EFFICIENT METHODS FOR READ MAPPING

A Dissertation
Presented to
The Academic Faculty

By

Haowen Zhang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering
College of Computing

Georgia Institute of Technology

August 2022

EFFICIENT METHODS FOR READ MAPPING

Thesis committee:

Dr. Srinivas Aluru, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Xiuwei Zhang
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Ümit V. Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Heng Li
Department of Biomedical Informatics
Harvard Medical School

Dr. Kostas Konstantinidis
School of Civil and Environmental Engineering
Georgia Institute of Technology

Date approved: June 30, 2022

For my mom and dad, Ying and Shoucai

ACKNOWLEDGMENTS

First and foremost, I owe a lot to my advisor Dr. Srinivas Aluru for his guidance, support, trust, and encouragement over the years. I was lucky to have the freedom to explore my research interests and find research collaborators. I enjoyed in-depth discussions with him on research projects, and I appreciate his sincere suggestions that helped me make decisions. I got everything I needed for my Ph.D. from him!

I would also like to thank my thesis committee members: Dr. Ümit V. Çatalyürek, Dr. Kostas Konstantinidis, Dr. Xiuwei Zhang, and Dr. Heng Li, for their valuable time, tremendous help, and constructive feedback, which made this thesis possible.

I also want to thank my mentors and collaborators I met during my research or industry experiences. I started reading papers on sequence analysis during my undergrad under the supervision of Dr. Weiguo Liu. Thanks to his training, I grew interested in this field and decided to dig deeper. Later I was lucky to know Dr. Tao Jiang and Dr. Tai Guo, from whom I got lots of precious advice. I never imagined that I would have the chance to build a read mapper under the supervision of Dr. Heng Li, the author of the most popular short read mapper. I learned a lot from his humility and passion for research. This internship became more delightful with the members of his group. Especially, I had tons of great discussions with Dr. Haoyu Cheng as we share the same research interests and experiences. I also want to thank Dr. Shirley Liu for her feedback and suggestions, which allowed me to finish the Chromap paper. Also, this work could not have been completed without the help of Dr. Li Song and other members of her group. Besides, I appreciate the suggestions and help from Dr. Kin Fai Au and Haoran Li when I was developing Sigmap, and Dr. Yu Gao when I was working on sequence to graph alignment. I was also lucky to have Dr. Vijayshankar Raman, Jason Chin, and Dr. Yixin Luo as my hosts during my summer internship at Google. Though the internship was virtual due to the pandemic, they made the experience enjoyable for me.

I am so fortunate to have such fantastic lab mates for all the discussions, fun, and friendship. Mainly, I want to thank Dr. Chirag Jain for spending plenty of time with me on many research works and Dr. Patrick Flick for motivating and helping me get my first and only industry internship experience. I am also sure I will miss all the friends I met during my years in Atlanta so severely after I leave this lovely city. Life here would be so dull without the time with all of you. As I can never be patient when I learn things, I am particularly grateful to Alvin and Kevin. They patiently coached me in tennis and skiing, which have become my favorite sports.

Last but not least, I could have never gotten this far without my loving parents. They provided me with the best they have for everything at any time.

TABLE OF CONTENTS

Acknowledgments iv

List of Tables x

List of Figures xii

Summary xv

Chapter 1: Introduction and motivation 1

1.1 Background 1

1.1.1 Short read sequencing for chromatin profiling 2

1.1.2 Nanopore long read targeted sequencing 3

1.1.3 Graphical reference genomes 5

1.2 Objectives and overview 6

Chapter 2: An ultrafast method for short read mapping and preprocessing . . . 9

2.1 Related work 9

2.2 The Chromap Methods 10

2.2.1 Overview of Chromap and improvements to minimap2 10

2.2.2 Index construction and query 12

2.2.3 Adapter removal 12

2.2.4 Candidate generation 13
2.2.5 Candidate cache 14
2.2.6 Candidate supplement 15
2.2.7 Candidate verification 15
2.2.8 Split mapping 15
2.2.9 Deduplication 16
2.2.10 Barcode correction 16
2.3 Results 17
2.3.1 Simulated and sequencing data for evaluation 17
2.3.2 Evaluating performance for simulated and ChIP-seq data 17
2.3.3 Evaluating performance for Hi-C data 20
2.3.4 Evaluating performance for scATAC-seq data 24
2.4 Summary 28

Chapter 3: An efficient method for mapping nanopore raw reads in real time . . 29

3.1 Related work 30
3.2 Methods 30
3.2.1 Indexing 31
3.2.2 Signal pre-processing 34
3.2.3 Seeding 35
3.2.4 Chaining 36
3.2.5 Streaming mapping 37
3.3 Experimental Results 38

3.3.1 Experimental setup 38
3.3.2 Comparison with Uncalled 41
3.4 Summary 47

Chapter 4: An Improved Algorithm for sequence alignment to graph genomes . 48
4.1 Related work 48
4.2 Preliminaries 50
4.3 Methods 50
4.3.1 Linear Gap Penalty 51
4.3.2 Affine Gap Penalty 58
4.4 Summary 60

Chapter 5: Fast sequence to graph alignment using graph wavefront algorithm . 61
5.1 Related work 62
5.2 Methods 63
5.2.1 Sequence to graph alignment algorithm 64
5.2.2 Diagonal recurrence 65
5.2.3 Graph wavefront algorithm 67
5.2.4 Graph wavefront pruning 70
5.2.5 Graph alignment walk traceback 71
5.3 Results 72
5.3.1 Experimental setup 73
5.3.2 Runtime comparison 75
5.3.3 Memory usage 77

5.4 Summary 79

Chapter 6: Validating paired-end read mappings in graph genomes 80

6.1 Related work 81

6.2 Problem Formulation 82

6.3 Related Problems in Graph Theory 83

6.4 An Index-based Polynomial-time Algorithm 85

6.4.1 Exploiting Sparsity in Sequence Graphs 87

6.5 Results 90

6.5.1 Datasets 91

6.5.2 Index construction 93

6.5.3 Querying Performance 94

6.6 Summary 95

Chapter 7: Conclusions 96

Appendices 98

Appendix A: Fast sequence to graph alignment using graph wavefront algorithm 99

References 106

LIST OF TABLES

2.1 The statistics for the ChIP-seq, Hi-C and 10x Genomics scATAC-seq data sets. 17

2.2 The computational cost of different methods on ChIP-Seq data. The pre-processing steps include MAPQ filtering, sorting and deduping. 20

2.3 The normalized mutual information (NMI) and adjusted rand index (ARI) of cell type annotations and cell clusters from MAESTRO on 10K PBMC 10x Genomics scATAC-seq data using Chromap and CellRanger v1.2.0 and v2.0.0. MAESTRO obtained 15, 16, 15 clusters from CellRanger v1.2.0, CellRanger v2.0.0 and Chromap results respectively. 26

2.4 The normalized mutual information (NMI) and adjusted rand index (ARI) of cell type annotations and cell clusters from ArchR on 10K PBMC 10x Genomics scATAC-seq data. ArchR obtained 11, 12, 13 clusters from Cell-Ranger v1.2.0, CellRanger v2.0.0 and Chromap results respectively. 26

2.5 The normalized mutual information (NMI) and adjusted rand index (ARI) of cell clusters from MAESTRO on 10K PBMC 10x Genomics scATAC-seq data using Chromap_bulkdedup and CellRanger v1.2.0 with BWA and Bowtie2 as aligners. MAESTRO obtained 15, 14, 15 clusters from BWA, Bowtie2 and Chromap results respectively. 27

3.1 List of benchmarking data sets. 39

3.2 Performance comparison between Sigmap and Uncalled on yeast genome. . 41

3.3 Performance comparison between Sigmap and Uncalled on green algae genome. Data set D3 was used for the tests. 42

4.1 Comparison of run-time complexity achieved by different algorithms for the sequence to graph alignment problem when changes are allowed in the query sequence alone. 49

5.1 List of benchmark sequence graphs. 73

5.2 Runtimes (in seconds) of the methods to align queries to real sequence graphs. Dijkstra's algorithm and generalized Navarro's algorithm cannot finish in 24 hours when mapping queries to the MHC graphs. The LRC region of HG002.2 was not fully assembled. The GraphAligner heuristic would fragment the alignment into multiple pieces when its runtime is marked with *. 76

5.3 Memory usage (MB) of the methods to align queries to real sequence graphs. Dijkstra's algorithm and generalized Navarro's algorithm cannot finish in 24 hours when mapping queries to the MHC graphs. The LRC region of HG002.2 was not fully assembled. The GraphAligner heuristic would fragment the alignment into multiple pieces when its runtime is marked with *. . 78

6.1 List of 20 *Bacillus anthracis* strains used to build the sequence graphs G5-G7. We used the first strain in G5, the first five strains in G6, and all the 20 strains in G7. 92

6.2 Directed sequence graphs used for evaluation. In these graphs, each vertex is labeled with a DNA nucleotide. Four acyclic graphs are derived from segments of human genome and variant files from the 1000 Genomes Project (Phase 3). Three cyclic graphs are de Bruijn graphs built using whole-genome sequences of *Bacillus anthracis* strains, with k-mer length 25. 92

6.3 Performance measured in terms of wall-clock time and memory-usage for building index matrix using all input graphs and different distance constraints. *nnz* represents number of non-zero elements in the index matrix, to indicate its size. Our implementation uses 4 bytes to store each non-zero of a matrix in memory. 94

6.4 Time to execute a million queries using all the graphs and distance constraints. Each query is a random pair of vertices in the graph. 95

A.1 Runtime (in seconds) of the methods to align queries to the linear sequence graphs. 105

A.2 Memory usage (MB) of the methods to align queries to the linear sequence graphs. 105

LIST OF FIGURES

2.1 Overview of Chromap. a Workflow of Chromap mapping a read-pair R1 and R2. First, their minimizers are extracted and then queried in the candidate cache and the minimizer index. The set of three minimizers of R1 is in the cache and the candidate mapping start positions are returned by the cache. The set of two minimizers in R2 is not in cache. So each of them is searched in the minimizer index and the occurrences of the minimizers are used to derive the candidate mapping positions. Then all the candidates are verified, which results in the final mapping. b Accuracy of methods on the simulated data with different read lengths. c Consensus of read alignments from Chromap, BWA-MEM, and Bowtie2 on bulk ChIP-seq data. d Overlapped peaks called from the alignments reported by different methods on bulk ChIP-seq data. 11

2.2 An example of adapter removal This read pair has forward R1 TTGACTGGACACGA and backward R2 GTCCAGGCAATCGT (reverse-complement ACGATTGCCTGGAC denoted as $R2^T$). Suffix of $R2^T$ can be matched to the prefix of R1 with an overlap size of 10 including 1 mismatch, indicating fragment length is shorter than read length. Therefore, Chromap removes ACGA from R1 and $R2^T$ (TCGT in R2) as parts of the adapter sequences. . 13

2.3 Chromap on the large data set. a. Comparison of Hi-C contact matrices at 25 kb resolution and insulation scores for TADs analysis derived from Chromap and BWA-MEM alignments. b. Cluster annotation and NMI of the PBMC 10x Genomics scATAC-seq data based on the results of Chromap and CellRanger. c. Running time of Chromap and workflows based on BWA-MEM on ChIP-seq, Hi-C, and 10x Genomics scATAC-seq data. 19

2.4 Intersections of peaks called from Accel-Align, STAR and minimap2 alignments respectively with the peaks called from BWA-MEM and Bowtie2 alignments on bulk ChIP-seq data. 20

2.5 Annotations of peaks called by MACS2 using BWA-MEM, Bowtie2, minimap2, STAR, Accel-Align and Chromap alignments on bulk ChIP-seq data. 21

2.6 The number of overlapped peaks generated using MACS2 on BWA-MEM, Bowtie2, Chromap alignments of ChIP-seq replicate 1 and on BWA-MEM alignments of ChIP-seq replicate 2 (denoted as BWA (replicate)). 22

2.7 Contact matrices at 100kb resolution and compartment consistency based on Chromap and BWA-MEM. 24

2.8 Chromatin loops based on Chromap and BWA-MEM. 24

2.9 Average CTCF supports around Chromap-unique and BWA-unique loop anchor site. 25

3.1 Overview of the proposed algorithm. The reference genome is first converted to a sequence of events $e_1^s, e_2^s, \ldots$ (red lines) using the expected current value of each k-mer in the pore model. For simplicity of illustration, we use 2-mers in this example. Now every pair of consecutive events (e_i^s, e_{i+1}^s) is a point in two-dimensional space, thus a spatial index for these points (red triangles) can be created. For visualization purpose we set dimension to 2, but higher dimensions may be used. In the mapping stage, raw signals (blue dots) are first segmented into events $e_1^r, e_2^r, \ldots$ (red lines). Then seeds are selected to query the index with range search and hits on the reference are chained to get the mapping (in the blue rectangle). 32

3.2 Boxplots showing the mapping time distributions of Uncalled and Sigmap on mapping real reads in D2 and D3. Center lines denote the median, box limits are the quartiles and the whiskers extended from the boxes represent 5% and 95% confidence intervals. 43

3.3 Number of chunks processed by Sigmap to correctly map reals read in D2 and D3. Most of the reads were mapped using ≤ 10 chunks. 45

3.4 Index size and mean read mapping time with respect to the maximum number of points allowed in a leaf node of the k-d tree. 46

4.1 An example to illustrate the construction of an alignment graph from a given sequence graph and a query sequence. Multiple colors are used to show weighted edges of different categories in the alignment graph. The red, blue and green edges are weighted as insertion, deletion and substitution costs respectively. 52

4.2 An example to illustrate the construction of an alignment graph for sequence to graph alignment using affine gap penalty. The alignment graph now contains three sub-graphs separated by the gray dash lines. The deletion and insertion weighted edges in the alignment graph for linear gap penalty are shifted to deletion sub-graph and insertion sub-graph, respectively. Their weights are also changed to the gap extension penalty. Besides, more edges are added to connect the sub-graphs with each other. For simplicity, we use the highlighted vertex as an example to illustrate how to open a gap and extend it. The weight of magenta colored edges is the sum of gap open penalty and gap extension penalty, and the weight of the black colored edges is 0. 59

6.1 Visualizing distance constraints while mapping paired-end reads to sequence graphs. 82

6.2 Visualizing non-zero structure of adjacency matrix of a chain graph. We also show how the structure changes after exponentiation. This is useful to count $bitops$ during SpGEMM. 90

SUMMARY

DNA sequencing is the mainstay of biological and medical research. Modern sequencing machines can read millions of DNA fragments, sampling the underlying genomes at high-throughput. Mapping the resulting reads to a reference genome is typically the first step in sequencing data analysis. The problem has many variants as the reads can be short or long with a low or high error rate for different sequencing technologies, and the reference can be a single genome or a graph representation of multiple genomes. Therefore, it is crucial to develop efficient computational methods for these different problem classes. Moreover, continually declining sequencing costs and increasing throughput pose challenges to the previously developed methods and tools that cannot handle the growing volume of sequencing data.

