of conservation can provide useful insights into the evolutionary relationships between different organisms. In [68], they analyzed many microbial chromosomes and found connection between GC-content in regions between the operons is higher compared to other non-coding regions, and they concluded that this is related to the ability to conserve the operonic gene order. Moreover, gene order and gene order conservation can be very useful in predicting the function of unknown genes and proteins. This case was made in [260], where genes that maintain the same order even over long phylogenetic distances, can indicate very strong evolutionary pressure to keep these genes together. This occurs despite lateral gene transfer events. The conservation and importance of gene order has also been studied in plants [12, 274] and eukaryotes [69] as well.

One way to look at the gene order, is to build a graph where each node represents a gene, and each directed edge between two nodes represents the order of these two genes in relation to each other. This idea was demonstrated also in [84], where they developed an algorithm to detect conserved gene clusters and align orthologous gene orders. More recently, this was also demonstrated in [156], where they generate the gene graphs from long-read alignments against a linear reference genome, and they built a gene graph from the alignments. As PanPA is able to align DNA sequences to amino acid graphs, and is able to perform local alignments, i.e., if part of the query sequence aligns to one graph, and another part aligns to another graph, both alignments will be reported separately in the output GAF file. Therefore, we are then able-for a long query sequence that goes through several graphs-to order the graphs or proteins that the query sequence aligns to. Thus, PanPA is also able to bring the idea of gene graphs to the panproteome world.

To demonstrate this, we used some of the genes and prokaryotes used in Figure. 5 in [222]. In their Figure, they show a gene order table, where they used *E. coli* as the template and compared the gene order of other organisms against it. We reused the panproteome of *E. coli* built in Section 2.4.1. First, we wanted to check whether we can retrieve the same order presented in [222], to do that, we aligned the *E. coli* reference with accession number "GCF_000005845" back to the panproteome. However, as PanPA builds a complete DP table in the alignment step, using the complete reference genome as a query sequence

would be extremely costly. Therefore, to solve this, we simply cut the reference genome into overlapping windows of 10kb, with an overlap of 5kb. The reason we kept an overlap is to make sure that the windows extracted always include a complete coding region. For example, if our window is from position 5,000 to 15,000 and half of the coding region is before the position 15,000 and the other half is after, then we lose that coding region in the alignment, but this coding region will be captured in the next window from 10,000 to 20,000.

Using this idea, we generated 929 DNA query sequences of 10kb each from the *E. coli* genome, and then aligned them back against the *E. coli* panproteome. This took around 5 minutes for and used only 100 Mb of memory. Looking at the output alignments, we first ordered them based on the subsequence original coordinates, then, if more than one protein graph aligned to the same subsequence, we ordered the aligned proteins based on their location on the subsequence alignment. Additionally, if the reverse complement of the DNA subsequence is aligned, then we need to reverse the order of the genes. This resulted in the same order of genes shown in [222] (Shown in Figure 2.14 following the thick black arrows), i.e., `PanPA` was able to retrieve the correct gene order from aligning DNA sequences to a panproteome.

To compare other organisms, we downloaded four reference genomes of the following organisms: *Bacillus subtilis*, *Mycobacterium tuberculosis*, *Haemophilus influenzae*, and *Thermotoga maritima* with the RefSeq accession numbers GCF_000009045, GCF_000195955, GCF_000165525, and GCF_000230655 respectively. Similar to how we processed the *E. coli* reference genome, we extracted overlapping DNA subsequences from each assembly, aligned them back to the *E. coli* panproteome, and looked at the gene orders in the resulting alignments. The results are shown in Figure 2.14. We colored each species with a different color, and gave *E. coli* thicker, darker arrows because it forms the backbone of the graph generated. We see that the more phylogenetically distant the organism is, the fewer matches we found (e.g., *B. subtilis* and *T. maritima*). However, for closer organisms such as *H. influenzae*, which is also under in the same class as *E. coli* (Gammaproteobacteria), we see that they share most of the genes, but in almost completely the opposite order.

With this simple example, we were able to show that `PanPA` is, indeed, capable of not only performing fine-grained alignments against the panproteome graphs. Moreover, it can also utilize long and accurate DNA reads to extract the gene order and generate gene graphs. As mentioned at the beginning of this section, gene order graphs are interesting and important to study in terms of evolutionary relationship between these genes, and there are already recent studies looking into this further. Therefore, bringing this to the panproteome world would be interesting and useful.

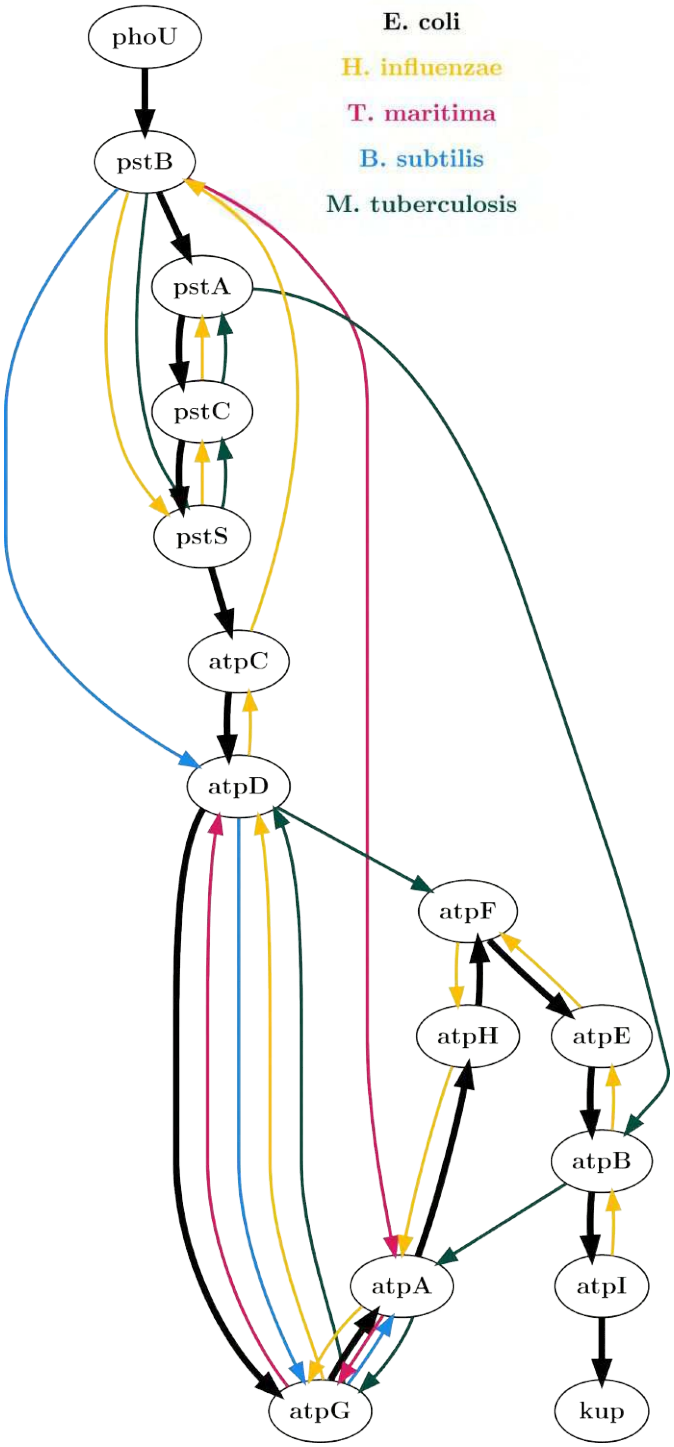


Figure 2.14: Gene order graph using the genes from [222], where the *E. coli* pangenome graphs for these genes are used, then the reference assembly of each of the organisms mentioned in the figure are aligned back to these gene graphs. Following the thick black arrows, that follow the *E. coli* assembly alignment, we recreate the same order in [222], which further validates that our method can capture the correct order of the genes.

2.6 Conclusion and Discussion

In this chapter we presented PanPA, a command-line toolkit for building panproteomes, indexing them, and aligning DNA and amino acid query sequences against them. We showed that building individual graphs to represent proteins and using this collection of graphs as a "reference" yields several positive results. To explore this further, we first investigated whether the idea behind PanPA works, and whether it has the ability to produce correct alignments. This was demonstrated in Section 2.4, where we showed that PanPA produces correct alignments when aligning the same sequences back to the panproteome graphs. We further argued that aligning over longer phylogenetic distances has its importance, especially for prokaryotes that are not well studied or do not have a standard reference, and therefore, moving to an amino acid space can increase both the number of alignments and alignment identity, which was then demonstrated in Section 2.5.2, where Figure 2.12 illustrated that alignments of *S. enterica* against *E. coli* panproteome did indeed have higher alignment identity. Figure 2.11 represented the intersection of the numbers of alignments captured, where PanPA was clearly able to align more sequences compared to the aligners in the DNA space. Later, in Section 2.5.3 we aligned DNA reads against the *E. coli* panproteome, and showed that PanPA is also able to handle DNA alignments, even if they have indels that result in a frameshift, as PanPA is able to perform frameshift-aware alignments as explained in Section 2.2.3.2. Furthermore, in Section 2.5.4, we showcased the effectiveness of PanPA in uncovering genetic mechanisms underlying phenotypic traits, including antimicrobial drug resistance. We see that this can be very useful in aligning new sequences to very well studied panproteomes, and be able to extract phenotypic traits or annotate sequences that were not aligned in the DNA space. In Section 2.5.5, we compared the alignments of PanPA with HMMER, which is a very famous tool for homologs search and can also align a protein sequence to protein HMM profiles. Comparing the results, we see that both PanPA and HMMER almost completely aligned the sequences to the same graphs/profiles, with only very few alignments that HMMER was not able to align. However, this does not necessarily points out a weakness in HMMER, as these few alignments had a very low alignment identity score. As a last use case, we showed in Section 2.5.6 how PanPA can reconstruct the gene order using long and accurate DNA sequences aligned against the panproteome, by utilizing the alignment locations, a gene graph can be constructed where the nodes represent genes and edges represent the gene adjacency in the genome being investigated. Such a gene graph was shown in Figure 2.14.

We demonstrated that PanPA operates efficiently in terms of computational resources. It is readily deployable on any modern laptop or desktop without requiring access to high-performance computing clusters. Additionally, PanPA supports parallelization, allowing users to high performance computational clusters to significantly accelerate the alignment process. However, one drawback of PanPA, is that it performed slower compared to other

linear aligners it was compared against, namely BWA and GraphAligner. One reason is that PanPA constructs a full dynamic programming (DP) table, fills all its cells, and employs several substitution matrices with different scores instead of relying on edit distance. Consequently, PanPA cannot utilize optimization techniques such as bounded edit distance [270] or the fast bit-vector algorithm for string matching [178], which has also been extended to graphs [217] and allows for much faster alignments. The performance bottleneck of PanPA is not in the number of graphs within the panproteome, but in the size of these graphs and the sparsity of their corresponding MSAs. For example, Supplementary Section A.4 shows that PanPA can handle very sparse MSAs, however, at a reduced speed. Therefore, further algorithmic improvements and optimizations are needed to enable PanPA to handle more complex graphs, such as those representing alignments of sparse protein families. Nevertheless, PanPA still performed effectively on real datasets, and its small memory footprint allows it to run on local machines or small compute nodes, where the use of additional CPUs can further speed up the alignment process. Another improvement or an avenue for further investigation, is the downstream analysis of the alignments produced by PanPA. In this chapter, we have mainly focused on the inner workings of PanPA and on validating that the alignments produced are correct and useful in different settings. However, panproteomes can tackle a problem such as annotation, e.g., genetic and coding region information can be embedded in the graph's paths; when reads or contigs are then aligned, one can use the alignment path to infer characteristics about the aligned sequence. Moreover, one can further investigate the usefulness of PanPA in helping to disentangle metagenomic sequencing samples, e.g., by analyzing the alignments, one can better cluster the sequencing reads as a preprocessing step to metagenomic assembly. Another avenue to explore further is the use of the alignments for small variants detection and calling, This was shown for example in [47], where they were able to recover more rare SNPs using a pangenome of *E. coli*, which would otherwise not be detected with conventional linear reference methods.

Chapter 3

Software Toolkits for Genome and Pangenome Graphs

This chapter introduces methods and software toolkits for working with both GFA graphs and GFA sequence-to-graph alignments. The toolkits all share a common graph infrastructure and are outlined below:

(1) `GFASubgraph` [53] is a command line tool and an Application Programming Interface (API) for working and manipulating GFA graphs, it provides important functionalities for the user such as reading, writing, finding subgraphs, removing nodes, edges, and other utilities for working with GFA graphs. A basic implementation of the GFA graph API was first developed as part of [56]. (2) `extgfa` is also a GFA graph command line tool and API that uses parts of the graph API from `GFASubgraphs`. `extgfa` however, allows for an external memory representation of the graph which enables the user to investigate large graphs without having to load it completely into memory RMA/Memory. (3) `gaftools` is a command line tool for working with pangenome sequence alignments in GAF format. `gaftools` provides the user with several utilities for working and manipulating GAF files, such as viewing (subsetting), sorting, realigning, changing coordinates type, and other utilities. My role in this project was mainly focused on implementing internal functionality and classes for working with the GFA graphs, especially sorting the graph, which is a precursor for other functionalities in `gaftools`.

The work in Section 3.3 related to the `extgfa` tool, reuses materials from the preprint [52] of which I am the sole author. The work in Section 3.4 related to `gaftools` reuses materials from [196] in which I am a co-author. Specifically, I contributed to the GFA class infrastructure of `gaftools`, GFA ordering (Section 3.4.1.1), realignments parallelization (Section 3.4.1.3), and in writing the manuscript. Figures 3.7 and 3.6 were done by myself, Table 3.2 is a joint contribution, and Table 3.3 is a contribution of the other authors of the paper.

3.1 Introduction

This section reuses materials from [52] of which I am the sole author, and [196] of which I am co-author.

As discussed in Sections 1.2.2 and 1.3, genome graphs are an important component in modern genome analysis. Especially with the recent transition from a linear reference to a graphical one, it led to a growing demand for software tools that can effectively perform various important analysis tasks. Tasks such as storing, processing, and analyzing the graphs efficiently, particularly as the size of the graphs increasing with more data. For instance, looking at some of the graphs generated by the Human Pangenome Consortium [157], e.g., the raw graph produced using the Minigraph-Cactus method [120] has a file size of 48 GB that contains 92,879,580 vertices, and the graph produced using the Pangenome Graph Builder pipeline (PGGB) [89] has a raw size of 89 GB and contains 110,884,673 vertices. Therefore, it is not a trivial task to be able to work with such graphs effectively, especially given that the graph size will only increase with the addition of more assemblies. To address these challenges, several software toolkits have been developed for working with large genome graphs [88, 151, 89]. Additionally, algorithms for search, subgraph detection, and indexing of genome graphs have been developed [248, 136, 56]. These toolkits, however, generally load the complete graph in Random Access Memory (RAM), even when if only a small part of the graph is required. Despite this plethora of tools and software implemented to analyze genome graphs, there remains a significant gap in the availability of user-friendly, interactive software libraries that allow users to investigate these graphs.

Since graph data structures have been studied in the computer science field for decades, and the obstacle of processing large graphs is not particularly a new one, computer scientists have investigated the possibilities of using external or disk memory instead of RAM, e.g., external-memory breadth-first search [168], external-memory depth-first search [117], and other external-memory algorithms [38]. However, in these theoretical studies, researchers have mostly focused on extending a specific algorithm to allow external memory, but have not presented a multipurpose external-memory graph data structure in which any graph algorithm can then be implemented.

In this chapter, we present several graph toolkits geared towards working with genome graphs, and alignments against these graphs. First, we introduce `GFASubgraph`, a simple `Python` toolkit and interface for manipulating graphs, it provides functionalities such as reading, writing, extracting subgraphs or neighborhoods in the graph, and other functionalities. We then present `extgfa`, a proof-of-concept `Python` toolkit and interface that introduces the idea of an external memory representation of graphs in the Graphical Fragment Assembly (GFA) format. Furthermore, we show that `extgfa` is able to reduce the memory profile by more than one order of magnitude on large graphs. Finally, we introduce `gaftools`, also a

Python toolkit for working with GAF sequence alignments, specifically against rGFA graphs. gaftools introduces several important functionalities that close the gap between manipulating alignments against linear references, and alignments against a graph reference, such as sorting, indexing, viewing, and realigning.

3.2 GFASubgraph and GFA class

GFASubgraphs is a command-line toolkit designed for manipulating GFA graphs. Specifically, extracting subgraphs and graph components into separate GFA files to facilitate visualization of smaller parts of large graphs, or to study components separately further downstream. It is a dependency-free tool that allows for easy installation and deployment on different systems. Internally, GFASubgraphs implements a GFA class API that users can interface with to implement their own algorithms and tools for downstream processing.
The main reason behind developing such a tool was the lack of simple tools for manipulating and working with GFA graphs in Python.

3.2.1 GFA Class

GFASubgraph internally implements an updated version of the GFA API developed in [56]. This Graph class can be simply imported by the user and it offers the ability to read, write, investigate, and manipulate GFA graphs in Python.

Figure 3.1 shows a simple Unified Modeling Language (UML) diagram of the GFA class implementation. We see that the main GFA class simply contains each segment or node as an object stored in a dictionary or a hash table, with the key being the node ID as a string. It also implements simple functionality such as adding, removing, accessing neighbors or children of nodes. The Node class on the right side shows what a node object stores. Genome graphs are bidirectional, i.e., they can be traversed in both directions, taking into consideration the direction of the sequence in the segment. Therefore, in this graph class representation, an edge or a link has four mandatory attributes: (1) “from” node ID, (2) “from” node direction, (3) “to” node ID, (4) “to” node direction. Where the direction can tell us whether we need to take the forward or the reverse complement of the sequence in the node.

In our implementation, edges are stored in each node object, and each node object has two sets: one for edges from the start of the node and one for edges from the end of the node. Consequently, taking one node, its ID indicates the “from” attribute of the edge, for the “from” direction, this is taken based on whether we look at the “start” or “end” sets in the node object, it is binary encoded (0 for start and 1 for end). The tuples in these sets store the “to” node ID , the “to” direction, and the overlap value. Storing edges in the node object directly allows for faster access to the edges compared to other implementations of GFA graphs we have found, where both nodes and edges are stored in separate map data structures causing a single graph operation (such as finding neighboring nodes) to trigger multiple hash calls. However, our model then requires slightly more memory because both

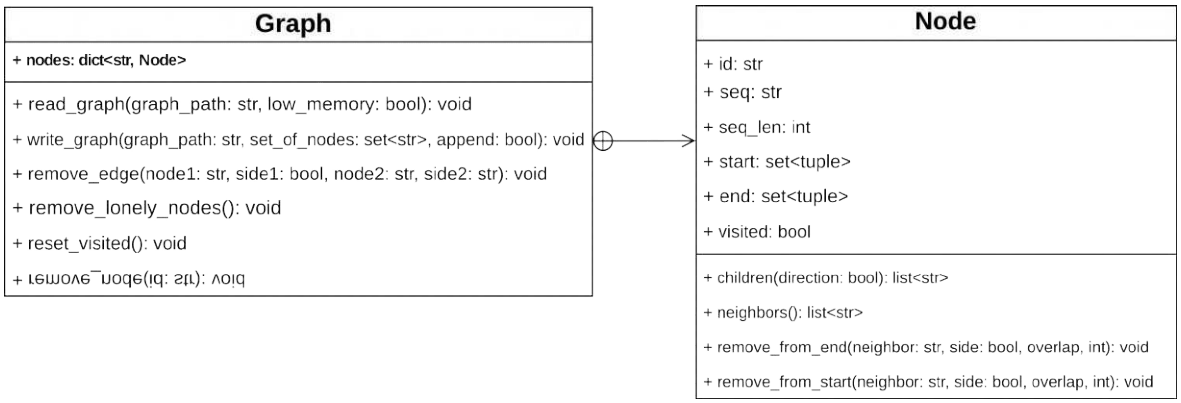


Figure 3.1: This is a simple Unified Modeling Language (UML) diagram explaining how the GFA class is implemented. We can see that the main graph class stores a dictionary of node objects. A node object contains the information related to the node, most importantly, each node object has a start and an end set of edges, where each edge is tuple of `(neighbor_id, direction, overlap)`, where the direction here refers to where the edge enters the neighbor node. The direction here is a binary, referring to 0 for the node start, and 1 for the node end.

directions of an edge are stored in both connected nodes. For example, in the graph shown in Figure 1.6, node "s3" will have one edge from the start connecting it with "s1" and one edge from the end connecting it with "s4". The start edge would then be a tuple encoding `(s1, 1, 2)` where the first element in the tuple is the other node's ID, the second is the direction (0 for start and 1 for end), meaning that "s3" connects to "s1" from its end, and the number 2 indicates the overlap size. The GFA class can also load a GFA graph without loading the sequence (low-memory version), which can save memory especially for graphs that have fewer nodes but encode very long sequences, such as assembly graphs. Supplementary Materials section B.1 provides more information on how the graph shown in Figure 1.6 is stored internally using the GFA class.

3.2.2 GFA Class Benchmarking

`GFASubgraph` or its API is not the only `Python` implementation for GFA graphs developed to date. Accordingly, a comparative analysis was conducted with other implementations that we were able to find, install, and interface with through their respective APIs. We compared with `gfapy` [96], `gfagraphs` [64], and `mygfa` [50]. All of these tools have been implemented in `Python`, and provide a GFA graph API for working with the graphs. In this comparison, both low-memory and standard implementations of the GFA class in `GFASubgraphs` were used.

Three aspects were compared: the time required to load a graph from a GFA file, the memory consumed after graph is fully loaded, and the time required to find all the connected components using the Breadth-First-Search (BFS) algorithm. To test the limits of all compared implementations, four different pangenome graphs with varying sizes from the

recently published HPRC pangenome graphs were used [157]:

- The complete HPRC Minigraph graph (V1.0 CHM13) [154].
- Chromosome 22 component of the HPRC Minigraph graph.
- The complete HPRC Minigraph-Cactus graph (V1.1 CHM13) [120].
- Chromosome 22 component of the HPRC Minigraph-Cactus graph.

To extract the component representing chromosome 22, we used GFASubgraphs to extract all components, and used the component with nodes tagged with "Chr22" in their "SN" tags. It took GFASubgraphs about 30 seconds to extract all the components of the HPRC Minigraph graph and output them into a separate GFA file; and it took it about 40 minutes to do the same for the HPRC Minigraph-Cactus graph. Table 3.1 shows the values for graph loading time, memory, and component search time for all the tools. The evaluation was performed by writing a custom script to load the graph using the API provided by each tool. Subsequently, a BFS-based component finding function was implemented. More details regarding the implementation of this test can be found in Supplementary Materials Section B.3. Looking at the table, we see that the GFA graph implementation of GFASubgraphs demonstrated its ability to load all graphs regardless of their size. It also showed the best performance in terms of load time, memory profile, and BFS component search time compared to the other tools. We see that both gfagraphs and mygfa exhibited comparable graph loading times to GFASubgraphs, however, GFASubgraphs's API still performed better. Conversely, gfapy had a graph-loading time that was about an order of magnitude higher than the other tools. We also note that both gfapy and mygfa encountered an assertion error that we were unable to solve; these errors prevented them from loading the complete HPRC Minigraph-Cactus graph. In terms of memory consumption, GFASubgraphs consumed the least amount of memory compared to the other tools. In particular, the low-memory version which shows the most benefit for the HPRC Minigraph; this makes sense, of course, since this graph has fewer nodes compared to other HPRC graphs, with many nodes representing very long sequences. This is due to the way the graph was constructed using minigraph [154]. More on Minigraph's graph construction pipeline can be found in Section 3.4.1.1 and Figure 3.6.

Finally, GFASubgraphs and its graph API showed the most efficient performance in terms of component finding time, successfully identifying components for all the graphs tested. mygfa was unfortunately incapable of producing results in this test, primarily due to the lack of inherent functionality for retrieving edges of nodes. After examining mygfa's source code, we found that all edges are stored in a separate array, and that there is no invokable subroutine that connects the nodes stored in a Node ID-Node Object dictionary to the corresponding edges. On the other hand, both gfapy and gfagraphs have the ability to retrieve edges from nodes in their APIs, with gfapy performing well for small graphs, and its runtime

for this step not scaling up as fast as `gfagraphs`. After further investigation into `gfagraphs`, we found that it does not scale well with larger graphs, the main reason attributed to its subroutine for retrieving edges, which involves a linear search of the entire edges array for both incoming and outgoing edges of a node, resulting in an $O(n)$ time complexity for each edge search call.

3.3 `extgfa` for External Memory GFA Representation

In this section of the chapter we present `extgfa`, a proof-of-concept method and its implementation for a general purpose external memory representation of a graph in the GFA format that is inspired by open-world video games and how they manage memory usage. We demonstrate that this implementation improves the memory profile when running an algorithm such as BFS on a large graph, and is able to reduce the memory profile by more than one order of magnitude for certain BFS parameters.

3.3.1 `extgfa` Method

3.3.1.1 External memory in video games

This section reuses materials from [52] of which I am the sole author.

In procedurally generated or open-world video games, storing the entire world in memory/RAM[1] is extremely inefficient and, for many games, simply infeasible. To address this, game developers needed to come up with ways to keep only small parts of the world in memory while the rest is kept on disk (external memory), and develop a way to dynamically load more parts from disk seamlessly, without affecting the performance of the game [86, 207]. A notable example of such a game is *Minecraft* [169], a procedurally generated open-world video game, where the game's world is constructed from different blocks consisting of various materials (e.g., sand, rock, grass, etc.). Furthermore, the game's world is organized into chunks, each measuring 16×16 blocks and extending up to 30 million blocks in each cardinal direction. To keep the gameplay as smooth and playable as possible, and to prevent the game from overloading the memory, chunks are stored on disk and only loaded into memory when the player is of a certain distance from the chunk [171].

Figure 3.2 shows an abstract representation of the concept behind video games, such as Minecraft. The green block, corresponding to the player's location, represents the part of the map that has been fully loaded into memory. The adjacent yellow blocks represent parts of the map that are only partially loaded into memory. These can include distant features such as trees, mountains, and houses, which have not yet been fully populated with all of the

[1] From here on out, we use memory and RAM interchangeably to mean the fast Random Access Memory, in comparison to using "external memory" to mean the slower external disk memory

		GFA-Subgraph Low Memory	GFA-Subgraph	gfapy	gfagraphs	mygfa
HPRC Minigraph Chr22	Load Sec	0.3	0.33	29.8	0.89	3.5
	Mem Mb	33.11	95.18	112.08	181.4	122.38
	Comp Sec	0.04	0.04	2.22	178.81	NA[1]
HPRC Minigraph	Load Sec	10.24	12.12	478.92	26.96	161.26
	Mem Mb	673.09	3,978.7	5,391.97	4,727.03	4,167.56
	Comp Sec	1.56	1.67	66.27	NA[2]	NA[1]
HPRC Minigraph -Cactus Chr22	Load Sec	67.7	73.12	2,907.01	200.29	198.04
	Mem Mb	3,522.65	3,637.63	7,841.62	8,205.19	8,086.15
	Comp Sec	19.7	18.4	245.25	NA[2]	NA[1]
HPRC Minigraph -Cactus	Load Sec	910.63	909.66	NA[3]	13,994.27	NA[3]
	Mem Mb	109,764.87	114,985.98	NA[3]	623,875.88	NA[3]
	Comp Sec	298.03	304.46	NA[3]	NA[2]	NA[3]

Table 3.1: In this table, we tested the following three parameters, **Load**: Graph Load Time in wall clock seconds, **Mem**: Memory used in megabytes, and **Comp**: Components Finding Time based on BFS in wall clock seconds.
Four different graphs were used: The Chr22 of the HPRC Minigraph graph, the full HPRC Minigraph graph, Chr22 of the HPRC Minigraph-Cactus graph, and the full HPRC Minigraph-Cactus graph. Chr22 component was extracted using `GFASubgraphs`. It took `GFASubgraphs` about 30 seconds to extract all the components of the HPRC Minigraph graph, and about 42 minutes to extract all the components of the HPRC Minigraph-Cactus graph. The *NA* entries in the table resulted from different reaspons:
NA[1]: in `mygfa`'s GFA class, there was no direct way to retrieve edges corresponding to nodes, the class did not provide any subroutines for this. Therefore, we could not run the component-finding algorithm. **NA[2]**: `gfagraphs` process had to be terminated after running for more then 24 hours. Looking into their code, the reason for this is that edges were stored in a list, and when calling the subroutine `.get_edges()`, it searches the list twice to get the in and out edges, resulting in $O(n)$ search time for each retrieval. **NA[3]**: An assertion error in both `mygfa` and `gfapy` when running on the HPRC Minigraph-Cactus graph, which we were unable to solve.

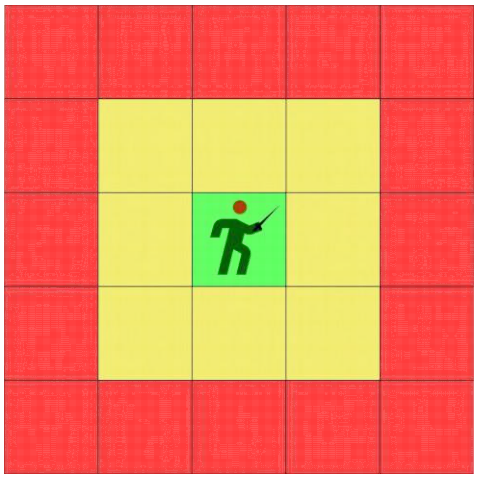


Figure 3.2: This figure is a description of how the map is loaded in *Minecraft*. The green square in the middle is where the player is, where this chunk is fully loaded; then the chunks are not fully loaded the further away they are from the player, with several levels of information being left out when loading. *Figure taken from [52].*

game's aspects. This technique preserves the feeling of a vast, open world while conserving memory. Finally, the red blocks represent unloaded parts of the map.

In the context of the game, the player's direction of movement dictates the status change of the map block (e.g., green, yellow, or red). Specifically, the farther away the chunks are from the player, the more their status changes and they become inactive, i.e., they are unloaded from memory onto the disk, while other chunks in the direction of the player's movement are loaded from disk into memory. This process then maintains a constant number of loaded map chunks in memory and prevents the game from overloading the memory.

3.3.1.2 Graph Chunking Pipeline

This section reuses materials from [52] of which I am the first author.

The main takeaway of memory management in open world games is that the whole world does not need to be loaded into memory, only the map chunks that surround the player. Inspired by this, we developed the following pipeline to chunk the GFA graph and create an index that allows us fast access to parts of the graph stored on disk, instead of keeping the entire graph in memory, this then allows us to dynamically load and unload parts of the graph between disk and memory. The pipeline consists of the following steps:

- **1. Cutting the graph into neighborhoods:** This step aims to partition the graph into non-overlapping chunks, where each chunk is a connected subgraph or community smaller than the original graph. Community detection in graphs is an old problem that has been studied and explored for decades; many algorithms have already been developed to solve this problem with varying degrees of sensitivity, specificity, and

time and memory complexity [143]. More about the specific algorithms tested and used in extgfa can be found in Section 3.3.2.

- **2. Recursive chunking:** Depending on the algorithm used to cut the graph into chunks, the chunks may not be balanced in terms of the number of nodes, and the sizes may vary tremendously. It is not strictly necessary for this method to have similarly sized chunks, however, this helps in keeping the loading and unloading times and memory usage of chunks uniform across the chunked graph. To accomplish that, an upper and a lower threshold for the number of nodes per chunk is used to limit the size of the chunks. The upper threshold is defined as the maximum number of nodes a chunk may have before it is recursively cut into smaller chunks; the lower threshold is defined as the minimum number of nodes a chunk may have before getting merged, if possible, with a neighboring chunk. This step is run recursively until the remaining chunks have a size between the upper and lower thresholds.
- **3. Producing reordered GFA and indexes:** After the previous steps, we are left with a set of non-overlapping chunks with sizes between the upper and the lower thresholds, where all the chunks jointly comprise the original graph. Using this information, we can reorder the GFA file, such that the nodes and edges of a chunk are written consecutively in the ordered GFA file, forming a continuous block in the GFA file, before starting with the next chunk. While writing the reordered GFA file to disk, we keep track of the chunk's start offset and number of lines, enabling us to later load any arbitrary chunk from the file into memory without having to read the complete GFA file. We also create a database linking each node ID to its corresponding chunk ID, allowing us to retrieve the chunk id and the chunk for any node dynamically and quickly.

3.3.2 extgfa Implementation

3.3.2.1 Graph Partitioning

This section reuses materials from [52] of which I am the sole author.

extgfa is written in Python and uses the NetworkX library to execute the community algorithms [108]. Thereafter, it uses a custom GFA class similar to the one presented in Section 3.2.1 to read, write, and manipulate GFA graphs. For the graph cutting step, we tested several graph community detection algorithms already implemented in the NetworkX library, such as the Kernighan-Lin algorithm [138], edge betweenness partition [83], Louvian communities [21], and the Clauset-Newman-Moore greedy modularity maximization algorithm [41]. We found that the last algorithm worked best compared to the others; it was relatively fast even for large graphs, and produced communities that were similar in size in terms of the number of nodes.

In short, the Clauset-Newman-Moore greedy modularity maximization algorithm tries to find sets of nodes or "communities", where each community is more densely connected internally than to other communities. This is achieved by starting with each node as its own community, then merging pairs of communities that maximize the "modularity", until further merging does not increase the modularity. Modularity here can be simply explained as the value that maximizes the number of edges in a community compared to the number of edges between communities [185].

In `extgfa`, after recursively detecting communities/chunks using the Clauset-Newman-Moore algorithm, we arbitrarily assign a unique integer ID starting from 1 to each produced chunk. From there, we can now generate three files that encapsulate the information of the chunks and allow dynamic loading and unloading of chunks between external disk storage and ram. Figure 3.3 shows an example graph represented in a GFA file format and visualized with `Bandage` [289], with the three files generated by our implementation, which are:

1. A reordered GFA file, where chunks are written consecutively as blocks in the output file.

2. A chunk offset index, it consists of key-value pairs, where the key is the integer chunk ID and the value is a tuple of two values; the first value points to the start offset in the reordered GFA file for that chunk, and the second value is the number of lines to read starting from the offset.

3. A `dbm`[2] file built with the `shelve` library in `Python`, this file represents an external database (not stored in RAM) of key-value pairs, where the key is the node's string ID and the value is the integer chunk ID.

With the three files generated, we now have the ability for any node in the graph, to retrieve the chunk ID to which that node belongs to using the `dbm` file. Subsequently, using the offset index, we can load the chunk from the reordered GFA file without having to load the entire graph into memory. Note, however, that with this formulation, we cannot edit the graph, i.e., remove or add nodes and edges, as this would require rebuilding of both the `dbm` database, the offsets, and the reordered GFA file.

3.3.2.2 Chunked Graph Class

This section reuses materials from [52] of which I am the sole author.

We have implemented two similar graph classes, `Class::Graph` and `Class::ChGraph`, where the former loads the GFA graph completely, i.e., stores all the nodes and edges in memory, while the latter uses the three files previously generated to dynamically load and unload chunks as needed. Both classes have the same internal functions and data structures, which

[2] A `dbm` is a library or a database with single hashed keys that point to some value and provide fast access to the data stored. Values can be retrieved from this database without having to load the data-base into memory.

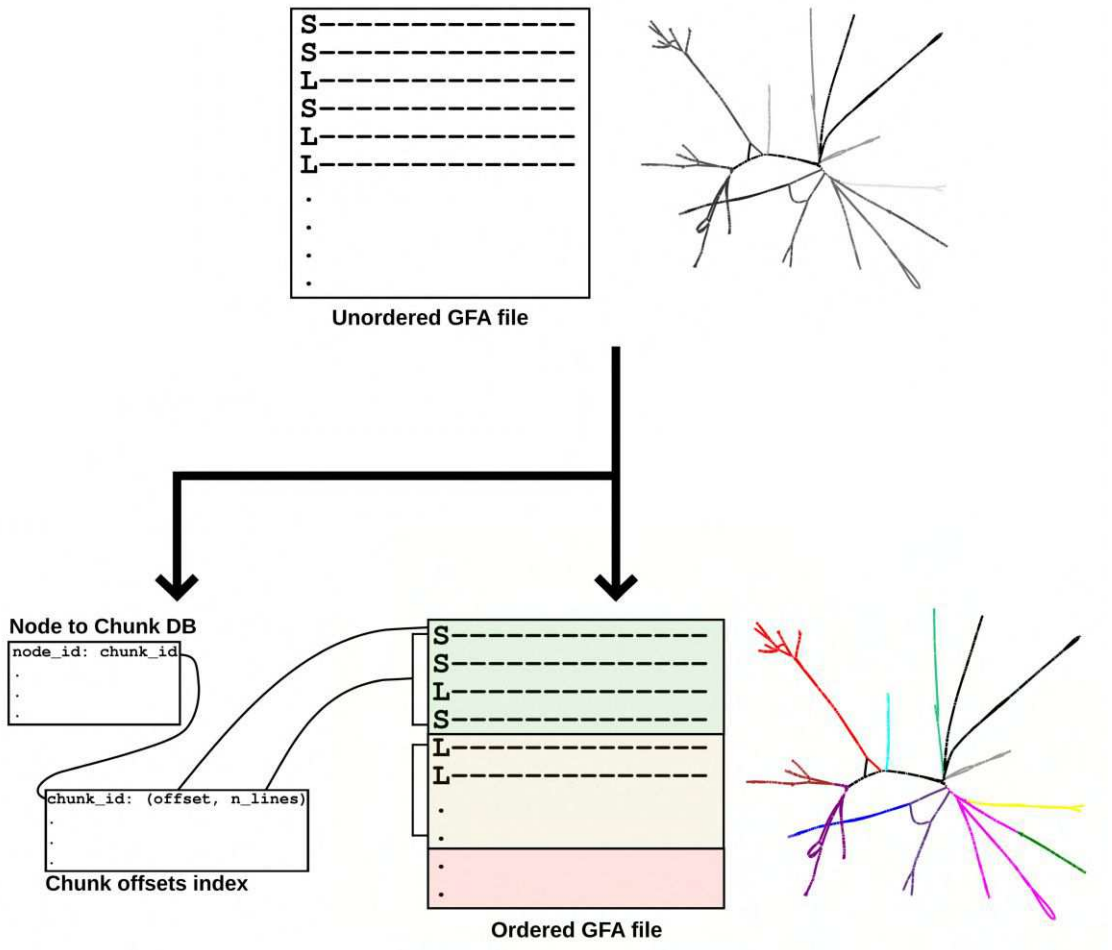


Figure 3.3: This is the `extgfa` pipline. First, it detects chunks in the GFA graph as described in Section 3.3.2.1. Once the chunks are found, `extgfa` produces three files: (1) a database `dbm` of key-value pairs, where the keys are the node IDs and the values are the chunk IDs to which the node belongs to. (2) A binary file that is a key-value pair, where the key is the chunk ID and the value is a tuple of a file offset in the GFA and the number of lines to read from that offset. (3) A reordered GFA file, where each chunk is written consecutively in the file. *Figure taken from [52].*

allows a direct comparison between them, and gives the user the ability to reuse the same code with one or the other class.

With the `Class::ChGraph`, there is no need to load any nodes or chunks at the beginning, only once the user tries to retrieve a node, the class retrieves the chunk ID associated with that node using the node ID-chunk ID database. It then finds the offset and number of lines to read in the reordered GFA file, and finally retrieves the chunk that the node belongs to. In addition, the user can set a cutoff on how many chunks can be loaded into memory before older chunks are removed. This is done using a First-In-First-Out (FIFO) queue that keeps track of the chunks loaded. When the specified threshold is exceeded, the first chunk that was loaded is removed from memory and new a chunk is loaded and added to the queue.

This queue is used to mimic how some video games unload the chunks that are farther away from the player as the player moves.

Both classes implement basic functionality related to graphs, such as finding edges, nodes, node contents (sequence, length, tags, etc.), graph traversals, and other functionality. Additionally, `Class::ChGraph` is able to automatically detect when it needs to load or unload a chunk without user intervention, allowing for a seamless implementation without the need for the user to manually manage what needs to loaded or unloaded.

3.3.3 `extgfa` Chunked and Unchunked Graphs Comparison

This section reuses materials from [52] of which I am the sole author.

To test `extgfa`, the graph representing chromosome 22 from the HPRC PGGB V1 graph [89] was used. This graph consists of 3,759,736 vertices and 5,224,421 edges. To cut the graph into chunks, we used the Clauset-Newman-Moore algorithm as described in the Section 3.3.2.1. We set the upper threshold to $|V|/2000$ and the lower threshold to $|V|/5000$, where $|V|$ denotes the number of nodes in the graph. This process resulted in 3,084 chunks with an average number of nodes of 1,219 per chunk.

To evaluate the performance of both graph classes, a standard BFS algorithm was implemented using the two classes. The algorithm start at a random node in the graph and traverses the graph until it either reaches a user-specified size cutoff (BFS size) or finds no additional nodes to traverse. In this experiment, a random starting node was selected and the BFS algorithm was executed with different BFS cutoff sizes (50, 100, 1,000, 5,000, 10,000, 50,000, 100,000, 500,000, and 1,000,000). Additionally, for the chunked version, we ran each BFS cutoff size experiment with 7 different chunk queue sizes (1, 5, 10, 50, 100, 500, 1,000).

Figure 3.4 shows a scatter plot comparing running the BFS algorithm on both classes with the different BFS cutoff sizes; the top plot shows the memory profile and the bottom plot shows the time profile. Looking at the top plot, we see that the unchunked version (fully loading the graph) has a constant memory profile of approximately 8 GB for all the BFS cutoff sizes. This is expected, of course, because the unchunked version loads the entire graph into memory before running the algorithm, which would always result in the same memory profile regardless of the BFS size. In contrast, for the chunked graph version, the memory profile is much smaller, and is affected by the BFS cutoff size, with a maximum memory usage of approximately 3 GB. This is to be expected, since the smaller the BFS cutoff size is, the fewer chunks need to be loaded and held in memory. Looking at the bottom plot, which shows the time profile, the unchunked version behaves similarly to the top plot, with a constant time profile of about 90 seconds, but with a slight increase in time at a BFS cutoff size of 1,000,000. This is due to the additional time required to run the BFS algorithm.

In the case of the chunked version, however, considerable variability in the time profile is observed, especially for large BFS cutoff sizes. This is due to two reasons. First, as the BFS size increases, more chunk loading operations must be performed, including database lookups, finding the chunk location in the reordered GFA file, and loading the chunks into memory. Second, the maximum number of chunks allowed in memory contributes to the time variability for certain BFS sizes. As mentioned in Section 3.3.2.2, once the chunk FIFO queue is full, older chunks must be unloaded from memory. Since `extgfa` does not yet implement multithreading, the loading and unloading are done sequentially on the same thread rather than concurrently. This, in turn, has a negative effect on the processing time.

To explore the effect of chunk queue size on both time and memory, and to further explain the variability in the runtime of the chunked version, individual values for each run of BFS on the chunked graph version only are shown as a scatter plot in Figure 3.5. The points are colored from light blue to dark blue based on the smallest to largest queue size. We see that for smaller BFS cutoff sizes, the effect of the chunks queue size becomes negligible. This can be attributed to the fact that for smaller BFS cutoff sizes, we only need to load one or very few chunks into memory. However, the effect increases as the cutoff gets larger. These results underscores the expected tradeoff between memory consumption and execution time. As the number of loaded chunks increases, more nodes are also loaded into memory, facilitating rapid access to nodes and edges, and accelerating the BFS algorithm. Conversely, when a only a small number of chunks are loaded, less memory is used, but more execution time is now required to load more chunks to access their nodes and edges, thereby hindering the efficiency of the BFS algorith.

3.4 `gaftools` for Working with Pangenome Alignments

In this section, we present `gaftools`, a multi-purpose toolkit written in `Python` for working with graph sequence alignments against pangenome graphs. `Gaftools` introduces several important functionalities that are standard in world of linear sequence alignment, but have not been extended to the graph alignment formats. Some of these functionalities include sorting, indexing, subsetting (viewing), and generating statistics on the alignments. It also it introduces a realignment step using the wavefront alignment algorithm [164] to realign sequences on the graph with higher accuracy.

3.4.1 `gaftools` Commands

In the following sections, we will describe the different functionalities that `gaftools` provides to users in more details.

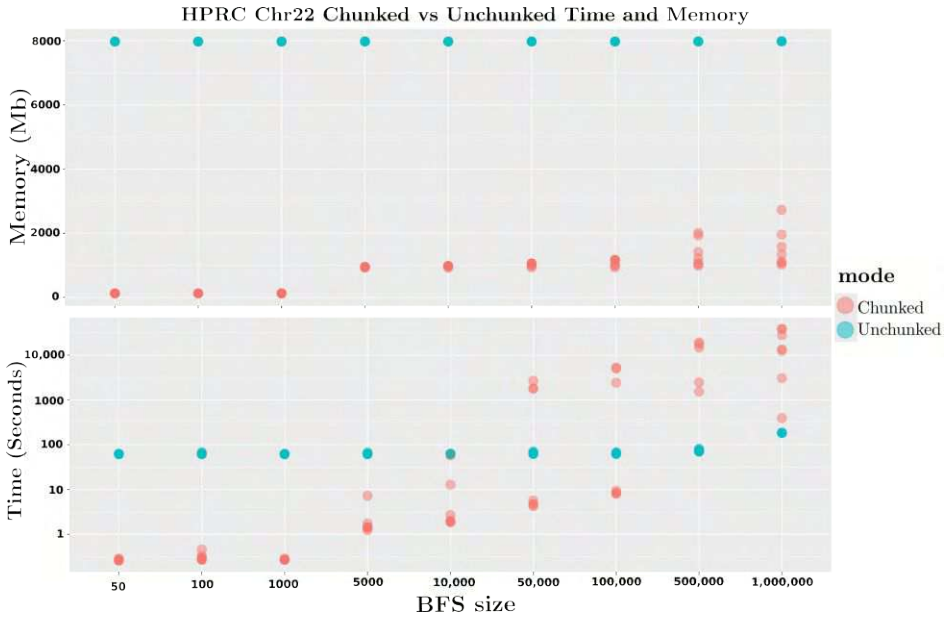


Figure 3.4: Scatter plot comparing the chunked and unchunked versions in terms of time and memory. We see that for the unchunked version, the time and memory are mostly constant, because we always need to load the complete graph, and this operation takes much more time compared to running the BFS algorithm. In contrast, for the chuncked version, we see more variability in terms of time and memory, which can be explained by the number of chunk loading and unloading operations required, and the effect of the maximum number of chunks allowed in memory. *Figure taken from [52].*

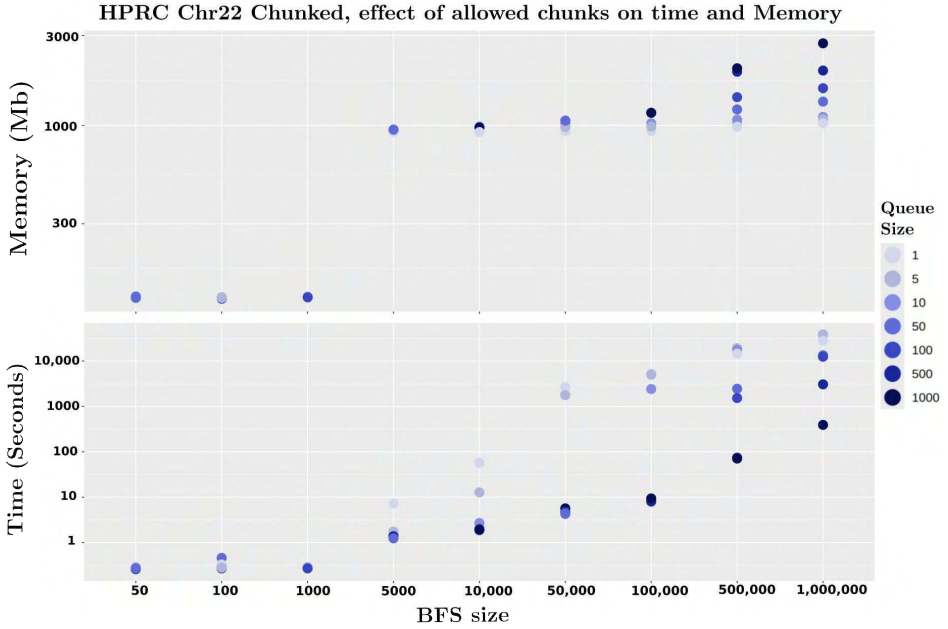


Figure 3.5: Scatter plot showing the effect of the chunks queue size on both memory and time in the chunked version of the graph. We see that the bigger the BFS cutoff size is, the bigger the effect of queue size. Furthermore, the queue size has a contrasting effect on time and memory; the bigger the queue, the less time it takes to run the BFS and the more memory it requires, and *vice versa. Figure taken from [52].*

3.4.1.1 GFA Ordering and GAF Sorting

This section reuses materials from [196], specifically materials that were my contribution to the work.

One of the problems in moving from a linear reference to a graphical reference, is having a coherent and consistent coordinate system. In the linear world, when processing sequence alignments against a reference genome, unique coordinates can be given to the alignment, e.g., alignment start, alignment end, indels location, etc., using the coordinates on the reference genome. Additionally, having these coordinates can help in operations such as sorting and indexing the alignments, which, in turn, allows for faster access to subsets of the alignments. However, in pangenomes, the graph structure lacks the simple coordinate system of the linear world, which hinders the ability to easily sort, index, or manipulate the alignments. To tackle this issue, we advise an ordering and sorting approach here specifically designed for reference graphs (rGFAs) (Section 1.4.3.1). This functionality is called `order_gfa` and it is a subcommand of `gaftools`.

Before delving into the method itself, we need to shed some light on why this works only on rGFAs and not all GFAs, particularly, on the rGFAs produced by `minigraph`. `Minigraph` has two main functionalities, the incremental generation of a reference graph (in rGFA) and the alignment of long sequences (such as contigs, or complete genomes) against the generated graph. The graph generation step is based on sequence-to-graph alignments, where `minigraph` internally uses a modified version of `minimap2` [148] to allow for seed-chaining and alignment of a longer query sequence, where other sequence graph aligners such as the ones presented in [216, 88] are unable to do so. `Minigraph` does not perform a base-resolution alignment, however, it uses chains of local hits to find the approximate mapping location on the graph. Figure 3.6 shows a simple schematic of the recursive graph building. In this figure, `minigraph` starts with the first genome in red, denoted here as the reference genome (rank 0). When adding the next genome (drawn in green), we see that part of the second genome does not align to the reference genome, resulting in the generation of two nodes, or a bubble [3], where the source and sink of the bubble are the parts of the reference genome. Next, adding a third genome (drawn in blue), we see that it also creates a bubble with the reference, and this third genome will get the rank 2. In the final graph generated, all the reference segments (with rank 0) should form a continuous path If this path is followed, one can reconstruct the reference of rank 0. Furthermore, we see that we end up with two bi-connected components that share an articulation point in the middle. Because of this incremental way of building the graph, and having a linear reference as the backbone of the graph, and not performing base resolution alignments, all result is a graph with a chain of bi-connected components representing the structural variants between the added sequences, and with articulation points between the bubbles that always belong to

[3] Here, we define the bubble as a bi-connected subgraph with two disjoin paths, the articulation points are called source and sink

the reference genome. Here, we call these articulation points “scaffold nodes”.

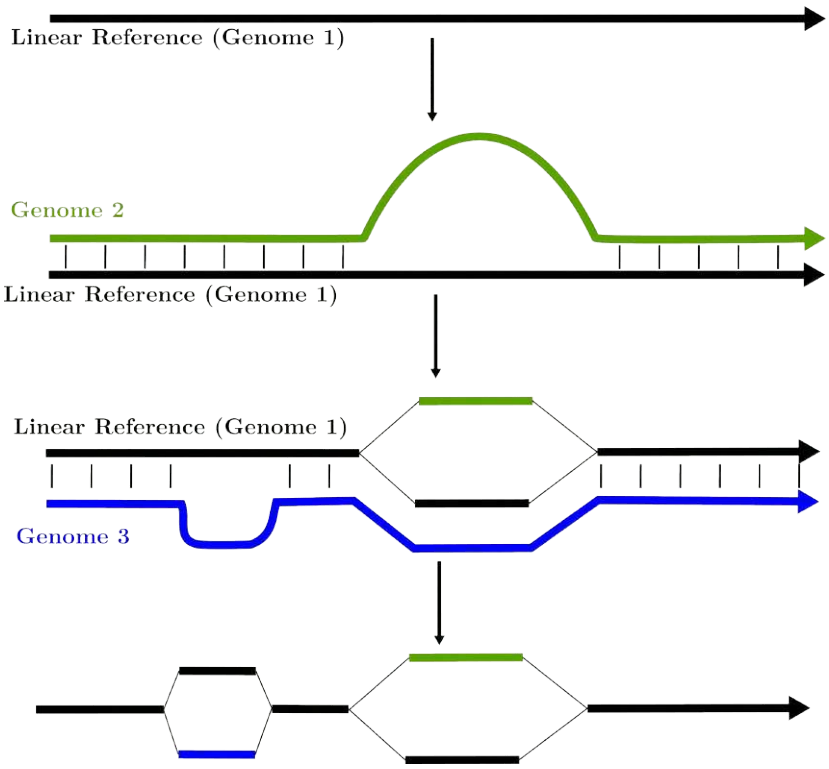


Figure 3.6: Here, we show a simple schematic of the incremental construction of an rGFA using `minigraph`. We start with a linear sequence (black), which is marked as rank 0 in the rGFA output file. Then, `minigraph` aligns the next genome, haplotype, or contigs. The variation between the two will generate bi-connected components (bubbles), and nodes belonging to only the aligned sequence will have the rank 1. This now happens incrementally with each genome added, e.g., adding the blue genome, and depending on the alignments, new nodes and new bi-connected components are generated in the graph to describe the variability between the different sequences. *Figure inspired by Figure 2 in [154].*

The `gaftools` command `order_gfa` utilizes the features of the rGFA built with `minigraph` to devise an ordering. First, it detects the bi-connected components, and for that, we implemented a detection algorithm inspired by the implementation in `NetworkX` [108], which is based on the algorithm presented in [123]. This is described in more details in the Supplementary Materials Section B.2. Furthermore, `gaftools` internally uses a similar graph data structure/class as the one described in Section 3.2. Once we find all the articulation points and bi-connected components, we can chain the components using the shared articulation points, and order them based on the coordinates on their “SO” tag. With this ordering, we can introduce new tags to the nodes that reflect this ordering, these new tags are called “BO” (Bubble Order) and “NO” (Node Order) tags. The BO tags are used to sequentially label (starting from 1) the bi-connected components detected and ordered in the graph. All nodes in a bi-connected component receive the same BO tag. The NO tags are used to sequentially label (starting from 1) the nodes within the bi-connected component. This is based on the

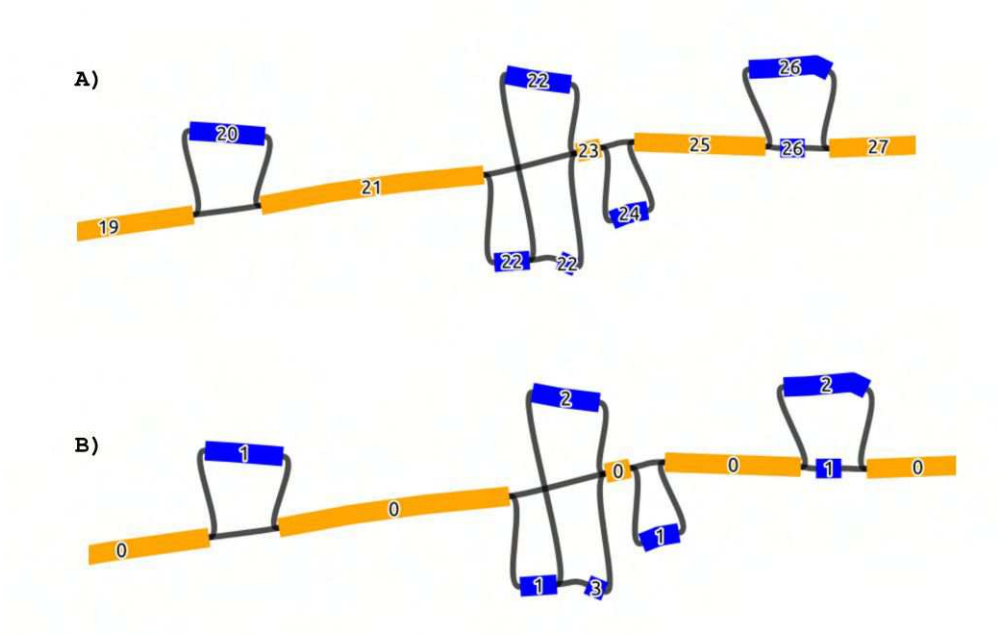


Figure 3.7: This figure depicts the BO (A) and NO (B) tags. Blue nodes are the bubble and orange ones are the scaffold nodes. *Figure taken from in [196].*

lexicographic order of the node IDs within the bi-connected component. Figure 3.7 shows a simple chain of four bi-connected components and five scaffold nodes. Part (A) and (B) of the figure show the assigned BO and No tags, respectively. Articulation points are given the value 0 for their NO tags. Now, we can sort the GAF alignments using the BO and NO tags, where we first consider the BO tag, and the alignment that aligns to a node with the smaller BO tag comes first in the sorted output. If the BO tags are the same for two alignments, we look at the NO tags, if these are also the same, then we look at the alignment position in the segment.

3.4.1.2 GAF Indexing and Viewing

This section is adapted from [196], and the work presented here was done by Samarendra Pani.

The `view` subcommand has a similar functionality to the one in `SAMtools`, where it can "view" or extract a subset of the ordered GAF alignments based either on reference genome coordinates or graph node IDs. This subcommand is also able to perform conversion between the "stable" and "unstable" coordinate systems (these systems were described in Section 1.4.3.1), using the information in the rGFA graph. An example of this conversion, looking at Figure 1.8, `gaftools` would then convert the node's unstable coordinates (alignments marked with "_u"), to the contig or reference-based stable coordinates (alignments marked with "_s"). Furthermore, `gaftools` also provides an indexing scheme for the sorted GAF alignments; the index provides fast access to the alignments when using the `view` subcommand. The index here is an hashmap lookup, with unique keys as the segment or node

IDs, and the values are arrays of offsets pointing to the alignments that have aligned against these segments in the GAF file. This then allows fast access to the alignments if the user desires so subset alignments that mapped to a specific segment in the graph.

3.4.1.3 Wavefront (Re)Alignment

This section is adapted from [196], and the work presented here was done by Arda Soylev, with the alignment parallelization part done by myself.

An important feature of `gaftools` is the ability to realign an alignment from the input GAF file. Here, we realign the sequence to the path in the GAF file provided, i.e., we extract the sequence of path in the graph, and realign the sequence from the GAF to the sequence extracted from the path. We use the wavefront algorithm [164] as it is fast, and uses gap-affine penalty between the two sequences aligned, which results in better alignments. The reason for performing realignments, is that in [253] they have found that other sequence to graph aligners had some problems aligning certain sequences. For example, they found that `GraphAligner` produced alignments with gaps at positions that did not make sense biologically; and `minigraph` was not able to align some sequences continuously and produced fragmented or clipped alignments for one sequence. In addition, gap-affine penalty is better at capturing certain qualities that are more biological when aligning two sequences against each other, compared to edit-distance based alignments. To speed up the realignment process, `gaftools` is able to use multiple CPU threads and parallelize the process.

3.4.1.4 Miscellaneous Functionalities

This section is adapted from [196], the first functionality was contributed by myself, and the second and third contributed by the other authors.

Our software also provides several other functionalities that are useful to users in downstream analysis steps, such as:

1. Retrieving the sequence of a certain path provided in the graph using the `find_path` subcommand.

2. Generating statistics for the GAF alignment file, such as, number of primary and secondary alignments, total aligned bases, and average mapping scores (average quality, identity, and mapping ratio), using the `stat` subcommand. Using the extended mode with the `--cigar` flag, `gaftools` reports more information related to the CIGAR string of the alignments, such as the number of insertions, deletions, matches, and mismatches.

3. Introduction of phasing to alignments, adding information about which haplotype the alignment maps to is added. Here, we add the tag “ps:Z:” for the phase set information,

	Graph alignments			Linear alignments
	gaftools	vg	minigraph	SAMtools
Coord. Conversion	✓	-	✓	N/A
Align. Subsetting	✓	✓	-	✓
Align. formats	GAF	VG,GAM,GAF	GAF	SAM, BAM, CRAM
Align. Indexing	✓	✓	-	✓
Graph ordering	✓	-	-	N/A
Align. sorting	✓	✓	-	✓
Haplotype tags	✓	-	-	-
(Re)align	✓	-	-	-
Path to Sequence	✓	-	✓	N/A
Align. statistics	✓	✓(GAM)	-	✓

Table 3.2: This feature table outlines the functionalities of gaftools, alongside other tools offering similar capabilities. The "N/A" is for features that are only applicable to graphs. We see that `minigraph` is also able to convert coordinate systems, however, one needs to run the alignment again to change the coordinate systems. While `gaftools` is able to do so directly on the GAF file without having to realign the sequences. Align. stands for alignments, and coord. stands for coordinates. *Table taken from [196]*

and the tag "ht:Z:" for the haplotype. This information is based on the output of `WhatsHap` [165], specifically using the output of its `haplotag` command.

3.4.2 Comparison and Benchmarking

This section is adapted from [196], and the work presented here is a contribution by all authors.

Looking at the recently developed pangenome toolkits, we see that `gaftools` offers a distinctive set of features not found in other tools. In particular, `gaftools` introduces utilities that were previously only available for linear sequence alignments. A comparison of the features provided by gaftools with those of other tools, both graph-based and linear-based, for graph and alignment processing is presented in Table 3.2. The tools included in this comparison are `vg` and `minigraph`, which work with graphs and graph alignments, and `SAMtools`, which works with linear alignments. As shown in Table 3.2, a checkmark indicates that a given toolkit is capable of performing that specific functionality. We can see from the table that `gaftools` is capable of performing a multitude of functionalities that bridges the gap between the different tools.

To test our tool, we ran it on the Oxford Nanopore Technologies (ONT) long reads of the sample NA12878 sample from the 1000 Genomes Project [1]. This sample had a depth coverage of approximately 14X. The sample was then aligned against the HPRC Minigraph graph, which was constructed using the CHM13 reference using `minigraph`. Table 3.3 shows the runtime and memory consumption of each command provided by `gaftools`. The results show that `gaftools` is fast and consumes a reasonable amount of memory, with the exception of realign, which consumes a significant amount of memory and time. The reason

Command	Runtime (hh:mm)	Memory (GB)
`view`	$<$0:01	$<$1
`view --format`	0:20	2.2
`index`	0:01	2.2
`order_gfa`	0:01	2.2
`sort`	0:08	3.2
`phase`	0:05	1.8
`realign`	64:34	47
`find_path`	$<$0:01	5.5
`stat`	0:04	1.7
`stat --cigar`	0:40	1.7

Table 3.3: Graph alignments of NA12878 ONT (Oxford Nanopore Technologies) reads ($\sim$14X depth of coverage) from the 1000 genomes project, aligned to HPRC-r518 T2T-CHM13 using Minigraph. Results show that gaftools is fast and memory efficient for all the commands except "realign". Since "realign" requires Wavefront alignment, where higher runtime and memory requirement is expected. *Table taken from [196]*

why realignment is an outlier in terms of time and memory, is that it uses the Wavefront alignment algorithm, which requires a considerable amount of resources for long sequences. However, the realignment is parallelized, so the time can be reduced if more memory and CPU cores are available.

3.5 Conclusion and Discussion

In this chapter, we have presented a collection of toolkits for working with genome graphs in the GFA format and with sequence-to-graph alignments in the GAF format. We believe that such toolkits are very valuable to fill the gaps created by moving from the linear-reference world to the graph-reference world.

We first (Section 3.2.1) described the design and implementation of a simple and efficient GFA graph `Python` API. This API allows the user to easily work with and manipulate with GFA graphs. Furthermore, we demonstrated the use of this API in `GFASubgraph`, a tool capable of extracting subgraphs and components from large graphs for downstream analysis and visualization purposes. This implementation of a GFA class also serves as an important building block for both `extgfa` and `gaftools` that were presented here. It also highlights the importance of having an easy-to-use and fast software API for working with genome graphs, where other functionality can then be easily built on top of this infrastructure. This also gives researchers the ability and freedom to implement their own ideas without having to redesign this data structure.

We then introduced `extgfa` (Section 3.3), a method for a low memory GFA graph representation inspired by the world of video games and their ability to provide players with open-world maps without exceeding the limits of the RAM. Although we labeled this implementation as a proof-of-concept implementation, it is, however, operational for examining

numerous subgraphs or loci in large graphs. It is flexible enough to allow users to implement their own algorithms using the API provided. Further improvements to the design and implementation are still possible. For example, implementing a prefetching scheme that loads neighboring chunks behind the scenes before the algorithm reaches that part of the graph. Or, loading only the topological information of a chunk and not all of the node's information such as sequence and tags. We see that this concept of external memory is also very important for graph visualization, since the current visualization tools such as [289, 97] cannot handle very large graphs. Therefore, integration techniques such as the one in `extgfa` can be used to chunk the graph and visualize only what is needed, which can help tremendously and allow users with smaller machines to work with very large graphs.

Finally, we presented `gaftools` (Section 3.4), a versatile tool for working with sequence-to-graph alignments in the GAF format. We demonstrated the various utilities provided by `gaftools` and their benefits, such as realignment, graph ordering, alignment indexing and sorting, alignment viewing, and so on. We believe that `gaftools` serves as a first step in addressing some of the missing functionality and algorithms that are readily available for alignments in the linear reference world, but have not yet found their way into the graph world. One drawback of `gaftools` is that it only works on rGFAs produced by `minigraph`, as it requires the graph to have a linear chain of bubbles/bi-connected component, when collapsed, result in a line graph of reference segments. However, continuous work is still being done to allow other rGFAs (e.g., produced by `minigraph-cactus` or`PGGB`) to be utilized.

Looking at the rapid advancement in pangenomics, and the continuous production of large graphs, the need for efficient tools is definitely clear. The tools presented in this chapter can be seen as first steps towards closing the gap between the linear reference world and the graph pangenomics world. We believe that the tools presented here can serve as a stepping stone to build better, and more efficient software to perform similar tasks. Particularly, re-implementing an improved graph data structure in a typed programming language such as `C` or `Rust`; as such programming languages offer much more efficiency in terms of speed and memory consumption. Moreover, there is a need in standardizing the GFA format and producing a binary indexable version of it, which can further help in avoiding the task of loading the complete graph in memory for any analysis. Efforts done in the linear reference world, such as the production of `HTSlib` [25] for working with SAM, BAM, CRAM, and VCF files can serve as a great example for what to aim for in the future,

Chapter 4

Multi-Platform Investigation in Cancer Structural Variants and Subclones

This chapter presents an unpublished collaborative work on cancer cell line NCI-H2087 and its matched normal cell line NCI-BL2087. First, we assemble both samples using sequencing reads from different technologies, and we investigate the quality of these assemblies and the potential causes of a fragmented assembly. In addition, we generate a set of high confidence somatic structural variants by using five different variant callers and intersecting the calls to purify the set of somatic SVs. We further investigated the role of genome graphs and their visualization to aid in the subclonal assembly and structural variant detection problem. To that end, we develop a graph toolkit called `graphdraw` that helps in extracting parts of the graph based on their alignments, moreover, can draw parts of the graph along a reference for better visualization.

This work is in collaboration with Dr. Jan Korbel and his research group at EMBL, they have provided us with data and important incites to bring this project together. My role in this project was focused on the graph investigation, development of `graphdraw`, and SV calling and intersection. The assemblies generated with the `PGAS` pipelie were done by Dr. Peter Ebert, and Figure C.3 was generated by Dr. Bernardo Rodriguez-Martin who also provided continuous supervision on the project.