This dissertation seeks to advance the state-of-the-art in the established field of read mapping by proposing more efficient and scalable read mapping methods as well as tackling emerging new problem areas. Specifically, we design ultra-fast methods to map two types of reads: short reads for high-throughput chromatin profiling and nanopore raw reads for targeted sequencing in real-time. In tune with the characteristics of these types of reads, our methods can scale to larger sequencing data sets or map more reads correctly compared with the state-of-the-art mapping software. Furthermore, we propose two algorithms for aligning sequences to graphs, which is the foundation of mapping reads to graph-based reference genomes. One algorithm improves the time complexity of existing sequence to graph alignment algorithms for linear or affine gap penalty. The other algorithm provides good empirical performance in the case of the edit distance metric. Finally, we mathematically formulate the problem of validating paired-end read constraints when mapping sequences to graphs, and propose an exact algorithm that is also fast enough for practical use.

CHAPTER 1
INTRODUCTION AND MOTIVATION

1.1 Background

DNA sequencing is the procedure to determine nucleotide sequences of DNA molecules comprised of adenine (A), cytosine (C), guanine (G), and thymine (T). Reading the DNA sequences was initially a laborious and expensive task. However, it gradually became an automated and low-cost process with the extensive development of DNA sequencing technologies over the last few decades. The significantly dropped sequencing cost and increased sequencing throughput have enabled the growth of the sequencing scale from a few kilobases to the first human genome [1, 2], and even to all vertebrate species [3].

Despite tremendous progress in DNA sequencing technologies, the whole genome cannot be sequenced perfectly from one end to the other. Instead, myriad DNA fragments called reads much shorter than the whole genome are generated, and sequencing errors can also be introduced in the reads during the sequencing process. These reads can be corrected and then assembled to recover the whole genome. However, performing whole-genome sequencing (WGS) and obtaining high-quality genome assemblies is still costly. The reason is that the genome needs to be sequenced with a relatively high depth so that enough reads cover each genome position to overcome the sequencing errors when determining the nucleotide at that position.

Due to the high cost of producing full-length genomes without errors, standard practice leverages existing high-quality assemblies of an organism as the reference genome (e.g., human reference genome GRCh38) to analyze the sequencing data of other individuals from the same species. This type of analysis is called *genome resequencing*, which is a cost-efficient method to discover variants at population level. A classic example of large-scale

genome resequencing is the 1000 Genome Project [4], which has performed low-coverage WGS of a few thousand individuals.

Resequencing data analysis is a distinct task from genome assembly. An essential step in the resequencing data analysis is aligning the sequenced reads to a proper reference genome, which is called *read mapping*. Then the read alignments can be processed to identify genetic variants from the differences between the reads and the reference for later downstream analyses. The read mapping process usually costs more time than other analysis steps, which makes it the bottleneck in many sequencing data processing pipelines.

From a computational perspective, the read mapping problem mainly has two inputs: a set of reads and a reference. The methods to handle various types of reads and references can differ significantly. Currently, multiple sequencing technologies are available, and they generate reads with different lengths and error profiles [5]. Moreover, the single reference genome is currently transitioning to a graph-based model, or a graph genome, which can encode the genetic variants at a population scale and represent a set of genomes [6]. In the following sections, we overview various types of reads, and reference genomes as input data for the read mapping problems addressed in this dissertation.

1.1.1 Short read sequencing for chromatin profiling

Illumina sequencing, which uses a sequencing-by-synthesis approach, is currently the mainstay of short-read sequencing, also named next-generation sequencing (NGS) technology. It can generate short reads ($<$300 bp) with a low error rate ($<$0.1%) at high throughput. For example, NovaSeq 6000, a state-of-the-art Illumina sequencing platform, can generate up to 6,000 Gb data, i.e., around 20 billion reads, in a 44-hour run.

The standard workflow for Illumina short-read sequencing contains several steps. The first step is to prepare a library for sequencing, during which the DNA sample is cut into smaller DNA fragments, followed by ligation with sequencing adapters at the fragment ends. The library is then loaded into the flow cell of the sequencing instrument and ampli-

fied for sequencing. During the sequencing process, either one or both ends of the DNA fragments is sequenced, yielding single-end and paired-end reads, respectively. Finally, the reads are mapped to a reference, and the read alignments are used in various downstream analyses to gain new biological insights.

In eukaryotic cells, DNA is complexed with histones to form nucleosomes, which are the basic unit structures of chromatin. The nucleosomes can be depleted at certain chromatin locations to interact with regulatory elements such as transcription factors to promote or suppress gene expression, which leads to "open" or "accessible" chromatin regions. Chromatin profiling techniques, such as ChIP-seq [7], ATAC-seq [8], and Hi-C [9], have been widely used to study transcription factor binding [10], chromatin accessibility [11], and higher-order chromatin organization [12, 13], respectively. Single-cell ATAC-seq (scATAC-seq) further enables the profiling of cis-regulatory elements in individual cells [14]. These chromatin profiling assays process the chromatin in various ways to extract and enrich certain chromatin regions of interest and perform high-throughput short-read sequencing. After read mapping, peak calling is usually performed to discover the enriched regions with high coverage (or more mapped reads) for other downstream analyses.

1.1.2 Nanopore long read targeted sequencing

Oxford Nanopore Technologies (ONT) sequencers produce millions of long reads with $>$10 Kbp N50 in a single 48 to 72-hour run. These long reads can span repetitive regions of a genome that are hard to resolve using short reads, thus enabling assemblies with high continuity [15]. Direct RNA sequencing through nanopores can sequence full-length RNA transcripts without amplification, which can significantly aid in *de novo* transcriptome analysis [16]. Without additional library preparation, amplification-free nanopore sequencing also enables the detection of nucleotide modifications [17].

Nanopore sequencers work by measuring ionic current as a molecule passes through a

pore. Since different molecules in the pore modulate the current in specific ways, individual nucleotides can be inferred by the base calling of the raw current signal. For various ONT pore versions (e.g., R7, R9), the current signal is mainly affected by five or six nucleotides (i.e., k-mers where $k = 5$ or 6) occupying the pore at a given time point. These current readings usually have a low signal-to-noise ratio, making it hard to identify the corresponding k-mers. To tackle this problem, many base callers have been developed to "translate" the raw signals to nucleotide sequences [18]. State-of-the-art base callers (e.g., ONT official base caller Guppy) can achieve around 90% accuracy. However, base calling is computationally expensive and can last days on a high-end central processing unit (CPU) or hours on a graphical processing unit (GPU), even for a relatively low throughput run with only $\sim$20 Gbp data.

The ONT MinION is a portable device that typically yields up to 30 Gbp sequencing data using a single flow cell at a low cost. Portability of the MinION sequencer allows sequencing to be performed in the field or the clinic, for example, surveillance for Ebola virus in West Africa [19] and fast detection of SARS-CoV-2 with high sensitivity [20]. The MinION device is compatible with recently released Flongle flow cells with even lower prices while reducing the sequencing throughput to $\sim$2 Gbp for smaller analyses and tests. However, this throughput is usually too low for many applications requiring high sequencing depth, making targeted sequencing necessary.

Targeted sequencing allows for enriched coverage of desired genomic regions, which reduces sequencing costs and labor to achieve high coverage at regions of interest. Typical targeted sequencing approaches do not work well with nanopore sequencing due to loss of nucleotide modifications, high input requirements, low throughput, or long protocols [21]. On the other hand, the targeted sequencing protocol designed specifically for nanopore sequencing [21] addressed some of these issues but still requires additional preparation time and is limited by the maximum size and number of targeted regions.

Alternatively, targeted sequencing can be performed with the selective sequencing fea-

ture of ONT sequencers in real time, assisted by computational methods [22, 23, 24]. This is achieved by temporarily reversing the voltage across a nanopore, thereby rejecting an undesired molecule and making the pore available for other molecules. Thus, if there is a sufficiently fast computational method that can identify whether reads come from regions of interest, one can quickly eject undesired reads and leave the pores for reads of interest. As a result, undesired genomic regions are not sampled, and regions of interest are enriched.

1.1.3 Graphical reference genomes

Recent advances in long-read sequencing technologies and assembly methods have enabled the production of human genome assemblies that meet or even exceed the quality of the reference human genome [25, 26, 27]. As the set of available high-quality human genome assemblies grows, it is important to shift from a single reference genome to a new reference genome model that can support the representation and analysis of the variants from a collection of assembled haplotypes [6]. In this transition, pan-genomics has emerged and extended rapidly as a new research subarea of computational biology [28]. The idea currently gaining momentum is to replace the single reference genome with a graphical genome model, or a genome graph, which fully encodes the variations of individual genomes in the population [29, 30]. Multiple recent studies have shown that read mapping accuracy can be improved by using a graph-based genome as the reference [31, 32, 33].

Graph-based representations provide a natural mechanism for compact representation of related sequences and variations among them. Typically, either the graph vertices or the edges are associated with sequence labels so that concatenating the labels of the vertices or edges along a specific walk in the graph spells a genome. Thus, mapping a read to the graph is equivalent to finding a walk in the graph such that the read can be aligned to the sequence represented by the walk, which requires new algorithms different from the

methods for mapping reads to a linear reference genome.

Multiple different graph models are frequently used in bioinformatics (we defer their introduction to Section 4.1). In this dissertation, we use a generic graph model called sequence graph. It is a directed graph, and each vertex is labeled with either a character or a string. We use this graph structure because it provides a good abstraction for solving the alignment problem on various types of graphs used in bioinformatics. Commonly used graphs can be converted into an *equivalent* sequence graph. In the context of solving the alignment problem, equivalence implies that any sequence (i.e., concatenation of vertex labels in a walk) in the first graph exists if and only if it exists in the second graph.

1.2 Objectives and overview

Although numerous efforts have been undertaken on improving methods for read mapping, new read mapping methods are constantly needed for several reasons. First, the volume of sequencing data keeps increasing as the sequencing cost continues to decrease, which allows sequencing at deeper coverage. Thus, more efficient methods are needed to keep computational resources and time spent on read mapping feasible as the sequencing data set grows larger and larger. Besides, different sequencing technologies generate various types of reads, which differ in lengths and error profiles. Method development efforts to handle multiple types of reads are necessary since no universal solution is currently available. Moreover, the transition from a single genome reference to a graph-based genome reference is currently underway. However, the algorithms designed to map reads to a linear reference genome cannot be trivially extended to map reads to the graph genome, making development of new methods a compelling necessity.

In this dissertation, we aim to develop efficient read mapping methods that can process large sequencing data sets within a reasonable time and design algorithms to handle various input types of reads and references mentioned above. The rest of the dissertation is structured as follows.

In Chapter 2, we present Chromap, an ultrafast method for aligning and preprocessing short reads generated by high throughput chromatin profiling assays. Chromap takes advantage of the observation that chromatin profiles are enriched only in a subset of the whole genome to accelerate the read mapping process. Chromap is comparable to the state-of-art tools in mapping accuracy and is over ten times faster than traditional workflows on bulk ChIP-seq/Hi-C and scATAC-seq profiles.

In Chapter 3, we present Sigmap, a new streaming method that can map nanopore raw reads for real-time selective sequencing. Rather than converting read signals to bases, we propose to convert reference genomes to signals and entirely operate in the signal space. We demonstrate the superior performance of Sigmap compared with other tools on both simulated and real ONT long-read sequencing data.

In Chapter 4, we present an improved algorithm for the sequence-to-graph alignment problem. Considering a query sequence of length m and a directed graph $G(V, E)$ with character-labeled vertices, we propose an algorithm that achieves $O(|V| + m|E|)$ time bound for both linear and affine gap penalty cases, superior to the best existing algorithms in terms of time complexity.

In Chapter 5, we propose the graph wavefront algorithm (Gwfa) to find the optimal global sequence to graph alignment with unit edit cost. Besides, we develop heuristics to reduce alignment runtime and present methods to trace an optimal alignment walk in the graph. We construct graphs for biologically important regions rich in polymorphism and demonstrate the advantages of Gwfa empirically in terms of runtime and memory usage compared with other graph alignment methods.

In Chapter 6, we provide the first mathematical formulation of the problem of validating paired-end distance constraints in graphs and propose an exact algorithm to solve it. The proposed algorithm exploits sparsity in sequence graphs to build an index, which can be queried quickly using a simple lookup during the read mapping process. We provide formal arguments to shed light on why our indexing procedure is efficient and demonstrate

the practical performance of our algorithm on various graphs. Finally, in Chapter 7, we summarize the contributions made in this dissertation.

CHAPTER 2
AN ULTRAFAST METHOD FOR SHORT READ MAPPING AND PREPROCESSING

As the sequencing depth of chromatin studies continually grows deeper for sensitive profiling of regulatory elements or chromatin spatial structures, aligning and preprocessing these sequencing data have become the bottleneck for analysis. However, currently widely used read mapping tools were developed in the last decade; thus, they are not fast enough to process the growing volume of sequencing data.

In this chapter, we present an efficient short read mapping method named Chromap. Taking advantage of the characteristics of chromatin profiles, Chromap can achieve an order of magnitude speedup on a variety of chromatin profiling data compared with other state-of-the-art mapping tools without losing accuracy. Besides read mapping, Chromap also incorporates sequencing adapter trimming, duplicate removal, and scATAC-seq barcode correction, further improving the processing efficiency.

The rest of the chapter is organized as follows. In Section 2.1, we reviewed the standard practice for mapping and processing short reads generated by chromatin profiling assays. In Section 2.2, we present the Chromap methods. In Section 2.3, we provide details on benchmarking the methods and show the performance of Chromap together with other tools on processing different types of chromatin profiling data. And in Section 2.3, we summarize our work.

2.1 Related work

Standard analysis workflows, such as those used by the ENCODE project [34], start with read mapping by the popular short read aligner BWA-MEM [35] or Bowtie2 [36], along with alignment sorting and deduplication by SAMtools [37] and Picard [38]. These steps

are the common bottlenecks which may take hours or days to complete, compared to the downstream analysis steps such as peak calling by MACS2 [39] which usually takes minutes. One reason for such inefficiency is that the comprehensive base-level alignment results for the purpose of variant calling are unnecessary for most chromatin biology studies. Furthermore, alignment filtering, deduplication, and other preprocessing steps are handled by different methods sequentially in a standard workflow, and each step requires parsing from compressed files. Such repeated I/O significantly increases the running time.

Minimap2 [40] is an efficient read aligner based on the minimizer sketch [41]. It was initially designed for long reads of high error rate and then extended for short accurate reads. Although a few times faster than FM-index-based short-read aligners such as BWA-MEM and Bowtie2, minimap2 more frequently misses short alignments that lack sufficient minimizer seeds. This becomes a severe issue in mapping scATAC-seq data when a large portion of the read sequence is used for barcoding and indexing, and the remaining genomic sequence in a read can be as short as 50bp. Moreover, minimap2 has to slowly scrutinize the alignment to resolve the high sequencing error rate inherent in the long reads even when in the short-read mode, which could be unnecessary for the highly accurate Illumina short-read sequencing data.