4.1 Introduction

The hallmarks of cancer are a set of functional capabilities acquired by cells during its development. The hallmarks comprise six capabilities: (1) sustaining proliferative signaling, (2) evading growth suppressors, (3) resisting cell death, (4) enabling replicative immortality, (5) inducing the formation of new blood vessels for the cancer, and (6) activating invasion and metastasis [110]. The list of hallmarks is subject to variation, with additional hallmarks

being added to the list, such as the ability to evade immunity destruction [82], and the ability to reprogram cellular energy metabolism to sustain the continuous growth of the cancer [202]. The combination of these hallmarks results in the cancer's ability to survive, multiply, and spread [110].

Cancers generally originate from a single cell, which undergoes a series of somatic mutations, resulting in the acquisition of the aforementioned hallmarks by the cell's progeny. However, the descendants of this cell can undergo diverse sets of mutations, separation, and evolutionary selection, thereby giving rise to genetic heterogeneity and the emergence of distinct populations of cells within a single tumor [262]. These genetically distinct populations from one cancer are called "subclones", and due to the genetic differences between the subclones, their behavior and response to cancer treatments can differ [27]. Consequently, subclone detection and reconstruction are imperative to understand the cancer and to influence the cancer's progression and therapy options [262].

With the rapid advancements of sequencing technology and sequence algorithms (Section 1.2), scientists were able to understand cancer better, attempt to reconstruct cancer subclones, detect important variations, and apply this to personalized clinical treatments [104]. With these advancements, large cancer studies have identified that structural variants (SVs) are the predominant class of driver mutation in many cancer types [51]. Moreover, due to the difficulties in detecting SVs compared to point mutations (SNPs), they remain underexplored [51]. Large studies such as [206] produced a comprehensive catalog of cancer somatic mutations; and most notable in recent years, the Pan-Cancer Analysis of Whole Genome (PCAWG) Consortium analyzed a large number of cancer whole-genomes and their matched normal, across many cancer types allowing for the detection of a large number of cancer-associated structural variants [2]. Furthermore, studies also showed that cancer does not necessarily accumulate mutations gradually, but "cellular crisis events" can occur in one of the cells causing what is called "chromothripsis", where hundreds of genomics rearrangements take place [256]. This also adds to the difficulty of assembling and constructing high quality SV call sets. However, multi-platform sequencing studies, such as [76], attempt to combine different sequencing technologies and leverage their characteristics in order to identify a set of high-quality somatic SVs in cancer.

In this chapter, we investigate the cancer and matched normal cell lines pair (NCI-H2087 and NCI-BL2087), where NCI-H2087 is a cell line from stage 1 adenocarcinoma, and NCI-BL2087 is a B lymphoblast cell line from peripheral blood. We use different sequencing technology to capture different levels of information. Moreover, we assemble both cell lines and call the structural variants using five different computational methods. Subsequently, using the match normal structural variants, we attempt to curate a set of high-quality somatic structural variants.

	PacBio		Illumina		Strand-seq	
	avg. cov	avg. len.	avg. cov.	len.	avg. cov.	len.
BL2087	39.5	16,032	268.5	151	2.1	81
H2087	40.4	19,101	180	151	2.8	81

Table 4.1: Information on the sequencing data for both the cancer sample H2087 and the matched normal BL2087. For Strand-seq data, the coverage is calculated only on the high quality cells chosen by ASHLEYS, which is explained further in Section 4.3.1

4.2 Data

In this work, we used three different sequencing technologies on both cell lines. PacBio CCS or HiFi long reads (Section 1.2), Illumina paired-end short reads, and single cell Strand-seq reads [77]. For the Strand-seq data, three separate plates at different dates were used, each containing 96 cells for the H2087 cell line, and only one plate for the BL2087. Table 4.1 shows information about average coverage and sequence length for each technology for both cell lines.

Single-cell template strand sequencing (Strand-seq) is a technology that uniquely resolves the individual homologs withing a cell by limiting the sequencing to template strand of the DNA during replication. This method exploits the directionality of DNA, distinguishing each strand based on its 5′–3′ orientation. By culturing cells with a thymidine analog (bromodeoxyuridine or BrdU) for one cell division cycle, incorporates it in the nascent strand during DNA replication; this is followed by the degradation of the nascent strand to isolate and sequence the template strand [77, 228]. This approach bypasses genomic preamplification and avoids amplifying labeled nascent strands to preserve directionality of the template strnad [228]. The resulting single-cell libraries are multiplexed, pooled, and sequenced on an Illumina platform [77]. Strand-seq offers several capabilities, it can sort long-reads or contigs by chromosome, which improves *de novo* assembly [92], order and orient contigs [191], and provides a chromosome-wide phase signal regardless of physical distance [213, 211, 229, 115].

4.3 Results

4.3.1 Genome Assembly

To assemble the cell lines, we used both the Strand-seq and HiFi reads together in a pipeline called PGAS [212]. First, Strand-seq libraries usually require an initial quality control step, which has been conventionally done manually by domain experts. However, in [102], they developed the Automatic Selection of High-quality Libraries for the Extensive analYsis of Strand-seq data (ASHLEYS) pipeline. This pipeline uses a linear support vector classifier (SVC) trained on a large data-set evaluated by domain experts for Strand-seq cells quality control. We used ASHLEYS to assess the quality of our Strand-seq data on both cell lines, we

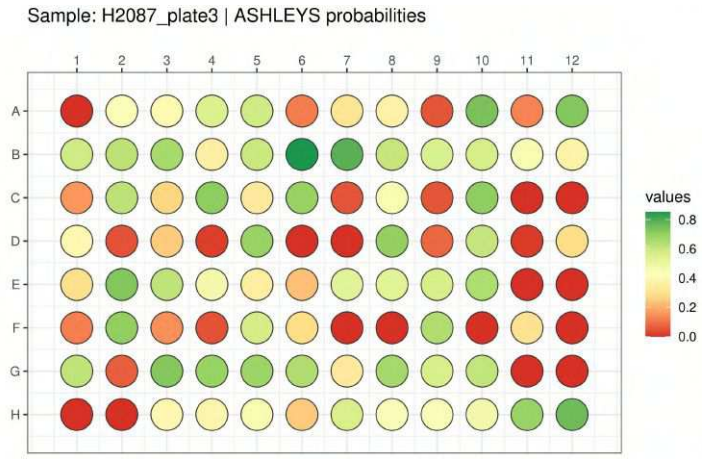


Figure 4.1: Example of how ASHLEYS report the probabilites for each cell in the Strand-seq sequencing plate. The pipline also produces cell selections based on a probability cutoff, these are shown in Supplementary Figure C.1

then only used the cells that ASHLEYS predicted as good cells in the assembly step. Figure 4.1 shows an example of the probabilities that ASHLEYS produces for each cell on a plate. Here, we use the cells that have a probability of over 50%. Supplementary Figure C.1 shows the predictions for all the plates.

The Phased Genome Assembly using Strand-seq (PGAS) pipeline is a comprehensive workflow designed to produce high-quality, haplotype-resolved diploid genome assemblies by integrating long-read sequencing data with Strand-seq technology [212]. The process of PGAS begins by generating *de novo* assemblies from the long reads; this assembly is an unphased or "squashed" haploid assembly. Subsequently, Strand-seq reads are aligned to the contigs generated in the assembly step to infer their strand inheritance patterns, and using this pattern, we can cluster the contigs so that, ideally, all contigs in a cluster are from the same chromosome. In the next step, both long reads and Strand-seq data are aligned back to the clustered contigs to call heterozygous single-nucleotide variants (SNVs), which serve as markers for phasing. These SNVs are phased globally using WhatsHap [165], combining Strand-seq, and PacBio reads to reconstruct chromosome-length haplotypes [212]. The phased SNVs are then used to tag and separate long reads by haplotype, followed by independent *de novo* assemblies for each parental homolog. Several tools can be used for the long-reads assembly within PGAS, however, in this chapter, hifiasm [37] is used to perform the assembly step.

Table 4.2 shows basic statistics for the assemblies produced by PGAS using the HiFi long-reads and the selected high-quality cells from the Strand-seq reads. We can see from the assembly statistics that the assembly for the matched normal cell line BL2087 has fewer contigs and a higher N50 score compared to the cancer cell line H2087. This is expected as PGAS and hifiasm are designed for deiploid genomes and aim to produce a diploid assembly. Moreover, due to the structural variant accumulation [51], subclonal heterogeneity [263],

		Num. Contigs	Assembly Size (bp)	N50 (bp)
BL2087	Hap.1	853	3,146,273,789	53,691,686
	Hap.2	723	3,129,660,653	66,540,667
H2087	Hap.1	1746	3,254,831,318	20,826,101
	Hap.2	1544	3,214,936,662	24,144,225

Table 4.2: Statistics on the `PGAS` assemblies for both cell lines. N50 here is defined as the contig size where half of the genome sequence is covered by contigs larger than or equal to it. Generally, the bigger the N50 value is, the better the assembly quality.

chromothripsis [256], breakage-fusion-cycles (BFB) [147], and other genome rearrangement events in cancers result in a lower quality assembly compared to germline cell lines. Moreover, Figure 4.2 plots the distribution in the number of contigs and the number of alignments for each chromosome for both cell lines. We can see that there is a correlation between the number of contigs and number of alignments. However, we also notice that for some chromosomes, the number of alignments is much higher than the number of contigs. For example, Chromosome 8 in BL2087 and Chromosome 2 in H2087, they have a small number of contigs but a high number of alignments. This can happen when the assembler is able to assemble a complex region, that the aligner later struggles to align contiguously. Figure 4.3 shows the alignments along the reference of the contigs produced by `PGAS` for H2087 for chromosome 8, and the copy numbers. The plot shows clear fragmentation of both haplotypes and elevation of the copy number, which can point to complex genomic rearrangements events.

4.3.2 Structural and Copy Number Variation Calling

The detection of structural variants is an important task in many genomic studies, especially in cancer research, due to their role in cancer development [51]. Therefore, many methods, algorithms, and software tools have been developed using different sequencing technologies for SV detection [3]. To this end, we performed SV calling and copy number variation (CNV) calling using several software tools and methods.

For the SV calling, we used five different SV callers; the first set of SV callers include `Delly` [215], `pbsv` [193], and `Sniffles` [252] which are sequencing-based callers, i.e., they use the long reads to call the variants. The other callers are `PAV` [70] and `SVIM-asm` [114] which are assembly-based callers, and they use the contigs to call the variants. Using two different approaches increases the chance that any structural variants not detected by one group of callers will be detected by the other group. Table 4.3 shows the number of variants for each category of structural variants for both cell lines. Furthermore, the commands and software tool versions used for long-reads alignment and SV calling are described in Supplementary Sections C.1 and C.2.

We see from Table 4.3 that the number of SVs in the matched normal sample is higher

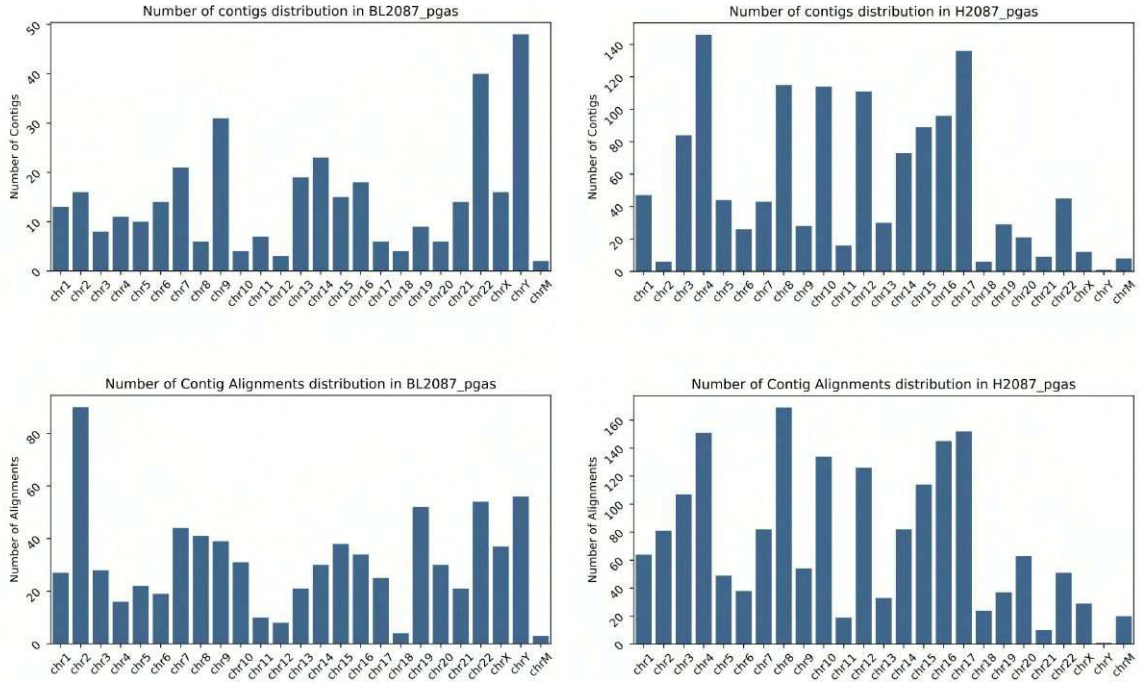


Figure 4.2: This plot shows the distribution of the number of contigs and the number of alignments for each chromosome and for both cell lines. The alignments here were done with `minimap2`.

than the ones in the cancer sample. This might sound counter-intuitive, given that cancers go through more mutations and accumulate SVs. However, one reason is related to the difficulty for callers to call SVs in cancer samples due to their complexity [271], another reason is the loss of parts of the chromosome (aneuploidy) and complex rearrangements [15]. We also see that assembly-based callers are able to call more variants compared to the others. This can be attributed to the fact that assembly-based callers are better at detecting large SVs that long-read-based callers fail to detect [160]. Moreover, they are more robust against coverage fluctuation [160]. Figure 4.4 for example, shows the distribution of SVs for all callers along chromosome 8 for the H2087 cancer cell line. We see that around the centromere, `PAV` reports more variants compared to the other callers. The centromere region is highly variable and contains complex repeats, which may explain why assembly-based callers may call variants there, whereas read-based callers produce poor quality alignments for such regions and discard them. However, this does not necessarily mean that the assembly-based callers are calling true structural variants in these complex regions, and could be false positives. Supplementary Figure C.2 plots the numbers of SVs across each chromosome per caller and per sample.

Copy number variations are a type of duplication or deletion, i.e., they are a type of structural variation. Here, the CNV we detect is more on macroscale level, based on the depth of alignment over the reference, usually followed by a normalization step to calculate the gains or the gains or losses in copy number across these windows. We used the three

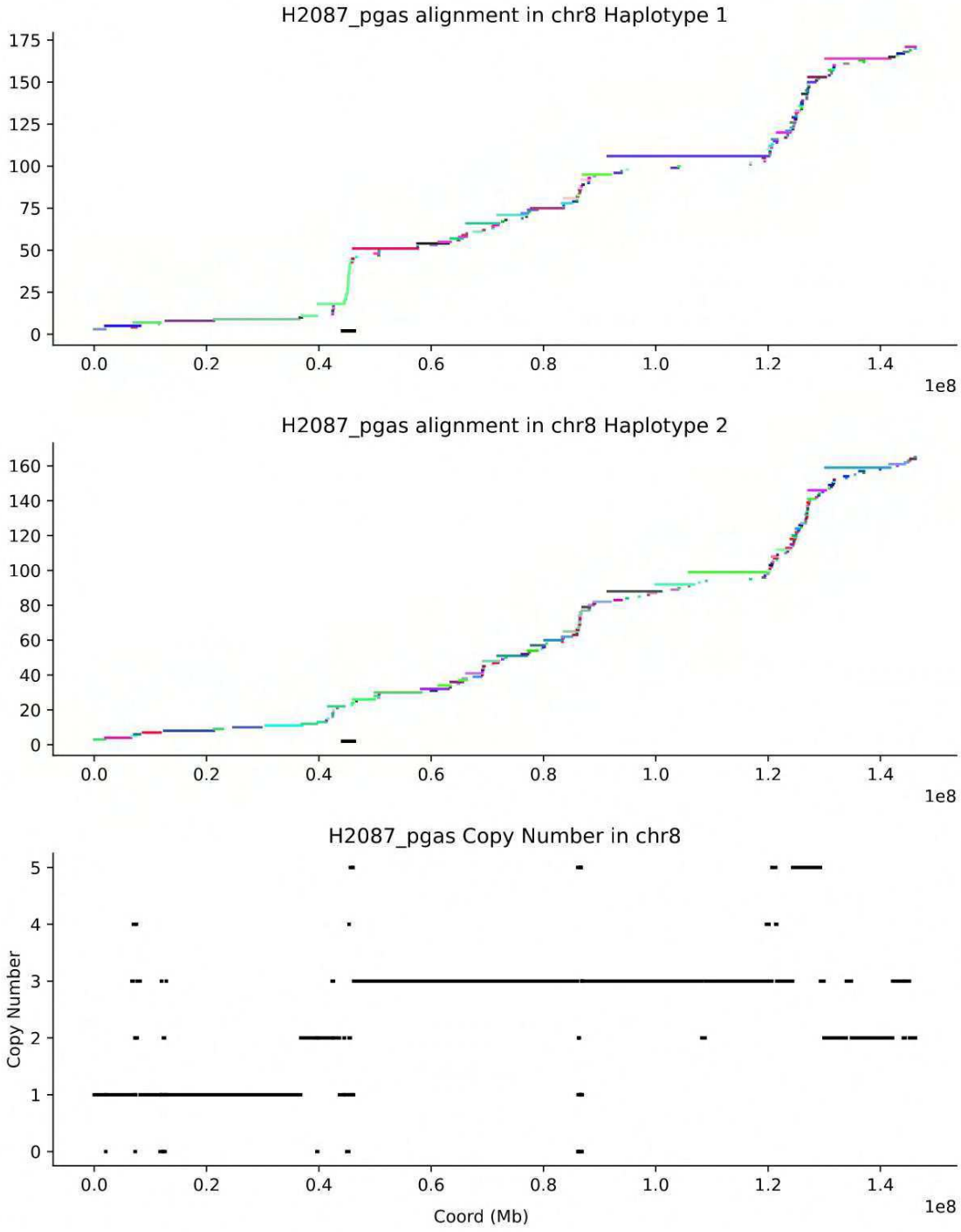


Figure 4.3: This plot shows the PGAS assembly contig alignments and copy number of chromosome 8. While both haplotypes are fragmented, we can see that haplotype 2 is more so. Looking at the copy number in the bottom plot. Looking at the copy number, we see it is elivated which could indicate complex genomic rearragnements. Hence, the poor quality of the assembly.

H2087	DEL > 50bp	INS > 50bp	INV	DUP	BND
PBSV	11,825	12,290	20	3,108	442
Sniffles	11,956	10,146	100	100	231
Delly	12,137	9,990	156	248	255
PAV	15,687	15,833	50	0	0
SVIM-asm	16,627	16,215	38	81	557

BL2087	DEL > 50bp	INS > 50bp	INV	DUP	BND
PBSV	14,036	14,169	26	3,688	86
Sniffles	13,878	11,343	45	82	124
Delly	14,577	11,153	90	189	138
PAV	19,299	19,284	51	0	0
SVIM-asm	18,728	18,415	31	83	205

Table 4.3: Structural Variant number across the 5 callers used for the H2087 cancer sample and the matched normal BL2087

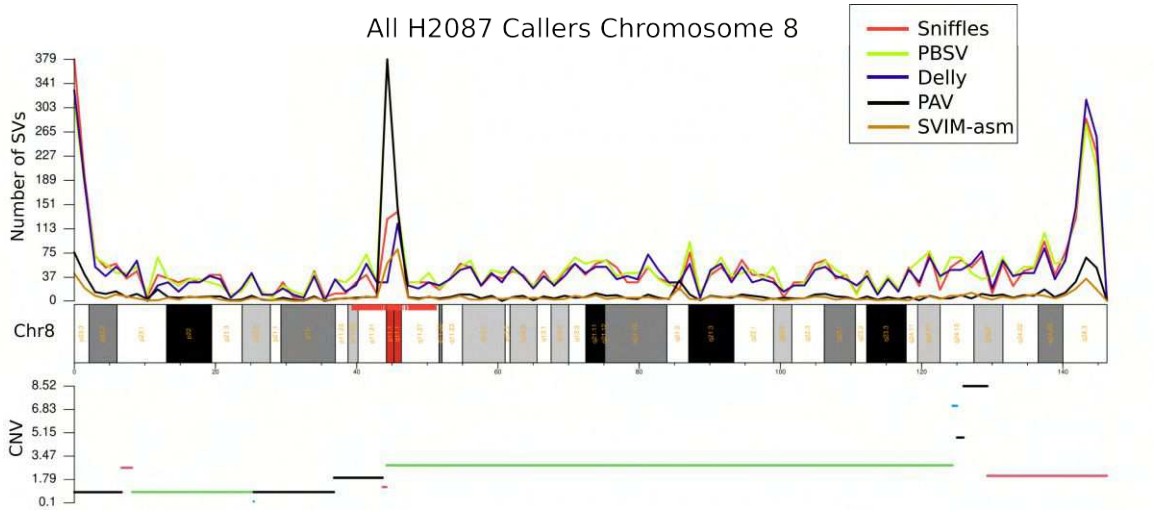


Figure 4.4: Plotting the distribution of the SV calls from the 5 callers along chromosome 8 of the H2087 cancer cell line. We can see that around the centromere (highly variable regions), `PAV` was able to call many more SVs compared to the other callers, which could explain why assembly-based callers have a higher number of calls.

sequencing technologies available for calling the CNVs. For Strand-seq sequencing, we used `Mosaicatcher2` [283] which is based on the older version developed here [229], and we only used the high-quality cells selected by `ASHLEYS`. For CNV calling on both short Illumina reads and long PacBio HiFi reads, `Delly` was used. Commands and versions used can be found in Supplementary Section C.2. Figures 4.5 and 4.6 show the copy number variations for all three sequencing technologies and for both cell lines. We see that all three technologies show similar CNV profiles, however, with different resolutions. This further confirms the quality of the data. We see for the cancer cell line, the copy number is very variable between and within the chromosomes, which is expected in cancers [245]. As for the matched normal cell line, we see that some chromosomes, such as chromosomes 8 and 12, have an elevated copy number (3 instead of 2); this can result from chromosomal instability due to prolonged culturing of the cell line [113], or aneuploidy due to cellular stress in the cell culture [295]. This can also occur due to sequencing or lab errors, however, less likely in our case, as it was reported by all three sequencing technologies.

4.3.3 SV Calls Intersection

To intersect the SV callsets produced, we using this pipeline produced by Dr. Jana Ebler [71], however, slightly edited to work on our SV calls. The first step in the pipeline is to add all variants to a list and sort based on the start position of the variant. The second step goes through all the variants, and a variant is added to a cluster if its start position is not bigger than the end of the previous variant plus an offset (here 200). The previous end here is the maximum between the first end of the cluster and the current end. Once a cluster is generated, we check if we can "merge" the variants in the cluster, i.e., check if they match under certain criteria, which are:

- Between two variants of the same type, the reciprocal overlap is 50% of the length of the variant.
- Or, between two variants of the same type, the difference between the start position and end position is smaller than 200 and they differ by at most 50% in length.

When a cluster is merged, the variants in the cluster are compared to the first variant in the cluster; where the variants in the cluster are ordered based on the input order of the SV calls to the pipeline. The first variant in the cluster becomes the representative SV of the cluster; this means that the comparison depends on the order of the input SV calls, which can be helpful if the most accurate SV caller is first in the order. However, when we tested different combinations of input orders in our case, the differences in the intersection results were negligible. Furthermore, we validated the pipeline's intersection with `svpack match` [194] that matches the SV calls from one set to another. We compared our H2087 SV calls intersection with the `svpack` intersection and got nearly the same results, which gave us further confidence that the pipeline used is producing trusted intersections.

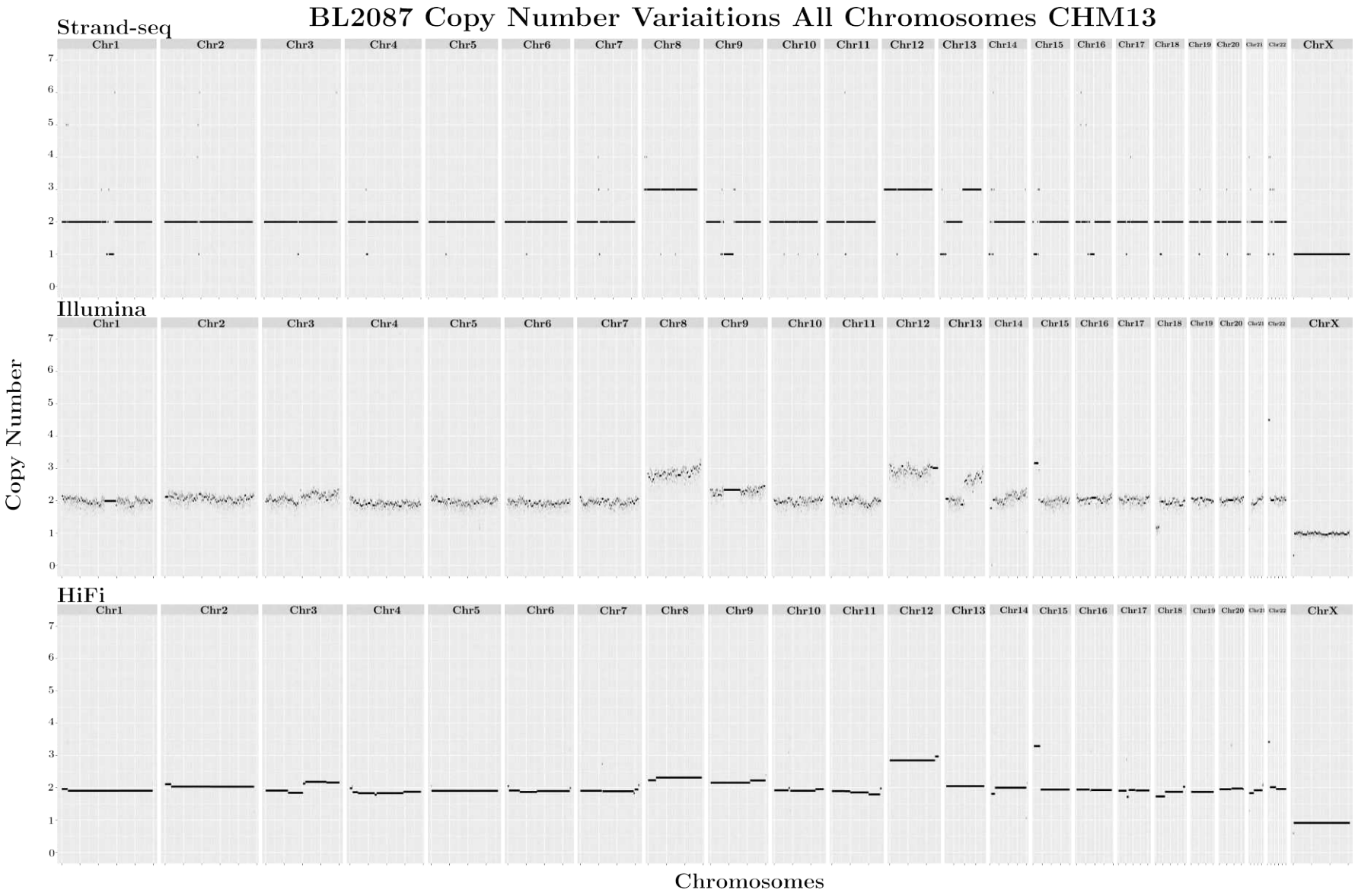


Figure 4.5: Copy Number variation for the matched normal BL2087 cell line with three sequencing technologies. For Illumina and PacBio, `Delly` was used to calculate the CNV, and for Strand-seq, `Mosaicatcher` was used

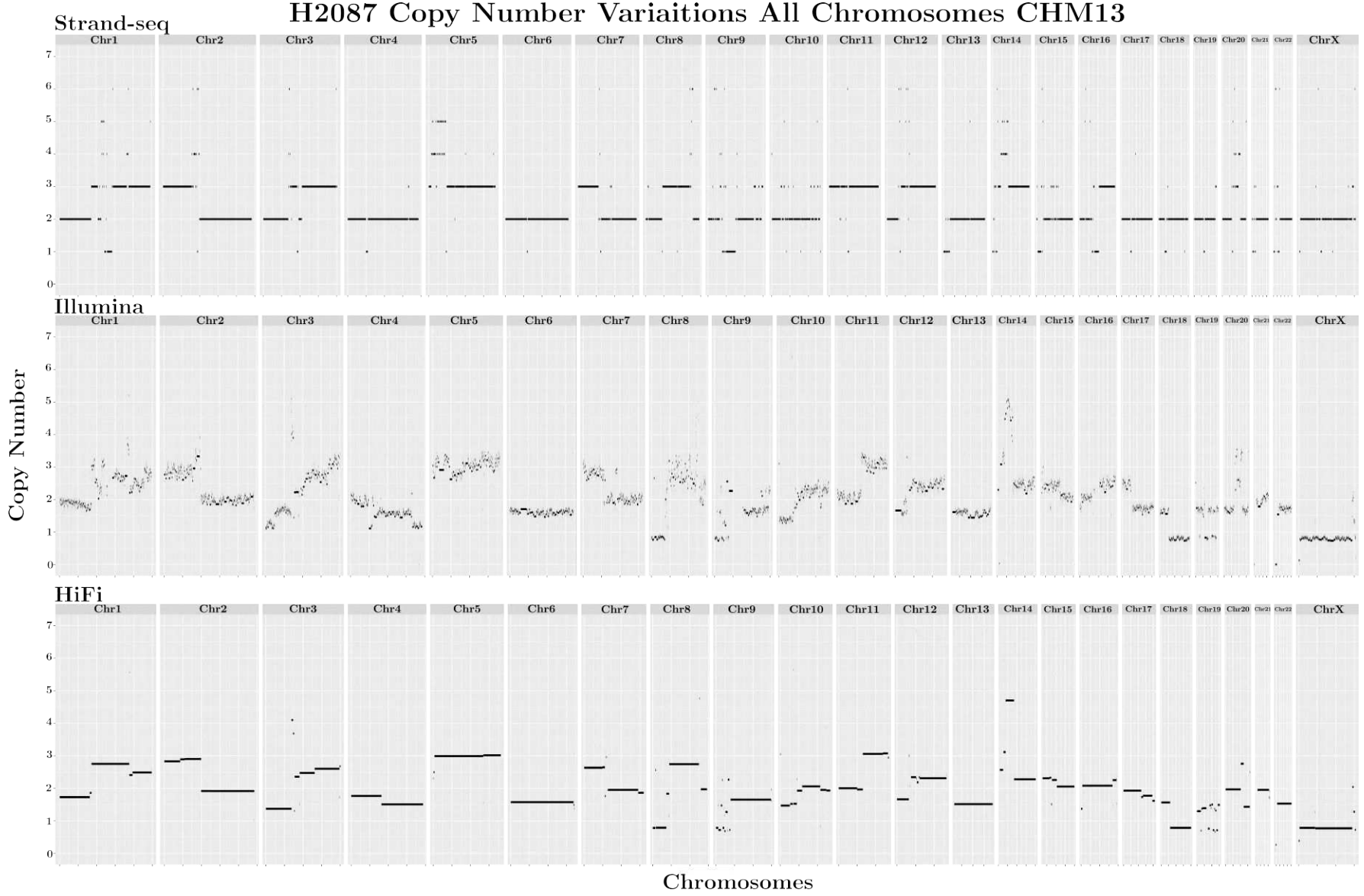


Figure 4.6: Copy Number variation for the matched normal H2087 cell line with three sequencing technologies. For Illumina and PacBio, `Delly` was used to calculate the CNV, and for Strand-seq, `Mosaicatcher` was used

Figure 4.7 shows an upset plot of the SV calls intersection for both cell lines. In both cell lines, assembly-based callers have a high number of unique calls that were not intersected with other callers. Further investigation is needed to determine why assembly-based callers have a high number of unique variants that do not intersect with other callers. One reason could be the low quality of the assemblies or their sensitivity to complex regions in the genomes that result in higher false positives. resulting in higher false positives. However, this could also indicate that their calls are not fully reliable.

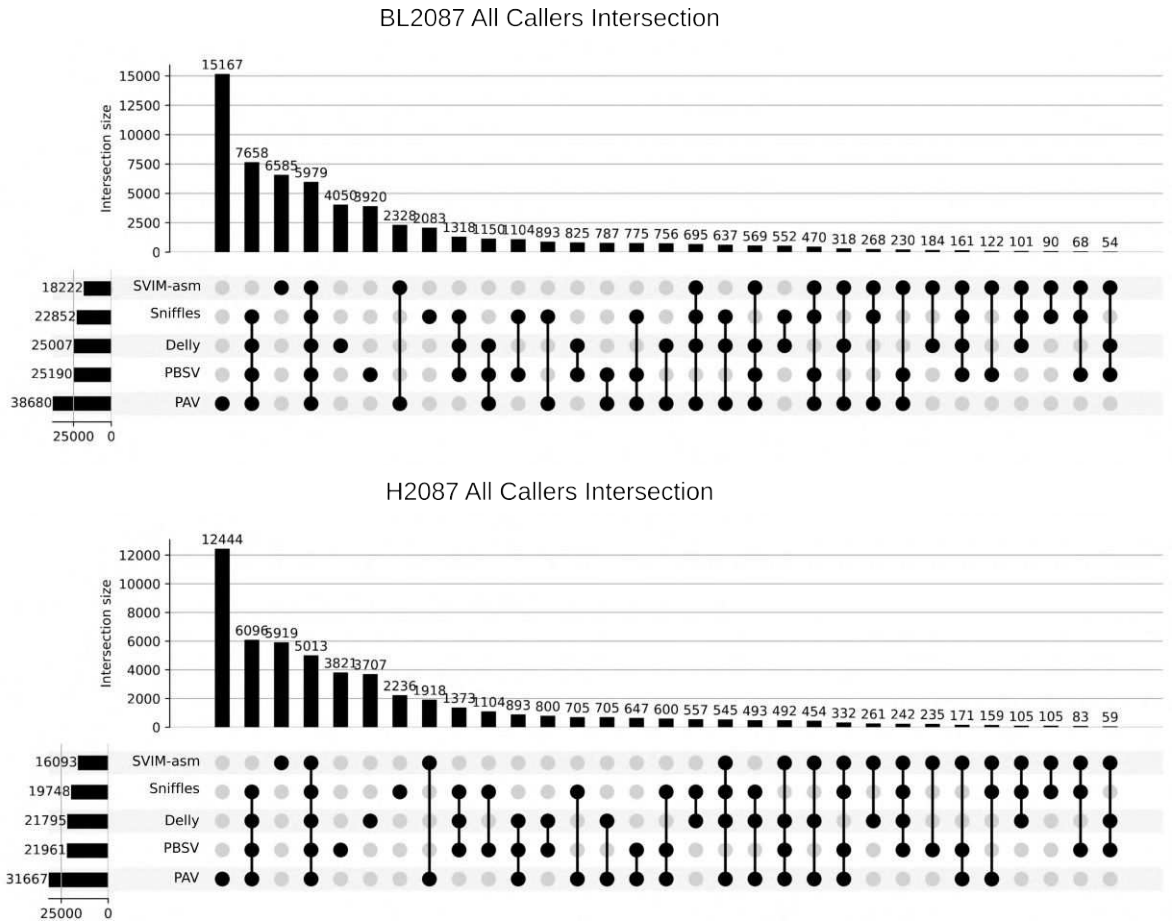


Figure 4.7: Upset plot for the intersection between the 5 callers for each cell line. We see that the assembly-based callers, especially `PAV` has a high number of SVs that do not intersect. However, we can still that there is high concordance between 4 of the 5 callers and all 5 callers.