2.2 The Chromap Methods

2.2.1 Overview of Chromap and improvements to minimap2

Though both Chromap and minimap2 build the minimizer index and extract minimizers from sequences as seeds to map the reads, they use distinct algorithms for seeding and for identifying alignment candidates. Minimap2 applies an expensive chaining procedure on the seeds to generate candidate mapping positions and then runs a slow dynamic programming algorithm that supports affine-gap penalty to verify those candidates. This complex procedure was initially designed for long reads and adapted for short reads later. It is overkilling and inefficient for short reads. Chromap, on the other hand, takes advantage of

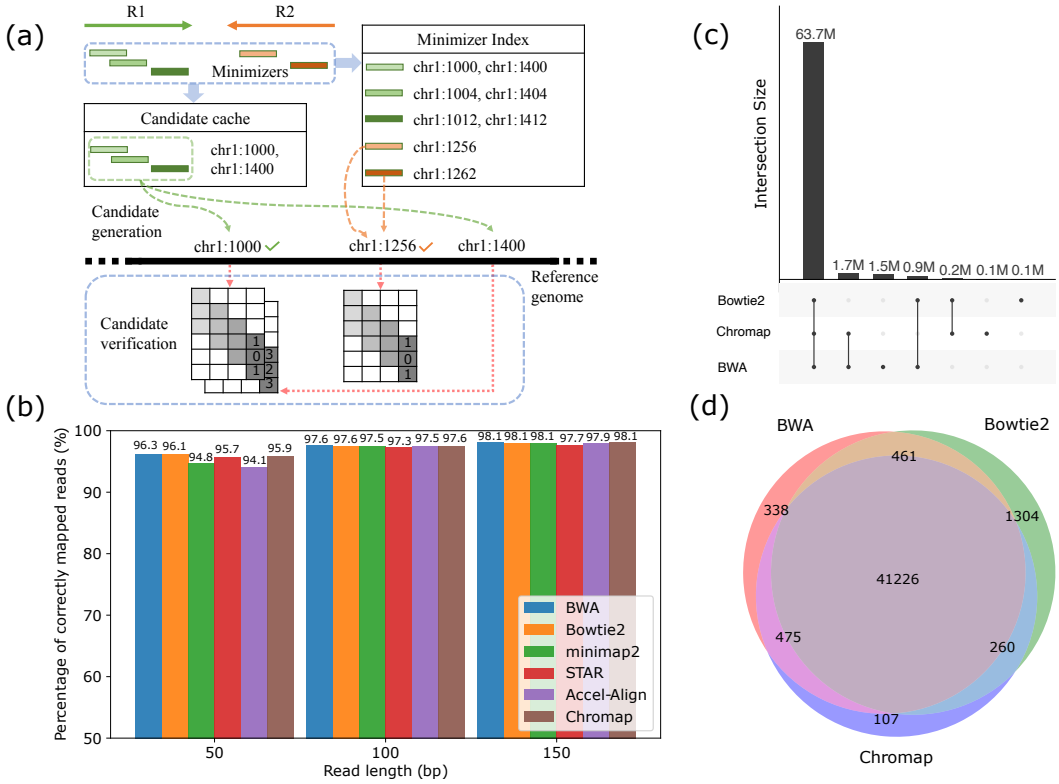

Figure 2.1: Overview of Chromap. a Workflow of Chromap mapping a read-pair R1 and R2. First, their minimizers are extracted and then queried in the candidate cache and the minimizer index. The set of three minimizers of R1 is in the cache and the candidate mapping start positions are returned by the cache. The set of two minimizers in R2 is not in cache. So each of them is searched in the minimizer index and the occurrences of the minimizers are used to derive the candidate mapping positions. Then all the candidates are verified, which results in the final mapping. b Accuracy of methods on the simulated data with different read lengths. c Consensus of read alignments from Chromap, BWA-MEM, and Bowtie2 on bulk ChIP-seq data. d Overlapped peaks called from the alignments reported by different methods on bulk ChIP-seq data.

a light-weight candidate generation method, which is fast and sensitive to find candidate mapping positions for short reads. The candidates are supplemented using the read-pair information to improve mapping accuracy in repetitive regions, which minimap2 lacks. To verify alignment candidates, Chromap uses an efficient method to compute the edit distances of the read and its candidate mapping regions. Note that Chromap is not only an aligner like minimap2 but also an integrated tool that can pre-process the reads to remove adapters and correct the barcodes, and post-process the mappings to remove duplicates. The details of Chromap are described below.

2.2.2 Index construction and query

Double-strand minimizers of reference genomes are collected and indexed using a hash table with minimizer sequences as keys and their sorted order of occurrences along the reference as values (Figure 2.1a). When mapping a read, Chromap retrieves the genome coordinates for each minimizer of the read. Due to the repetitive regions in the reference, some minimizers have high frequency, which can cause false positive mappings and reduce mapping speed significantly. Thus by default, we mask minimizers occurring > 500 times on the reference during query.

2.2.3 Adapter removal

For ATAC-seq or scATAC-seq, when a read contains the adapter sequence at the 3'-end, its fragment length can be shorter than the read length. To remove the adapters, for a pair of reads, if a prefix of one read has ≤ 1 Hamming distance compared with a suffix of the other read in the pair and the overlapped region is longer than a threshold l_{ovp}, we trim the bases outside the overlap (Figure 2.2). We extract $l_{ovp}/2$ long seeds from one read, find the hits of the seeds in the other reads and verify those hits. This algorithm accelerates the trimming step and still guarantees finding overlaps within Hamming distance of 1.

Figure 2.2: An example of adapter removal This read pair has forward R1 TTGACTGGACACGA and backward R2 GTCCAGGCAATCGT (reverse-complement ACGATTGCCTGGAC denoted as R2^T). Suffix of R2^T can be matched to the prefix of R1 with an overlap size of 10 including 1 mismatch, indicating fragment length is shorter than read length. Therefore, Chromap removes ACGA from R1 and R2^T (TCGT in R2) as parts of the adapter sequences.

2.2.4 Candidate generation

We define candidates for a read to be possible mapping start locations on the reference genome, which are estimated by exact minimizer hits (i.e., anchors) between the read and the reference. Formally, an anchor is a pair (x, y) where x denotes the minimizer start position on the reference and y denotes the minimizer start position on the read. Then the candidate can be estimated by this anchor as $x - y$. Co-linear anchors (i.e., chains) are a set of anchors that appear in ascending order in both the read and reference, which can be found by a dynamic programming algorithm [40] in quadratic time with respect to the number of anchors. While this algorithm can robustly identify chains for noisy long reads (> 1000 bp with 5%$\sim$10% error rate), we present a more efficient algorithm that can generate candidates for short reads with a low error rate. We generate candidates using all the anchors and then sort the candidates. During a linear scan on the sorted candidates, we merge the same candidates or candidates that have smaller than error threshold difference generated from multiple anchors. The error threshold is a user-defined parameter that constrains the edit distance between read and the genomic region. By allowing error threshold in candidate merging, Chromap accommodates the insertions and deletions when generating the final candidates for a read. During the merging, Chromap records the multiplicity for each candidate, which is also the number of supporting anchors, and filters the candidates with fewer support than the user-defined threshold. For paired-end reads, chains were

first generated for each end and then filtered by the fragment length constraint.

2.2.5 Candidate cache

Chromap stores the raw candidates in a cache for frequent reads to avoid repeated candidate generation for reads from peak regions. The cache is a hash table, where the key is a vector of minimizers and the value is the vector of candidates generated from the set of minimizers. The minimizers vector stores the M minimizers sequences m_i and the $M-1$ offsets between adjacent minimizers m_i. Chromap uses the function $h(m) = (m_1 + m_M) \mod N$ to quickly map the vector to the $h(m)$-th entry in the hash table of size $N = 2,000,003$. The advantage of this mapping function is that the identical reads from both strands can access the same cached information. Furthermore, reads that are nearby in the genome have a greater likelihood of generating the same minimizer vector, and they can also share the same cache information.

Inspired by the count-min sketch [42], Chromap maintains a small count array of size $N' = 103$ in each cache entry to identify the most frequent minimizer vector from hundreds of different vectors mapped to the same cache entry, namely cache collidings. Chromap uses the function $f(m) = (m_1 \oplus m_M) \mod N'$ to map the vector to the $f(m)$-th entry in the count table by computing the XOR of the minimizer codings, which has the same advantage of ignoring the read strand. Chromap then updates the cache table if and only if the count for the minimizer vector is more than 20% of the total count in the count array and is the dominant minimizer vector (show up more than half times) among the vectors mapped to the count array entry $f(m)$. As a result, Chromap not only stores in cache the candidates from frequent minimizer vectors, but also avoids unnecessary cache updates from the background noises.

2.2.6 Candidate supplement

Chromap supplements the candidates with read-pair information to recover the lost candidates due to the minimizer occurrence limit. For each read end, Chromap will pick its mate's candidate supported by the most number of anchors and use this mate's candidate as the estimation for the read coordinate. As a result, for each minimizer in the read end, instead of extracting all the occurrences on the reference, Chromap applies a binary search in the index entry to only select the occurrences within the range estimated read coordinate determined by the fragment size distribution. Chromap then executes the same candidate generation algorithm to supplement the candidates with the minimizer occurrences from the binary searches.

2.2.7 Candidate verification

Since each read can have multiple candidate mapping positions, we implemented a banded Myers' bit-parallel algorithm [43] to pick the optimal candidate coordinate with minimum edit distance to the reference genome. To further accelerate the verification step, we parallelized the algorithm using SIMD instructions on the CPU to align the read with multiple candidates on the reference simultaneously. We also modified the algorithm to efficiently trace back the alignment so that accurate start and end mapping positions can be obtained.

2.2.8 Split mapping

When the edit distance exceeds the threshold during the candidate verification step, we check if the length of the mapped read is greater than a certain length threshold. If the length of the mapping passes the length filter, the mapping is kept with an estimated mapping score as the mapped read length minus the edit distance. Note that for some of the Hi-C reads, there can be a small region (< 20 bp) which cannot be mapped at the beginning of its 5' end. To resolve this issue, when the mapping length is too short, the first 20 bp of the read is excluded and a second round of mapping of the remaining region is performed.

If a mapping generated in this way passes the length filter, the mapping is then extended backward from its beginning to its maximum exact match. For paired-end data in split-alignment mode, Chromap ignores the constraints from the read-pair, such as the fragment length or strandness.

2.2.9 Deduplication

When the data set is small, all the mappings can be kept in the memory and sorted to remove duplicates. For large data sets or limited memory, we provide a low memory mode. It saves mappings in chunks temporarily on the disk and uses external sort to merge them into the final mapping output in a low memory footprint. For scATAC-seq data, duplicates can be removed at either bulk level or cell level (default) based on the users' choice.

2.2.10 Barcode correction

Using the barcode whitelist provided by 10x Genomics, we correct barcodes that are not on the whitelist. Prior to the correction, the barcodes are converted to their bit representations and the abundance of each barcode is computed efficiently using a hash table. For barcodes outside the whitelist, all whitelisted barcodes within one Hamming distance from the barcode to correct are extracted by a set of efficient bit operations. Using the quality score of the mismatched base and the abundance of these whitelisted barcodes as a priori, we compute the posterior probability of correcting the observed barcode to the whitelisted barcodes. We make the correction if the highest probability of the observed barcode being a real barcode is $\geq 90\%$. The correction step is performed as part of the read mapping process which is in parallel of loading the next batch of reads.

2.3 Results

2.3.1 Simulated and sequencing data for evaluation

In this work, we evaluated Chromap on various data sets including simulated whole genome sequencing data, bulk ChIP-seq data, 10x Genomics scATAC-seq data, and Hi-C data (Table 2.1). One million fragments were simulated from the human reference genome GRCh38 using Mason [44] with average sequencing error rate 0.1% and read lengths 50bp, 100bp, and 150bp. The bulk CTCF ChIP-seq data on the human VCaP cell line were downloaded from ENCODE to test the tools on bulk sequencing data. The 10K PBMC scATAC-seq data set is publicly available from 10x Genomics and used to evaluate the performance of the tools on single cell data. To investigate the impact of alternating BWA-MEM with Chromap on chromatin conformation analysis, we combined the two Hi-C data replicates from a previous study [13].

Table 2.1: The statistics for the ChIP-seq, Hi-C and 10x Genomics scATAC-seq data sets.

Data set	Number of read pairs	Read length
ChIP-seq	37 million	101
Hi-C	1.4 billion	101
10x Genomics scATAC-seq	379 million	50

2.3.2 Evaluating performance for simulated and ChIP-seq data

Tools and parameters

We compared Chromap with five state-of-the-art short read aligners minimap2(v2.17), STAR (v2.7.9a), Accel-Align (GitHub commit code 7217a9f), BWA-MEM (v0.7.17) and Bowtie2 (v2.4.2). STAR is designed to align RNA-seq data which contains spliced alignments across introns, so we used the options "–alignIntronMax 1 –alignEndsType EndToEnd" to forbid spliced alignment. When testing on simulated data, we converted all the alignments into PAF format and used the paftools to calculate the accuracy of alignments. Using the bulk ChIP-seq data, we compared the consensus of alignments and peaks among

the aligners after filtering the alignments with MAPQs less than 30 based on ENCODE protocol (7 for Accel-Align). Accel-Align computes MAPQs in a different way, so we compared the distribution of MAPQs from all the aligners in the simulated data and found MAPQ 7 in Accel-Align was highly similar to MAPQ 30 in other aligners. All the methods were tested in a multiprocessing environment with 8 threads. Accel-Align was tested with option "-x" for the fast alignment-free mode.

Performance on simulated data

We compared Chromap with other chromatin profiling aligners, namely BWA-MEM, Bowtie2, minimap2, STAR17 (no-splicing mode) and Accel-Align [45] on three simulated whole genome sequencing data sets with various read lengths (Figure 2.3b). Except for STAR, the accuracy of these aligners was similar on the 100bp and 150bp paired-end data, about 98% for the five methods. On 50bp paired-end data, BWA-MEM, Bowtie2 and Chromap had similar accuracy of around 96%, while minimap2, STAR and AccelAlign had worse performance at 94.1%$\sim$95.6%. The comparison showed that Chromap achieved comparable alignment accuracy to BWA-MEM and Bowtie2 for a wide range of read lengths.

Performance on real ChIP-seq data

Next, we evaluated Chromap along with other aligners on real ChIP-seq data. On a CTCF ChIP-seq data set from the ENCODE project, we first compared Chromap with BWA-MEM and Bowtie2. Among the 68 million fragments reported by any of the three methods (MAPQ$\geq$ 30), Chromap aligned 3% fewer fragments than BWA-MEM and 1.2% more than Bowtie2, and 99.8% of Chromap alignments were supported by either BWA-MEM or Bowtie2 (Figure 2.1c).

We next investigated the effects of the alignment methods on peaks called by MACS2 and included minimap2, STAR, Accel-Align in the evaluation. Peaks from Chromap align-

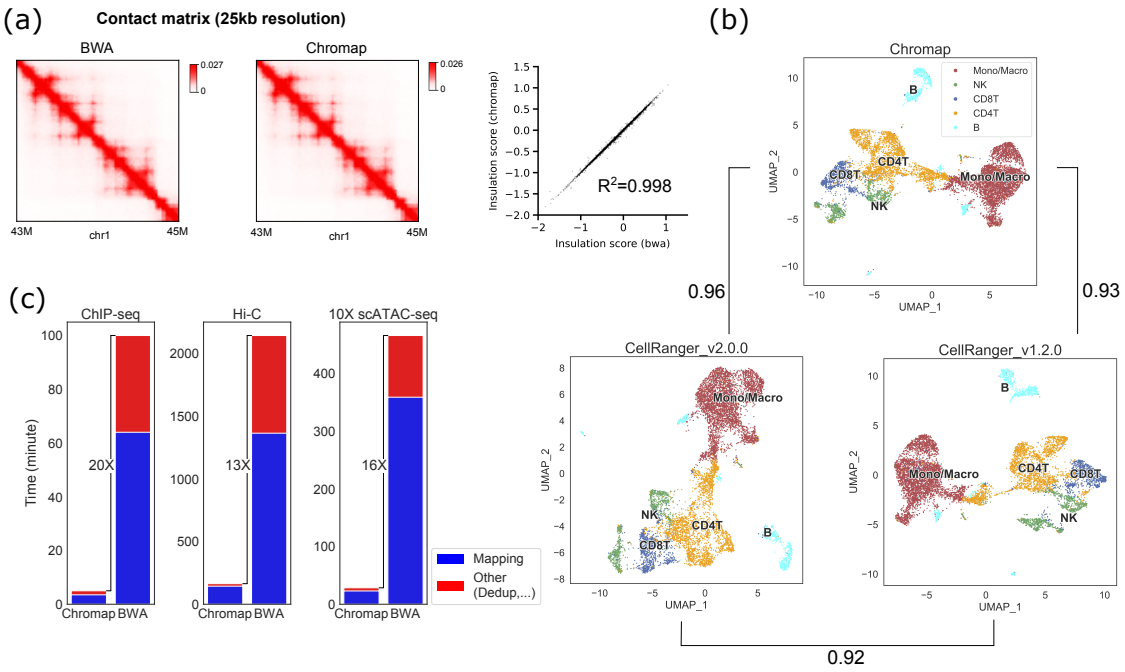

Figure 2.3: Chromap on the large data set. a. Comparison of Hi-C contact matrices at 25 kb resolution and insulation scores for TADs analysis derived from Chromap and BWA-MEM alignments. b. Cluster annotation and NMI of the PBMC 10x Genomics scATAC-seq data based on the results of Chromap and CellRanger. c. Running time of Chromap and workflows based on BWA-MEM on ChIP-seq, Hi-C, and 10x Genomics scATAC-seq data.

ment overlapped 99.8% with those from BWA-MEM and Bowtie2. While Chromap generated a comparable number of peaks as other methods, it created the fewest aligner-unique peaks (Figure 2.1d, Figure 2.4). Annotation of the peaks with ChIPseeker [46] did not find any aligner-specific bias in peaks from the alignment methods (Figure 2.5). In addition, the differences of peak sets from the BWA-MEM, Bowtie2 and Chromap were significantly smaller than those between data replicates (Figure 2.6).

Notably, Chromap only took less than 5 minutes to complete the mapping, sorting, and deduplication process, while the second fastest workflow based on Accel-Align, SAMTools and Picard required about 42 minutes. On the mapping step, Chromap (3.5min) was 75% to 24.5 times faster than other alignment methods, supporting the efficiency improvement of Chromap (Table 2.2). We note that Chromap also reduced half an hour on the sorting and deduplication steps, confirming the advantage of integrating alignment and preprocessing

in chromatin profiling analysis.

Table 2.2: The computational cost of different methods on ChIP-Seq data. The preprocessing steps include MAPQ filtering, sorting and deduping.

Method	Time: alignment (min)	preprossing (min)	Memory (GB)
Chromap	3.5	1.5	18.4
Accel-Align	6	35	19.4
STAR	10	35	29.4
Minimap2	21	35	12.9
BWA-MEM	64	35	7.3
Bowtie2	86	35	3.5

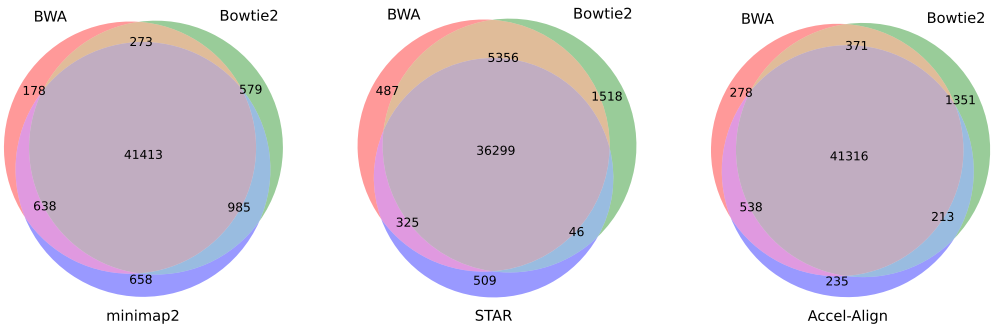

Figure 2.4: Intersections of peaks called from Accel-Align, STAR and minimap2 alignments respectively with the peaks called from BWA-MEM and Bowtie2 alignments on bulk ChIP-seq data.

2.3.3 Evaluating performance for Hi-C data

Chromap supports split-alignment, thus is compatible with Hi-C analysis. We compared Chromap and the standard 4D Nucleome Hi-C processing pipeline, which is based on BWA-MEM and pairtools [47], on a large Hi-C data set for human cell line K562 by evaluating the downstream chromatin features such as chromatin compartments, topologically associating domains (TADs), and chromatin loops. We filtered the alignments with MAPQ 0, which follows the default parameter settings in pairtools. Due to complexity introduced by the ligation junction in a Hi-C experiment, direct comparison of alignment coordinates would underestimate the consistency between the methods. Therefore, we compared the contact maps derived from the alignments at various resolutions. We compared the overall

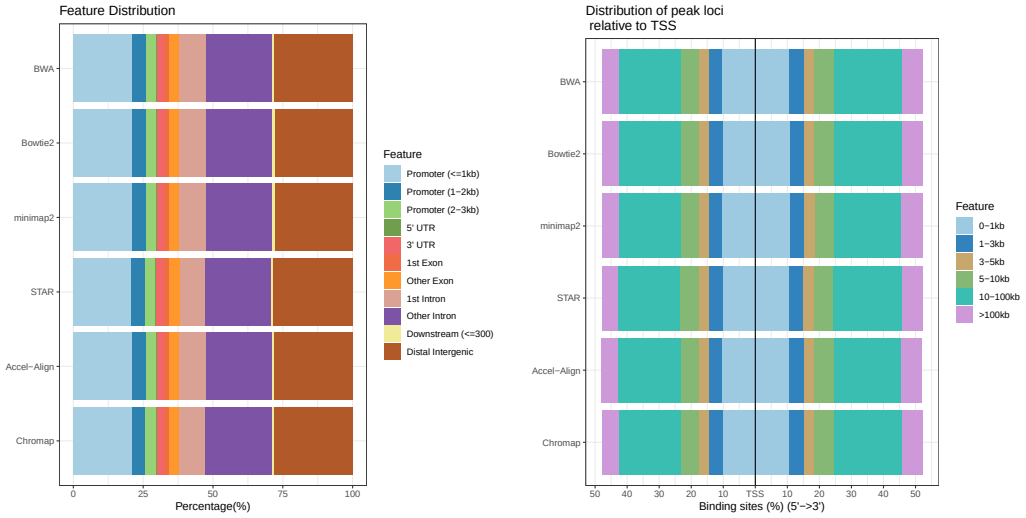

Figure 2.5: Annotations of peaks called by MACS2 using BWA-MEM, Bowtie2, minimap2, STAR, Accel-Align and Chromap alignments on bulk ChIP-seq data.

distribution of chromatin contacts at 25 kb resolution by using the stratum-adjusted correlation coefficients (SCC); the chromatin compartments measured by the first eigenvector of the normalized contact matrices at 100 kb; the TAD boundary strength measured by the insulation score at 25kb resolution; and the identified chromatin loops at 10 kb.

The chromatin compartments (measured by the first eigenvector) and TADs (measured by the insulation score) called from the two aligners gave highly similar results, achieving Pearson correlation coefficients of 0.995 and 0.998 respectively (Figure 2.3a, Figure 2.7). Although there is some divergence on the chromatin loops called by the two aligners, CTCF enrichment at the loop anchors supported these aligner-unique loops as genuine chromatin interaction loops (Figure 2.8, Figure 2.9). On this large data set with about 1.4 billion read fragments, Chromap spent 164 minutes to produce a processed alignment file in the pairs format [47] ready for downstream analysis. It was 13 times faster than a standard workflow with BWA-MEM and pairtools [47].

To confirm that the difference between Chromap alignments and BWA-MEM alignments was smaller than the difference between biological replicates, we computed SCCs by using a Python implementation of HiCRep (https://pypi.org/project/hicreppy/, v0.0.6) [48] between Chromap and BWA-MEM on the same replicate (Chromap R2 vs. BWA-MEM

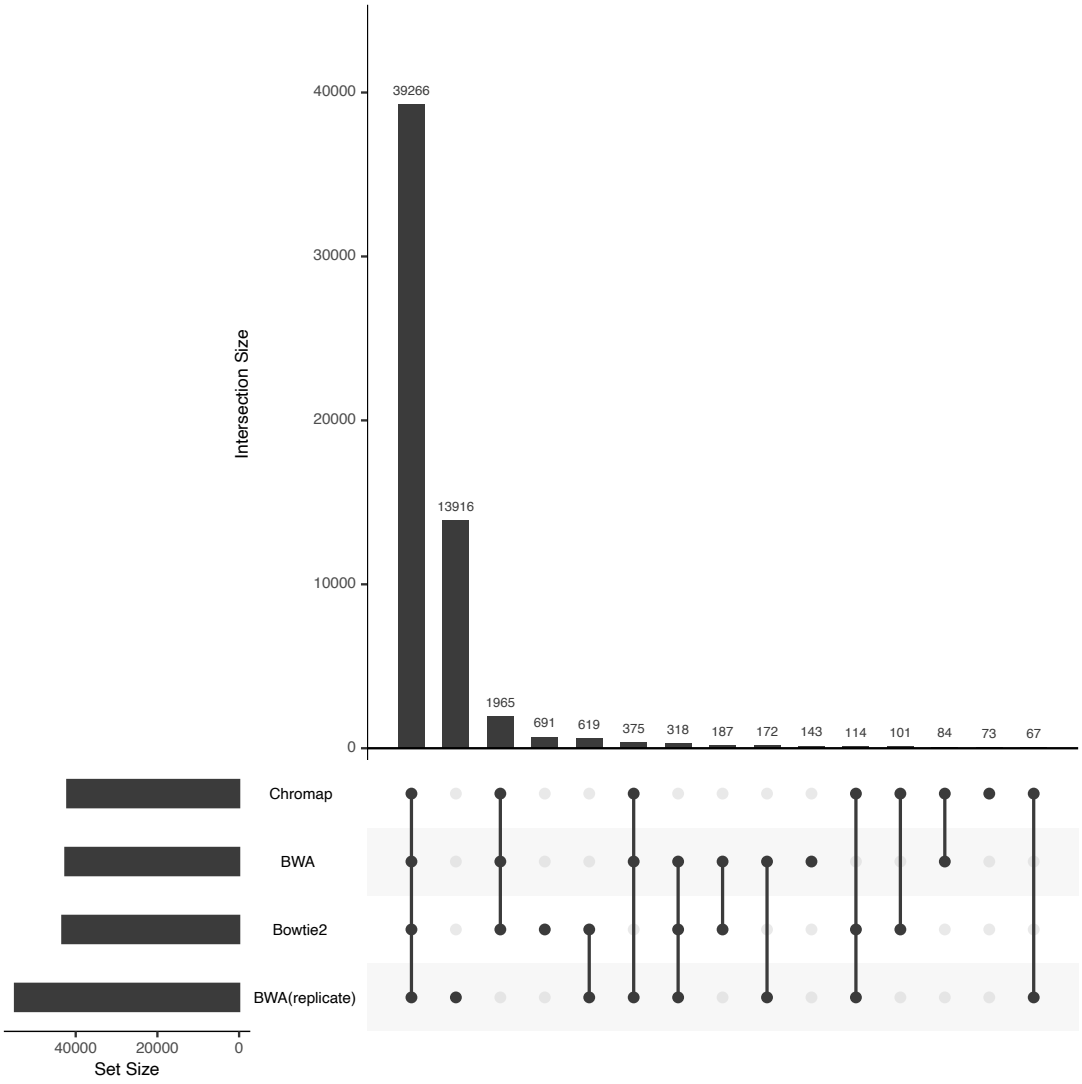

Figure 2.6: The number of overlapped peaks generated using MACS2 on BWA-MEM, Bowtie2, Chromap alignments of ChIP-seq replicate 1 and on BWA-MEM alignments of ChIP-seq replicate 2 (denoted as BWA (replicate)).

R2) or between two replicates (BWA-MEM R1 vs. BWA-MEM R2). Because the original two replicates have disparate sequencing depths (R1: 1,048,612,352 vs. R2: 317,616,493), we first down-sampled R1 to make it match the sequencing depth of R2. The resulting SCC between Chromap R2 and BWA-MEM R2 was 0.998, which was significantly higher than SCC between BWA-MEM R1 and BWA-MEM R2 (0.945). HiCRep was run at 25 kb resolution, and the smoothing factor and the maximum genomic distance were set to 5 and 2 Mb, respectively. For the following chromatin conformation analysis, we merged the alignment results from the two replicates.

Both compartments and TADs were estimated using cooltools (https://pypi.org/project/cooltools/, v0.3.2). For compartments, the eigenvalue decomposition was performed on the 100 kb intra-chromosomal contact maps, and the first eigenvector (PC1) was used to capture the "plaid" contact pattern. The original PC1 was oriented according to a K562 DNase-Seq track (ENCODE accession code: ENCFF338LXW) so that positive values correspond to active genomic regions and negative values correspond to inactive regions. The Pearson correlation of PC1 was 0.995 between Chromap and BWA (Figure 2.7). For TADs, genome-wide insulation scores (IS) were calculated at 25 kb with the window size setting to 1 Mb. The Pearson correlation of the IS scores was 0.998 between the results from Chromap and BWA-MEM (Figure 2.3a). Finally, we identified chromatin loops using HiCCUPS at 10 kb (https://pypi.org/project/hicpeaks/, v0.3.4). Among the 9,455 and 9,950 loops identified from Chromap and BWA-MEM respectively, we found 8,385 of them were supported by both methods (Figure 2.8). Furthermore, we found loop anchor sites that were uniquely identified by Chromap or BWA-MEM had a similar enrichment of CTCF binding peaks (Figure 2.9), suggesting those aligner-specific anchors could be biologically meaningful.

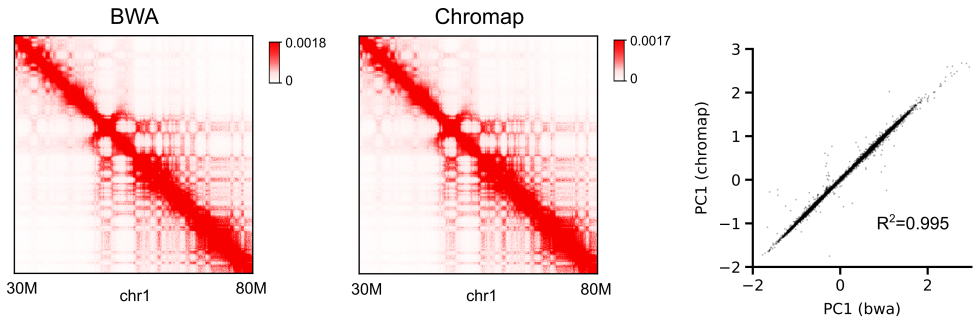

Figure 2.7: Contact matrices at 100kb resolution and compartment consistency based on Chromap and BWA-MEM.