To create a somatic set of SVs, we need to intersect the calls between the cancer and the matched normal samples, and extract only those SVs that are unique to the cancer sample. However, we want to be very strict in this intersection and eliminate potential false positives. Therefore, we first extracted a set of SVs from the cancer sample where the SV was present in at least two of the five callers. This resulted in a set of 21,629 SVs. In addition, we filtered the SVs in the centromere region, which is more prone to false positives due to the complexity of the region, which resulted in a set of 20,471 SVs. Subsequently, we intersected this set with

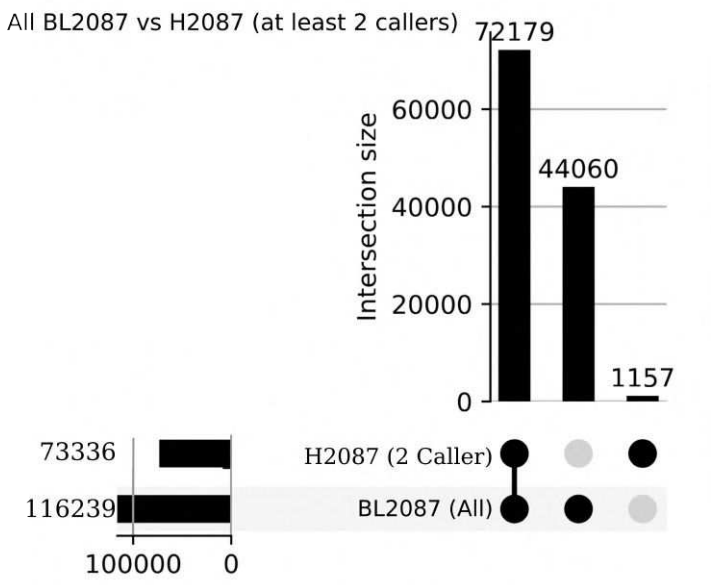


Figure 4.8: Upset plot for the intersection the SV set made from variants that showed up in at least two callers for the H2087 cancer cell line, and the complete set of all the SV calls of the BL2087. The bar plot with 1,376 represents the SVs that are only in H2087, i.e., somatic SVs.

all the matched normal calls, i.e., the combination of all five callers, which produced a set of 116,239 SVs. In this way, any SV in this "2-callers" cancer SV set that might match any SV from any caller in the matched normal is filtered out. Figure 4.8 shows the upset plot of this described intersection; we see that we retrieve 1,157 somatic SVs. The numbers of SVs for the cancer 2-callers set might seem incorrect in the figure, but the reason for this elevated number is that the pipline calculates all pair matches, i.e., one SV in the cancer sample might match to more than one SV in the matched normal sample (because we combined all the callers together). Therefore, the same SV in the cancer call set might appear more than once in the final intersection table produced, and this table is then used to produce the figure in the pipeline. However, this does not affect the final column in the upset plot (only H2087 sample), because these variants are cancer-callset specific, so they do not show up more than once in the table.

4.3.4 Graph Drawing

To understand our results better, and investigate the subclones of the cancer sample and the final set of somatic SVs obtained. We decided to focus our attention on the raw assembly graphs that `hifhiasm` produces along with the contigs. These graphs contain more information that is sometimes lost when `hifiasm` attempts to produce diploid contigs. To this end, we developed a graph toolkit we call `graphdraw` [54] that aids in such investigation and offers several useful subcommands:

(1) **Graph coloring:** Colors an assembly graph based on chromosomes. It uses the alignments of the node sequences in the graph against a reference genome to retrieve the chromosome information of the node to color it.

(2) **Contig alignment plotting:** This command plots the contig alignments along the chromosomes for one or both haplotypes, as well as the copy number variation if available. Figure 4.3 is produced using this subcommand, and the coloring of the alignments shown is random. However, alignments belonging to the same contig will be colored the same, i.e., if a contig is not completely aligned, but instead is partially aligned, each of these alignments will produce a separate line in the plot, but with the same color.

(3) **Node coloring:** This command colors the graph based on two sequence-to-graph alignment samples, where it counts for each node the number of alignments from the first sample and the second sample, then the node is colored red or blue (sample 1 or sample 2) if the number of reads from one sample crosses a threshold, by default is 90%. For example, Supplementary Figure C.3 constructed using `Mosaicatcher2` by Dr. Bernardo Rodriguez-Martin shows the potential subclones in the Strand-seq reads. Therefore, we can pool the reads of each subclone together and align them back to the assembly graph, using this command, we can then color the nodes where one subclone is predominantly represented. An example of the nodes coloring can be seen in Supplementary Figure C.4, where in this case, the graph shows a bubble chain from an assembly graph produced by mixing both cancer and matched norma HiFi long reads. The long-reads are then aligned back using `GraphAligner` [216] to the graph and used as an input for the command. Red nodes indicate one sample, the blue ones indicate the other, and the black indicate that both samples aligned across the node.

(4) **Subgraph extraction:** This command allows the user to extract a subgraph from the graph based on reference coordinates. First, the sequences in the nodes need to be aligned to a reference, then using the alignments, `graphdraw` extracts the nodes that align to these coordinates. Furthermore, it gives the user the option to extract a neighborhood of nodes around the nodes that aligned to the specified coordinates using Breadth-first search (BFS). It also produces a `Bandage` [289] acceptable CSV file with coloring and information about the nodes that aligned to the coordinates specified. The subgraph extracted and colored in Figure 4.9 was produced using this command, and it shows the effects of a somatic insertion on the unitigs[1] alignments and how this, in turn, affects the graph. Specifically, the insertion affects a subset of the unitigs and causes a bubble to form, using `graphdraw`, we supply the graph, the alignments of the unitigs, and the coordinates of the insertion, to also explore the neighborhood around the insertion, we provide `graphdraw` with a neighborhood size. Moreover, `graphdraw` colors the nodes that aligned to the coordinates provided in red, to distinguish them form the neighboring nodes/unitigs. This exercise then allows us to investigate better such structural variants.

[1] Here, we are talking about the raw unitigs graph that `hifiasm` produces that keeps all haplotype information before it does any merging to produce primary unitigs and contigs

(5) **Subgraph drawing:** Similar to the previous command, this command also extracts a subgraph from the assembly graph based on alignment coordinates. However, this command draws the graph and does not depend on other graph visualization tools. Particularly, most genome graph visualization tools visualize the graph topologically, e.g., using force-directing layout algorithms; however, without consideration to the sequence alignments of the nodes to a reference. Therefore, this command draws a subgraph but orders and places the nodes based on their alignments to a reference. Figure 4.10 shows an example from the H2087 cell line assembly graph produced by `hifiasm`, the nodes are first placed according to their alignment location on the reference, then edges are added using the information in the input graph file. Red nodes indicate an alignment on the reverse strand, where this is inferred from the BAM alignment file.

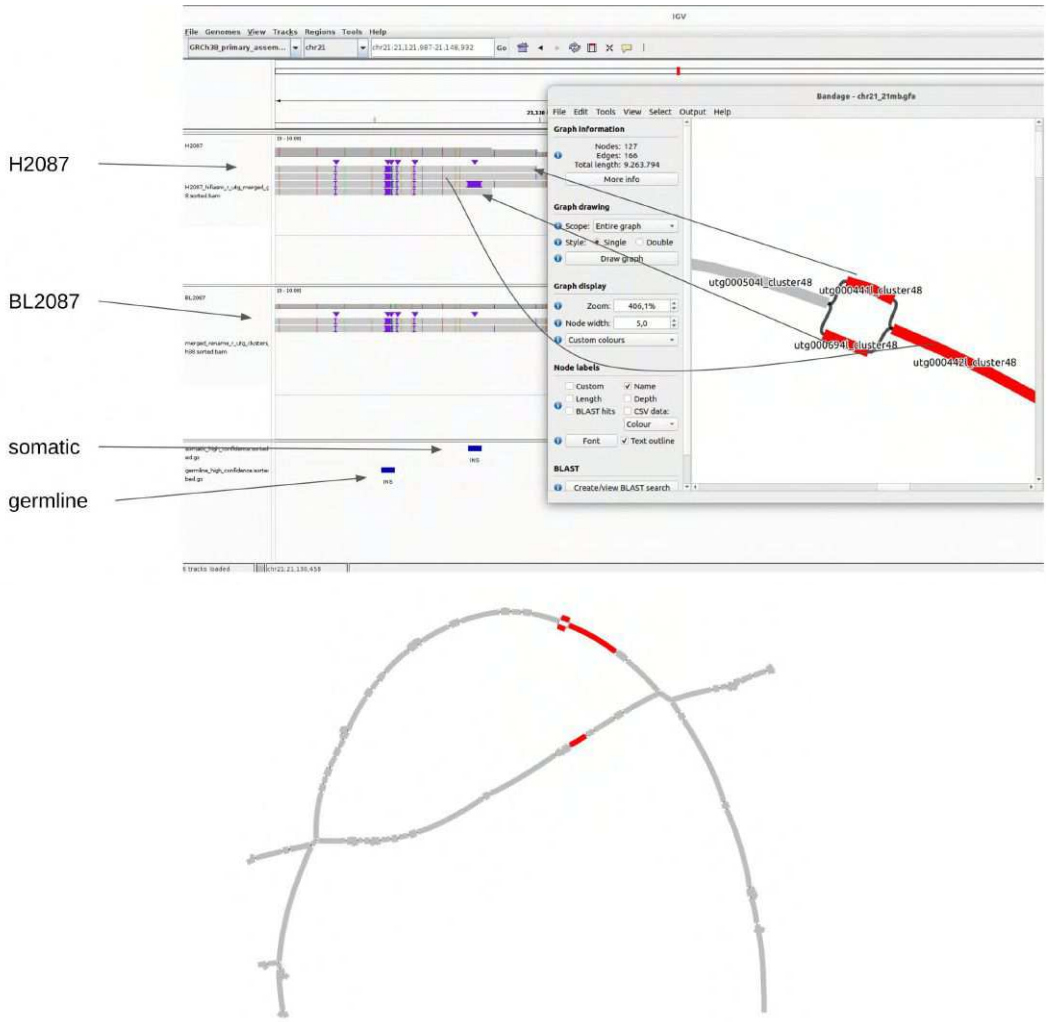


Figure 4.9: This figure shows part of the graph extracted with `graphdraw`, where a somatic insertion affects a subset of the cancer raw unitigs produced from `hifiasm` assembly, which causes a bubble in the graph. The alignments are visualized with `IGV` [221] and the graph visuzlied with `Bandage`. This bubble can indicate the difference between the two subclones.

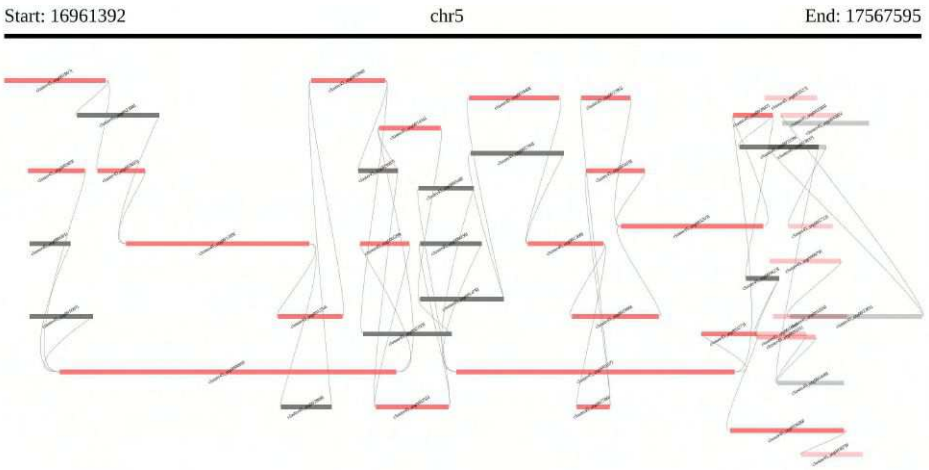


Figure 4.10: This figure demonstrates how `drawgraph` draws a subgraph with a reference. This subgraph is taken from the H2087 cell line assembly graph produced by `hifiasm`

4.4 Conclusion and Discussion

In this chapter we presented preliminary results for the analysis of the cancer cell line H2087 and its matched normal cell line BL2087. We first described the different sequencing data we have for both cell lines (Seciton 4.2). The data consists of short Illumina reads, long PacBio HiFi reads, and single-cell Strand-seq reads. We also presented some simple statistics on this data in terms of coverage and average read length for both cell lines. Next, we described the work done to assemble both cell lines (Section 4.3.1), where the single-cell Strand-seq data is leveraged in the `PGAS` pipeline to aid in better clustering and assembling the long HiFi reads. We also discussed the reasons behind having a lower quality assembly for the cancer sample. Structural variants were called on both samples using five different callers (Section 4.3.2), the callers were grouped into two groups; three callers of which are long read alignment based, and the other two are assembly-based. This guarantees a wide range of structural variants to be detected. Subsequently, we described our protocol to intersect the different SV callsets (Section 4.3.3). This is not a trivial problem, and there is not one standard way to do this task. However, this pipeline has been used in other projects successfully; we also compared its results to another tool, which gave us similar results. Moreover, to guarantee a higher quality final somatic SV set, we applied a more stringent filtering on the H2087 cell line compared to the BL2087. To investigate the assembly graphs further, we presented a simple graph toolkit called `graphdraw` that offers a set of useful utilities for further analyzing genome graphs and assemblies (Section 4.3.4). The subgraph extraction and subgraph drawing commands are particularly useful in facilitating the analysis of certain parts of the assembly graph that are of interest. For example, in Figure C.5 we used `IGV` to visualize the alignments of the raw unitigs produced by `hifiasm`, and the somatic mutations to guide us to areas of interest in the graph. In particular, we see in that figure how the somatic insertion only affects a subset of the unitigs, which might indicate that some of the

unitigs belong to one of the subclones in the cancer cell line; this also shows in the graph produced, where we see that small variants and SNPs cause smaller bubbles to form in the graph, however, the big bubbles caused by the somatic insertion. Another such example can be seen in Supplementary Figure C.5, where small variants cause a small bubble to form. However, when zooming out in the graph, we see that the somatic insertion caused a bigger bubbles to form. This kind of analysis was made possible using `graphdraw`, that allows us to quickly separate and visualize subgraphs.

There are still several other avenues to further explore in this project to obtain yet better assemblies and structural variant callsets. For example, properly analyzing the alternative contig graphs produced by the `hifiasm` assembly step of `PGAS`. These graphs consist of all contigs that the assembler was not able to assemble; these contigs could be, of course, caused by errors. However, more likely in our case, they contain important information related to the cancer subclones, where the assembler ignores in favor of fitting a diploid assembly on the sample. Another way to improve the assembly is to produce ultra-long Nanopore reads[2], which then enables us to run other assembly pipelines such as `Verkko` [218]. In our project, this has been attempted, however, the first batch of ultra-long reads were of low quality, and further lab optimizations to the technique are required before producing another set. The use of assembly graphs along with Strand-seq data has been shown to be beneficial, for example, in this study [115], they were able to produce chromosome-scale haplotypes for a diploid genome by leveraging Strand-seq signal with assembly graph topology. Therefore, we believe that digging deeper into the assembly graphs and taking advantage of the special features of Strand-seq data may be the key to unraveling the subclones and even allowing us to generate complete assemblies for each subclone in the future.

[2] `https://nanoporetech.com/document/ultra-long-dna-sequencing-kit-sqk-ulk114`

Chapter 5

EpiPAMPAS: Epistasis Detection Using Parsimonious Ancestral State Reconstruction and Mutation Counting

This chapter introduces EpiPAMPAS, an R and Python command-line tool for epistasis detection between mutations. EpiPAMPAS employs a statistical method that uses a hierarchical clustering dendrogram instead of a phylogenetic tree, Sankoff parsimony algorithm, and mutation counting for epistatic interaction detection. The efficacy of EpiPAMPAS was evaluated through testing on both simulated and real sequencing data. In the context of the simulated data, our tool was able to detect the simulated epistatic pairs of mutations effectively. In a real-world application, we tested the influenza proteins N1, N2, H1, H3, and HIV-1 envelope protein subtypes A, B, and C. Our results on the Influenza A proteins show that EpiPAMPAS detects a smaller number of interacting paris than comparable statistical approaches. However, the overlap between the detected positions with another approach is significant. Furthermore, some of the amino acids from the detected pairs have been previously identified as deleterious for viral fitness.

This chapter is based on this publication [58], of which I am a co-first author. The original statistical method was developed by the other co-first author Kristina Thedinga, and expanded by myself on protein sequences. All results and figures generated on the real data were performed by myself. Results and figures on the original statistical method and simulated data were done by Kristina Thedinga (Figure 5.1, Figure D.1, and Figure 5.3). In this chapter, materials used from [58] will be indicated.

5.1 Introduction

This section reuses some materials from [58] of which I am a first co-author

The term "epistasis" was first introduced by William Bateson in 1902, where he described the phenomenon as a form of genetic interaction whereby the expression of one gene can be either suppressed or activated by another gene [16]. Subsequently, in 1907, Muriel Wheldale Onslow provided a comprehensive account of epistasis in her paper on the color inheritance of *Antirrhinum Majus*. For example, she talked about how the color of factor I is ivory, although this is dependent on the modification of factor L: If factor L modifies factor I, then the color of factor I is magenta [287].

Since then, several definitions of epistasis have been presented and can be generalized to describe the complex interactions between genes or genetic loci that cause a phenotypical effect [203]. Studies have discussed many roles of epistasis, especially in its effects on evolution. For example, the importance of epistasis in adaptation and selection [111]. Or in protein evolution, where epistasis impacts how mutations alter protein structure and function, as it affects the evolutionary trajectories of protein by restricting or opening new pathways [255].

Other studies have also shown that epistatic interactions between mutations are possible [22, 161, 227], and more recently, in the context of compound heterozygosity of rare variant pairs within genes in humans [107]. In [141], they showed that up to 70% of rare missense mutations are deleterious, and are associated with fitness loss. However, it has been shown that epistasis then plays an important role in compensating against these deleterious mutations, and can help restore lost fitness [224, 257, 208]. Moreover, this compensatory effect plays a role in human diseases [133]. This positive compensation via epistasis is called "positive epistasis" [26]. The opposite, i.e., when the interaction between mutations causes loss of fitness is called "negative epistasis" [13]. Epistasis is also a major driver of evolution in viruses, especially that viruses evolve faster and are under stronger adaptive pressure to evade immunity. This has been observed in viruses like influenza A [259, 31, 162], other RNA viruses such as vesicular stomatitis RNA virus [232], and in the human immunodeficiency virus type-1 (HIV-1) [26].

Given the important role epistasis in evolution, the development of robust statistical and algorithmic models for its detection is imperative. To this end, many statistical methods have been developed for detecting epistasis in different scenarios and settings. For example, in this study [242], the authors looked into 36 computational methods for detecting epistasis, and categorized them into three major categories; methods that search for epistasis using exhaustive search, stochastic search, or heuristic search. Another group of research did an extensive survey on the different statistical methods for epistasis detection as well [186].

A majority of the reviewed models were based on the use of genotypes and phenotypes of the Single Nucleotide Polymorphisms (SNPs) and their interactions; however, they did not take into consideration the complete sequence. By contrast, other methods, such as those outlined in [139, 95], examined mutual information using sequence alignments. Nevertheless, these methods did not consider the phylogenetic relationships between the sequences, a consideration that subsequent methods did incorporate [124, 243, 85].

In this chapter, we focused on the method presented in [140] for detecting epistasis in proteins using both sequence alignments and phylogeny. This method assumes that under positive epistasis, nonsynonymous substitutions at linked sites follow each other in rapid succession. The method initially constructs a phylogenetic tree from the sequences using a maximum likelihood model using `PHYML` [105]. Subsequently, their method measures an "epistasis statistic" $E(i, j)$, where i and j are designated as "leading" and "trailing" sites, respectively. This statistic basically detects an acceleration of non-synonymous substitution at the trailing site j, after a non-synonymous substitution at the leading site i. They applied their method on the sequence data of the surface proteins hemagglutinin and neuraminidase of influenza A virus subtypes H3N2 and H1N1. Furthermore, they were able to detect numerous epistatic interactions between different sites, with some having been previously confirmed *in vitro* by other researchers and associated with drug resistance against oseltamivir. [10, 175].

Here, we propose a fast and simple method that relies on hierarchical clustering of sequences instead of full phylogeny reconstruction. Additionally, instead of detecting events that happen faster or slower than expected after each other (as introduced in [140, 184]), our method, however, detects pairs of mutations that happen more than once independently in this proxy of the phylogenetic tree. We demonstrate the robustness of the method on simulated data and further apply it to Influenza A HA and NA sequences for a direct comparison with [140] and HIV-1 envelope protein from subtypes A, B, and C. We also demonstrate an agreement of our results with [140] and evaluate the spatial distribution of the epistatically interacting pairs in the protein three-dimensional structures.

5.2 Methods

This section reuses materials from [58] of which I am a first co-author

As an overview, `EpiPAMPAS` bases the detection of epistasis on a dendrogram, instead of a properly calculated phylogenetic tree, representing the evolutionary relationships between the analyzed samples. The dendrogram is built on multiple sequence alignments (MSA) of amino acid sequences using hierarchical clustering with Ward's clustering criterion [279, 176]. `EpiPAMPAS` then exhaustively goes through each pair of variable positions in the MSA and label the leaf nodes with the genotype. Subsequently, using the Sankoff parsimony

algorithm, it reconstructs the most-likely genotype of the inner nodes in the dendrogram which correspond to the ancestral states. Using the reconstructed genotypes, our method counts the number of mutations in a certain direction of one position in comparison to the other position, and tries to fit the counts to a binomial distribution of 0.5 probability. In the following subsections, the different steps will be explained in more details.

5.2.1 Constructing the Dendrogram

This section reuses materials from [58] of which I am a first co-author

EpiPAMPAS constructs a dendrogram from the sequences in the MSA using the Ward error sum of squares hierarchical clustering. The goal here is to distinguish between false positives caused by the population structure and true positives arising from epistatic interactions. This approach results in a tree that resembles a phylogenetic tree but requires less computational power to construct. We hypothesize that this representation will be a reliable one, as closely related samples will exhibit similar variant patterns, leading to their close clustering on the tree.

The leaf nodes of the tree can then be labeled with the genotype of a specific variable position under investigation. In the context of EpiPAMPAS, the genotype is binary in nature, because we look at two amino acids for each position at a time, with the tree-building process being constrained to sequences that contain one of these amino acids. As illustrated in Figure 5.1, the tree is constructed using 13 samples/sequences and is labeled with their respective genotypes.

5.2.2 Sankoff Algorithm

This section reuses materials from [58] of which I am a first co-author

The Sankoff parsimony algorithm [233, 234, 42] is an algorithm designed to reconstruct the sequences in the internal nodes of a finite tree where the leaves represent some sequences belonging to a finite alphabet, this reconstruction is performed with minimal costs, where costs are derived from a cost matrix, associating a cost for transitioning between different letters in the alphabet.

In our case, the alphabet simply consists of $\{0, 1\}$, which represents the two possible states the sample can have for a certain variable position in the sequence, i.e., the two possible genotypes in that position. To reconstruct the states of the inner nodes in the dendrogram, which correspond to the ancestral genotype in the sequence position of interest, the Sankoff parsimony is applied to the dendrogram with labeled leaves. The Sankoff parsimony algorithm counts the minimal number of evolutionary changes or mutations in a phylogenetic tree – here represented by the dendrogram – to find the most likely ancestral state. Initially, each node of the phylogenetic tree is assigned a cost vector. This cost vector

contains one cell for each possible state of the node storing the minimal evolutionary cost, which is the minimal number of evolutionary changes in the phylogenetic subtree rooted at this node. As mentioned, in our case, the states are 0 or 1. Initially, the cell in the cost vector corresponding to the evolutionary state observed in each leaf node is set to 0, since there are no evolutionary changes necessary to reach the observed state, and all other cells in the cost vector are set to infinity since they are infeasible for the leaf node. The cost vectors of the inner nodes are still unknown at this point, so they are also assigned infinity. Then, starting from the leaf nodes, the cost vectors of all inner nodes up to the root of the phylogenetic tree are calculated according to the following formula:

$$S_i^{(p)} = \min_j(c_{ij} + k_j^{(l)}) + \min_k(c_{ik} + k_k^{(r)}) \tag{5.1}$$

where i is the evolutionary state, p denotes the parent node whose cost vector is to be calculated, l and r are the two child nodes of p with already known cost vectors $S^{(l)}$ and $S^{(r)}$, and c_{ij} and $c_{i,k}$ are entries of the cost matrix C containing the costs of evolutionary changes from state i to state j, which in our case are 0 if $i = j$ and 1 if $i \neq j$. Thus, the total cost for reaching the states of the leaf nodes in the subtree of node p from a state i in node p is the cost of transitioning from state i to the states of the child nodes of p plus the minimal cost to reach the states of the leaf nodes from the child nodes.

5.2.2.1 Sankoff Algorithm Example

In Figure 5.1, we see an example of a dendrogram with 13 samples (leaf nodes), and the cost vector for each node in the tree is calculated step-by-step in the figure. Here, our cost vector $S^{(p)}$ has two possible states representing the genotype. If more than two genotypes are available for that position in the MSA, we subset our samples and take all combinations of two genotypes for a pair of positions and calculate a tree based on this subsample. In this Figure, genotypes are labeled as “o” and “x”, where the left cell in the cost vector represents the genotype “o” and the right cell represents the “x” genotype. When initializing the tree with the cost vector of the leaf node, the genotype of that leaf node gets value 0, and the other genotype is assigned ∞. For example, looking at the left pair of leaf-nodes, the first one has the genotype “x” and is initialized with $[\infty, 0]$, and the second leaf node has the genotype “o” and is initialized with $[0, \infty]$. Once the leaf nodes are labeled, we can start calculating the values for each parent node using Equation 5.1. Because our cost value is of size two, and we only have two genotypes, we can expand that equation and calculate the values for each node as follows:

$$node[o] = min(lcn[o], lcn[x] + 1) + min(rcn[o], rcn[x] + 1)$$
$$node[x] = min(lcn[o] + 1, lcn[x]) + min(rcn[o] + 1, rcn[x])$$

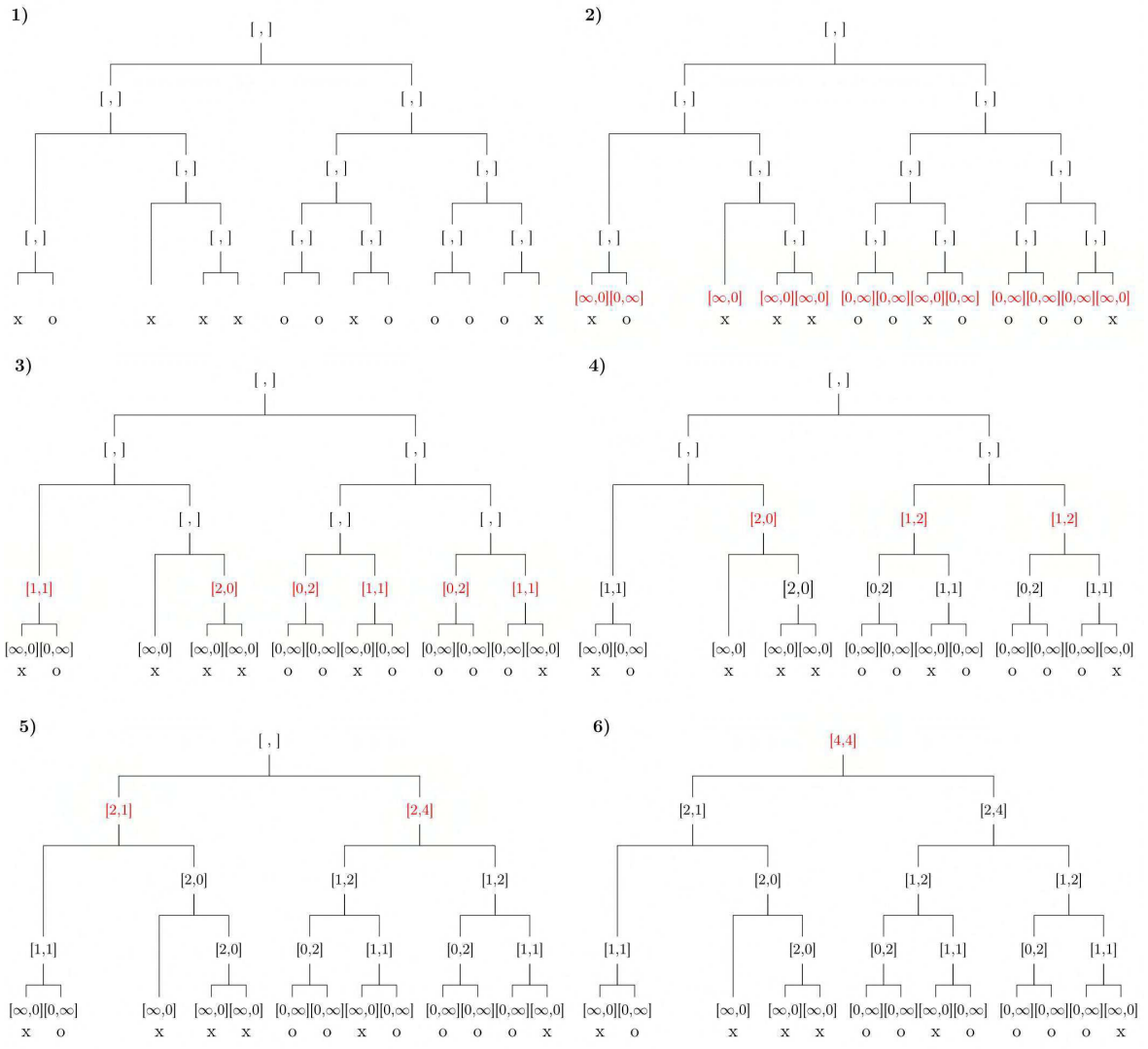


Figure 5.1: Example on how Sankoff algorithm works on a dendrogram constructed from 13 samples/sequences. In each step, we see how the score vector is calculated for each inner nodes using the two child nodes. *Figure taken from [58]*

where $node[o]$ is the value for genotype "o" in the node, and $node[x]$ the value for genotype "x"; lcn means "left-child node", and rcn means "right-child node". For example, looking at step 4 in the figure, in the middle we have the calculated vector $[1, 2]$, under that node, we have two leaf nodes ($[0, 2]$, and $[1, 1]$), the first value "1" was calculated with ($min(0, 2+1)+min(1, 1+1) = 0+1 = 1$), and the second value "2" was calculated with ($min(0+1, 2)+min(1+1, 1) = 1+1 = 2$). The same goes for the root node $[4, 4]$, this was calculated using the two child nodes ($[2, 1]$ and $[2, 4]$), and the first value calculated with ($min(2, 1+1)+min(2, 4+1) = 2+2 = 4$), and the second value calculated with ($min(2+1, 1)+min(2+1, 4) = 1+3 = 4$).

5.2.3 Mutation Direction and Counting

This section reuses materials from [58] of which I am a first co-author

As mentioned in Section 5.2.1, samples or sequences that are close together will probably have a similar mutation profile and cluster together in the tree. However, variants caused by epistasis are more likely to occur by samples that are less closely related, and would be located further from each other on the tree. Furthermore, `EpiPAMPAS` takes into account the order in which mutations occur within the dendrogram to distinguish between epistatic and random effects. The underlying idea is that if there are epistatic interactions between two protein locations that are being mutated over time, mutation events at these two locations will most likely not occur independently of each other due to selection pressure. For instance, if two locations a and b are linked by epistasis and a mutation introducing a variant at location a takes place that decreases the fitness of the organism, it is likely that location b will also be mutated, compensating the effect of the variant at location a. On the other hand, if locations a and b both have variants compensating each other, it is less likely that a mutation changing only one of the locations a or b occurs.

After reconstructing the ancestral evolutionary states of the pair of protein locations that are analyzed for epistasis with the Sankoff parsimony, each of the two protein locations is analyzed for mutation directions with respect to the other location. To this end, mutations in both directions are counted across the whole dendrogram for each of the two locations. If somewhere in the dendrogram the protein location under consideration mutates to the same state (i.e., 0 or 1) as the other location in the pair at the same node in the dendrogram, this is considered a *same direction* mutation, while mutations leaving the pair of protein locations in different evolutionary states are counted towards the *opposite direction* mutations. Pairs of protein locations that are mutated independently are expected to show comparable numbers of *same direction* and *opposite direction* mutations and should thus fit into a binomial distribution with a probability of 0.5, while location pairs linked by epistasis do not mutate independently and are expected to deviate from the binomial distribution. Hence, to detect epistasis, we apply two-tailed binomial tests with probability 0.5 on the counts of *same direction* and *opposite direction* mutations obtained from the dendrogram for each of the locations in the pair to obtain p-values. Supplementary Section D.1 and Supplementary Figure D.1 expand more on this step.

5.3 Implementation

This section reuses materials from [58] of which I am a first co-author

`EpiPAMPAS` is mostly written in `R` and requires a minimum number of external libraries to run. A `Python` preprocessing script is used to convert an MSA into several tables, which is then utilized by the `R` module. Figure 5.2 presents an example of an MSA comprising

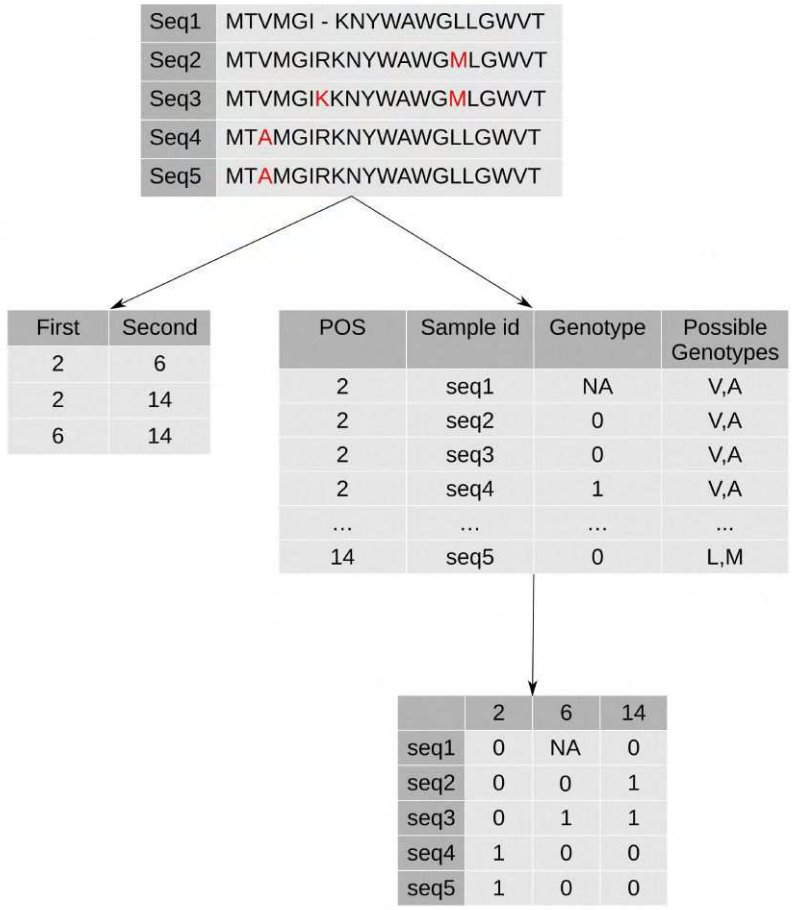


Figure 5.2: An example of an MSA with 5 sequences and 3 variable positions, the middle table to the left is the 3 possible pairs of these 3 positions in this MSA, the middle right table is the VCF table with the information related to each sample (0 indexing is used here), the variant position and the variant value. The last table is the matrix representing the VCF-style table that is used to build the dendrogram. *Figure taken from [58]*

five sequences. It illustrates the preprocessing of these sequences by the Python module, resulting in the generation of a VCF and a table listing all possible pairs of variant loci. Subsequently, the VCF is converted into a matrix, with rows representing samples and columns representing potential variant positions, and values denoting the "genotype" in the `R` module. For each pair of positions, only the samples that contain that pair of positions in the matrix are retained. That is to say, in the event that a gap was present at a given position, such as in sequence 1 position 7, the value in the matrix will be designated as "NA". The matrix is then used to construct the dendrogram through the implementation of hierarchical clustering, and the most probable genotype is reconstructed (the genotype that necessitates the minimal number of mutations) employing Sankoff parsimony.

5.4 Results

5.4.1 Simulated Data

This section reuses materials from [58] of which I am a first co-author

To test that our dendrogram-based algorithm does actually detect a pair of mutations that independently happen in the same direction, simulated dendrograms were used. In the first step, balanced dendrograms are constructed from n samples, where each node has exactly two child nodes. We then mark the nodes with the same genotype "0" which denotes that for a pair of positions, both positions have the wildtype genotype. We start introducing same direction mutations where both positions mutate to have the same genotype, and opposite direction mutations where they have different genotypes. This introduction is done with different probabilities, where p_{same} is the probability of introducing same direction mutations, and $p_{opposite}$ is the probability of introducing opposite mutations. We vary p_{same} between 0 and 0.5 with 0.05 steps, and $p_{opposite}$ is defined with the following equation: $p_{opposite} = \frac{p_{same}}{f}$ where f is a factor that relates between p_{same} and $p_{opposite}$, and takes a value between 1 and 4.

Mutations are introduced starting from the root node of the tree, where each node is mutated with probability $p_{same} + p_{opposite}$, and if the mutation is introduced at some node v, the genotype is propagated through the subtree rooted at v. Here, we limit the p_{same} to 0.5 to make sure to never have a total probability of over 1, when summing up both probabilities. This mutation procedure is repeated 100 times for each parameter combination and each number of samples (10, 50, 100, 500) to get a representative pool of results.

Testing on simulated data confirms the validity of our approach (Figure 5.3). We see that the higher the value for p_{same}[1], the lower the p-values are detected by our method. Moreover, larger values of f (which result in smaller values for $p_{opposite}$ compared to p_{same}) also result in more significant p-values. We can also see that the number of samples affects the p-value reported by our method, where the bigger the sample size is, the more significant the reported p-value is.

5.4.2 Viral Data

This section reuses materials from [58] of which I am a first co-author

Two viral datasets were used for testing our method. First, we used the same dataset of Influenza A hemagglutinins and neuraminidases of subtypes H1, H3 and N1, N2, respectively, as in the [140] study in order to enable a direct comparison with their results. The second dataset is the HIV-1 envelope gp160 glycoprotein (Env) sequences from HIV-1 subtypes A, B, and C, taken from the Los Alamos National Laboratory (LANL) HIV Sequence

[1] Higher p_{same} indicates more mutations in the same direction, but only if p_{same} is larger than $p_{opposite}$

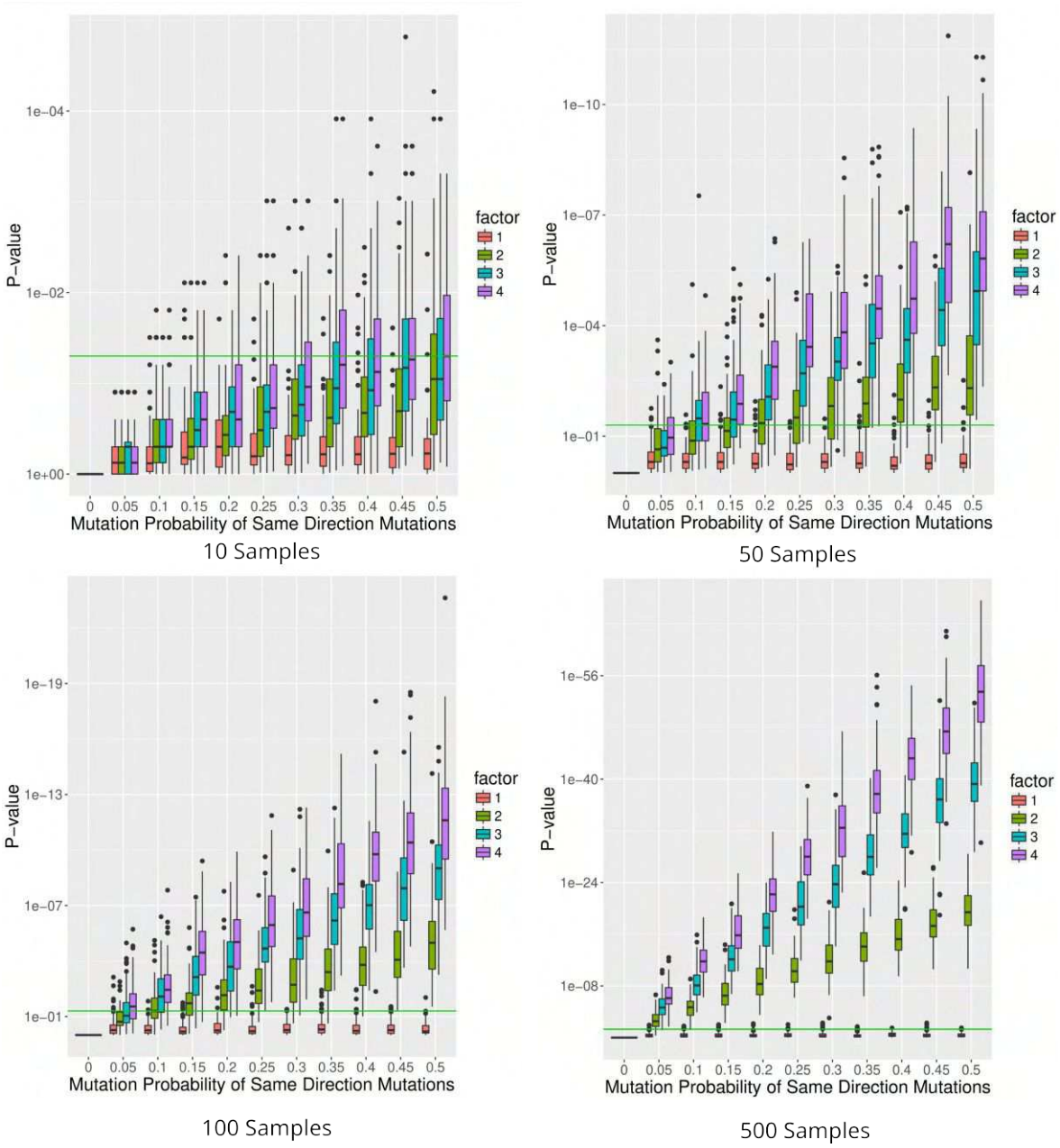


Figure 5.3: Boxplots for the simulated trees. Different sample sizes were used for this test. The x-axis represented the different p_{same} probabilities, the different colors for the boxes represent the different f values, and the y-axis represents the p-value measured by our method. The green line indicates the 0.05 p-value. *Figure taken from [58]*

Database [142].

We ran EpiPAMPAS on each MSA of the proteins, EpiPAMPAS then produces a table with all possible variable pair-positions for each MSA and the corresponding p-value for the pair after counting the mutations and fitting the binomial distribution as described in Section 5.2.3. As EpiPAMPAS only considers two amino acids at one position at a time, in the cases where

more than two amino acids can be observed in a certain position, we take all combinations of two amino acids when running `EpiPAMPAS`, i.e., for a pair of positions, our algorithm can be run more than once to investigate all combinations, which can lead to inflated p-values due to multiple testing. This is corrected with Bonferroni correction in those cases. Furthermore, to investigate the potential functional impact of the detected epistatic interactions, we looked into their location in the corresponding protein three-dimensional structures. For each protein, a structure from the Protein Data Bank [18] was chosen. For the Influenza proteins we used the same structures as described in [140], and for the HIV-1 envelope we searched for the most complete structures of gp120 and gp41 (the end products in the env gene) from the corresponding subtypes using `StructMAn` [101]. Tables 5.1 and 5.2 show the number of sequences for each protein MSA, the number of pair-positions with a significant p-value after the Bonferroni correction, and the protein structures chosen for each protein.

Protein	**H1**	**H3**	**N1**	**N2**
Number of Sequences	1219	2149	1836	2339
Number of Pairs p-value <0.1	7	33	19	18
Number of Pairs p-value <0.05	3	23	1	7
Number of Pairs p-value <0.01	1	5	0	3
Structure PDB ID	1RUZ	2VIU	3BEQ	1NN2

Table 5.1: Real-world dataset used in this study on Influenza A hemagglutinin and neuraminidase sequences. Bonferroni multiple tests correction applied. *Table taken from [58]*

Protein	**HIV-1 Sub. A**	**HIV-1 Sub. A**	**HIV-1 Sub. A**
Number of Sequences	223	2035	1265
Number of Pairs p-value <0.1	310	485	320
Number of Pairs p-value <0.05	145	348	202
Number of Pairs p-value <0.01	53	183	105
Structure PDB ID	6B0N	6B0N	6MYY

Table 5.2: Real-world dataset used in this study on HIV-1 Env sequences. Bonferroni multiple tests correction applied. *Table taken from [58]*

For the Influenza proteins, after filtering pairs with p-value less than 0.05, we were left with only very few pairs, therefore, we opted to take a more relaxed cutoff of 0.1. Notably, in [140], they predicted many more epistatically interacting pairs, in line with their mentioned higher false discovery rate (FDR). Nevertheless, the results of both methods agree very well (Figure 5.6, right). Both methods detect the same sets of amino acid positions; however, the predicted interacting pairs (Figure 5.6, left) are largely different. Only for H1, we detect three interacting pairs, which were all discovered by [140].

To calculate the pairwise distance between the detected pairs, the sequences were mapped

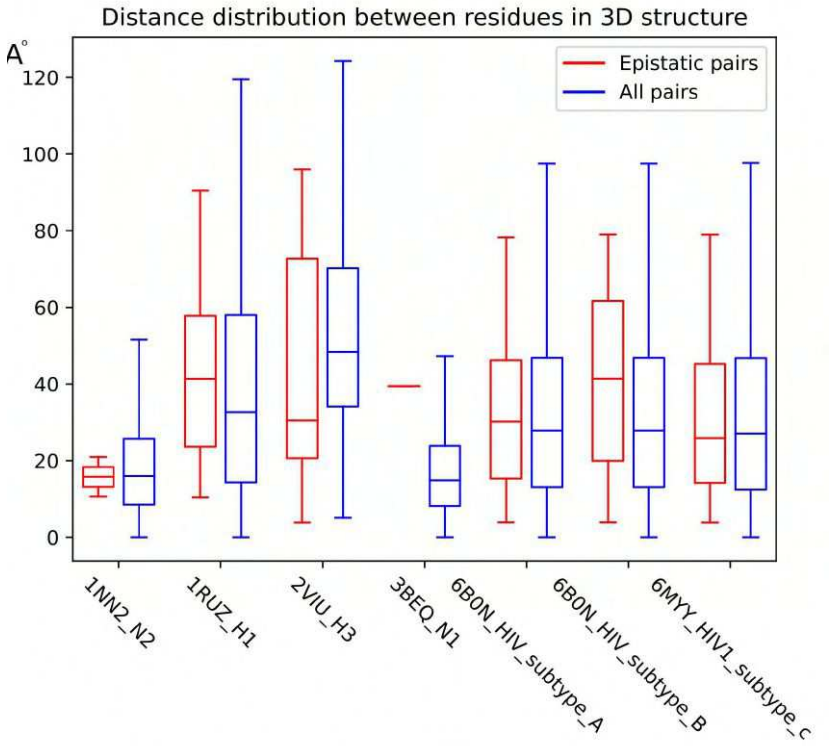


Figure 5.4: Boxplot with distance distribution of all pairs in a 3D structure (in blue) and only the epistatic interacting pairs that we detected (in red). *Figure taken from [58]*

onto their corresponding structures, and the pairwise distances between the nearest atoms in the corresponding amino acids were calculated. We compared the distance distribution of detected pairs with the background distance distribution of all amino acids in the corresponding protein three-dimensional structures using the Wilcoxon signed-rank test (Tables 5.3 and 5.4). We see that in influenza, the proteins H1, H3, and N1 had a significant p-value but only in H3 the distance was significantly smaller (Figure 5.4). However, with H1 and N1, the number of pairs was much smaller compared to H3. For HIV1 we see that the p-value is significant for subtypes B and C, but looking at Figure 5.4, the average distances are not that different. Moreover, the corresponding AUC values are low, indicating that this significance probably is simply due to the higher number of detected pairs in these datasets.

In the HIV1 dataset, looking at the overlap between the pairs for each subtype, we found that between subtypes A and B only 2 pairs were the same, between A and C also only 2 pairs, and between A and C only 4 pairs overlap; this is a small number compared to the actual number of pairs for each subtype which is 145, 348, and 202 for subtypes A, B, and C respectively. This points to the evolutionary diversity between the subtypes that has been seen before, which adds to the challenges of effectively controlling the virus [264].

In influenza, some mutations have been associated with resistance to oseltamivir, e.g., mutation in position 274 the mutation from Histidine to Tyrosine (H274T) has been shown

Protein	H1	H3	N1	N2
Test statistic W	35391.5	908535.5	6539.5	6885
P-value	0.04	0.005	0.051	0.466
AUC	0.79	0.35	0.97	0.48

Table 5.3: Comparing distances between potentially epistatically interacting pairs detected using our method for p-value threshold of 0.1 with all pairwise distances in the structure. *Table taken from [58]*

Protein	HIV-1 Sub. A	HIV-1 Sub. B	HIV-1 Sub. C
Test statistic W	309836.5	1005287	642826
P-value	0.308	0.0002	0.088
AUC	0.52	0.6	0.54

Table 5.4: Comparing distances between potentially epistatically interacting pairs detected using our method filtered against distances between all amino acid pairs for p-value <0.05. *Table taken from [58]*

to give the virus resistance to oseltamivir drug [175]. Moreover, mutations at position 222 between Arginine and Glutamine (R222Q) and at position 234 between Histidine and Tyrosine (H234Y) regain the viral resistance against the drug [23]. Both R222Q and H234Y were detected by our method with significant p-values.

Comparing our detected positions in the HIV sequences with a list of biologically relevant residues, such as those in the CD4 binding site or antigenic epitopes taken from [109], looking at Table 5.5 we see that the fraction of amino acid residues in biologically important regions from [109] among detected potentially epistatically interacting positions is around 39%, 43%, and 42% for HIV-1 subtypes A, B, and C respectively. For all amino acids in Env, the fraction of the ones in biologically relevant regions is around 32%. To see if this increase of percentage is significant, we ran a Fisher exact test, and we see that for HIV-1 subtypes B and C, the p-value is smaller than 0.05. The structural classes (e.g., interactions with other proteins or small ligands, core and surface residues) of all detected positions were evaluated using `StructMAn` and compared to the background distribution of structural classes for all Env amino acids (Figure 5.5). Potential epistatically interacting positions from all three subtypes follow a similar trend with slight depletion from the core compared to the background. Interestingly, there is no enrichment on the protein-protein interaction interfaces, which is in a slight contrast with the previous observation, since most biologically relevant residues from [109] signify an interaction with another protein, and epistatically interacting residues are enriched among them. Hence, epistatically interacting residues must be less common in the interaction interfaces between the Envelop subunits in the trimer.

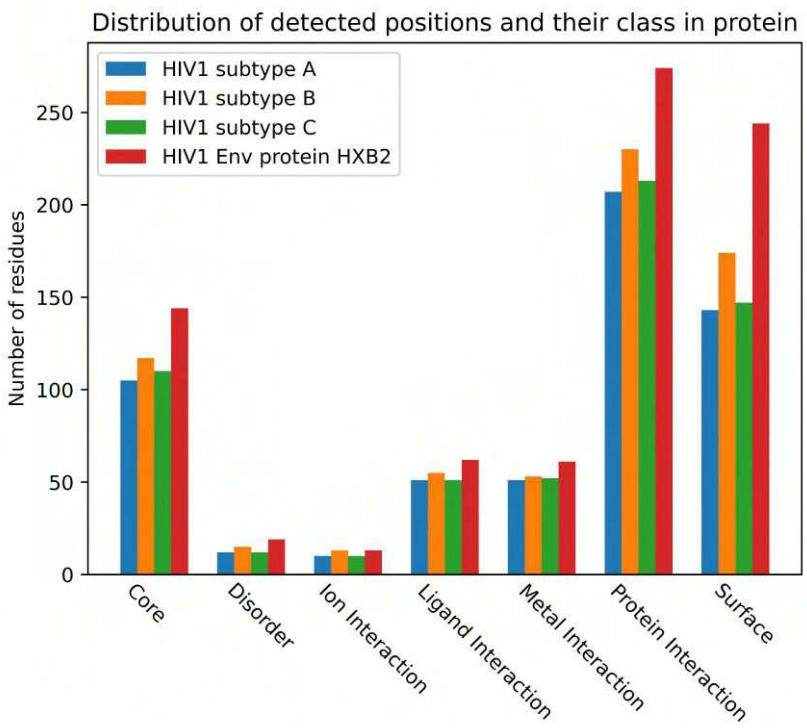


Figure 5.5: Bar plot with distribution of number of detected residues as potential epistatic pairs against the reference sequence HXB2. *Figure taken from [58]*

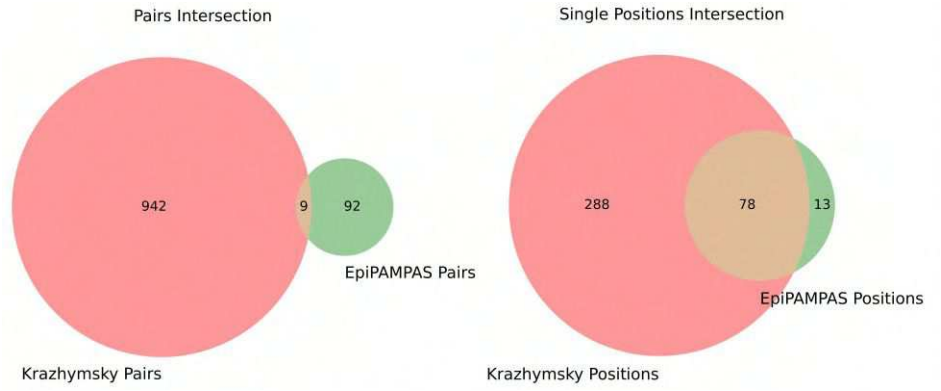


Figure 5.6: This plot is showing the intersection between the individual positions detected between our method and [140] method, and the intersection of the detected pairs. We see that the majority of our positions were also detected in their results. However, when it comes to pairs, the intersection is much smaller, indicating that the pairs we detected are different from their pairs. The results represented here are filtered for a p-value smaller than 0.05 except for N1, where we used 0.1 because there was only one pair detected with a p-value cutoff of 0.05. *Figure taken from [58]*

Protein	HIV-1 Sub. A	HIV-1 Sub. B	HIV-1 Sub. C	HXB2
# With Annotations	25	72	41	279
# Without Annotations	39	93	55	577
Percentage %	39.06	43.63	42.70	32.59
Fisher exact test	0.177	0.0044	0.031	NA

Table 5.5: Here we show the number of annotated positions in HIV-1 envelope protein taken from [109], we looked at how many of the positions we detected had annotations compared to the reference protein, and ran a Fisher exact test to see if the inflation in the number of annotated positions to not annotated positions is more significant in the positions we detected. *Table taken from [58]*

5.5 Conclusion and Discussion

This section reuses materials from [58] of which I am a first co-author

In this chapter, we presented `EpiPAMPAS`, a novel statistical method for detecting epistatic interactions between mutations. Our approach is based on the counting of same-direction mutations on a dendrogram after reconstructing the most-likely genotype of the ancestors in the dendrogram. Our method is different compared to other tree-based methods by not requiring a phylogenetic tree, which reduces the computational burden, especially for an exhaustive method that checks for all mutation-pair combinations. We evaluated our method through a comparative analysis of both simulated (Section 5.4.1) and real viral protein data (Section 5.4.2), demonstrating its efficacy in the former context. For the viral protein data, we compared our results to another tree-based method, and we found that `EpiPAMPAS` identified many positions that were also reported by [140], but reported fewer positions in total. However, the interactions between the positions were detected differently (Figure 5.6). Furthermore, we identified that some of the epistatic interacting positions `EpiPAMPAS` detected are known to be deleterious from the literature, but provide resistance against certain antiviral drugs. From the protein structure perspective, we did not identify a clear signal that the pairs detected are closer to each other in the 3D protein structure compared to all possible pairs of mutations (Figure 5.4 and Tables 5.3 and 5.4). One reason for this is that our statistical definition of epistasis does not reflect the actual fitness-based definition. A more plausible explanation is that the sequence data on viral proteins, despite our comprehensive efforts to collect it, is inadequate for detecting epistasis. Two arguments support this perspective. First, the simulation data demonstrates very good performance. Second, for certain datasets, we observe a maximum of one pair at $p < 0.05$. An alternative explanation could be that the simulation method does not accurately reflect the underlying mechanisms of epistasis compensation, leading to a positive bias in our simulation results. Ultimately, we believe that `EpiPAMPAS` does provide comparable results to other methods that are much more computationally intensive, and can provide valuable insights in pinpointing potential epistatically-interacting pairs of mutations before further analysis.

Summary

This dissertation has presented published and unpublished research related to algorithmic methods and software toolkits for various genomic analysis, especially, in the field of genome graphs and pangenomics. In Chapter 1, the general concepts in this field were introduced, specifically the algorithms and toolkits developed so far to solve problems such as sequence alignment and mapping, sequence assembly, genome graphs, and pangenomes. In Chapter 2, we presented the concept of a panproteome and `PanPA`, a software toolkit for building, indexing, and aligning panproteomes. We defined a panproteome as a collection of graphs, where each graph represents a multiple sequence alignment of a protein or a protein cluster. We argued and demonstrated that especially in the case of prokaryotes, building such panproteomes and moving to the amino acid space is indeed beneficial. We showed that the amino acid space improves the sequence alignment by comparing `PanPA` to other linear reference and graph sequence aligners in the DNA space, where we were able to align more sequences that would, otherwise, not be aligned by the other tools. We also developed a frameshift-aware alignment algorithm to improve the alignments of DNA sequences against panproteomes, enabling `PanPA` to align raw sequencing reads directly with better precision. Furthermore, the utility of `PanPA` in unveiling genetic mechanisms underlying phenotypic traits, including antimicrobial drug resistance, was demonstrated. Finally, we showed the ability of `PanPA` to infer the gene ordering of a genome from its sequences. In Chapter 3, we presented three software toolkits for working with genome graphs and sequence-to-graph alignments. First, we presented `GFASubgraph`, a toolkit and API for manipulating genome graphs effectively. It offers users various utilities to aid in building their own downstream analysis. We compared `GFASubgraph` efficacy to several other genome graph toolkits implemented in the `Python` programming language. Our tool demonstrated the best performance in terms of memory footprint and processing speed, even when dealing with large graphs. Next, we introduced `extgfa`, another software toolkit and API designed for working with genome graphs, and build upon `GFASubgraph`. Inspired by the design principles of some video games, it offers a method for partitioning large graphs and only loading the necessary parts of the graph into memory, with the remaining parts residing on the disk. The API operates seamlessly behind the scenes to load and unload parts of the graph between memory and disk, enabling users to investigate and manipulate large graphs on personal machines with limited memory. Finally, we presented `gaftools`, a collaborative software toolkit for working with sequence-to-graph alignments. `Gaftools` attempts to bridge the

gap between the linear reference and the pangenome reference software ecosystem; it provides utilities for reference graph ordering, indexing and viewing alignments based on their alignment location, realigning sequences using the Wavefront alignment algorithm, and other functionalities. A distinguishing feature of the toolkits presented in this chapter is the utilization of a uniform graph processing module, which underscores the significance of designing an efficient genome graph programming library. This library then can seamlessly integrate with other toolkits and pipelines, thereby enhancing the efficiency of graph-based research in the scientific community. Chapter 4 presents unpublished collaborative work in analyzing and assembling a cancer cell line and its matched normal. In this work, we employ several sequencing technologies to produce high-quality assemblies and structural variant sets. Moreover, using the matched normal, we produce a set of somatic mutations with high confidence. Subsequently, we investigate further the effect of subclones and the heterogeneity in cancer and their negative effect on producing higher-quality assemblies and structural variants. To facilitate this investigation, we have developed a graph toolkit called `graphdraw`, which assists in visualizing graph components, coloring important nodes, and enabling us to swiftly ascertain the location and impact of subclonal SVs on the assembly graph. Moreover, this graph investigation sheds light on the potential role of assembly graphs in disentangling cancer subclones and haplotypes, and their role in producing high quality cancer assemblies. Chapter 5 presents `EpiPAMPAS`, a novel statistical method designed to detect epistatic interactions between mutations in protein multiple sequence alignments. A pivotal aspect of the method is its ability to circumvent the construction of a phylogenetic tree, which is computationally intensive. Instead, it employs a hierarchical clustering dendrogram in lieu of the tree, and the Sankoff parsimony algorithm to reconstruct the most probable ancestral states. Subsequently, a mutation direction counting method is utilized to deduce potential compensating mutations. To assess the efficacy of our method, we tested on both simulated and real datasets. Furthermore, we compared our tool to another tree-based method and found that `EpiPAMPAS` identified several positions that were also reported by the other method. Additionally, we observed that some of the epistatically interacting positions detected by our method were previously documented in the literature as deleterious. Subsequent investigation of the pairs detected in terms of protein 3D structure, we did not find a clear signal that these pairs are closer to each other in 3D space. A reason behind this could be that our definition of epistasis does not reflect the actual fitness-based definition; or simply the amount of viral protein data we analyzed was inadequate and a further collection of data is needed. Finally, the methods presented in this dissertation seek to bridge the gap towards transitioning to the graph pangenome space. Furthermore, they underscore the significance of this transition and the advantages it offers for the genome research field. Additionally, they highlight the importance of well-designed and efficient genome graph and sequence-to-graph alignment data structures to address the various challenges that accompany this transition.

Bibliography

[1] 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, Sept. 2015.

[2] L. A. Aaltonen, F. Abascal, A. Abeshouse, H. Aburatani, D. J. Adams, et al. Pan-cancer analysis of whole genomes. *Nature*, 578(7793):82–93, Feb. 2020.

[3] M. U. Ahsan, Q. Liu, J. E. Perdomo, L. Fang, and K. Wang. A survey of algorithms for the detection of genomic structural variants from long-read sequencing data. *Nature Methods*, 20 (8):1143–1158, June 2023.

[4] T. Akutsu. A linear time pattern matching algorithm between a string and a tree. In *Combinatorial Pattern Matching*, volume 684 of *Lecture Notes in Computer Science*, pages 1–10, Berlin/Heidelberg, 1993. Springer-Verlag.

[5] S. F. Altschul and M. Pop. Chapter 20.1: Sequence alignment. In *Handbook of Discrete and Combinatorial Mathematics. 2nd edition*. CRC Press/Taylor and Francis, 2017.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, Oct. 1990.

[7] R. I. Amann, W. Ludwig, and K. H. Schleifer. Phylogenetic identification and in situ detection of individual microbial cells without cultivation. *Microbiological Reviews*, 59(1):143–169, 1995.

[8] A. Amir, M. Lewenstein, and N. Lewenstein. Pattern matching in hypertext. *Journal of Algorithms*, 35(1):82–99, Apr. 2000.

[9] F. Andreace, P. Lechat, Y. Dufresne, and R. Chikhi. Comparing methods for constructing and representing human pangenome graphs. *Genome Biology*, 24(1), Nov. 2023.

[10] F. Y. Aoki, G. Boivin, and N. Roberts. Influenza virus susceptibility and resistance to oseltamivir. *Antiviral Therapy*, 12(4 Pt B):603–616, 2007.

[11] H. Ashkenazy, I. Sela, E. Levy Karin, G. Landan, and T. Pupko. Multiple sequence alignment averaging improves phylogeny reconstruction. *Systematic Biology*, 68(1):117–130, June 2018.

[12] M. Aydın Akbudak and V. Srivastava. Effect of gene order in dna constructs on gene expression upon integration into plant genome. *3 Biotech*, 7(2), May 2017.

[13] R. B. R. Azevedo, R. Lohaus, S. Srinivasan, K. K. Dang, and C. L. Burch. Sexual reproduction selects for robustness and negative epistasis in artificial gene networks. *Nature*, 440(7080): 87–90, Mar. 2006.

[14] S. Bagel, V. Hüllen, B. Wiedemann, and P. Heisig. Impact of gyrA and parC mutations on quinolone resistance, doubling time, and supercoiling degree of escherichia coli. *Antimicrobial Agents and Chemotherapy*, 43(4):868–875, Apr. 1999.

[15] T. M. Baker, S. Waise, M. Tarabichi, and P. Van Loo. Aneuploidy and complex genomic rearrangements in cancer evolution. *Nature Cancer*, 5(2):228–239, Jan. 2024.

[16] Bateson, Mendel, and Leighton. *Mendel's principles of heredity, by W. Bateson*. Cambridge [Eng.]University Press,, 1902.

[17] S. A. Benner, M. A. Cohen, and G. H. Gonnet. Amino acid substitution during functionally constrained divergent evolution of protein sequences. *Protein Engineering, Design and Selection*, 7(11):1323–1332, 1994.

[18] H. Berman, K. Henrick, and H. Nakamura. Announcing the worldwide protein data bank. *Nature Structurla Biology*, 10(12):980, Dec. 2003.

[19] N. Biggs, Lloyd, and R. J. Wilson. *Graph theory, 1736-1936*. Clarendon Press, Feb. 1986. ISBN 0198539169.

[20] O. R. P. Bininda-Emonds. transalign: using amino acids to facilitate the multiple alignment of protein-coding DNA sequences. *BMC Bioinformatics*, 6:156, June 2005.

[21] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, Oct. 2008.

[22] J. D. Bloom, S. T. Labthavikul, C. R. Otey, and F. H. Arnold. Protein stability promotes evolvability. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 103(15):5869–5874, Apr. 2006.

[23] J. D. Bloom, L. I. Gong, and D. Baltimore. Permissive secondary mutations enable the evolution of influenza oseltamivir resistance. *Science*, 328(5983):1272–1275, June 2010.

[24] M.-L. Bondeson, N. Dahl, H. Malmgren, W. J. Kleijer, T. Tönnesen, et al. Inversion of the ids gene resulting from recombination with ids-related sequences in a common cause of the hunter syndrome. *Human Molecular Genetics*, 4(4):615–621, 1995.

[25] J. K. Bonfield, J. Marshall, P. Danecek, H. Li, V. Ohan, A. Whitwham, T. Keane, and R. M. Davies. Htslib: C library for reading/writing high-throughput sequencing data. *GigaScience*, 10(2), Jan. 2021.

[26] S. Bonhoeffer, C. Chappey, N. T. Parkin, J. M. Whitcomb, and C. J. Petropoulos. Evidence for positive epistasis in hiv-1. *Science*, 306(5701):1547–1550, Nov. 2004.

[27] S. W. Brady, J. A. McQuerry, Y. Qiao, S. R. Piccolo, G. Shrestha, et al. Combating subclonal evolution of resistant cancer phenotypes. *Nature Communications*, 8(1), Nov. 2017.