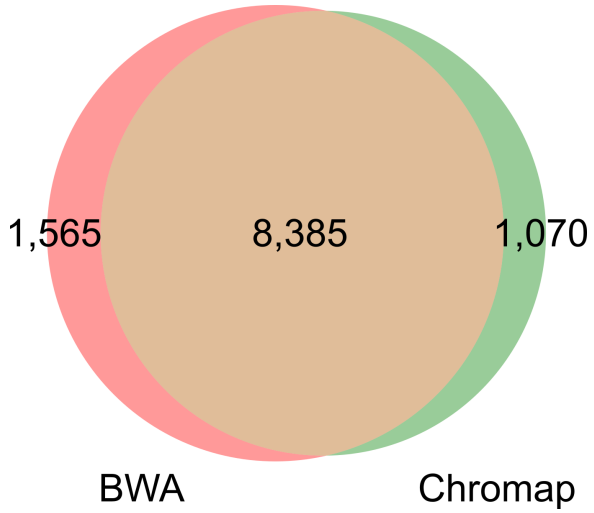

Figure 2.8: Chromatin loops based on Chromap and BWA-MEM.

2.3.4 Evaluating performance for scATAC-seq data

Last but not least, we tested Chromap on a 10K PBMC scATAC-seq data set from 10x Genomics with about 758 million reads and compared the results with CellRanger v1.2.0 and CellRanger v2.0.0, the official pipelines for processing scATAC-seq data developed by 10x Genomics based on BWA-MEM. Released in May 2021, CellRanger v2.0.0 substantially improves the computational efficiency over its predecessor along with other updates in preprocessing steps, such as deduplication criteria. We used all three methods for alignment and preprocessing followed by MAESTRO [49] for cell clustering and cell type annotation (Figure 2.3b). We evaluated the consistency of cell type annotation using normalized mutual information (NMI), and found Chromap and CellRanger v2.0.0 generated nearly iden-

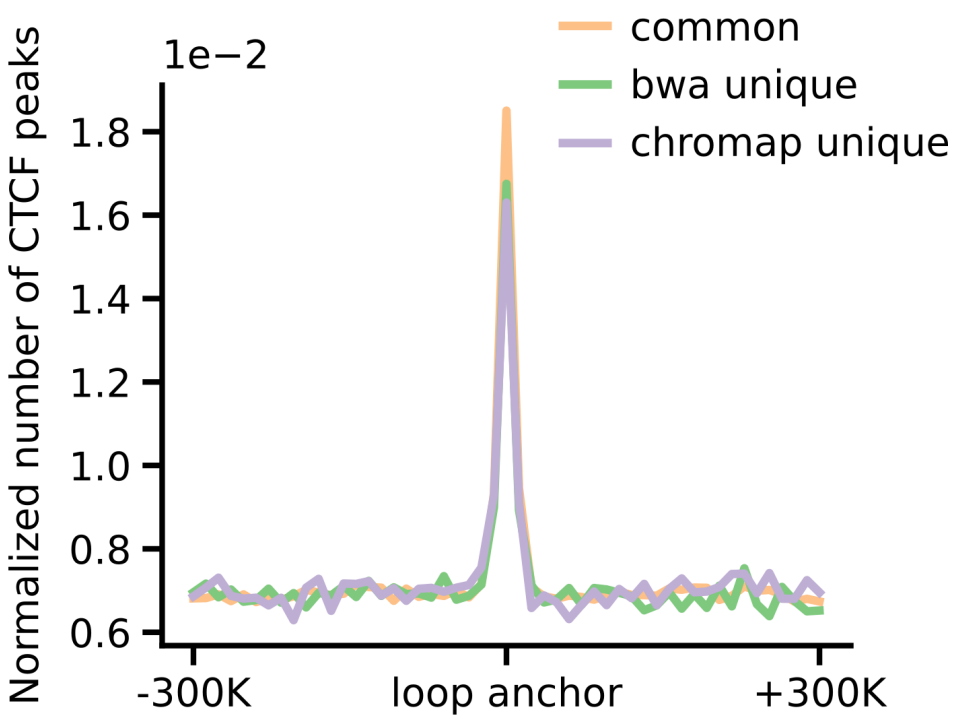

Figure 2.9: Average CTCF supports around Chromap-unique and BWA-unique loop anchor site.

tical results with NMI more than 0.96, higher than the NMI between the two CellRanger versions (Figure 2.3b, Table 2.3). The lower consistency between CellRanger v1.2.0 and v2.0.0 suggested that alternating BWA-MEM and Chromap had less impact on the analysis than changing other preprocessing strategies. The clustering profiles were also highly similar between Chromap and CellRanger v2.0.0, no matter whether clustering was performed using the peak-based approach in MAESTRO or the bin-based approach in ArchR [50] (Table 2.4). On performance, Chromap generated the final alignment file in less than 30 minutes. It was 68 times faster than CellRanger v1.2.0 (33 hours) and 16 times faster than CellRanger v2.0.0 (8 hours). On this data set, Chromap directly obtained the candidates for about 120 million reads from the candidate cache of size 2 million entries which reduced the alignment time by 4%. The memory usage of Chromap is around 21GB, of which the candidate cache consumed about 1.7GB. Since the memory usage is dependent on the index file size, it is stable with respect to sequencing depth and regardless of applications to

ChIP-seq, Hi-C, or scATAC-seq.

Table 2.3: The normalized mutual information (NMI) and adjusted rand index (ARI) of cell type annotations and cell clusters from MAESTRO on 10K PBMC 10x Genomics scATAC-seq data using Chromap and CellRanger v1.2.0 and v2.0.0. MAESTRO obtained 15, 16, 15 clusters from CellRanger v1.2.0, CellRanger v2.0.0 and Chromap results respectively.

Cluster

	CellRanger_v1.2.0 vs CellRanger_v2.0.0	CellRanger_v1.2.0 vs Chromap	CellRanger_v2.0.0 vs Chromap
NMI	0.822	0.832	0.932
ARI	0.677	0.701	0.914

Cell type

	CellRanger_v1.2.0 vs CellRanger_v2.0.0	CellRanger_v1.2.0 vs Chromap	CellRanger_v2.0.0 vs Chromap
NMI	0.922	0.933	0.964
ARI	0.963	0.969	0.983

Table 2.4: The normalized mutual information (NMI) and adjusted rand index (ARI) of cell type annotations and cell clusters from ArchR on 10K PBMC 10x Genomics scATAC-seq data. ArchR obtained 11, 12, 13 clusters from CellRanger v1.2.0, CellRanger v2.0.0 and Chromap results respectively.

	CellRanger_v1.2.0 vs CellRanger_v2.0.0	CellRanger_v1.2.0 vs Chromap	CellRanger_v2.0.0 vs Chromap
NMI	0.865	0.881	0.899
ARI	0.896	0.905	0.928

We conducted comprehensive evaluations between Chromap and CellRanger on the 10K PBMC 10x Genomics scATAC-seq data to show that the clustering results were not affected by replacing CellRanger with Chromap. We compared the consistency of the cell type annotations or cell clusters using normalized mutual information (NMI) and adjusted rand index (ARI) calculated by the Python package scikit-learn. We first computed the baseline NMI and ARI between CellRanger v1.2.0 and CellRanger v2.0.0. Chromap vs CellRanger v2.0.0 achieved a higher consistency score than the baseline score, suggesting the results from Chromap were highly consistent with CellRanger and were more consistent than CellRanger version changes (Table 2.3).

To confirm the difference in the consistency score was insignificant, we created two

replicates of the data set by randomly sampling 95% of the read fragments in the data set and applied CellRanger v2.0.0 to process these two replicates. The cluster-level NMI between the two downsampled replicates (0.888) was lower to the NMI of the clusters generated from CellRanger v2.0.0 and Chromap (0.932), supporting that the impact from alternating CellRanger and Chromap is small.

In addition, we also applied a bin-based scATAC-seq analysis method ArchR on this data set to evaluate the difference in the clustering caused by using Chromap and two CellRanger versions. Similar to the results on MAESTRO, we found alternating CellRanger to Chromap had tiny effects on the clustering results generated by ArchR (Table 2.4). Though CellRanger v1.2.0 is slow, it is easier to modify, and we were able to adapt it to use Bowtie2 as the alignment method (CellRanger v1.2.0_Bt2). Therefore, we could examine the impact of alternating the alignment methods on cell clusters. In this case, we ran Chromap with bulk level deduplication (Chromap_bulkdedup) as the setting in CellRanger v1.2.0. The NMI and ARI scores among CellRanger v1.2.0, CellRanger v1.2.0_Bt2 and Chromap_bulkdedup are all high (NMI> 0.9, ARI> 0.88, Table 2.5), suggesting that alternating the alignment methods BWA-MEM, Bowtie2 and Chromap had little impact on scATAC-seq analysis. CellRanger is a pipeline including data analysis steps after alignment and preprocessing, we measured its running time until the last "WRITE_ATAC_BAM" step in the log file.

Table 2.5: The normalized mutual information (NMI) and adjusted rand index (ARI) of cell clusters from MAESTRO on 10K PBMC 10x Genomics scATAC-seq data using Chromap_bulkdedup and CellRanger v1.2.0 with BWA and Bowtie2 as aligners. MAESTRO obtained 15, 14, 15 clusters from BWA, Bowtie2 and Chromap results respectively.

	CellRanger_v1.2.0 vs CellRanger_v1.2.0_Bt2	CellRanger_v1.2.0 vs Chromap_bulkdedup	CellRanger_v1.2.0_Bt2 vs Chromap_bulkdedup
NMI	0.903	0.918	0.927
ARI	0.884	0.919	0.916

2.4 Summary

In summary, Chromap implements an efficient and accurate alignment and processing method for chromatin profiles. It is significantly faster than general-purpose aligners by taking full advantage of the nature of chromatin studies, i.e., read coordinate locations are more important for downstream analyses (Figure 2.3c). Chromap further improves efficiency by integrating the adapter trimming, alignment deduplication, and barcode correction processing steps in the standard chromatin biology data workflows. With the decreasing cost of high throughput sequencing and increasing deeper sequencing coverage of chromatin profiles, Chromap will continue to expedite biological findings from chromatin studies in the future.

CHAPTER 3

AN EFFICIENT METHOD FOR MAPPING NANOPORE RAW READS IN REAL TIME

ONT sequencers can interact with computational methods to select desired molecules to sequence. This is achieved by processing nanopore raw reads consisting of ionic current readings measured during the sequencing process and ejecting undesired molecules from the pore. To perform selective sequencing on ONT sequencing instruments in real time, efficient computational methods are required to map nanopore raw reads and decide whether they are generated from the targeted genomic regions.

In this chapter, we present a new streaming method that can map nanopore raw signals for real-time selective sequencing. Our method features a new way to index reference genomes using k-d trees, a novel seed selection strategy, and a seed chaining algorithm aware of the current signal characteristics. We implemented the method as a tool Sigmap. Then we evaluated it on both simulated and real data, and compared it to the state-of-the-art nanopore raw signal mapper Uncalled. Our results show that Sigmap yields comparable performance on mapping yeast simulated raw signals, and better mapping accuracy on mapping yeast real raw signals with a 4.4x speedup. Moreover, our method performed well on mapping raw signals to genomes of size $>$100 Mbp and correctly mapped 11.49% more real raw signals of green algae, which leads to a significantly higher F_1-score (0.9354 vs. 0.8660).

The rest of the chapter is organized as follows. In Section 3.1, we introduce the related work and its limitations on processing data for nanopore selective sequencing in real time. In Section 3.2, we provide details on the Sigmap method. In Section 3.3, we evaluate the performance of Sigmap and other tools that can map nanopore raw signal data. Finally, we summarize our work in Section 3.4.

3.1 Related work

Loose *et al.* [51] took advantage of the selective sequencing feature of the MinION sequencer and performed real-time targeted sequencing for amplicon enrichment. In their work, they use dynamic time warping (DTW) to align raw signals to reference genomes to decide whether reads are of interest. Since the time complexity of DTW is quadratic in terms of sequence length, it only works on small genomes that are kilobase pairs long. To address this issue, methods based on base calling followed by read mapping were proposed [22, 23]. However, base callers are not optimized to work on small chunks of reads; thus, they may generate sub-optimal read sequences, which makes mapping challenging [24]. As base calling is a computationally intensive process, enough compute power (e.g., sufficiently powerful GPUs) to achieve real-time base calling may not always be available outside laboratories.

To avoid these drawbacks, Uncalled [24] was developed to map raw signals in real time without base calling. It builds an FM-index [52] for reference genomes, segments the raw signals into events (collapsed current readings for each k-mer), and converts the events into possible k-mers using the ONT pore model. High-probability k-mers are used to query the index and extended. Since raw signals are noisy, Uncalled keeps track of all possible positions of each k-mer as the mapping proceeds. After removing false positive locations by a seed clustering method, the final mapping is reported if one of the locations is sufficiently better than the others. The authors demonstrated successful use of Uncalled on targeted sequencing of small genomes ($<$30 Mbp) and reported that it cannot work properly on mapping raw signals to large genomes that have high repeat content.

3.2 Methods

Seed-and-extend is a widely applied strategy to map erroneous long reads [53, 54, 55, 40]. Typically, exact or approximate word matches between reads and reference genomes are

extracted and then co-linear matches (a sequence of matches that occur in ascending order in both reads and reference genomes) are identified to generate final alignments. Our algorithm also follows the seed-and-extend strategy (see Figure 3.1 for an overview) but is specifically designed to handle noisy raw signal data. Prior to mapping, the reference genome is converted to events and an index of the reference is built once (Section 3.2.1). In the mapping step, raw current signals are first segmented into events and normalized (Section 3.2.2). Then seeds that are less likely to contain segmentation errors are selected from the processed raw signal and used to query the index (Section 3.2.3). After collecting the seed hits (anchors) on the reference, we designed and implemented a chaining algorithm tailored towards the current signal characteristics to find co-linear anchors as chains (Section 3.2.4). The chains are filtered by their scores to ignore sub-optimal mappings. To do real-time selective sequencing, we presented a streaming version of the proposed algorithm (Section 3.2.5). The details of each step are as follows.

3.2.1 Indexing

Different pore models are provided by ONT for various pore versions since current readings are affected by different number of nucleotides occupying the pore at each sequencing time point. In this probabilistic model, current readings for each k-mer are assumed to follow a Gaussian distribution with known parameters. Thus using the pore model, one can estimate the probability of a given event being any of the k-mers, or convert a nucleotide sequence to an event sequence by simply substituting k-mers with their expected current readings.