[28] D. Y. C. Brandt, V. R. C. Aguiar, B. D. Bitarello, K. Nunes, J. Goudet, et al. Mapping bias overestimates reference allele frequencies at the hla genes in the 1000 genomes project phase i data. *G3 Genes|Genomes|Genetics*, 5(5):931–941, May 2015.

[29] T. A. Brown. Genome anatomies. In *Genomes. 2nd edition*. Wiley-Liss, 2002.

[30] B. Calippe, X. Guillonneau, and F. Sennlaub. Complement factor h and related proteins in age-related macular degeneration. *Comptes Rendus Biologies*, 337(3):178–184, Mar. 2014.

[31] F. Carrat and A. Flahault. Influenza vaccine: the challenge of antigenic drift. *Vaccine*, 25 (39-40):6852–6862, Sept. 2007.

[32] K. Carruthers-Smith. Sliding window minimum implementations. `https://people.cs.uct.ac.za/~ksmith/articles/sliding_window_minimum.html`, 2011. [Accessed 20-03-2022].

[33] R. A. Cartwright. Logarithmic gap costs decrease alignment accuracy. *BMC Bioinformatics*, 7 (1), Dec. 2006.

[34] J. V. Chamary, J. L. Parmley, and L. D. Hurst. Hearing silence: non-neutral evolution at synonymous sites in mammals. *Nature Reviews Genetics*, 7(2):98–108, Feb. 2006.

[35] N.-C. Chen, B. Solomon, T. Mun, S. Iyer, and B. Langmead. Reference flow: reducing reference bias using multiple population genomes. *Genome Biology*, 22(1), January 2021.

[36] N.-C. Chen, B. Solomon, T. Mun, S. Iyer, and B. Langmead. Reference flow: reducing reference bias using multiple population genomes. *Genome Biology*, 22(1):1–17, 2021.

[37] H. Cheng, G. T. Concepcion, X. Feng, H. Zhang, and H. Li. Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nature Methods*, 18(2):170–175, Feb. 2021.

[38] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, et al. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, page 139–149, USA, 1995. Society for Industrial and Applied Mathematics. ISBN 0898713498.

[39] R. Chikhi, Y. Dufresne, and P. Medvedev. Constructing and personalizing population pangenome graphs. *Nature Methods*, 21(11):1980–1981, Oct. 2024.

[40] D. M. Church, V. A. Schneider, T. Graves, K. Auger, F. Cunningham, et al. Modernizing reference genome assemblies. *PLoS Biology*, 9(7):e1001091, July 2011.

[41] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, 70(6), Dec. 2004.

[42] J. C. Clemente, K. Ikeo, G. Valiente, and T. Gojobori. Optimized ancestral state reconstruction using sankoff parsimony. *BMC Bioinformatics*, 10:51, Feb. 2009.

[43] M. Cobb and N. Comfort. What Rosalind Franklin truly contributed to the discovery of DNA's structure. `https://www.nature.com/articles/d41586-023-01313-5`, 2023. [Accessed 24-01-2025].

[44] P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The sanger FASTQ file format for sequences with quality scores, and the solexa/illumina fa0stq variants. *Nucleic Acids Research*, 38(6):1767–1771, Dec. 2009.

[45] F. M. Cohan and A. F. Koeppel. The origins of ecological diversity in prokaryotes. *Current Biology*, 18(21):R1024–R1034, Nov. 2008.

[46] P. W. Collingridge and S. Kelly. Mergealign: improving multiple sequence alignment performance by dynamic reconstruction of consensus multiple sequence alignments. *BMC Bioinformatics*, 13(1), May 2012.

[47] R. M. Colquhoun, M. B. Hall, L. Lima, L. W. Roberts, K. M. Malone, M. Hunt, et al. Pandora: nucleotide-resolution bacterial pan-genomics with reference graphs. *Genome Biology*, 22(1): 267, Sept. 2021.

[48] P. E. C. Compeau, P. A. Pevzner, and G. Tesler. How to apply de bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991, Nov. 2011.

[49] Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, Jan. 2018.

[50] Cornell Capra: Computer architecture and programming abstractions at Cornell University. mygfa. `https://github.com/cucapra/pollen/commit/7e9f6207ea9f4321c3664d34a9eca81c9c9c07ea`, 2024.

[51] M. R. Cosenza, B. Rodriguez-Martin, and J. O. Korbel. Structural variation in cancer: Role, prevalence, and mechanisms. *Annual Review Genomics Human Genetics*, 23:123–152, Aug. 2022.

[52] F. Dabbaghie. extgfa: A low-memory on-disk representation of genome graphs. *bioRxiv*, page 2024.11.29.626045, Dec. 2024.

[53] F. Dabbaghie. Gfasubgraph. `https://github.com/fawaz-dabbaghieh/gfa_subgraphs/commit/255d48e4ec01f63e126a2b637ee42ed2f7da684c`, 2024.

[54] F. Dabbaghie. graphdraw. `https://github.com/fawaz-dabbaghieh/graphdrawing_toolkit/commit/e3ba6d0fa68ec40c13aa6590a95f1ddab746be67`, 2024.

[55] F. Dabbaghie. MSA to GFA. `https://github.com/fawaz-dabbaghieh/msa_to_gfa/commit/cffa15d50951e7b3fccf80fdf38849c8840b5d55`, 2024.

[56] F. Dabbaghie, J. Ebler, and T. Marschall. BubbleGun: enumerating bubbles and superbubbles in genome graphs. *Bioinformatics*, 38(17):4217–4219, Sept. 2022.

[57] F. Dabbaghie, S. K. Srikakulam, T. Marschall, and O. V. Kalinina. PanPA: generation and alignment of panproteome graphs. *Bioinformatics Advances*, 3(1), Jan. 2023.

[58] F. Dabbaghie, K. Thedinga, G. A. Bazykin, T. Marschall, and O. Kalinina. EpiPAMPAS: Rapid detection of intra-protein epistasis via parsimonious ancestral state reconstruction and counting mutations. *bioRxiv*, page 2024.12.13.628430, Dec. 2024.

[59] P. Dadgostar. *Antimicrobial resistance: implications and costs*. Infection and drug resistance. Taylor & Francis, 2019.

[60] J. Daily. Parasail: SIMD c library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics*, 17(1), Feb. 2016.

[61] A. Danek, S. Deorowicz, and S. Grabowski. Correction: Indexes of large genome collections on a pc. *PLOS ONE*, 10(5):e0128172, May 2015.

[62] R. Das and M. Soylu. A key review on graph data science: The power of graphs in scientific studies. *Chemometrics and Intelligent Laboratory Systems*, 240:104896, Sept. 2023.

[63] J. J. Davis, A. R. Wattam, R. K. Aziz, T. Brettin, R. Butler, et al. The PATRIC bioinformatics resource center: expanding data and analysis capabilities. *Nucleic Acids Research*, 48(D1): D606–D612, Jan. 2020.

[64] S. Dubois. gfagraphs. `https://github.com/dubssieg/gfagraphs/commit/7947b18115fdeae34aafa91fefa9a057b58b754b`, 2024.

[65] D. Duchen, S. Clipman, C. Vergara, C. L. Thio, D. L. Thomas, et al. A hepatitis b virus (hbv) sequence variation graph improves sequence alignment and sample-specific consensus sequence construction for genetic analysis of hbv. *bioRxiv*, Jan. 2023.

[66] P. V. Dunlap. Microbial diversity. In S. A. Levin, editor, *Encyclopedia of Biodiversity (Second Edition)*, pages 280–291. Academic Press, Waltham, second edition edition, 2001. ISBN 978-0-12-384720-1.

[67] R. Durbin. Efficient haplotype matching and storage using the positional burrows–wheeler transform (pbwt). *Bioinformatics*, 30(9):1266–1272, Jan. 2014.

[68] A. Dutta, S. Paul, and C. Dutta. GC-rich intra-operonic spacers in prokaryotes: Possible relation to gene order conservation. *FEBS Letters*, 584(22):4633–4638, Oct. 2010.

[69] M. Dávila López, J. J. Martínez Guerra, and T. Samuelsson. Analysis of gene order conservation in eukaryotes identifies transcriptionally and functionally linked genes. *PLoS ONE*, 5(5): e10654, May 2010.

[70] P. Ebert, P. A. Audano, Q. Zhu, B. Rodriguez-Martin, D. Porubsky, et al. Haplotype-resolved diverse human genomes and integrated analysis of structural variation. *Science*, 372(6537), Apr. 2021.

[71] J. Ebler. callset-comparison. `https://github.com/eblerjana/callset-comparison/commit/ad8c0f4bcbe4a0d03d6a97fa4573eac77965b1eb`, 2024.

[72] J. Ebler, P. Ebert, W. E. Clarke, T. Rausch, P. A. Audano, et al. Pangenome-based genome inference allows efficient and accurate genotyping across a wide spectrum of variant classes. *Nature Genetics*, 54(4):518–525, Apr. 2022.

[73] R. C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, Mar. 2004.

[74] J. M. Eizenga, A. M. Novak, J. A. Sibbesen, S. Heumos, A. Ghaffaari, G. Hickey, et al. Pangenome graphs. *Annual Review of Genomics and Human Genetics*, May 2020.

[75] B. Ekim, B. Berger, and R. Chikhi. Minimizer-space de bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell Syst.*, 12(10):958–968.e6, Oct. 2021.

[76] J. Espejo Valle-Inclan, N. J. M. Besselink, E. de Bruijn, D. L. Cameron, J. Ebler, et al. A multi-platform reference for somatic structural variation detection. *Cell Genom*, 2(6):100139, June 2022.

[77] E. Falconer, M. Hills, U. Naumann, S. S. S. Poon, E. A. Chavez, et al. Dna template strand sequencing of single-cells maps genomic rearrangements at high resolution. *Nature Methods*, 9(11):1107–1112, Oct. 2012.

[78] M. Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, Jan. 2007.

[79] D.-F. Feng and R. F. Doolittle. Progressive sequence alignment as a prerequisitetto correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, Aug. 1987.

[80] J. Figueroa, D. Castro, F. Lagos, C. Cartes, A. Isla, et al. Analysis of single nucleotide polymorphisms (snps) associated with antibiotic resistance genes in chilean piscirickettsia salmonis strains. *Journal of Fish Diseases*, 42(12):1645–1655, Oct. 2019.

[81] R. D. Finn, J. Clements, and S. R. Eddy. HMMER web server: interactive sequence similarity searching. *Nucleic Acids Research*, 39(Web Server issue):W29–37, July 2011.

[82] K. Fischer, P. Hoffmann, S. Voelkl, N. Meidenbauer, J. Ammer, et al. Inhibitory effect of tumor cell–derived lactic acid on human t cells. *Blood*, 109(9):3812–3819, Jan. 2007.

[83] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3–5):75–174, Feb. 2010.

[84] W. Fujibuchi. Automatic detection of conserved gene clusters in multiple genomes by graph comparison and p-quasi grouping. *Nucleic Acids Research*, 28(20):4029–4036, Oct. 2000.

[85] K. Fukami-Kobayashi, D. R. Schreiber, and S. A. Benner. Detecting compensatory covariation signals in protein evolution using reconstructed ancestral sequences. *Journal of Molecular Biology*, 319(3):729–743, June 2002.

[86] Gamin Industry. How can you optimize memory usage for large open-world games?, 2024. URL `https://www.linkedin.com/advice/0/how-can-you-optimize-memory-usage-large-open-world-vh7ae`. [Accessed 22-11-2024].

[87] E. Garrison and G. Marth. Haplotype-based variant detection from short-read sequencing, 2012.

[88] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, et al. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36(9):875–879, October 2018.

[89] E. Garrison, A. Guarracino, S. Heumos, F. Villani, Z. Bao, et al. Building pangenome graphs. *Nature Methods*, 21(11):2008–2012, Oct. 2024.

[90] K. Gerth, S. Pradella, O. Perlova, S. Beyer, and R. Müller. Myxobacteria: proficient producers of novel natural products with various biological activities–past and future biotechnological aspects with the focus on the genus sorangium. *Journal of Biotechnology*, 106(2-3):233–253, Dec. 2003.

[91] GFA Format Specification Working Group. Gfa-spec, 2024. URL `https://gfa-spec.github.io/GFA-spec/GFA1.html`. [Accessed 18-11-2024].

[92] M. Ghareghani, D. Porubskỳ, A. D. Sanders, S. Meiers, E. E. Eichler, et al. Strand-seq enables reliable separation of long reads by chromosome via expectation maximization. *Bioinformatics*, 34(13):i115–i123, June 2018.

[93] A. J. Gibbs and G. A. Mcintyre. The diagram, a method for comparing sequences: Its use with amino acid and nucleotide sequences. *European Journal of Biochemistry*, 16(1):1–11, Sept. 1970.

[94] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, June 2004.

[95] G. B. Gloor, L. C. Martin, L. M. Wahl, and S. D. Dunn. Mutual information in protein multiple sequence alignments reveals two classes of coevolving positions. *Biochemistry*, 44(19): 7156–7165, May 2005.

[96] G. Gonnella and S. Kurtz. Gfapy: a flexible and extensible software library for handling sequence graphs in python. *Bioinformatics*, 33(19):3094–3095, June 2017.

[97] G. Gonnella, N. Niehus, and S. Kurtz. Gfaviz: flexible and interactive visualization of GFA sequence graphs. *Bioinformatics*, 35(16):2853–2855, Dec. 2018.

[98] S. Goodwin, J. D. McPherson, and W. R. McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17(6):333–351, May 2016.

[99] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[100] O. Gotoh. Optimal alignment between groups of sequences and its application to multiple sequence alignment. *Bioinformatics*, 9(3):361–370, 1993.

[101] A. Gress, V. Ramensky, J. Büch, A. Keller, and O. V. Kalinina. StructMAn: annotation of single-nucleotide polymorphisms in the structural context. *Nucleic Acids Research*, 44(W1): W463–8, July 2016.

[102] C. Gros, A. D. Sanders, J. O. Korbel, T. Marschall, and P. Ebert. ASHLEYS: automated quality control for single-cell strand-seq data. *Bioinformatics*, 37(19):3356–3357, Apr. 2021.

[103] C. Groza, C. Schwendinger-Schreck, W. A. Cheung, E. G. Farrow, I. Thiffault, et al. Pangenome graphs improve the analysis of structural variants in rare genetic diseases. *Nature Communications*, 15(1), Jan. 2024.

[104] Y.-F. Guan, G.-R. Li, R.-J. Wang, Y.-T. Yi, L. Yang, et al. Application of next-generation sequencing in clinical oncology to advance personalized treatment of cancer. *Chinese Journal of Cancer*, 31(10):463–470, Oct. 2012.

[105] S. Guindon and O. Gascuel. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic Biology*, 52(5):696–704, Oct. 2003.

[106] T. Günther and C. Nettelblad. The presence and impact of reference bias on population genomic studies of prehistoric human populations. *PLOS Genetics*, 15(7):e1008302, July 2019.

[107] M. H. Guo, L. C. Francioli, S. L. Stenton, J. K. Goodrich, N. A. Watts, et al. Inferring compound heterozygosity from large-scale exome sequencing data. *Nature Genetics*, 56(1):152–161, Dec. 2023.

[108] A. Hagberg, P. J. Swart, and D. A. Schult. Exploring network structure, dynamics, and function using NetworkX. Technical Report LA-UR-08-05495; LA-UR-08-5495, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), Jan. 2008.

[109] A. Hake and N. Pfeifer. Prediction of HIV-1 sensitivity to broadly neutralizing antibodies shows a trend towards resistance over time. *PLoS Computation Biology*, 13(10):e1005789, Oct. 2017.

[110] D. Hanahan and R. A. Weinberg. Hallmarks of cancer: The next generation. *Cell*, 144(5): 646–674, feb 2011.

[111] T. F. Hansen. Why epistasis is important for selection and adaptation: Perspective. *Evolution*, 67(12):3501–3511, Aug. 2013.

[112] M. Hauser, M. Steinegger, and J. Söding. MMseqs software suite for fast and deep clustering and searching of large protein sequence sets. *Bioinformatics*, 32(9):1323–1330, May 2016.

[113] Z. He, A. Wilson, F. Rich, D. Kenwright, A. Stevens, et al. Chromosomal instability and its effect on cell lines. *Cancer Reports*, 6(6), Apr. 2023.

[114] D. Heller and M. Vingron. SVIM-asm: structural variant detection from haploid and diploid genome assemblies. *Bioinformatics*, 36(22–23):5519–5521, Dec. 2020.

[115] M. Henglin, M. Ghareghani, W. T. Harvey, D. Porubsky, S. Koren, et al. Graphasing: phasing diploid genome assembly graphs with single-cell strand sequencing. *Genome Biology*, 25(1), Oct. 2024.

[116] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 89(22): 10915–10919, Nov. 1992.

[117] J.-H. Her and R. Ramakrishna. An external-memory depth-first search algorithm for general grid graphs. *Theoretical Computer Science*, 374(1):170–180, 2007.

[118] D. Hernandez, P. François, L. Farinelli, M. Østerås, and J. Schrenzel. De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, 18(5):802–809, Mar. 2008.

[119] R. T. Hersh, R. V. Eck, and M. O. Dayhoff. Atlas of protein sequence and structure, 1966. *Systematic Zoology*, 16(3):262, Sept. 1967.

[120] G. Hickey, J. Monlong, J. Ebler, A. M. Novak, J. M. Eizenga, et al. Pangenome graph construction from genome alignments with minigraph-cactus. *Nature Biotechnology*, 42(4):663–673, May 2023.

[121] G. Holley and P. Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de bruijn graphs. *Genome Biology*, 21(1), Sept. 2020.

[122] T. Hon, K. Mars, G. Young, Y.-C. Tsai, J. W. Karalius, et al. Highly accurate long-read hifi sequencing data for five complex genomes. *Scientific Data*, 7(1), Nov. 2020.

[123] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, June 1973.

[124] T. A. Hopf, J. B. Ingraham, F. J. Poelwijk, C. P. I. Schärfe, M. Springer, C. Sander, and D. S. Marks. Mutation effects predicted from sequence co-variation. *Nature Biotechnology*, 35(2): 128–135, Jan. 2017.

[125] A. Hosseininasab and W.-J. van Hoeve. Exact multiple sequence alignment by synchronized decision diagrams. *INFORMS Journal on Computing*, Sept. 2020.

[126] Y. Hu, L. Fang, C. Nicholson, and K. Wang. Implications of error-prone long-read whole-genome shotgun sequencing on characterizing reference microbiomes. *iScience*, 23(6):101223, June 2020.

[127] P. Hugenholtz. Exploring prokaryotic diversity in the genomic era. *Genome Biology*, 3(2): reviews0003.1, Jan. 2002.

[128] R. M. Idury and M. S. Waterman. A new algorithm for dna sequence assembly. *Journal of Computational Biology*, 2(2):291–306, Jan. 1995.

[129] Illumina. Sequencing Technology | Sequencing by synthesis. `https://emea.illumina.com/science/technology/next-generation-sequencing/sequencing-technology.html`, 2020. [Accessed 25-01-2025].

[130] P. Ivanov, B. Bichsel, H. Mustafa, A. Kahles, G. Rätsch, et al. Astarix: Fast and optimal sequence-to-graph alignment. In R. Schwartz, editor, *Research in Computational Molecular Biology*, pages 104–119, Cham, 2020. Springer International Publishing.

[131] M. Jaillard, L. Lima, M. Tournoud, P. Mahé, A. van Belkum, V. Lacroix, and L. Jacob. A fast and agnostic method for bacterial genome-wide association studies: Bridging the gap between k-mers and genetic events. *PLoS Genetics*, 14(11):e1007758, Nov. 2018.

[132] C. Jain, A. Rhie, H. Zhang, C. Chu, B. P. Walenz, et al. Weighted minimizer sampling improves long read mapping. *Bioinformatics*, 36(Suppl_1):i111–i118, July 2020.

[133] D. M. Jordan, S. G. Frangakis, C. Golzio, C. A. Cassa, J. Kurtzberg, Task Force for Neonatal Genomics, et al. Identification of cis-suppression of human disease mutations by comparative genomics. *Nature*, 524(7564):225–229, Aug. 2015.

[134] I. K. Jordan, I. B. Rogozin, Y. I. Wolf, and E. V. Koonin. Essential genes are more evolutionarily conserved than are nonessential genes in bacteria. *Genome Research*, 12(6):962–968, May 2002.

[135] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, July 2021.

[136] M. Karasikov, H. Mustafa, G. Rätsch, and A. Kahles. Lossless indexing with counting de bruijn graphs. *Genome Research*, 32(9):1754–1764, May 2022.

[137] K. Katoh and D. M. Standley. MAFFT multiple sequence alignment software version 7: improvements in performance and usability. *Molecular Biology and Evolution*, 30(4):772–780, Apr. 2013.

[138] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, feb 1970.

[139] B. T. Korber, R. M. Farber, D. H. Wolpert, and A. S. Lapedes. Covariation of mutations in the V3 loop of human immunodeficiency virus type 1 envelope protein: an information theoretic analysis. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 90(15):7176–7180, Aug. 1993.

[140] S. Kryazhimskiy, J. Dushoff, G. A. Bazykin, and J. B. Plotkin. Prevalence of epistasis in the evolution of influenza a surface proteins. *PLoS Genetics*, 7(2):e1001301, Feb. 2011.

[141] G. V. Kryukov, L. A. Pennacchio, and S. R. Sunyaev. Most rare missense alleles are deleterious in humans: implications for complex disease and association studies. *The American Journal of Human Genetics*, 80(4):727–739, Apr. 2007.

[142] C. Kuiken, B. Korber, and R. W. Shafer. HIV sequence databases. *AIDS Reviews*, 5(1):52–61, Jan. 2003.

[143] A. Lancichinetti and S. Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E*, 80(5), Nov. 2009.

[144] C. Lee, C. Grasso, and M. F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, Mar. 2002.

[145] R. Leinonen, H. Sugawara, M. Shumway, and International Nucleotide Sequence Database Collaboration. The sequence read archive. *Nucleic Acids Research*, 39(Database issue):D19–21, Jan. 2011.

[146] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, feb 1966.

[147] C. Li, L. Chen, G. Pan, W. Zhang, and S. C. Li. Deciphering complex breakage-fusion-bridge genome rearrangements with ambigram. *Nature Communications*, 14(1), Sept. 2023.

[148] H. Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18): 3094–3100, Sept. 2018.

[149] H. Li. The graph alignment format (gaf), 2019. URL `https://github.com/lh3/gfatools/blob/master/doc/rGFA.md#the-graph-alignment-format-gaf`. [Accessed 18-11-2024].

[150] H. Li. The reference gfa (rgfa) format, 2019. URL `https://github.com/lh3/gfatools/blob/master/doc/rGFA.md#the-reference-gfa-rgfa-format`. [Accessed 18-11-2024].

[151] H. Li. gaftools. `https://github.com/lh3/gfatools`, 2022. [Acessed 16-02-2025].

[152] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, Mar. 2010.

[153] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, et al. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[154] H. Li, X. Feng, and C. Chu. The design and construction of reference pangenome graphs with minigraph. *Genome Biology*, 21(1):265, oct 2020.

[155] H. Li, S. Wang, S. Chai, Z. Yang, Q. Zhang, et al. Graph-based pan-genome reveals structural and sequence variations related to agronomic traits and domestication in cucumber. *Nature Communications*, 13(1):682, Feb. 2022.

[156] H. Li, M. Marin, and M. R. Farhat. Exploring gene content with pangene graphs. *Bioinformatics*, 40(7), July 2024.

[157] W.-W. Liao, M. Asri, J. Ebler, et al. A draft human pangenome reference. *Nature*, 617(7960): 312–324, May 2023.

[158] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227 (4693):1435—1441, Mar. 1985.

[159] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A tool for multiple sequence alignment. *Proceedings of the National Academy of Sciences*, 86(12):4412–4415, June 1989.

[160] Y. H. Liu, C. Luo, S. G. Golding, J. B. Ioffe, and X. M. Zhou. Tradeoffs in alignment and assembly-based methods for structural variant detection with long-read sequencing data. *Nature Communications*, 15(1), Mar. 2024.

[161] M. Lunzer, G. B. Golding, and A. M. Dean. Pervasive cryptic epistasis in molecular evolution. *PLoS Genetics*, 6(10):e1001162, Oct. 2010.

[162] D. M. Lyons and A. S. Lauring. Mutation and epistasis in influenza virus evolution. *Viruses*, 10(8):407, Aug. 2018.

[163] U. Manber and S. Wu. Approximate string matching with arbitrary costs for text and hypertext. In *Advances in Structural and Syntactic Pattern Recognition*, volume 5 of *Series in Machine Perception and Artificial Intelligence*, pages 22–33. World Scientific Publishing Co Pte Ltd, Feb. 1993.

[164] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics*, 37(4):456–463, Sept. 2020.

[165] M. Martin, M. Patterson, S. Garg, S. O. Fischer, N. Pisanti, et al. WhatsHap: fast and accurate read-based phasing. *bioRxiv*, page 085050, 14 Nov. 2016.

[166] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, et al. The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data. *Genome Research*, 20(9):1297–1303, July 2010.

[167] P. Medvedev and M. Pop. What do eulerian and hamiltonian cycles have to do with genome assembly? *PLOS Computational Biology*, 17(5):e1008928, May 2021.

[168] K. Mehlhorn and U. Meyer. *External-Memory Breadth-First Search with Sublinear I/O*, page 723–735. Springer Berlin Heidelberg, 2002. ISBN 9783540457497.

[169] Microsoft. Minecraft. `https://minecraft.net`, 2024. [Accessed 22-11-2024].

[170] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, June 2010.

[171] Minecraft Wiki. Minecraft fandom chunk. `https://minecraft.fandom.com/wiki/Chunk`, 2024. [Accessed 20-09-2024].

[172] J. Mistry, S. Chuguransky, L. Williams, M. Qureshi, G. A. Salazar, et al. Pfam: The protein families database in 2021. *Nucleic Acids Research*, 49(D1):D412–D419, Jan. 2021.

[173] K. I. Mohr. Diversity of Myxobacteria-We only see the tip of the iceberg. *Microorganisms*, 6 (3), Aug. 2018.

[174] F. Mölder, K. P. Jablonski, B. Letcher, M. B. Hall, C. H. Tomkins-Tinch, et al. Sustainable data analysis with snakemake. *F1000 Research*, 10(33):33, Jan. 2021.

[175] A. Moscona. Global transmission of oseltamivir-resistant influenza. *The New England Journal of Medicine*, 360(10):953–956, Mar. 2009.

[176] F. Murtagh and P. Legendre. Ward's hierarchical agglomerative clustering method: Which algorithms implement ward's criterion? *Journal of Classification*, 31(3):274–295, Oct. 2014.

[177] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, Mar. 2000.

[178] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, May 1999.

[179] E. W. Myers Jr. A history of dna sequence assembly. *it - Information Technology*, 58(3): 126–132, June 2016.

[180] National LIbrary of Medicine. Query input and database selection, 2024. URL `https://blast.ncbi.nlm.nih.gov/doc/blast-topics/`. [Accessed 18-11-2024].

[181] G. Navarro. Improved approximate pattern matching on hypertext. *Theoretical Computer Science*, 237(1):455–463, Apr. 2000.

[182] M. Nawaz, K. Sung, O. Kweon, S. Khan, S. Nawaz, et al. Characterisation of novel mutations involved in quinolone resistance in escherichia coli isolated from imported shrimp. *International Journal of Antimicrobial Agent*, 45(5):471–476, May 2015.

[183] S. B. Needleman and C. 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.

[184] A. D. Neverov, S. Kryazhimskiy, J. B. Plotkin, and G. A. Bazykin. Coordinated evolution of influenza a surface proteins. *PLoS Genetics*, 11(8):e1005404, Aug. 2015.

[185] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2), Feb. 2004.

[186] C. Niel, C. Sinoquet, C. Dina, and G. Rocheleau. A survey about methods dedicated to epistasis detection. *Frontiers in Genetics*, 6, Sept. 2015.

[187] C. Notredame, D. G. Higgins, and J. Heringa. T-coffee: a novel method for fast and accurate multiple sequence alignment 1 1edited by j. thornton. *Journal of Molecular Biology*, 302(1): 205–217, Sept. 2000.

[188] S. Nurk, S. Koren, A. Rhie, M. Rautiainen, A. V. Bzikadze, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, Apr. 2022.

[189] NYU Center For Genomics and Systems Biology in New York and Abu Dhabi. Fastq format, 2024. URL `https://learn.gencore.bio.nyu.edu/ngs-file-formats/fastq-format/`. [Accessed 18-11-2024].

[190] N. A. O'Leary, M. W. Wright, J. R. Brister, S. Ciufo, D. Haddad, et al. Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation. *Nucleic Acids Research*, 44(D1):D733–45, Jan. 2016.

[191] K. O'Neill, M. Hills, M. Gottlieb, M. Borkowski, A. Karsan, et al. Assembling draft genomes using contibait. *Bioinformatics*, 33(17):2737–2739, May 2017.

[192] PacBio. Sequencing 101: from DNA to discovery — the steps of SMRT sequencing. `https://www.pacb.com/blog/steps-of-smrt-sequencing/`, 2020. [Accessed 26-01-2025].

[193] PacificBioscience. pbsv. `https://github.com/PacificBiosciences/pbsv`, 2018. [Accessed 27-02-2024].

[194] PacificBioscience. pbsv. `https://github.com/PacificBiosciences/svpack`, 2020. [Accessed 05-05-2024].

[195] PacificBiosciences. How does CCS work. `https://ccs.how/how-does-ccs-work.html`, 2020. [Accessed 02-02-2025].

[196] S. Pani, F. Dabbaghie, T. Marschall, and A. Soylev. gaftools: a toolkit for analyzing and manipulating pangenome alignments. *bioRxiv*, Dec. 2024.

[197] K. Park and D. K. Kim. String matching in hypertext. In Z. Galil and E. Ukkonen, editors, *Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95, Espoo, Finland, July 5-7, 1995, Proceedings*, volume 937 of *Lecture Notes in Computer Science*, pages 318–329. Springer, Aug. 1995.

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

[199] H. Peltola, H. Söderlund, and E. Ukkonen. Seqaid: a dna sequence assembling program based on a mathematical model. *Nucleic Acids Research*, 12(1Part1):307–321, 1984.

[200] N. T. Perna, G. Plunkett, 3rd, V. Burland, B. Mau, J. D. Glasner, et al. Genome sequence of enterohaemorrhagic escherichia coli O157:H7. *Nature*, 409(6819):529–533, Jan. 2001.

[201] P. A. Pevzner. l-tuple dna sequencing: Computer analysis. *Journal of Biomolecular Structure and Dynamics*, 7(1):63–73, Aug. 1989.

[202] L. M. Phan, S.-C. J. Yeung, and M.-H. Lee. Cancer metabolic reprogramming: importance, main features, and potentials for precise targeted anti-cancer therapies. *Cancer Biol. Med.*, 11 (1):1–19, Mar. 2014.

[203] P. C. Phillips. Epistasis — the essential role of gene interactions in the structure and evolution of genetic systems. *Nature Reviews Genetics*, 9(11):855–867, Nov. 2008.

[204] R. Pinard, A. de Winter, G. J. Sarkis, M. B. Gerstein, K. R. Tartaro, et al. Assessment of whole genome amplification-induced bias through high-throughput, massively parallel whole genome sequencing. *BMC Genomics*, 7(1), Aug. 2006.

[205] D. Pinkel and D. G. Albertson. Comparative genomic hybridization. *Annual Review of Genomics and Human Genetics*, 6(1):331–354, Sept. 2005.

[206] E. D. Pleasance, R. K. Cheetham, P. J. Stephens, D. J. McBride, S. J. Humphray, et al. A comprehensive catalogue of somatic mutations from a human cancer genome. *Nature*, 463 (7278):191–196, Dec. 2009.

[207] Polydin Art Studio. The essentials of video game optimization, 2023. URL `https://polydin.com/video-game-optimization/`. [Accessed 22-11-2024].

[208] A. Poon and L. Chao. The rate of compensatory mutation in the DNA bacteriophage phiX174. *Genetics*, 170(3):989–999, July 2005.

[209] M. Pop. Genome assembly reborn: recent computational challenges. *Briefings in Bioinformatics*, 10(4):354–366, May 2009.

[210] R. Poplin, P.-C. Chang, D. Alexander, S. Schwartz, T. Colthurst, et al. A universal snp and small-indel variant caller using deep neural networks. *Nature Biotechnology*, 36(10):983–987, Sept. 2018.

[211] D. Porubsky, S. Garg, A. D. Sanders, J. O. Korbel, V. Guryev, et al. Dense and accurate whole-chromosome haplotyping of individual genomes. *Nature Communications*, 8(1), Nov. 2017.

[212] D. Porubsky, P. Ebert, P. A. Audano, M. R. Vollger, W. T. Harvey, et al. Fully phased human genome assembly without parental data using single-cell strand sequencing and long reads. *Nature Biotechnology*, 39(3):302–308, Dec. 2020.

[213] D. Porubský, A. D. Sanders, N. van Wietmarschen, E. Falconer, M. Hills, et al. Direct chromosome-length haplotyping by single-cell sequencing. *Genome Research*, 26(11): 1565–1574, Sept. 2016.

[214] E. Rakici, A. Altunisik, K. Sahin, and O. B. Ozgumus. Determination and molecular analysis of antibiotic resistance in gram-negative enteric bacteria isolated from pelophylax sp. in the eastern black sea region. *Acta Veterinaria Hungarica*, 69(3):223–233, Sept. 2021.

[215] T. Rausch, T. Zichner, A. Schlattl, A. M. Stütz, V. Benes, et al. DELLY: structural variant discovery by integrated paired-end and split-read analysis. *Bioinformatics*, 28(18):i333–i339, Sept. 2012.

[216] M. Rautiainen and T. Marschall. GraphAligner: rapid and versatile sequence-to-graph alignment. *Genome Biology*, 21(1):253, Sept. 2020.

[217] M. Rautiainen, V. Mäkinen, and T. Marschall. Bit-parallel sequence-to-graph alignment. *Bioinformatics*, 35(19):3599–3607, Oct. 2019.

[218] M. Rautiainen, S. Nurk, B. P. Walenz, G. A. Logsdon, D. Porubsky, et al. Telomere-to-telomere assembly of diploid chromosomes with verkko. *Nature Biotechnology*, 41(10):1474–1482, Feb. 2023.