Uncalled uses the prior strategy to generate high-probability k-mers from read events, while our method leverages the latter to convert the reference to events. Note that in the first case a full iteration on all the distributions is usually required to identify high-probability k-mers that an event may correspond to, which can be slow when many events in the read are processed simultaneously. But converting a k-mer to its expected current reading is a direct translation once a hash table is built for the pore model using k-mers as keys and

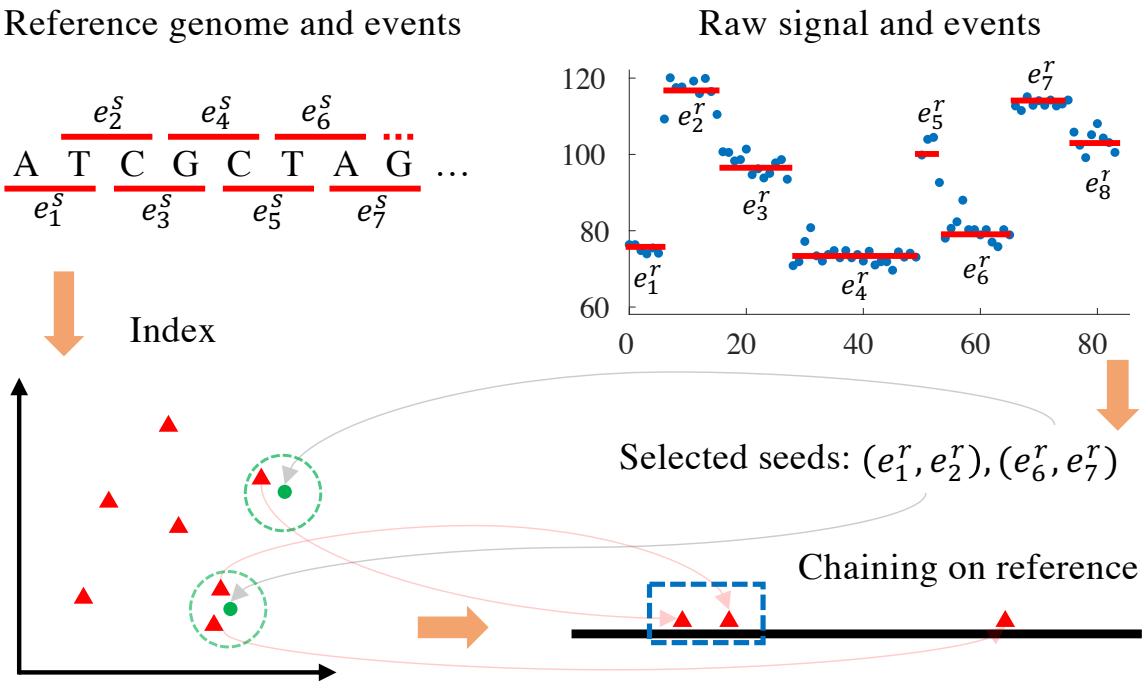

Figure 3.1: Overview of the proposed algorithm. The reference genome is first converted to a sequence of events $e_1^s, e_2^s, \ldots$ (red lines) using the expected current value of each k-mer in the pore model. For simplicity of illustration, we use 2-mers in this example. Now every pair of consecutive events (e_i^s, e_{i+1}^s) is a point in two-dimensional space, thus a spatial index for these points (red triangles) can be created. For visualization purpose we set dimension to 2, but higher dimensions may be used. In the mapping stage, raw signals (blue dots) are first segmented into events $e_1^r, e_2^r, \ldots$ (red lines). Then seeds are selected to query the index with range search and hits on the reference are chained to get the mapping (in the blue rectangle).

expected current as values. Since the conversion is only done once for reference genomes, we can save the overhead of applying pore models to read events to find high-probability k-mers during the mapping stage.

Formally, let $s = s_1 s_2 \dots s_n$ be a nucleotide sequence of length n over alphabet Σ and its corresponding sequence of k-mers be $K(s) = k_1 k_2 \dots k_{n-k+1}$, where $k_i = s_i s_{i+1} \dots s_{i+k-1}$. The pore model is defined as $f : \Sigma^k \to \mathbb{R}$, which gives the expected current corresponding to a k-mer. We create the corresponding event sequence as $E(s) = e_1^s e_2^s \dots e_{n-k+1}^s$, where $e_i^s = f(k_i)$. This is translated to a set of points $P(s) = \{p_i^s = (e_i^s, e_{i+1}^s, \dots, e_{i+d-1}^s), 1 \leq i \leq n-k-d+2\}$ in d-dimensional space. Similarly, for each raw signal sequence r, we generate its events $E(r) = e_1^r e_2^r \dots e_m^r$ (described in Section 3.2.2). The reads are also translated to points $P(r) = \{p_i^r = (e_i^r, e_{i+1}^r, \dots, e_{i+d-1}^r), 1 \leq i \leq m-d+1\}$ in d-dimensional space, some of which are used as seeds in the mapping step. Therefore we need a data structure to organize points of the reference sequence in d-dimensional space so that given a query point p^r of the read, we can efficiently retrieve points $p_{i_1}^s, p_{i_2}^s, \dots$ of the sequence near p^r, i.e., $\left\| p^r - p_{i_j}^s \right\|_2 \leq \epsilon$ where ϵ is the threshold for this range search.

The k-d tree [56] is a data structure designed for partitioning space and organizing points with a binary tree. The leaf nodes of the tree are points while every non-leaf node implicitly divides a subspace into two parts by a hyperplane within that subspace. The points on either side of this hyperplane are associated with the left/right subtrees, respectively. In a balanced k-d tree, the time complexity of range search is $O(dn^{1-\frac{1}{d}})$ in worst case for a fixed range size [57]. But in practice, this typically takes $O(\log n + 2^d)$ time, where logarithmic time is spent in finding the nodes "near" the query point and $O(2^d)$ time is spent to explore their neighborhoods. Therefore we use the k-d tree to organize points generated from the reference to handle large number of queries efficiently during mapping process. Note that construction of the index requires $O(n \log n)$ time when using an $O(n)$ median of medians algorithm [58], and the index only needs to be built once prior to map-

ping. In the implementation, we used the highly-optimized k-d tree package nanoflann (https://github.com/jlblancoc/nanoflann), which supports k-d tree construction and queries.

3.2.2 Signal pre-processing

There are two signal pre-processing steps: signal segmentation and normalization. For R9.4 pore, the DNA molecule transits through the pore with an average speed of 450 bp/s and the electric current is sampled at 4 kHz, which means on average each k-mer has around 8 current samples. The purpose of signal segmentation is to collapse the current readings of the same k-mer into an event. However, speed of the molecule passing through the pore varies significantly. As a result, some k-mers may stay longer in the pore and generate more current readings (*stay errors*) while some k-mers may have no recorded current as the time they reside in the pore is too short (*skip errors*), which makes it hard to segment signals accurately. Moreover, to process signals in real time, we need a fast segmentation method.

Scrappie (https://github.com/nanoporetech/scrappie) is a base caller from ONT, which has a segmentation step prior to fine-grained base calling. It uses t-test over rolling window on the raw signal to detect where the current changes significantly, thereby segmenting the signal. Similar to this method, we also use the Welch's t-test to segment the signal. We choose a fixed window size w and for raw current samples in every two adjacent windows we compute the t-statistics $t = (\bar{x}_1 - \bar{x}_2)/\sqrt{(y_1^2 + y_2^2)/w}$ where $\bar{x}_i$ is the current sample mean and y_i is the current sample standard deviation in the window. Then all the local maxima and minima are identified among the computed t-statistics along the sequence. When a local extremum passes a significance threshold, its position is selected to segment the signal. Due to the various molecule transiting speeds, t-statistics should be computed using multiple window sizes. Local extrema are chosen as segmentation positions using the smallest possible window size if the local extrema reach the significance threshold of that window size. After the signal is segmented, the detected events are normalized to account

for the shift or drift during sequencing.

3.2.3 Seeding

After reference genomes and raw signals are converted into events, the mapping problem is as follows: given read events $E(r)$ and reference events $E(s)$, find consecutive events $E_{i,j}(s) = e_i^s e_{i+1}^s \dots e_j^s$ in $E(s)$ such that $E(r)$ can be aligned to $E_{i,j}(s)$ with high confidence. Note that the mapping can be found by using subsequence dynamic time warping (sDTW) [59]. But the time to compute DTW distance is quadratic in the length of events sequences, which is too slow to compute for long reads in real time. Since the reads are long, though they are erroneous, there are still many subsequences shared in a high confidence mapping region of the read and the reference. Taking advantage of this fact, long read aligners such as minimap2 [40] can efficiently map reads using the seed-and-extend strategy and so does our method.

As the reference points are indexed for fast queries, we can use read points $P(r) = \{p_i^r = (e_i^r, e_{i+1}^r, \dots, e_{i+d-1}^r), 1 \leq i \leq m - d + 1\}$ as the seeds. Note that the number of seeds (or points) needed to query the index is roughly the length of the event sequence. For real-time mapping, the reads have to be mapped within their first few hundreds of base pairs (events). Thankfully, searching for all the seeds can be completed in reasonable time. However, more seeds also lead to more hits on the reference, thereby potentially increasing the time spent in chaining the hits. For organisms like yeast, the number of hits is limited by the small genome size and fewer repetitive regions. But for larger genomes with more repetitive structures, the number of hits can increase significantly, which makes the chaining step time consuming.

To address this problem, one can select seeds with a fixed step size l and only use a subset of all the read points $P(r)$ as seeds, $P_l(r) = \{p_i^r = (e_i^r, e_{i+1}^r, \dots, e_{i+d-1}^r), 1 \leq i \leq m - d + 1, i \mod l = 0\}$. However, raw signals are noisy, which also makes the events erroneous. Simply picking seeds with a fixed step size could miss some "error free" seeds

(query points that have true hits in the index within a certain range) and reduce mapping accuracy. This problem is even more serious when mapping reads in a streaming manner, where the read is supposed to be mapped with only its first few hundreds of base pairs sequenced.

As an alternative, if the quality of the seed can be measured by a score, then error-free seeds can be preferred during seed selection procedure. Formally, we define a scoring function $g : \mathbb{R}^d \to \mathbb{R}$ which computes the score for a given point in d-dimensional space. Note that during sequencing, stay errors happen more frequently than skip errors. Affected by the noise during sequencing, stay errors result in many current samples for the same k-mer with large variance, which leads to over segmentation of the raw signal. If a seed contains stay errors, range search can fail to find true hits of the seed.

We present a method to avoid seeds that are likely to contain stay errors. For a seed (query point) $p_i^r = (e_i^r, e_{i+1}^r, \ldots, e_{i+d-1}^r)$, we define the seed scoring function as $g(p_i^r) = \sum_{j=i+1}^{i+d-1} |e_j^r - e_{j-1}^r|$, which is the sum of the differences between every pair of consecutive events in the seed. Then with step size l, top $\lceil (m-d+1)/l \rceil$ seeds are selected based on their scores. Note that seeds with more abrupt changes in their events are considered better since the segmentation is more reliable in that case.

3.2.4 Chaining

The time for computing an optimal alignment between two sequences is quadratic in the length of the sequences. To avoid this computational bottleneck for aligning long sequences, chaining approaches [40] have been proposed and used to efficiently find mapping positions of long reads in large reference genomes.

Inspired by the chaining method of minimap2, we present a dynamic programming algorithm to identify a set of co-linear anchoring point matches. Formally, each seed hit (anchor) is a triple (u, v, h), which represents a read point p_u^r matching a reference point p_v^s with distance h, i.e., $\|p_u^r - p_v^s\|_2 = h$. Given a list of anchors sorted by their position

on the reference, the best chaining score up to the ith anchor can be computed using the recurrence $D_i = \max\{\max_{1\leq j<i}\{D_j + \alpha_{ji} - \beta_{ji}\}, (1 - h_i/\epsilon)d\}$, where $\alpha_{ji} = (1-h_i/\epsilon)*\min\{u_i - u_j, v_i - v_j, d\}$ is the bonus for the seed hit and β_{ji} is the gap penalty. Let $a_{ji} = |(u_i - u_j) - (v_i - v_j)|$ denote the gap length and $b_{ji} = |(u_i - u_j)/(v_i - v_j)|$ denote the gap scale. The gap penalty β_{ji} is set to ∞ when $v_i < v_j$ (ith anchor is not co-linear with the jth anchor), or gap length a_{ji} or gap scale b_{ji} is too large. Due to stay and skip errors, the gap length and scale are usually unpredictable. Hence, we do not penalize the gap as long as its length and scale are below certain thresholds. Instead, when computing the bonus α_{ji} for seed hits, we scale it down by the factor $(1 - h_i/\epsilon)$.

Note that the time of the chaining algorithm is quadratic in the number of anchors, which is slow. In practice, we use similar heuristics as in minimap2 chaining to reduce the number of anchors to examine. When computing D_i, we start the iteration from $j = i - 1$ and stop when no better chaining score is found after c iterations. For n_a anchors, this heuristic reduces the average time to $O(cn_a)$. The default c is set to the same value used in minimap2 since it led to reasonable speed and accuracy on mapping reads to various genomes empirically. There are theoretically faster chaining algorithms [60] but they are usually not adapted to generic gap functions, or have large hidden constants in their time complexity.

3.2.5 Streaming mapping

In nanopore real-time sequencing, the signal is returned in chunks, and each chunk by default is one second's worth of signal and contains 4,000 current samples or roughly 450 bp. We developed a streaming method to map raw signals by chunks. The signal preprocessing and seeding are performed on each chunk individually. As for chaining, the anchors in the good chains (chaining scores are at least half of the best score) generated using previous chunks are kept and used in the chaining together with the anchors in the current chunk. Each time after a chunk is processed, we compute the ratio between the best chaining score

and the second best chaining score. If the ratio exceeds a certain threshold, we stop mapping more chunks and report the best chain as the mapping. By default, we set this ratio to 1.4. If this ratio cannot exceed this threshold after mapping the first 30 chunks of the read, the mapping process of this read will be stopped and the read will be reported as unmapped. These parameters can be adjusted by users to increase mapping speed or lower false positive rate based on the applications if necessary.

3.3 Experimental Results

We demonstrate empirically the advantages of our method on both simulated and real data sets on two different genomes. The implementation of our proposed method is termed *Sigmap*, which is available at https://github.com/haowenz/sigmap. We compare Sigmap with Uncalled (v2.1).

3.3.1 Experimental setup

Benchmarking data sets

We used one simulated and two real data sets to test the methods. The number of reads, N50 values, genome sizes and average coverage for these data sets are shown in Table 3.1. Simulated raw signals of *Saccharomyces cerevisiae* (yeast) were generated using DeepSimulator [61] with its context-dependent model (-M 0) and sequencing coverage set to 20x (-K 20). For real data sets, 100,000 raw reads were randomly selected from nanopore sequencing of *S. cerevisiae* using ONT R9.4 chemistry (available at NCBI under the study PRJNA510813). The first run of *Chlamydomonas reinhardtii* (green algae) nanopore sequencing using ONT R9.4 chemistry was also used (under study PRJEB31789 on EMBL-EBI) in the evaluation. Note that in real-time targeted sequencing applications, the regions of interest are usually from $\sim$10 Mbp to $\sim$100 Mbp and the coverage of target regions is around 20x [24, 62]. Thus in the evaluation, the yeast and green algae sequencing data were used as their genome sizes are appropriate and their whole genome sequencing data are

subsampled to the proper coverage for real-time targeted sequencing applications. Besides, since Uncalled only supports R9.4 chemistry so far, we used R9.4 data in our evaluation. But with some parameter tuning for both methods, they might also be able to work on R10 data with the R10 pore model (https://github.com/jts/nanopolish/tree/r10/etc/r10-models) trained using Nanopolish [17].