[219] F. Riaz and K. M. Ali. Applications of graph theory in computer science. In *2011 Third International Conference on Computational Intelligence, Communication Systems and Networks*, pages 142–145, jul 2011.

[220] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, Dec. 2004.

[221] J. T. Robinson, H. Thorvaldsdóttir, W. Winckler, M. Guttman, E. S. Lander, G. Getz, and J. P. Mesirov. Integrative genomics viewer. *Nature Biotechnology*, 29(1):24–26, Jan. 2011.

[222] I. B. Rogozin. Computational approaches for the analysis of gene neighbourhoods in prokaryotic genomes. *Briefings in Bioinformatics*, 5(2):131–149, Jan. 2004.

[223] I. B. Rogozin, K. S. Makarova, D. A. Natale, A. N. Spiridonov, R. L. Tatusov, et al. Congruent evolution of different classes of non-coding DNA in prokaryotic genomes. *Nucleic Acids Research*, 30(19):4264–4271, Oct. 2002.

[224] J. I. Rojas Echenique, S. Kryazhimskiy, A. N. Nguyen Ba, and M. M. Desai. Modular epistasis and the compensatory evolution of gene deletion mutants. *PLOS Genetics*, 15(2):e1007958, Feb. 2019.

[225] M. Ronaghi, M. Uhlén, and P. Nyrén. A sequencing method based on real-time pyrophosphate. *Science*, 281(5375):363–365, July 1998.

[226] N. Rusk. Torrents of sequence. *Nature Methods*, 8(1):44–44, Dec. 2010.

[227] M. L. M. Salverda, E. Dellus, F. A. Gorter, A. J. M. Debets, J. van der Oost, et al. Initial mutations direct alternative pathways of protein evolution. *PLoS Genetics*, 7(3):e1001321, Mar. 2011.

[228] A. D. Sanders, E. Falconer, M. Hills, D. C. J. Spierings, and P. M. Lansdorp. Single-cell template strand sequencing by strand-seq enables the characterization of individual homologs. *Nature Protocols*, 12(6):1151–1176, May 2017.

[229] A. D. Sanders, S. Meiers, M. Ghareghani, D. Porubsky, H. Jeong, et al. Single-cell analysis of structural variations and complex rearrangements with tri-channel processing. *Nature Biotechnology*, 38(3):343–354, Dec. 2019.

[230] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, Dec. 1977.

[231] T. Sanju. SOLiD Sequencing: Principle, Steps, Applications, Diagram. https://microbenotes.com/solid-sequencing/, 2024. [Accessed 26-01-2025].

[232] R. Sanjuán, J. M. Cuevas, A. Moya, and S. F. Elena. Epistasis and the adaptability of an rna virus. *Genetics*, 170(3):1001–1008, July 2005.

[233] D. Sankoff. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics*, 28 (1):35–42, 1975.

[234] D. Sankoff and P. Rousseau. Locating the vertices of a steiner tree in an arbitrary metric space. *Mathematical Programming*, 9(1):240–246, Dec. 1975.

[235] A. Santos-Lopez, C. Bernabe-Balas, M. Ares-Arroyo, R. Ortega-Huedo, A. Hoefer, et al. A naturally occurring single nucleotide polymorphism in a multicopy plasmid produces a reversible increase in antibiotic resistance. *Antimicrobial Agents and Chemotherapy*, 61(2), Feb. 2017.

[236] H. Satam, K. Joshi, U. Mangrolia, S. Waghoo, G. Zaidi, et al. Next-generation sequencing technology: Current trends and advancements. *Biology*, 12(7):997, July 2023.

[237] C. P. Schaaf, J. Wiszniewska, and A. L. Beaudet. Copy number and snp arrays in clinical diagnostics. *Annual Review of Genomics and Human Genetics*, 12(1):25–51, Sept. 2011.

[238] S. Schbath, V. Martin, M. Zytnicki, J. Fayolle, V. Loux, and J.-F. Gibrat. Mapping reads on a genomic sequence: An algorithmic overview and a practical comparative analysis. *Journal of Computational Biology*, 19(6):796–813, June 2012.

[239] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 76–85, June 2003.

[240] I. Schrauwen, Y. Rajendran, A. Acharya, S. Öhman, M. Arvio, et al. Optical genome mapping unveils hidden structural variants in neurodevelopmental disorders. *Scientific Reports*, 14(1), May 2024.

[241] S. Secomandi, G. R. Gallo, R. Rossi, C. Rodríguez Fernandes, E. D. Jarvis, et al. Pangenome graphs and their applications in biodiversity genomics. *Nature Genetics*, 57(1):13–26, Jan. 2025.

[242] J. Shang, J. Zhang, Y. Sun, D. Liu, D. Ye, and Y. Yin. Performance analysis of novel methods for detecting epistasis. *BMC Bioinformatics*, 12(1), Dec. 2011.

[243] B. Shapiro, A. Rambaut, O. G. Pybus, and E. C. Holmes. A phylogenetic method for detecting positive epistasis in gene sequences and its application to RNA virus evolution. *Molecular Biology and Evolution*, 23(9):1724–1730, Sept. 2006.

[244] S. L. Sheetlin, Y. Park, M. C. Frith, and J. L. Spouge. Frameshift alignment: statistics and post-genomic applications. *Bioinformatics*, 30(24):3575–3582, Dec. 2014.

[245] A. Shlien and D. Malkin. Copy number variations and cancer. *Genome Medicine*, 1(6):62, 2009.

[246] F. Sigaux. Cancer genome or the development of molecular portraits of tumors. *Bulletin de l'Académie Nationale de Médecine*, 184(7):1441–7; discussion 1448–9, 2000.

[247] J. Siren, N. Valimaki, and V. Makinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2): 375–388, Mar. 2014.

[248] J. Siren, N. Valimaki, and V. Makinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2): 375–388, March 2014.

[249] J. Sirén, J. Monlong, X. Chang, A. M. Novak, J. M. Eizenga, et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, 374(6574):abg8871, Dec. 2021.

[250] J. Sirén, P. Eskandar, M. T. Ungaro, G. Hickey, J. M. Eizenga, et al. Personalized pangenome references. *Nature Methods*, 21(11):2017–2023, Sept. 2024.

[251] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, Mar. 1981.

[252] M. Smolka, L. F. Paulin, C. M. Grochowski, D. W. Horner, M. Mahmoud, et al. Detection of mosaic and population-level structural variants with Sniffles2. *Nature Biotechnology*, 42(10): 1571–1580, Jan. 2024.

[253] A. Soylev, J. Ebler, S. Pani, T. Rausch, J. Korbel, and T. Marschall. Svarp: pangenome-based structural variant discovery. *bioRxiv*, Feb. 2024.

[254] R. Staden. A strategy of dna sequencing employing computer programs. *Nucleic Acids Research*, 6(7):2601–2610, 1979.

[255] T. N. Starr and J. W. Thornton. Epistasis in protein evolution. *Protein Science*, 25(7): 1204–1218, July 2016.

[256] P. J. Stephens, C. D. Greenman, B. Fu, F. Yang, G. R. Bignell, et al. Massive genomic rearrangement acquired in a single catastrophic event during cancer development. *Cell*, 144(1): 27–40, Jan. 2011.

[257] J. F. Storz. Compensatory mutations and epistasis for protein function. *Current Opinion in Structural Biology*, 50:18–25, June 2018.

[258] G. G. Sutton, O. White, M. D. Adams, and A. R. Kerlavage. TIGR assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1(1):9–19, 1995.

[259] Y. Suzuki. Natural selection on the influenza virus genome. *Molecular Biology and Evolution*, 23(10):1902–1911, Oct. 2006.

[260] J. Tamames. Evolution of gene order conservation in prokaryotes. *Genome Biology*, 2(6), June 2001.

[261] L. Tang. Circular consensus sequencing with long reads. *Nature Methods*, 16(10):958–958, Sept. 2019.

[262] M. Tarabichi, A. Salcedo, A. G. Deshwar, M. Ni Leathlobhair, J. Wintersinger, D. C. Wedge, P. Van Loo, Q. D. Morris, and P. C. Boutros. A practical guide to cancer subclonal reconstruction from DNA sequencing. *Nature Methods*, 18(2):144–155, Feb. 2021.

[263] M. Tarabichi, A. Salcedo, A. G. Deshwar, M. Ni Leathlobhair, J. Wintersinger, et al. A practical guide to cancer subclonal reconstruction from dna sequencing. *Nature Methods*, 18(2): 144–155, Jan. 2021.

[264] B. S. Taylor, M. E. Sobieszczyk, F. E. McCutchan, and S. M. Hammer. The challenge of HIV-1 subtype diversity. *The New England Journal of Medicine*, 358(15):1590–1602, Apr. 2008.

[265] H. Tettelin, V. Masignani, M. J. Cieslewicz, C. Donati, D. Medini, et al. Genome analysis of multiple pathogenic isolates of *Streptococcus agalactiae*: Implications for the microbial "pan-genome". *Proceedings of the National Academy of Sciences*, 102(39):13950–13955, 2005.

[266] The Eddy-Rivas Laboratory. Hmmer. `https://github.com/EddyRivasLab/hmmer`, 1997. Accessed 06-03-2023.

[267] The SAM/BAM Format Specification Working Group. Sequence alignment/map optional fields specification, 2024. URL `https://github.com/samtools/hts-specs/blob/master/SAMtags.pdf`. [Accessed 18-11-2024].

[268] 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.

[269] M. Toft and O. A. Ross. Copy number variation in parkinson's disease. *Genome Medicine*, 2 (9), Sept. 2010.

[270] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1): 100–118, Jan. 1985.

[271] I. A. E. M. van Belzen, A. Schönhuth, P. Kemmeren, and J. Y. Hehir-Kwa. Structural variant detection in cancer genomes: computational challenges and perspectives for precision oncology. *npj Precision Oncology*, 5(1), Mar. 2021.

[272] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, et al. The sequence of the human genome. *Science*, 291(5507):1304–1351, Feb. 2001.

[273] J. Vila, J. Ruiz, P. Goñi, and M. T. De Anta. Detection of mutations in parc in quinolone-resistant clinical isolates of escherichia coli. *Antimicrobial Agents and Chemotherapy*, 40(2): 491–493, Feb. 1996.

[274] T. J. Vision. Gene order in plants: a slow but sure shuffle. *New Phytologist*, 168(1):51–60, Aug. 2005.

[275] I. M. Wallace. M-Coffee: combining multiple sequence alignment methods with t-coffee. *Nucleic Acids Research*, 34(6):1692–1699, Mar. 2006.

[276] T. S. K. Wan. Cancer cytogenetics: Methodology revisited. *Annals of Laboratory Medicine*, 34 (6):413–425, Nov. 2014.

[277] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.

[278] Y. Wang, Y. Zhao, A. Bollas, Y. Wang, and K. F. Au. Nanopore sequencing technology, bioinformatics and applications. *Nature Biotechnology*, 39(11):1348–1365, Nov. 2021.

[279] J. H. Ward, Jr. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, Mar. 1963.

[280] R. L. Warren, G. G. Sutton, S. J. M. Jones, and R. A. Holt. Assembling millions of short dna sequences using SSAKE. *Bioinformatics*, 23(4):500–501, Dec. 2006.

[281] J. D. Watson and F. H. C. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, Apr. 1953.

[282] M. A. Webber, M. M. C. Buckner, L. S. Redgrave, G. Ifill, L. A. Mitchenall, et al. Quinolone-resistant gyrase mutants demonstrate decreased susceptibility to triclosan. *Journal of Antimicrobial Chemotherapy*, 72(10):2755–2763, Oct. 2017.

[283] T. Weber, M. R. Cosenza, and J. Korbel. Mosaicatcher v2: a single-cell structural variations detection and analysis reference framework based on strand-seq. *Bioinformatics*, 39(11), Oct. 2023.

[284] A. M. Wenger, P. Peluso, W. J. Rowell, P.-C. Chang, R. J. Hall, et al. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nature Biotechnology*, 37(10):1155–1162, Aug. 2019.

[285] R. Wernersson and A. G. Pedersen. RevTrans: Multiple alignment of coding DNA from aligned amino acid sequences. *Nucleic Acids Research*, 31(13):3537–3539, July 2003.

[286] A. Westbrook, J. Ramsdell, T. Schuelke, L. Normington, R. D. Bergeron, W. K. Thomas, et al. PALADIN: protein alignment for functional profiling whole metagenome shotgun data. *Bioinformatics*, 33(10):1473–1478, May 2017.

[287] M. Wheldale. The inheritance of flower colour in antirrhinum majus. *Proceedings of the Royal Society of London, Series B*, 79(532):288–305, July 1907.

[288] W. B. Whitman, D. C. Coleman, and W. J. Wiebe. Prokaryotes: the unseen majority. *Proceedings of the National Academy of Sciences*, 95(12):6578–6583, 1998.

[289] R. R. Wick, M. B. Schultz, J. Zobel, and K. E. Holt. Bandage: interactive visualization of de novo genome assemblies. *Bioinformatics*, 31(20):3350–3352, Oct. 2015.

[290] K. C. Worley, S. Richards, and J. Rogers. The value of new genome references. *Experimental Cell Research*, 358(2):433–438, Sept. 2017.

[291] L. Yang. A practical guide for structural variation detection in the human genome. *Current Protocols in Human Genetics*, 107(1), Aug. 2020.

[292] X. Yu, D. Zhang, and Q. Song. Profiles of gyra mutations and Plasmid-Mediated quinolone resistance genes in shigella isolates with different levels of fluoroquinolone susceptibility. *Infection and Drug Resistance*, 13:2285–2290, July 2020.

[293] C. Zhang, Q. Wang, Y. Li, A. Teng, G. Hu, et al. The historical evolution and significance of multiple sequence alignment in molecular structure and function prediction. *Biomolecules*, 14 (12):1531, Nov. 2024.

[294] Y. Zhou, Z. Zhang, Z. Bao, H. Li, Y. Lyu, et al. Graph pangenome captures missing heritability and empowers tomato breeding. *Nature*, 606(7914):527–534, June 2022.

[295] J. Zhu, H.-J. Tsai, M. R. Gordon, and R. Li. Cellular stress associated with aneuploidy. *Developmental Cell*, 44(4):420–431, Feb. 2018.

Appendix A

PanPA: PanProteome Graph Builder and Aligner

A.1 Supplementary Tables

This section reuses material from [57] of which I am the first author

In this section we present three supplementary tables:

1. Table A.1 contains the raw alignments numbers of the Salmonella *enterica* sequence alignments using three different tools.
2. Table A.2 contains the different intersection numbers between the different aligners
3. Table A.3 shows the effect of the different parameters on PanPA's performance in the context of its comparison against HMMER.

Aligner	BWA	GraphAligner	PanPA
Num. alignments	2,699,361	26,009,077	8,684,414
Num. filtered alignment 50% length	1,645,224	4,399,906	7,897,707
Num. filtered alignments 50% id	1,645,224	4,399,906	7,897,707
Num. filtered alignments 70% id	1,645,222	4,384,913	5,273,200

Table A.1: Number of alignments from the 4,839,981 sequences from Salmonella *enterica* annotations using BWA, GraphAligner and PanPA. We see that GraphAligner produced the most alignments. However, after filtering for an alignment length of at least 50% of the original sequence size, the number of alignments drops drastically. For PanPA, most of the alignments were long enough and only a small number got filtered. *Table taken from [57].*

Intersection	Number of alignments $\geq 50\%$ identity	Number of alignments $\geq 70\%$ identity
Not Aligned	744,964	1,012,744
BWA	1	1
BWA - GraphAligner	4,084	4,090
BWA - PanPA	1,294	1,294
GraphAligner	12,488	52,357
Graphaligner - PanPA	1,694,181	1,643,479
PanPA	744,033	487,086
BWA - GraphAligner - PanPA	1,638,936	1,638,930

Table A.2: Intersection of unique alignments of 4,839,981 sequences representing the annotations from Salmonella *enterica* assemblies from RefSeq, against E. *coli* linear reference, pangenome, and panproteome using `BWA`, `GraphAligner`, and `PanPA` respectively. *Table taken from [57].*

Number of sequences	Identity cutoff percentage	Number of threads	Graph limit	Aligning time mm:ss	total time mm:ss	memory
10,000	40	1	10	15:47	20:14	2.2 Gb
10,000	40	10	10	2:13	7:25	2.2 Gb
10,000	10	1	10	16:09	20:57	2.2 Gb
10,000	10	10	10	2:16	7:02	2.2 Gb

Table A.3: The effect of the different parameters on `PanPA`'s performance, we see that the identity cutoff does not affect the alignment time much, and this makes sense, as the alignment will be performed anyway to get an alignment identity score and check whether it is below or above the cutoff. *Table taken from [57].*

A.2 MSA to GFA

A simple command-line toolkit was first developed for converting an MSA to a GFA, it is called `msa_to_gfa`. Later, its internal functionality was integrated into `PanPA`.
This toolkit takes an MSA as an input and generates a DAG output in the GFA format. However, it outputs the path as a separate JSON file where each original sequence in the MSA has one continuous path in the output GFA. Moreover, it generates groups of sequences that share the same path. This grouping can be useful when coloring the paths in a graph visualizer like `gfaviz` [97], where each path will get a color, and when many paths share the same node, it can overcrowd the visualization. Therefore, a group can have one color and it represents several paths.

Supplementary Figure A.1 shows an example of six sequences in an MSA, their variations generate three bubbles. using `msa_to_gfa`, we convert the MSA to a GFA and get a JSON file with the paths for each sequence and three groups, where the sequences that follow the same path are grouped together. Namely, sequences 1, 2, and 3 form group 1; sequence 4

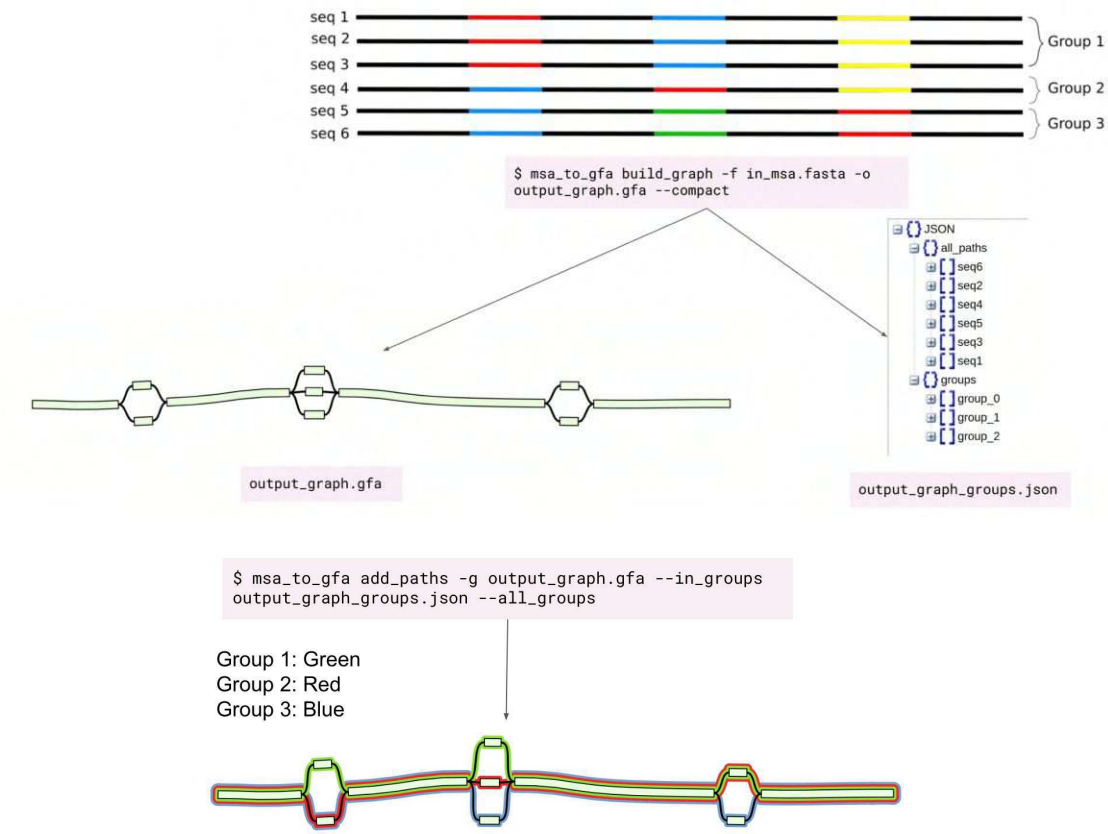


Figure A.1: First Step using the subcommand `build_graph`: taking the six sequences here that have three heterozygous positions. Running the command will output a graph in GFA format and the groups information as a JSON file. Same sequences are grouped together. Where the JSON file has information to which sequences belong to which group, and the path in the graph for each group. Second Step using the subcommand `add_paths`: Taking the graph outputted from the first step and the JSON file, users can either choose to add all grapus paths to the graph with `--all_groups` or select a subset of groups to add with `--some_groups`. For example, User can choose to visualize only Group 1 or only seq 6. Graph visualized using `gfaviz` and paths are colored using the groups in the JSON file.

forms group 2; and sequences 5 and 6 form group 3.

A.3 Random Sequences Selection Mechanism

This section reuses material from [57] of which I am the first author

The 32,289 sequences from the *E. coli* panproteome that were chosen for testing were chosen at random using a script that can be found on `PanPA`'s repository, it takes two arguments as input, both integers from 0 to 100 representing the percentage of different protein clusters to choose at random, and how many sequences to choose from each MSA. We gave the script the inputs 10 and 5, which then chooses at random 10% of the protein clusters, and then

from each cluster chooses 5% of the sequences at random. Because we know from which cluster each sequence belongs to, we can calculate the number of matches after doing the alignments.

A.4 Aligning to Sparse MSAs

This section reuses material from [57] of which I am the first author

To further evaluate and test the limits of PanPA, we tried to build a graph and align sequences back to a protein family, we took an MSA from Pfam (PF00006.28) representing the ATP Synthase Alpha/Beta family. Due to the nature of protein families, the MSAs tend to be very sparse as the sequences are evolutionary-related, but in terms of sequence identity, it is rather low. In cases like this, the graph resulting from the MSA tends to also be sparse, i.e., contains many nodes representing small substrings and many edges. For this protein family, the MSA contained 40,339 sequences. It took PanPA around 3 seconds to build the GFA, and about 2 minutes to align back a sample of 1,000 sequences of the same MSA back using 1 thread. This is relatively high compared to a more conserved MSA, for example, taking the MSA representing the gene *Araa* from the *E. coli* panproteome which contains 21,657 sequences and it only took PanPA about 10 seconds to align a sample of 1,000 sequences back to this graph using 1 thread.
The case of the ATP family can be considered an extreme case, as this MSA is quite sparse with many gaps, and the graph constructed consists of 13,463 nodes with a total concatenated sequence of length 15,303. Therefore, PanPA needs to build for each query sequence, a DP table the size of $n \times 15303$ where n is the size of the query sequence. Moreover, the average number of incoming edges has an effect here: in this example, for instance, each node had - on average - 5 incoming edges, which means that for calculating each cell in the DP table, PanPA needs to follow 5 different paths and calculate the scores before choosing the best one. However, this is an extreme case of an MSA and PanPA can still handle such graphs and alignments, albeit slower.

A.5 Indexing Time and Space

This section reuses material from [57] of which I am the first author

Supplementary Figures A.2 and A.3 show the relationship between the different indexing parameters and the indexing time and size, respectively. We can see that there is a tradeoff between time and index size, as extracting k-mers as seeds is faster than extracting (w, k)-minimizers. However, the index size is bigger when the seeds are k-mers. This is expected, as minimizers only take one k-mer out of a window of size w, which means that it includes less seeds in total in the final index, compared to taking each k-mer in the sequences as a

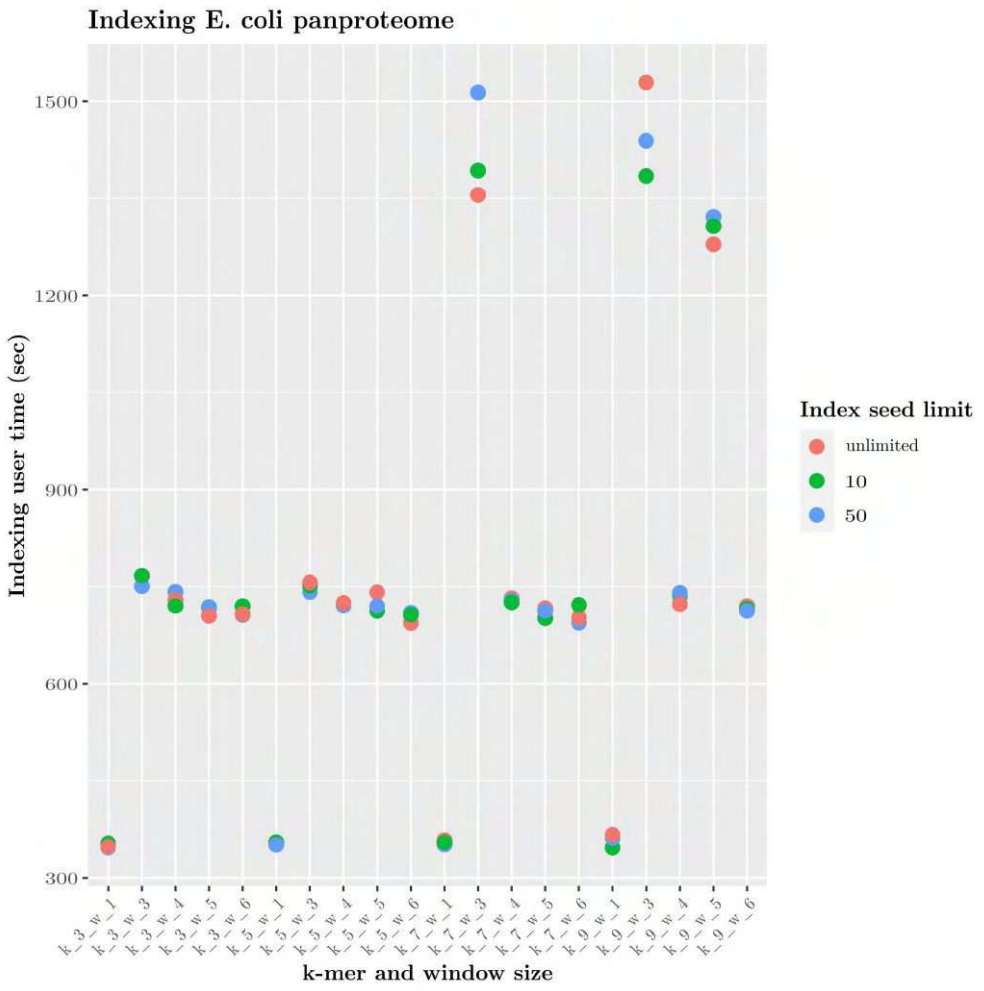


Figure A.2: This figure plots the relationship between the different parameters chosen to build the index, and the user time it took in seconds. We see that extracting k-mers is always faster than (w, k)-minimizers, which is expected, as extracting a single k-mer requires less operations than extracting a window of k-mers and taking the minimum.

seed for the index.

A.6 Command line tools and Parameters

A.6.1 Alignment comparison of *S. enterica* protein sequences

For aligning the DNA sequences from *S. enterica* against the *E. coli* reference genome, we use `BWA` with the following parameters:

```
$ bwa mem e_coli_reference_GCF_000005845.2_ASM584v2.fasta
   salmonella_refseqdna.fasta -t 60 >
   salmonella_refseqdna_ecoli_ref_genome_bwa.sam
```

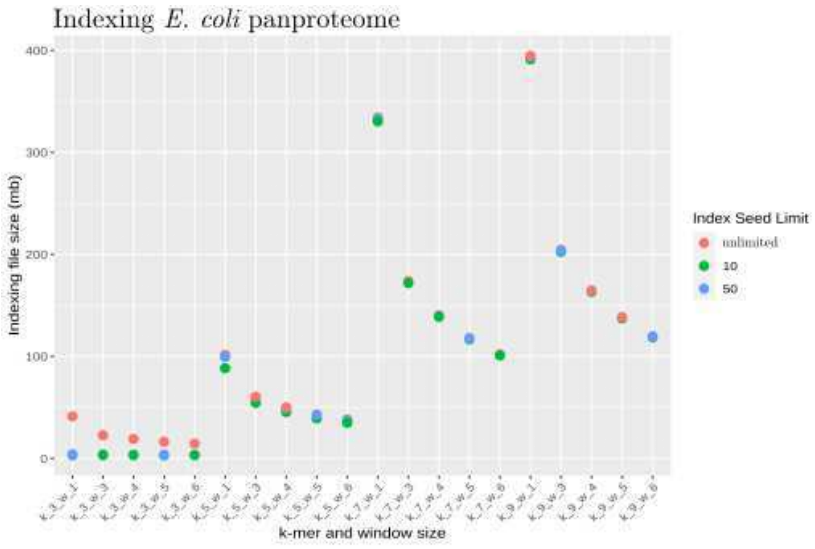


Figure A.3: This figure plots the relationship between the different parameters chosen to build the index, and the index file size, which also represents the index size. We see that when using k-mers index, the idnex size is bigger, compared to (w, k)-minimizers. This is expected, as a k-mer index saves each k-mer, while the (w, k)-minimizers only take one k-mer from a window, which then requires less k-mers or seeds to be stored in the index in total.

We used `GraphAligner` for aligning *S. enterica* DNA sequences against *E. coli* pangenome that was built using `minigraph` with the following parameters:

```
$ GraphAligner -f salmonella_refseq_dna.fasta -g e_coli_pangenome.
   gfa -a salmonella_refseqdna_ecoli_pangenome.gaf -x vg --threads
   60 2> graph_align.log

# for building the pangenome, these commands were used
# this command is the initial one to build a graph
minigraph -xggs -t20 e_coli_reference_GCF_000005845.2_ASM584v2.
   fasta e_coli_reference_GCF_000005845.2_ASM584v2.fasta >
   e_coli_pangenome.gfa

# updating the graph by adding one assembly every step
# assemblies_locations.txt is a list of each E. coli assembly to
   update the graph
$ while read r;do minigraph -xggs -t20 e_coli_pangenome.gfa $r >
   tmp && mv tmp e_coli_pangenome.gfa;done < assemblies_locations.
   txt
```

`PanPA` was used with the following parameters:

```
$ PanPA --log_file salmonella_aa.log align -d e_coli_gfa/ --index
```

```
   index_k_5_w_5_seed_lim_10.pickle -r salmonella_aa.fasta.gz -o
   salmonella_aa_ecoli_panproteome.gaf --min_id_score 0.5 --cores
   50 2> panpa_time.log
```

A.6.2 Aligning short reads parameters

For aligning the short reads a sample of *S. enterica* from SRA database with accession number SRR22756191. The following command was used for BWA:

```
$ bwa mem -t 5 reference_ecoli_GCF_000005845/ SRR22756191.fasta >
   alignments_SRR22756191.sam
```

For PanPA, the following command was used:

```
$ PanPA --log_file running_new_fs_panpa.log align -d
   index_k_5_w_3_seed_lim_0.pickle -r SRR22756191.fasta --dna -c 10
    -o SRR22756191_ecoli_panproteome_sf_panpa.gaf --min_id_score
   0.35 --seed_limit 20
```

A.6.3 Comparison with HMMER parameters

The following parameters were used with PanPA to align the 10,000 sequences of *S. enterica* against the *E. coli* panproteome.

```
$ PanPA --log_file panpa_alignment_10k_10core.log align -d
   e_coli_gfa/ --index e_coli_msas_index_k5_w3_no_limit.index -r
   random_10k_sequences.fasta -c 10 -o panpa_10k_alignments_0.4
   min_10core.gaf --min_id_score 0.4 --seed_limit 10
```

As for running HMMER, first, we need to convert each MSA into an HMM profile using hmmbuild

```
#!/bin/bash

home_dir=e_coli_hmms/
msa_dir=/e_coli_msa/

start=`date +%s`
for f in $msa_dir/*;
do
        base_fasta=$(basename --suffix .fasta $f)
        hmmbuild $base_fasta.hmm $f
        #echo $base_fasta.hmm
        #echo $f
done
```

```
end=`date +%s`
echo it took `expr $end - $start` seconds to run hmmer on all
   clusters > ../hmmer_time.txt

# Compressing all the hmms into one file using hmmpress
$ hmmpress all_hmms
# which would produced an indexed file with all hmmer profiles

# for search hmmsearch was used
$ hmmsearch --cpu 10 --tblout random_10k_sequences.txt -o
   hmmsearch_10k_output_mt.txt -A hmmsearch_10k_alignment.sto
   all_hmms random_10_sequences.fasta
```

The table produced by HMMER has the sequence hits against profiles, which we used to match with the alignments produced by PanPA

A.6.4 Gene Order Analysis parameters

In this experiment, we aligned genome assemblies back to a selection of gene graphs to find the order of these genes in these assemblies. We first needed to cut the assemblies into smaller overlapping sequences, as the complete genome assembly is too big to build one DP table for; we used a custom script for extracting these sequences (the script is available on the github repository of PanPA)

```
$ python3 extract_seqs_from_ref.py reference_file.fna 10000 >
   overlapping_sequences.fasta
```

Before aligning back these sequences against the graphs, we need to generate an index for the graphs of interest, we used the following parameters

```
$ PanPA --log_file build_index.log build_index -d gene_msas/ -o
   k_5_w_4_index -k 5 -w 4 --seed_limit 20
```

We can now use the index built and the overlapping sequences to align to the gene graphs of interest

```
$ PanPA --log_file align_ref.log align -d gfas/ --dna --index
   k_5_w_4_index -r overlapping_sequences.fasta -o
   overlapping_sequences.gaf --cores 2 --min_id_score 0.9 --
   seed_limit 10
```

We can then analyze the output GAF file to infer the gene order, this is done using a custom script that is also present on PanPA's repository.

```
$ python3 get_order_from_gaf.py overlapping_sequences.gaf
```

Appendix B

Graph toolkits: `GFASubgraphs`, `extgfa`, and `gaftools`

B.1 GFA representation in the GFA class

Below, shows how the GFA in Figure 1.6 is stored in the GFA class in `Python`.

```python
from GFASubgraphs.Graph import Graph
graph = Graph("example.gfa")
# to retrieve the edges at the end of node s1 for
print(graph['s1'].end)
# {('s3', 0, 2), ('s2', 1, 4)}
# We see how s1 connects to s3 from its start and to s2 from its end
# Looking at s2 end for example, graph['s2'].end, we get
graph['s2'].end
# {('s1', 1, 4)}
# indicating that s2 end connects to s1 end with an overlap of 4
```

B.2 Bi-Connected Component Detection

The bi-connected algorithm used attempts to find articulation points and bi-connected components using a non-recursive depth-first search that tracks the highest level reached by back edges in the DFS tree. A node is an articulation point if there is no back edge from any successor to any predecessor in the DFS tree. By tracking all edges traversed by the DFS, we can obtain the bi-connected components, since all edges of a bi-component will be traversed consecutively between articulation points.