Table 3.1: List of benchmarking data sets.

Data set	Type	Number of reads	N50 (bp)	Reference genome	Genome size (Mbp)	Average coverage
D1	Simulated	30,385	11,984	S. cerevisiae S288c	12.2	20x
D2	Real	100,000	8,348	S. cerevisiae S288c	12.2	58x
D3	Real	63,215	32,025	C. reinhardtii v5.5	111.1	12x

Hardware and software

For all experiments, we used a compute node with dual Intel Xeon Gold 6226 CPU (2.70 GHz) processors equipped with a total of 24 cores and 128GB main memory. We run Sigmap and Uncalled with all the available cores.

The k-d tree index constructed by Sigmap has two important parameters: dimension d and the maximum number of points associated with a leaf node, n_p. The empirical performance of k-d trees is usually good in low-dimensional spaces (e.g., 2D or 3D) but degrades in high-dimensional spaces as more tree branches need to be visited for each query. For this application a low d such as 2 or 3 cannot be chosen, as querying points in low-dimensional spaces usually results in too many hits, which can slow down mapping. Thus we set d to 6 by default. Since the ONT R9.4 pore model lists the expected current reading for each 6-mer, a point in the 6-dimensional space is analogous to an 11-mer, which is also a reasonable k-mer size for read mapping on genomes from tens of Mbp to several hundred Mbp. As for the other parameter, n_p controls the maximum number of points associated with a leaf node (points are stored in leaf nodes of k-d trees). A larger n_p can make the tree smaller but may cause more explorations of points during the search process

and increase the query time. On the other hand, a smaller n_p may reduce the number of points to inspect for a query but increase the tree size. By default, we set n_p to 20 and studied how it can affect memory usage and mapping time on D2. Moreover, to study the effect of seeding step size on mapping time, we evaluated Sigmap with various seeding step sizes l from 2 to 6 on D3 while other parameters are set to the default. We set the maximum amount of chunks to use for mapping a raw signal as 30 and the search radius ϵ to 0.08 by default since they led to proper mapping accuracy and time. These parameters can be adjusted by users according to their data and applications in practice.

To test Uncalled, we used default parameters for indexing reference genomes and mapping raw signals. Kovaka *et al.* [24] showed that masking repeats in genomes improved the mapping speed and accuracy of Uncalled. In the evaluation, we used recommended parameters and procedures stated in the Uncalled's user documentation for *C. reinhardtii* genome repeat masking.

Evaluation criteria

We followed a similar evaluation criteria previously used by Kovaka *et al.* [24]. Raw reads that are mapped to their true mapping locations are true positives (TP). Reads that are mapped by their raw signals but not to the correct locations are false positives (FP). Reads that have true mapping locations but are not mapped by their raw signals are false negatives (FN). Precision equals $TP/(TP + FP)$, recall equals $TP/(TP + FN)$, and F_1-score is calculated by $2 * precision * recall/(precision + recall)$. The percent of correctly mapped reads is the portion of reads that are mapped to their true mapping locations.

For simulated data set D1, we evaluated the mapping accuracy against the ground truth output by the simulator. For real data sets, we mapped the base-called read sequences with the well-established long read aligner minimap2 [40] and used the read alignments as ground truth to validate Sigmap and Uncalled. We excluded reads that are not mapped by minimap2 in the evaluation.

Moreover, we measured the mean mapping time of each read and the number of chunks used to map a read. In practical applications, mapping results are needed in real time to decide whether to eject a pore. Therefore, instead of cumulative mapping time, time spent on individual reads is an important metric to show whether most of the reads can be mapped fast enough for real-time decisions. To accurately measure the mapping time for individual reads, the mapping start time and end time of each read were recorded and the wall time for mapping each read was computed as the difference between these two values and then reported. This way of timing the mapping process for individual reads avoids the effect of loading index or the scalability of multi-thread implementation on measuring mapping time, which is a fair way to compare the two methods.

3.3.2 Comparison with Uncalled

We evaluated the performance of Sigmap and Uncalled on data sets D1-D3. The results on yeast genome are shown in Table 3.2. On the simulated data set D1, Sigmap achieved higher percentage of correctly mapped reads, precision and F_1-score while Uncalled has higher recall and faster speed. Since simulated data might not be as noisy as real data, the events were likely to be detected and converted to corresponding k-mers more reliably, which reduced the number of high-probability k-mers to explore in Uncalled and made it faster. On yeast real data set D2, 93,544 of the 100,000 reads were mapped by minimap2 and used in the evaluation. Sigmap achieved higher percent of correctly mapped reads, precision, recall and F_1-score. Notably, its speed of mapping a raw signal on average was 4.4 times faster than Uncalled.

Table 3.2: Performance comparison between Sigmap and Uncalled on yeast genome.

Data set	Method	Correctly mapped reads (%)	TP	FP	FN	Precision (%)	Recall (%)	F1-score	Mean time per read (ms)
D1	Sigmap	**97.66**	**29675**	**7**	661	**99.98**	97.82	**0.9889**	59
	Uncalled	97.47	29615	722	**47**	97.62	**99.84**	0.9872	**18.3**
D2	Sigmap	**87.54**	**81892**	**964**	**10683**	**98.84**	**88.46**	**0.9336**	**68.3**
	Uncalled	87.37	81725	1054	10765	98.73	88.36	0.9326	303.1

Next, we tested Sigmap and Uncalled on the green algae real data set D3, where minimap2 mapped 60,313 out of 63,215 reads. Table 3.3 shows the evaluation results. We denote Sigmap run with seeding step size 3 by Sigmap (l3), etc. Since the green algae genome is much larger than the yeast genome and has more repetitive regions, genome repeat masking was performed as suggested when using Uncalled to map raw signals. After repeat masking, both mapping accuracy and mean time to map a read improved. But Sigmap significantly outperformed Uncalled with or without repeat masking on the percentage of correctly mapped reads, recall, and F_1-score, while achieving comparable precision. Moreover, compared with Uncalled with and without masking respectively, Sigmap using default parameters was 1.3 and 1.2 times faster on mapping reads, and Sigmap using seeding step size 6 was 2.6 and 2.3 times faster. Though the mapping accuracy of Sigmap degraded when increasing the seeding step size, it was overall better compared to Uncalled. The reason for this observation is that using larger seeding step size reduces the number of picked seeds that go into chaining, which would reduce chaining time and thereby reducing mapping time. But picking fewer seeds also reduced the mapping accuracy since the true mapping location would have fewer supported seeds making it harder to distinguish from other false mapping locations.

Table 3.3: Performance comparison between Sigmap and Uncalled on green algae genome. Data set D3 was used for the tests.

Method	Correctly mapped reads (%)	TP	FP	FN	Precision (%)	Recall (%)	F1-score	Mean time per read (ms)
Sigmap	**87.86**	**52989**	1694	**5628**	96.90	**90.40**	**0.9354**	509.1
Sigmap (l3)	86.21	51998	1973	6338	96.34	89.14	0.9260	373
Sigmap (l4)	83.51	50370	2542	7397	95.20	87.20	0.9102	314.8
Sigmap (l5)	80.69	48669	3107	8532	94.00	85.08	0.8932	279.6
Sigmap (l6)	77.20	46564	3781	9962	92.49	82.38	0.8714	**261.2**
Uncalled	72.18	43534	883	15896	98.01	73.25	0.8384	677
Uncalled (mask)	76.37	46060	**881**	13372	**98.12**	77.50	0.8660	596.5

The mapping time distributions of Uncalled and Sigmap on D2 and D3 are shown in Figure 3.2. We observed that overall Sigmap achieved much shorter mapping time on

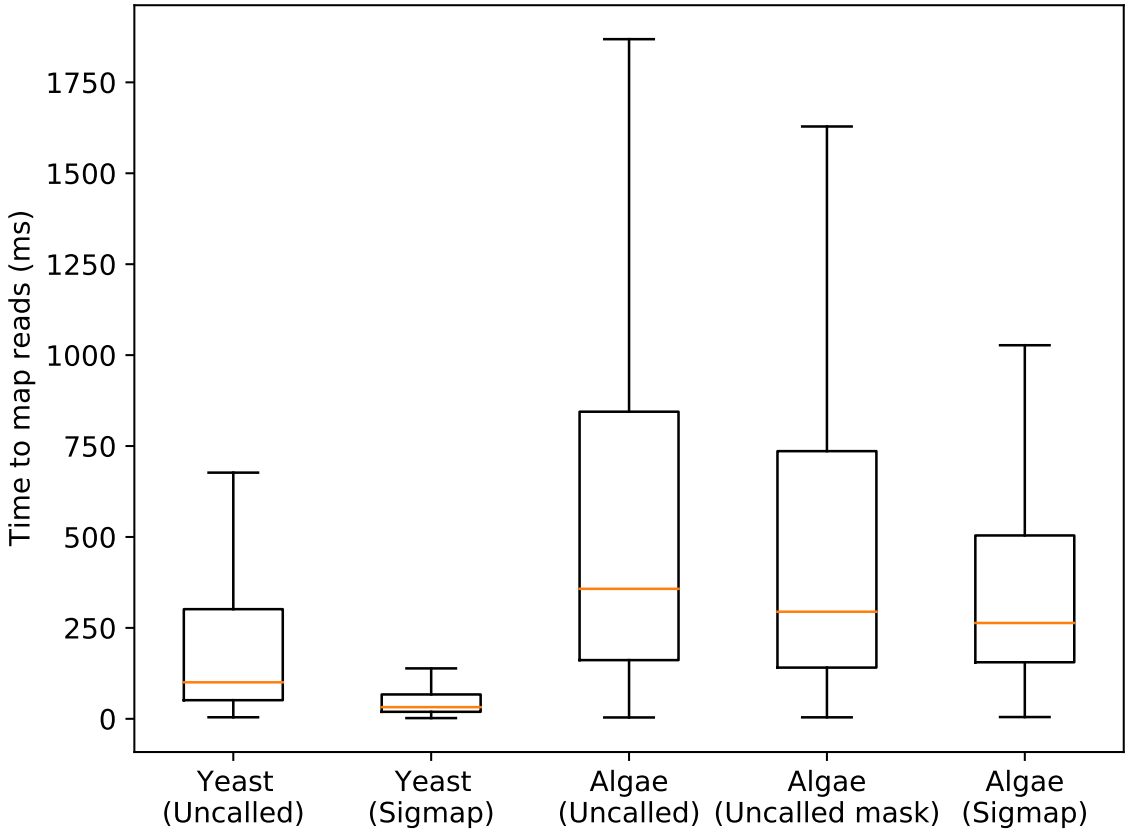

Figure 3.2: Boxplots showing the mapping time distributions of Uncalled and Sigmap on mapping real reads in D2 and D3. Center lines denote the median, box limits are the quartiles and the whiskers extended from the boxes represent 5% and 95% confidence intervals.

mapping yeast real raw reads compared with Uncalled. We noticed the speedup of mapping reads on green algae genome is not as significant as the speedup of mapping yeast reads. One reason is that the size of green algae genome is as around 9 times larger as the size of yeast genome. Given the fact that in practice the time of k-d tree queries is usually logarithmic in the number of points (explained in Section 3.2.1), which is roughly the size of the genome, the query time is supposed to increase accordingly. In addition, the green algae genome has more repetitive regions than the yeast genome and thus the number of signal chunks needed to map algae reads confidently on average is expected to be greater than that to map yeast reads. In the evaluation, we studied the number of chunks needed for Sigmap to map yeast and green algae reads correctly and present the results in Figure 3.3. We observe that using the same number of chunks, a smaller fraction of green algae reads were correctly mapped compared with yeast reads. This also indicates overall more chunks were needed to map green algae reads confidently, which increased the mapping time.

Besides mapping speed, we investigated the index size of Sigmap and Uncalled, which contributes to most of the memory usage in real-time signal mapping. Note that Uncalled mainly relies on an FM-index of the reference sequence which is a compressed full text index, hence expected to be space-efficient. The index size of the yeast genome and the green algae genome built by Uncalled is 21MB and 186MB respectively. Using default parameters, Sigmap built a 417MB index for the yeast genome and a 3.2GB index for the green algae genome, which are larger than the indices built by Uncalled but can still be accommodated on typically used computing systems.

As discussed in Section 3.3.1, increasing the maximum number of points associated with a leaf node, n_p, can trade off mapping speed for smaller index. We studied how mean time to map reads and index size vary with different $n_p = 10, 20, 50, 100, 200$ on D2 and showed the results in Figure 3.4. We observed that the mean mapping time increased and the index size decreased as n_p increased and when $n_p = 200$, the index size can be reduced by a half while the average time to map a read increased by about two times. Similarly, the

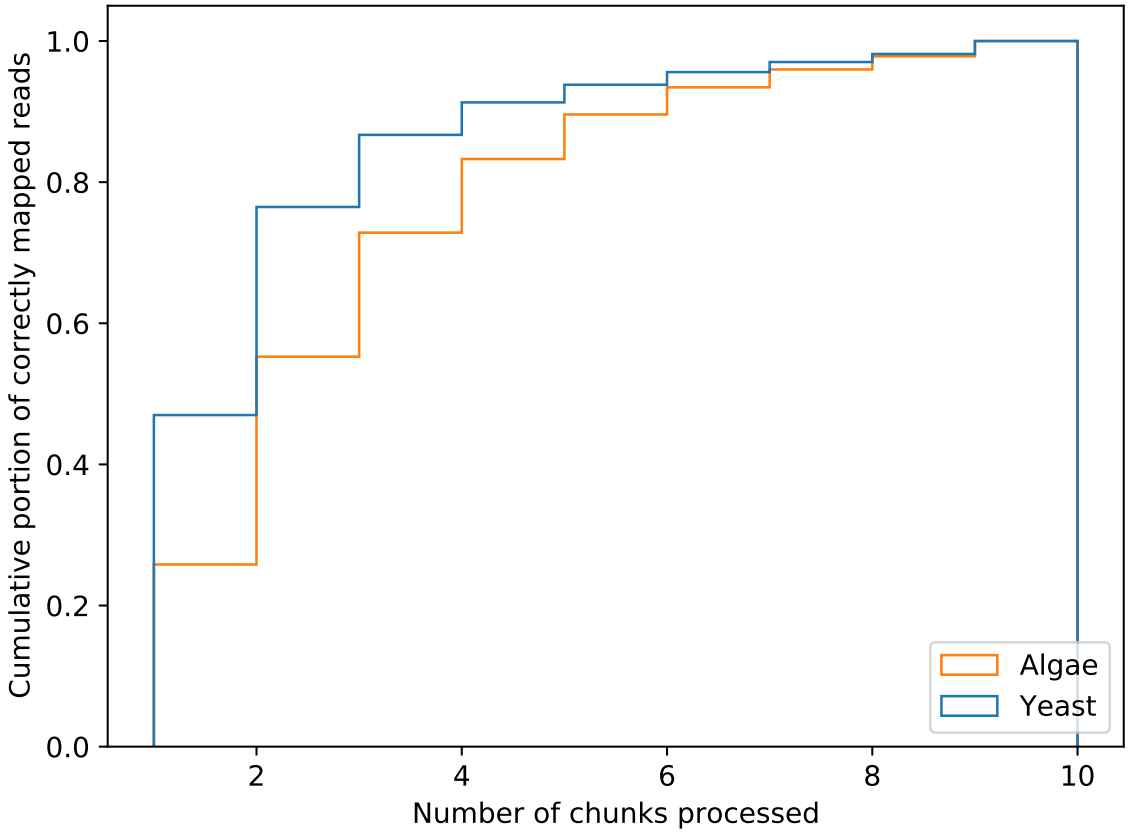

Figure 3.3: Number of chunks processed by Sigmap to correctly map reals read in D2 and D3. Most of the reads were mapped using ≤ 10 chunks.