B.3 GFA APIs Benchmarking

In order to compare to the different `Python` APIs for working with GFA graph, we used the following script that implements a simple component finding algorithm based on breadth-

first search algorithm, and we interfaced with each tool according to their description.

```
import GFASubgraph.Graph
import gfapy
import gfagraphs
import mygfa

def get_neighbors(nodeid, graph, library):
  if library in {0, 1}:         # gfasubgraph
    return graph[nodeid].neighbors()

  if library == 2:              # gfapy
    node = graph.try_get_segment(nodeid)
    return [x.to_name for x in node.all_references] + [x.from_name for x in node.
    all_references]

  if library == 3:              # gfagraphs
    neighbors = set()
    for e in graph.get_edges(nodeid):
      neighbors.add(e[0][0])
      neighbors.add(e[0][1])
    return list(neighbors)

def bfs(graph, start, library):
    visited = set()             # To keep track of visited nodes
    queue = deque([start])      # Initialize the queue with the starting node

    while queue:
        node = queue.popleft()  # Dequeue a node
        if node not in visited: # Only process if it's not visited
            visited.add(node)   # Mark the node as visited

            # Add all unvisited neighbors to the queue
            neighbors = get_neighbors(node, graph, library)
            for neighbor in neighbors:
                if neighbor not in visited:
                    queue.append(neighbor)
    return visited


def find_component(graph, start_node, visited, library):
    queue = []
    cc = set()
    queue.append(start_node)
    visited.add(start_node)
    while len(queue) > 0:
        start = queue.pop()
        if start not in cc:
            cc.add(start)
        else:
```

```
48              continue
49
50          visited.add(start)
51          neighbors = get_neighbors(start, graph, library)
52          for n in neighbors:
53              if n not in visited:
54                  queue.append(n)
55      return cc
```

Appendix C

Multi-Platform Investigation in Cancer Structural Variants and Subclones

C.1 Alignments

For aligning the long reads, both `pbmm2` and `minimap2` were used. We aligned against both the CHM13-T2T and GRCh38 human genome references. For indexing and calculating alignment depth, `samtools 1.6` was used. We needed to run alignments with `pbmm2` because the output alignments are needed for using `pbsv` caller. However, internally, `pbmm2` uses `minimap2` and produce similar results. The following commands were used for the alignments:

```
# these are the versions for pbmm2 and
pbmm2    : 1.13.1 (commit v1.13.1)
pbbam    : 2.5.0 (commit v2.5.0)
pbcopper : 2.4.0 (commit v2.4.0)
boost    : 1.81
htslib   : 1.17
minimap2 : 2.26
zlib     : 1.2.13

# example pbmm2 parameters
$ pbmm2 align --log-level INFO -J 4 -j 6 --preset CCS --sort --bam-
   index BAI --sample "BL2087" --rg '@RG\tID:m64093_221017_073615'

# and an example command for minimap2
$ ./minimap2 -ax map-hifi -t 10 /home/fawaz/projects/cancer_project
   /chm13_t2t_ref/chm13v2.0.fa temp/h2087_reads.fastq.gz
```

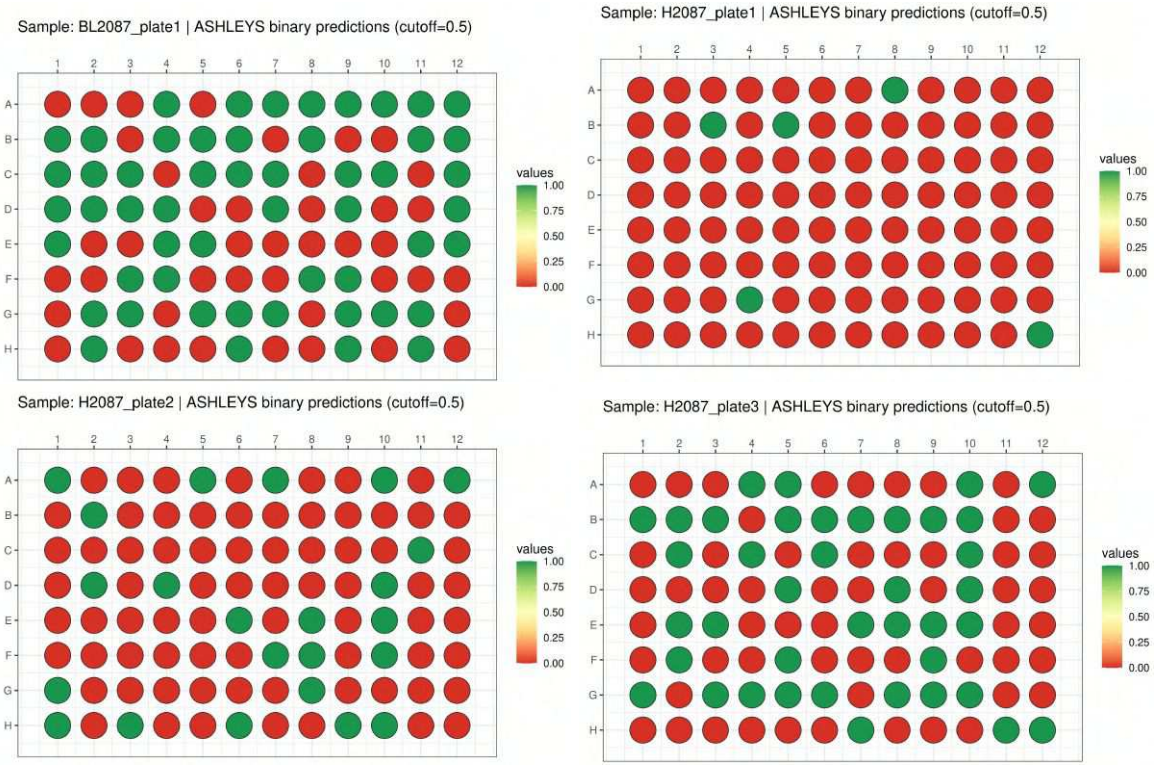


Figure C.1: `ASHLEYS` prediction for the good cells in each run of strand-seq sequencing. For the match normal BL2087 we ended up with 53 good cells, for H2087 Plate 1 we only had 5 good cells, for H2087 Plate 2 we got 23 good cells, and for H2087 Plate 3 we got 40 good cells.

For Illumina whole-genome short reads, `bwa mem 0.7.17-r1188` was used for alignments with default parameters.

Figure C.1 shows the predictions obtained by `ASHLEYS` for all the plates for both cell lines.

C.2 Structural Variants Calling

Figure C.2 shows a bar plot for the numbers of the SV types for each chromosome for both cell lines.

(1) Using the alignments produced by `pbmm2`, we can run `pbsv 2.9.0 commit v2.9.0-2-gce1559a`. The following commands were used for the structural variants calling with `pbsv`:

```
# first we need to run pbsv discover
$ pbsv discover --hifi -b human_chm13v2.0_maskedY_rCRS.trf.bed --
   region $i $in_bam.bam $in_bam.$i.svsig.gz
```

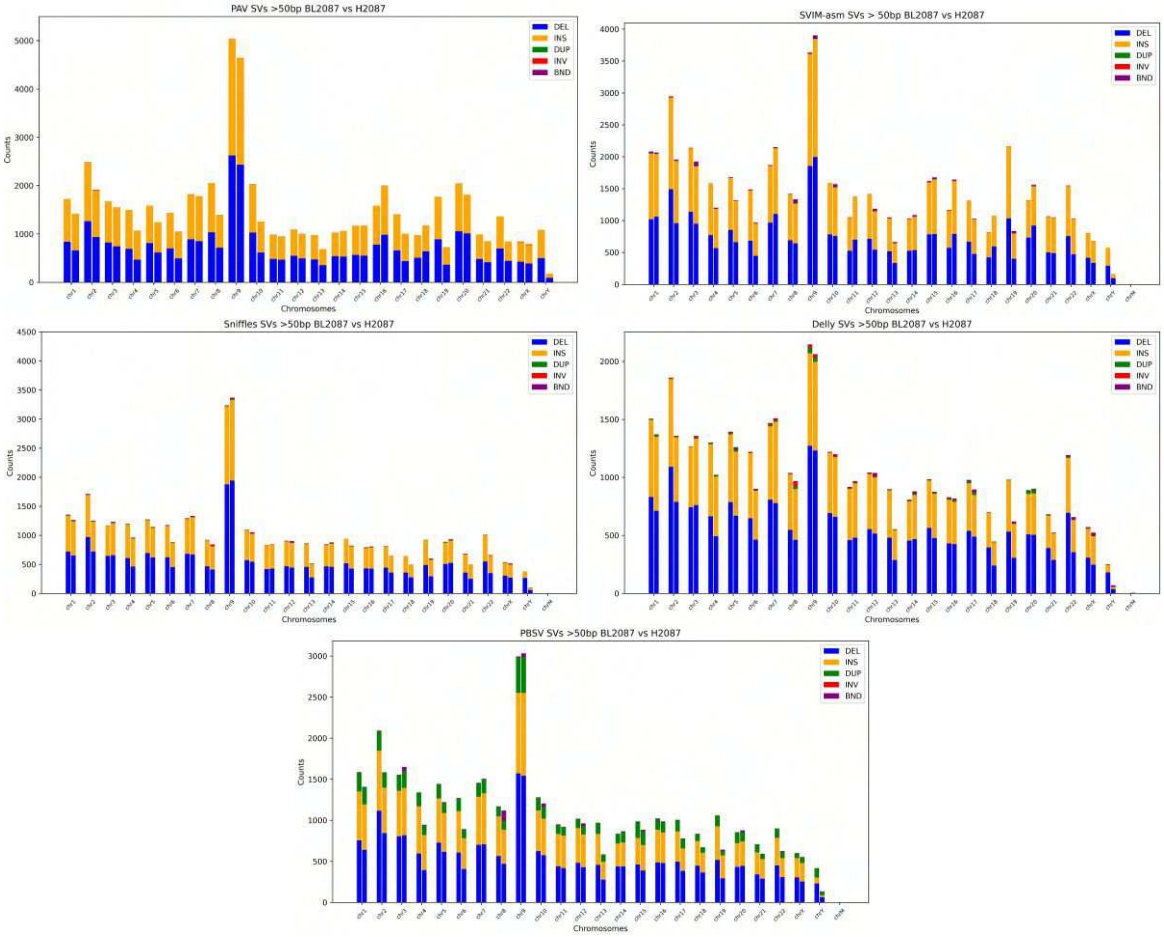


Figure C.2: Bar plots for the 5 different SV caller showing the distribution of the different SV types for each chromosome. At each chromosome on the x-axis, the left bar is for the BL2087 cell line and the right bar is for the H2087 cell line.

```
# this is looped for each chromosome separately and the tandem
   repeats are used from here
# https://github.com/PacificBiosciences/pbsv/tree/master/
   annotations

# then we need to run pbsv call on each chromosome
$ pbsv call -j 4 $ref $in_bam.*.svsig.gz $in_bam.pbsv.vcf
```

(2) Delly2 was used on both short and long reads variant calling and copy number variation calling. We used version 1.2.6 that uses Boost version 1.74.0 and HTSlib version 1.15.1. First, duplicate aligned reads needed to be marked, this was done using sambamba 1.0.0 LDC 1.28.1 / DMD v2.098.1 / LLVM12.0.0 / bootstrap LDC - the LLVM D compiler (1.28.1). default parameters were used for sambamba markdup, and the output was indexed with samtools 1.6. For getting the copy number variations using delly2 cnv we need mappability maps, which can be found in https://gear-genomics.embl.de/data/delly/. For the long reads, the author of delly2 suggested the parameters used. For short Illumina reads, to mark the duplicates, samtools fixmates was first used then samtools markdup was used to mark the duplicated, before running delly2 with default parameters.

```
# to run Delly2 on long reads for SV calling, we first need to mark
    the duplicate reads with sambamb
$ sambamba markdup input.bam output.bam
$ samtools index output.bam

# once duplicates are marked, we used delly lr
$ delly lr -g chm13.fa -o structural_variants.vcf in_bam.bam

# For calling copy number variations
# parameters recommended by Delly's author Tobias Rausch
$ delly cnv -i 10000 -j 10000 -w 10000 -g chm13.fa -m T2T-CHM13v1
   .1.fa.r101.s501.gz -c in_bam.delly.cnv.cov.gz -o in_bam.delly.
   cnv.bcf in_bam.bam >> delly_cnv_t2t.log 2>&1

# To run delly on the short reads alignemnts, it requires sorted,
   indexed, and duplicate-marked bam files
$ samtools sort -n -o tmp_sorted.bam -O BAM alignment.bam
$ samtools fixmates -m tmp_sorted.bam fixmates.bam
$ samtools sort -O sorted_fixmates.bam tmp_fixmates.bam
$ samtools markdup -r -s sorted_fixmates.bam final.bam
$ delly call -g example/ref.fa -o sr.bcf example/sr.bam
$ bcftools convert -O v $in_bam.delly.cnv.bcf > $in_bam.delly.cnv.
```

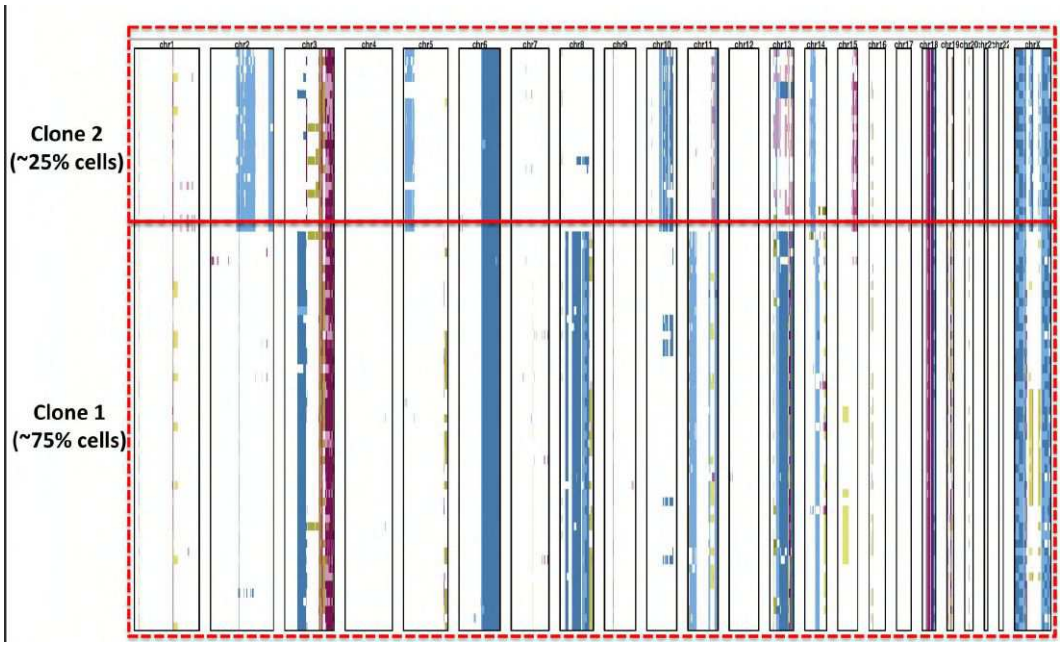


Figure C.3: Plot outputted by `Mosaicatcher` that colors the different variants found for each cell in the strand-seq over all the chromosomes. From this visualization, we can see there are two distinct signals that we believe corresponds to the two different subclones in the cancer sample.

```
    vcf
```

(3) We used `sniffles2 version 2.2` for calling the variants. It also requires a BED files with the tandem repeats and we used the same one that was used for running `pbsv`

```
$ sniffles --input $in_bam.bam --reference chm13.fa --vcf $in_bam.
    sniffles.vcf --tandem-repeats human_chm13v2.0_maskedY_rCRS.trf.
    bed --threads 6
```

(4) `pav version 2.3.4` for calling the variants on the assemblies produced by `PGAS`. Default parameters were used.
(5) We also used `SVIM-asm version 1.0.3` to call structural variants on the assemblies produced by `PGAS`, and default parameters were used.
(6) We also used `Hificnv version 0.1.7-70e9988` for calling copy number variation on the long-reads alignments produced by `pbmm2`.

```
$ hificnv --bam $in_bam.bam --ref chm13.fa --threads 6 --output-
    prefix $in_bam >> hificnv_bl2087_t2t.log 2>&1
```

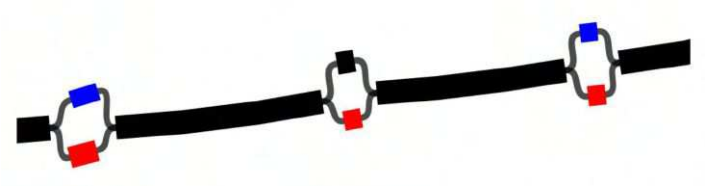


Figure C.4: Example on how the node coloring command from `graphdraw`, colors certain nodes based on the graph alignments provided. In this case, this was a bubble chain from an assembly graph produced by mixing both cancer and matched norma long reads. The long-reads are then aligned back to the graph and used as an input for the command.

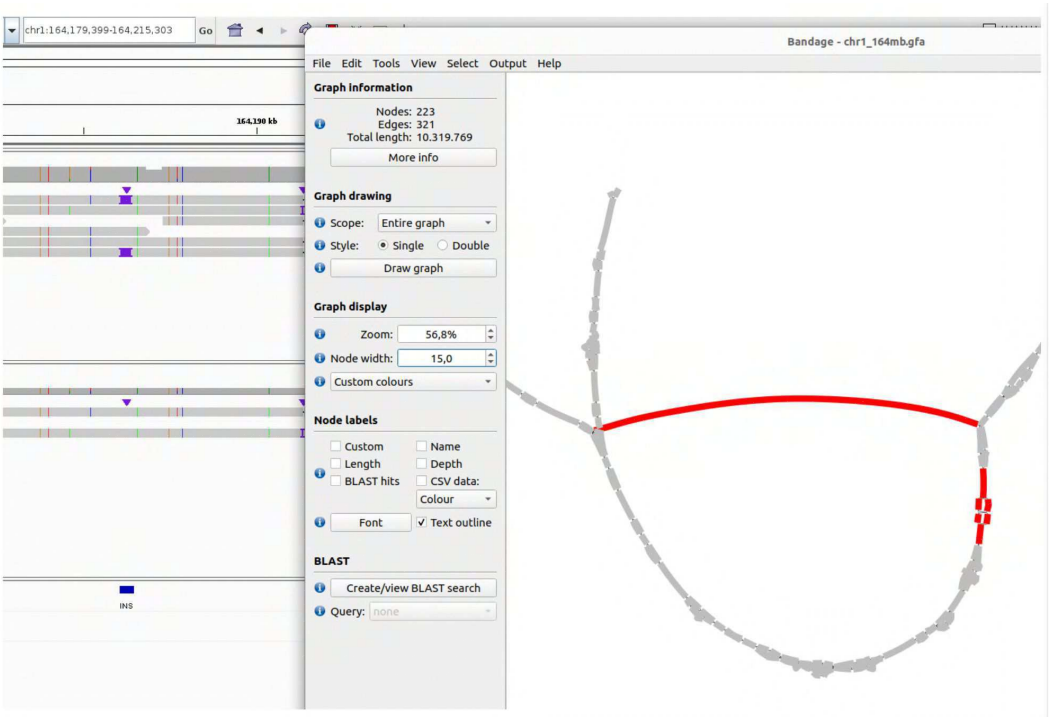


Figure C.5: This figure shows part of the graph extracted with `graphdraw`, where a somatic insertion affects a subset of the cancer raw unitigs produced from `hifiasm` assembly, which causes a bubble in the graph. The alignments are visualized with `IGV` [221] and the graph visuzlied with `Bandage`. This bubble can indicate the difference between the two subclones.

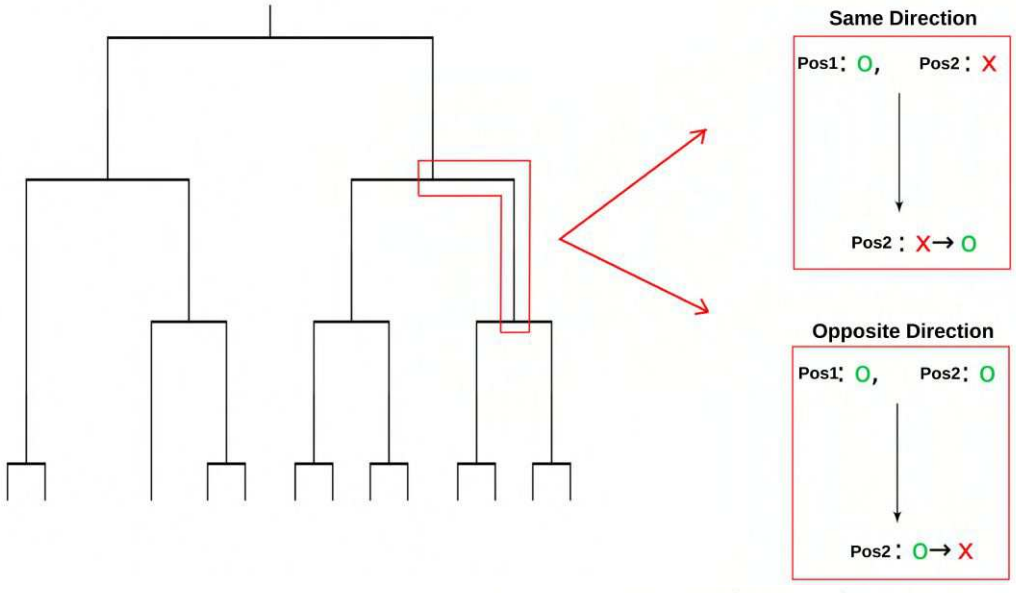
Appendix D

EpiPAMPAS: Epistasis Detection Using Parsimonious Ancestral State Reconstruction and Mutation Counting

This appendix reuses materials from [58] of which I am the first co-author

D.1 Mutation Direction Counting

Once we finished with the Sankoff algorithm and constructed the state of the inner nodes, i.e., reconstruct the most likely genotype of the inner nodes. We traverse the tree starting from the root and count the number of same and opposite direction mutations. We do two counting runs; first, we consider the first position in the protein position-pair as constant and count the mutation direction in the second position (as seen in Supplementary Figure D.1). Second, we consider the second position in the protein position-pair as constant and count the mutation direction of the first position.
Supplementary Figure D.1 also shows the two tables of the two possible counts, where same direction mutations are the sum $(b_2 + c_2)$ and opposite direction mutations are the sum $(a_2 + d_2)$.
In our method, we expect that variants/positions that do not have an epistatic interaction would fit a binomial distribution with a probability of 50%, and the ones with an epistatic interaction would deviate from this distribution. EpiPAMPAS offers the user to do a two-sided or one-sided binomial distribution, where a one-sided binomial can be then used to test if the count of one direction occurs more often than the other direction.



	Pos1 : o	Pos1 : x
Pos2: $o \to x$	a_1	b_1
Pos2: $x \to o$	c_1	d_1

	Pos2: o	Pos2: x
Pos1: $o \to x$	a_2	b_2
Pos1: $x \to o$	c_2	d_2

Figure D.1: The tree shows the same direction and opposite direction mutations of the second position (Pos2) while keeping (Pos1) constant. In red, is one event in the inner tree we are looking at, where we count 1 for the same direction mutation (top red box) if the mutation in the second position follows the first position and mutates to the same genotype, and count 1 for opposite direction mutations (bottom red box) if the mutation results in different genotypes. *Figure taken from [58]*

D.2 Supplementary Figures

Supplementary Figure D.2 showcases the different Venn diagrams for the comparison between EpiPAMPAS results and the results from [140]. Each row in the figure represents a different protein, the venn diagrams on the left are for the intersection of the positions detected by both methods, and the ones on the right show the intersection of the pairs detected.

Supplementary Figure D.3 shows scatter plots between the 1D and the 3D distance of the pairs detected in the proteins.

Figure D.2: Intersection between the positions and the pairs of positions EpiPAMPAS detected and the method from [140] for each of the viral proteins H1, H3, N1, and N2. We can see that the overlap of positions detected is big. However, the intersection when it comes to the pairs of interacting positions detected is rather small between the two methods. *Figure taken from [58]*

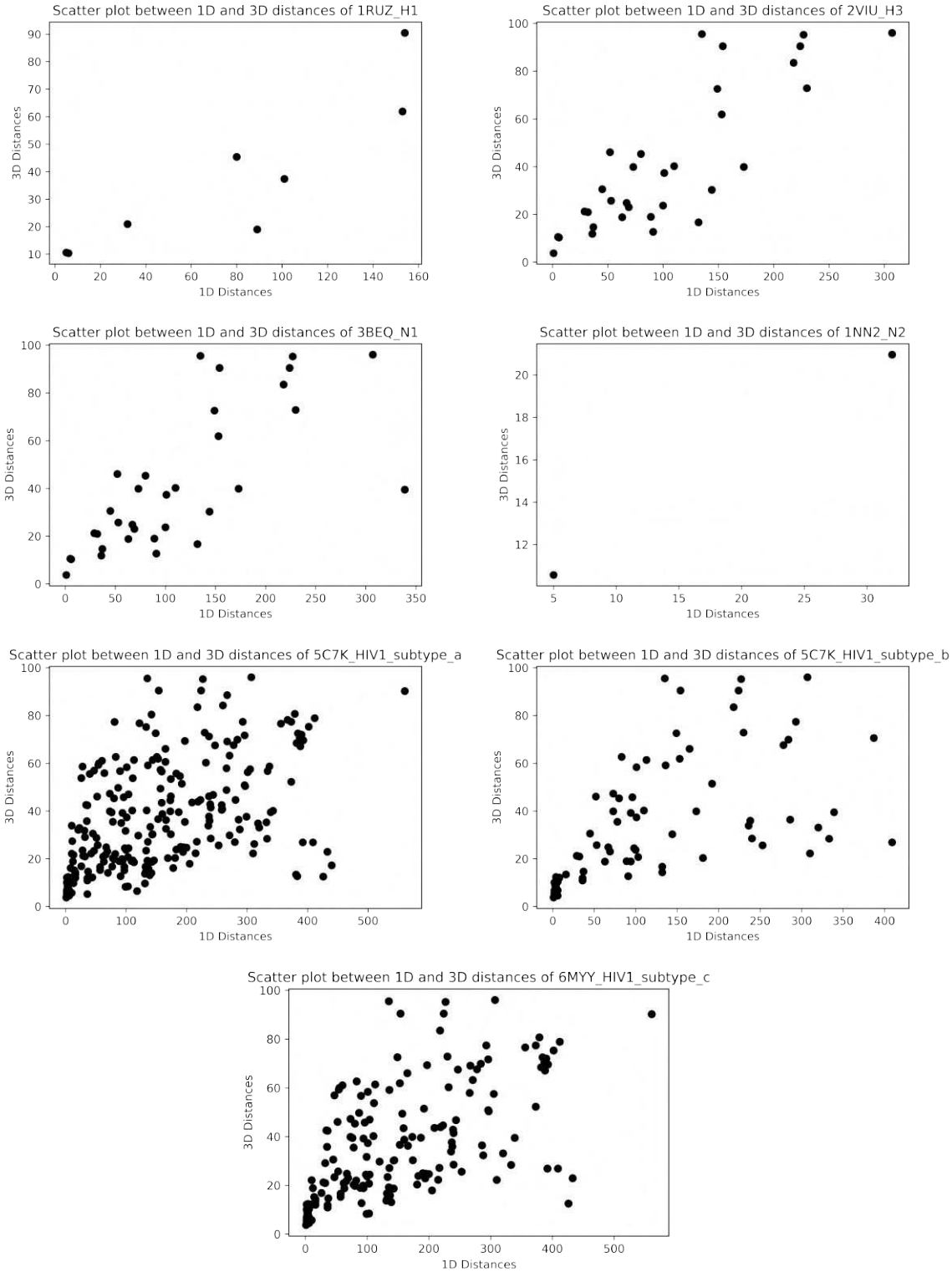


Figure D.3: Scatter plot of the 1D vs 3D distance of the pairs detected with EpiPAMPAS for H1, H3, N1, N2, HIV1 subtype a, HIV1 subtype b, and HIV1 subtype c using the structures 1RUZ, 2VIU, 3BEQ, 1NN2, 5C7K, 5C7K, and 6MYY respectively. We see that there is a trend where the longer the 1D distance, the longer the 3D distance. However, we would expect more of a trend where the 3D distance is smaller indicating that the pairs detected have some interaction in the 3D structure. *Figure taken from [58]*

Appendix E

Code Availability

The following list provides the URLs of the software and toolkits developed as part of this thesis:

1. The implementation of `PanPA` is available as an open-source code under the MIT license here: `https://github.com/fawaz-dabbaghieh/PanPA`
2. The implementation of `msa_to_gfa` is available as an open-source code under the MIT license here: `https://github.com/fawaz-dabbaghieh/msa_to_gfa`
3. The implementation of `GFASubgraph` is available as an open-source code under the MIT license here: `https://github.com/fawaz-dabbaghieh/gfa_subgraphs`
4. The implementation of `extgfa` is available as an open-source code under the MIT license here: `https://github.com/fawaz-dabbaghieh/extgfa`
5. The implementation of `gaftools` is available as an open-source code under the MIT license here: `https://github.com/marschall-lab/gaftools`
6. The implementation of `EpiPAMPAS` is available as an open-source code under the MIT license here: `https://github.com/kalininalab/EpiPAMPAS`
7. The implementation of `graphdraw` is available as an open-source code under the MIT license here: `https://github.com/fawaz-dabbaghieh/graphdrawing_toolkit`

Appendix F

Published articles underlying this thesis

F.1 `BubbleGun`: enumerating bubbles and superbubbles in genome graphs

The manuscript "`BubbleGun`: enumerating bubbles and superbubbles in genome graphs" [56] was published in *Bioinformatics*. Author information, author contributions, license and copyright information are listed in the subsections below.

F.1.1 Authors

Fawaz Dabbaghie, Jana Ebler, Tobias Marschall.

F.1.2 Contribution

As stated in the manuscript:
T.M. and F.D. designed the project and wrote the paper. F.D. implemented BubbleGun. J.E. helped with the bubble validation pipeline.

F.1.3 License and copyright information

As stated in the online version of the manuscript:
This is an Open Access article distributed under the terms of the Creative Commons Attribution License (`https://creativecommons.org/licenses/by/4.0/`), which permits unrestricted reuse, distribution, and reproduction in any medium, provided the original work is properly cited.

F.2 `PanPA`: generation and alignment of panproteome graphs

The manuscript "`PanPA`: generation and alignment of panproteome graphs" [57] was published in *Bioinformatics Advances*. Author information, author contributions, license and copyright information are listed in the subsections below.

F.2.1 Authors

Fawaz Dabbaghie, Sanjay K. Srikakulam, Tobias Marschall, Olga V. Kalinina.

F.2.2 Contribution

As stated in the manuscript:
F.D., T.M., and O.V.K. conceived the study. F.D. wrote PanPA, ran experiments, and wrote the manuscript. S.K.S. contributed to part of the code. T.M. and O.V.K. supervised the work and edited the manuscript. All authors read and approved the final version of the manuscript.

F.2.3 License and copyright information

As stated in the online version of the manuscript:
This is an Open Access article distributed under the terms of the Creative Commons Attribution License (`https://creativecommons.org/licenses/by/4.0/`), which permits unrestricted reuse, distribution, and reproduction in any medium, provided the original work is properly cited.

F.3 `extgfa`: a low-memory on-disk representation of genome graphs

The manuscript "`extgfa`: a low-memory on-disk representation of genome graphs" [52] was published as a preprint in *bioRxiv*. Author information, author contributions, license and copyright information are listed in the subsections below.

F.3.1 Authors

Fawaz Dabbaghie.

F.3.2 Contribution

This manuscript only has myself as an author and I am responsible for all the work in the manuscript.

F.3.3 License and copyright information

F.4 gaftools: a toolkit for analyzing and manipulating pangenome alignments

The manuscript "gaftools: a toolkit for analyzing and manipulating pangenome alignments" [196] was published as a preprint in *bioRxiv*. Author information, author contributions, license and copyright information are listed in the subsections below.

F.4.1 Authors

Samarendra Pani, Fawaz Dabbaghie, Tobias Marschall*, and Arda Söylev*.
The * indicates shared last authorship.

F.4.2 Contribution

The contribution is not mentioned in the preprint. However, S.P is the first author, and both T.M and A.S share the last authorship and supervised the work. As for this work, I contributed to the internal GFA class, graph ordering, and parallelizing the realignment step. I also contributed to writing the manuscript.

F.4.3 License and copyright information

F.5 EpiPAMPAS: Rapid detection of intra-protein epistasis via parsimonious ancestral state reconstruction and counting mutations

The manuscript "EpiPAMPAS: Rapid detection of intra-protein epistasis via parsimonious ancestral state reconstruction and counting mutations" [58] was published as a preprint in *bioRxiv*. Author information, author contributions, license and copyright information are listed in the subsections below.

F.5.1 Authors

Fawaz Dabbaghie*, Kristina Thedinga*, Georgii A Bazykin, Tobias Marschall†, Olga V. Kalinina†.
The * indicates shared first authorship, and the † indicates shared last authorship.

F.5.2 Contribution

The contribution is not mentioned in the preprint. However, K.T developed the original statistical method and did the simulated data experiments. I developed the method further to work on protein sequences and performed all the experiments on the real viral data obtained from G.B, and compared the results to another method. Both T.M and O.V.K supervised the work and share last authorship.

F.5.3 License and copyright information