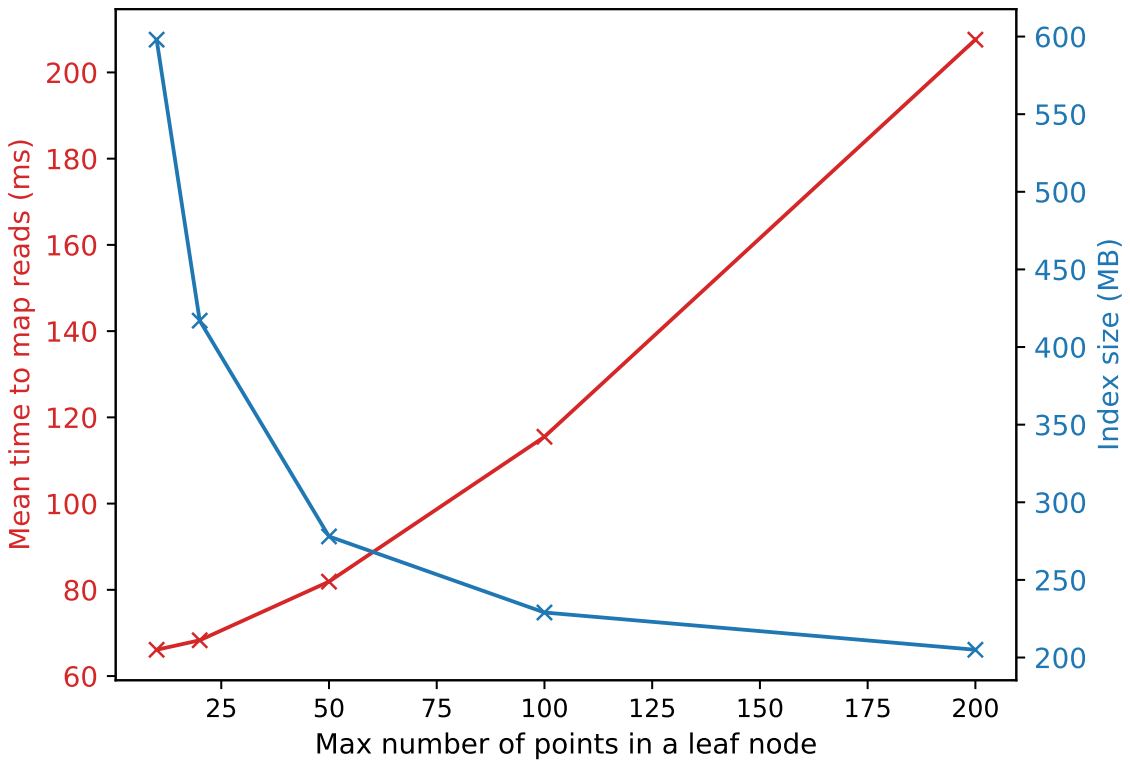

Figure 3.4: Index size and mean read mapping time with respect to the maximum number of points allowed in a leaf node of the k-d tree.

index size of green algae genome can be reduced to 1.8GB when setting $n_p = 200$.

Note that the space complexity of the k-d tree is linear in the number of points and the reason that Sigmap index size is large can be partly attributed to the implementation. Therefore, another possible way to reduce the index size without sacrificing mapping speed is to implement a memory efficient k-d tree customized for this application rather than using a generic k-d tree library, which is a useful direction for future work.

3.4 Summary

Mapping nanopore raw signals in real time is challenging under limited computing resources. Most mapping methods require base calling, which is computationally expensive. Uncalled is an efficient method that does not require base calling, but hits performance limitations on large genomes with higher repeat content. In this chapter, we introduced a new nanopore raw signal mapping method and implemented it as a tool Sigmap. On small genomes like yeast, while Sigmap has comparable performance with Uncalled on mapping simulated data, Sigmap is 4.4$\times$ faster than Uncalled on mapping yeast real raw signals and has the potential to support real-time signal mapping for high-yield run ONT sequencing devices with more pores (e.g., GridION), which previous mapping methods without base calling might not be able to achieve. Sigmap also has good performance on genomes of size $>$100 Mbp such as green algae, where Uncalled could not identify many correct mappings. The method avoids any conversion of signals to sequences and fully works in signal space, which holds promise for completely base-calling-free nanopore sequencing data analysis.

CHAPTER 4
AN IMPROVED ALGORITHM FOR SEQUENCE ALIGNMENT TO GRAPH GENOMES

As graph-based reference genomes are gaining momentum, new methods are needed to map reads to graph genomes. Decades of progress toward designing provably good algorithms for the classic sequence to sequence alignment problems serves as the foundation for mapping tools currently used in genomics. Similar efforts are necessary for sequence to graph alignment.

In this chapter, we present an improved algorithm for the sequence to graph alignment problem. Specifically, we propose an algorithm that achieves $O(|V| + m|E|)$ time bound to align a query sequence of length m to a directed graph $G(V, E)$ with character-labeled vertices for both linear and affine gap penalty cases, which is superior to the best existing algorithms. A vital attribute of the proposed algorithm is that it achieves the same time and space complexity as required for the easier problem of sequence alignment to acyclic graphs [63, 64], under both scoring models.

The rest of the chapter is organized as follows. In Section 4.1, we provide an overview of the related work on sequence to graph alignment. In Section 4.2 and Section 4.3, we formulate the sequence to graph alignment problem variants and design an algorithm to solve them. Then we summarize our work in Section 4.4.

4.1 Related work

Aligning sequences to graphs is becoming increasingly important in the context of several applications in computational biology, including variant calling [65, 66, 67, 32], genome assembly [68, 69, 70], long read error-correction [71, 72, 73], RNA-seq data analysis [74, 75], and more recently, antimicrobial resistance profiling [76]. Much of this has been driven

by the growing ease and ubiquity of sequencing at personal, population, and environmental-scale, leading to significant growth in availability of datasets. Graph based representations provide a natural mechanism for compact representation of related sequences and variations among them. Some of the most useful graph based data structures are de-Bruijn graphs [77], variation graphs [78], string graphs [79], and partial order graphs [80].

To address the growing list of biological applications that require aligning sequences to a graph, several heuristics [81, 82, 83, 84, 32] and a few provably good algorithms [85, 86, 87] have been developed in recent years. In addition, sequence to graph alignment has been studied much earlier in the string literature through its counterpart, approximate pattern matching to hypertext [63]. Since then, important complexity results and algorithms have been obtained for different variants of this problem [88, 64, 89]. Many versions of the classic sequence to sequence alignment problem were considered in the literature, e.g., different alignment modes – local/global, scoring functions – linear/affine/arbitrary gap penalty, and so on [90]. The list further proliferates when considering a graph-based reference. This is because the nature of the problem changes depending on whether the input graphs are cyclic or acyclic [64]. Table 4.1 provides an overview of these sequence to graph alignment methods and our algorithm.

	Linear gap penalty		Affine gap penalty
	Edit distance	Arbitrary costs	
Amir *et al.* [88]	$O(m(\|V\|\log\|V\|+\|E\|))$	$O(m(\|V\|\log\|V\|+\|E\|))$	-
Navarro [64]	$O(m(\|V\|+\|E\|))$	-	-
HybridSpades [68]	$O(m(\|V\|\log(m\|V\|)+\|E\|))$	$O(m(\|V\|\log(m\|V\|)+\|E\|))$	-
V-ALIGN [87]	$O(m\|V\|\|E\|)$	$O(m\|V\|\|E\|)$	$O(m\|V\|\|E\|)$
Rautiainen and Marschall [86]	$O(\|V\|+m\|E\|)$	$O(m(\|V\|\log\|V\|+\|E\|))$	$O(m(\|V\|\log\|V\|+\|E\|))$
This work	$O(\|V\|+m\|E\|)$	$O(\|V\|+m\|E\|)$	$O(\|V\|+m\|E\|)$

Table 4.1: Comparison of run-time complexity achieved by different algorithms for the sequence to graph alignment problem when changes are allowed in the query sequence alone.

4.2 Preliminaries

Let Σ denote an alphabet, and x and y be two strings over Σ. We use $x[i]$ to denote the i^{th} character of x, and $|x|$ to denote its length. Let $x[i,j]$ ($1 \leq i \leq j \leq |x|$) denote $x[i]x[i+1]\ldots x[j]$, the substring of x beginning at the i^{th} position and ending at the j^{th} position. Concatenation of x and y is denoted as xy.

Definition 4.1. *Char-labeled Sequence Graph: A char-labeled sequence graph $G(V, E, \sigma)$ is a directed graph with vertices V and edges E. Function $\sigma : V \rightarrow \Sigma$ labels each vertex $v \in V$ with $\sigma(v)$, one character in the alphabet Σ.*

Naturally, walk $w = v_i, v_{i+1}, \ldots, v_j$ in $G(V, E, \sigma)$ spells the sequence $\sigma(v_i)$ $\sigma(v_{i+1})\ldots\sigma(v_j)$. Given a query sequence q, we seek its best matching walk sequence in the graph. Alignment problems are formulated such that the distance between the computed walk and the query sequence is minimized. Typically, an alignment is scored using either a linear or an affine gap penalty function. The cost of a gap is proportional to its length, when using a linear gap penalty function. An affine gap penalty function imposes an additional constant cost to initiate a gap.

4.3 Methods

The sequence to graph alignment problem is polynomially solvable when changes are allowed on the query sequence alone [88, 64]. Here, we improve upon the state-of-the-art by presenting an algorithm with $O(|V| + m|E|)$ run-time. Our algorithm matches the run-time complexity achieved previously by Rautiainen and Marschall [86] for edit distance, while improving that for linear and affine gap penalty functions. In addition, it is simpler to implement because it only uses elementary queue data structures. Note that edit distance is a special case of linear gap penalty when cost per unit length of the gap is 1, and substitution penalty is also 1. We first present our algorithm for the case of a linear gap penalty function, and subsequently show its generalization to affine gap penalty.

4.3.1 Linear Gap Penalty

Alignment Graph

In the literature on the classic sequence to sequence alignment problem, the problem is either formulated as a dynamic programming problem or an equivalent graph shortest-path problem in an appropriately constructed edge-weighted *edit graph* or *alignment graph* [91]. However, formulating the sequence to graph alignment problem as a dynamic programming recursion, while easy for directed acyclic graphs through the use of topological ordering, is difficult for general graphs due to the possibility of cycles. As it turns out, formulation as a shortest-path problem in an alignment graph is still rather convenient, even for graphs with cycles [88, 86]. The alignment graph, described below, is constructed using the given query sequence, the sequence graph and the scoring parameters.

The alignment graph is a weighted directed graph which is constructed such that each valid alignment of the query sequence to the sequence graph corresponds to a path from source vertex s to sink vertex t in the alignment graph, and vice versa (Figure 4.1). The alignment cost is equal to the corresponding path distance from the source to the sink. Note that the alignment graph is a multi-layer graph containing m 'copies' of the sequence graph, one in each layer. A column of dummy vertices is required in addition to accommodate the possibility of deleting a prefix of the query sequence. Edges that emanate from a vertex are equivalent to the choices available while solving the alignment problem. A formal definition of the alignment graph follows:

Definition 4.2. *Alignment graph: Given a query sequence q, a sequence graph $G(V, E, \sigma)$, linear gap penalty parameters $\Delta_{del}, \Delta_{ins}$, and a substitution cost parameter Δ_{sub}, the corresponding alignment graph is a weighted directed graph $G_a(V_a, E_a, \omega_a)$, where $V_a = \big(\{1, \ldots, m\} \times (V \cup \{\delta\})\big) \cup \{s, t\}$ is the vertex set, and $\omega_a : E_a \to \mathbb{R}_{\geq 0}$ is the weight*

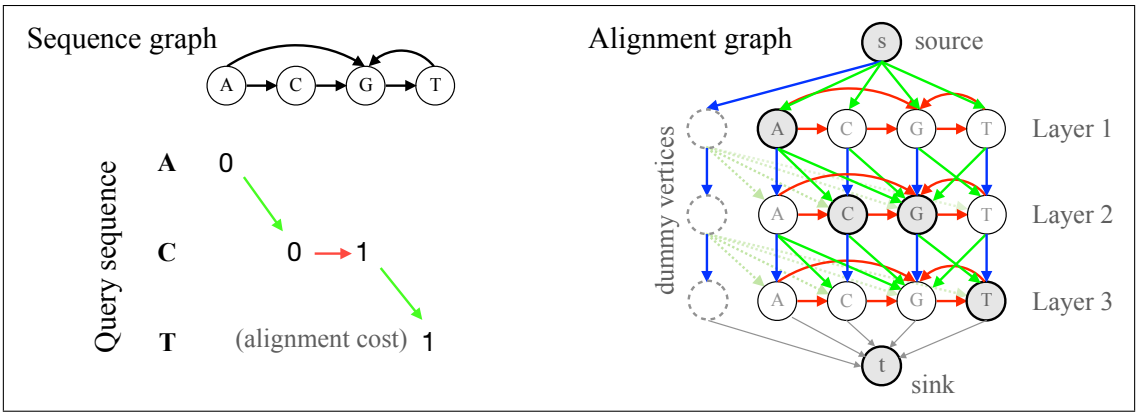

Figure 4.1: An example to illustrate the construction of an alignment graph from a given sequence graph and a query sequence. Multiple colors are used to show weighted edges of different categories in the alignment graph. The red, blue and green edges are weighted as insertion, deletion and substitution costs respectively.

function defined as

$$\omega_a(x,y) = \begin{cases} \Delta_{i,v} & x=(i-1,u), y=(i,v) \quad 1<i\leq m \ \ \& \ \ (u,v)\in E \\ \Delta_{ins} & x=(i,u), y=(i,v) \quad 1\leq i\leq m \ \ \& \ \ (u,v)\in E \\ \Delta_{del} & x=(i-1,v), y=(i,v) \quad 1<i\leq m \ \ \& \ \ (v,v)\notin E \\ \min(\Delta_{del},\Delta_{i,v}) & x=(i-1,v), y=(i,v) \quad 1<i\leq m \ \ \& \ \ (v,v)\in E \\ \textit{for source and sink vertices:} & \\ \Delta_{1,v} & x=s, y=(1,v) \quad v\in V \\ \Delta_{del} & x=s, y=(1,\delta) \\ 0 & x=(m,v), y=t \quad v\in V\cup\{\delta\} \\ \textit{for dummy vertices:} & \\ \Delta_{del} & x=(i-1,\delta), y=(i,\delta) \quad 1<i\leq m \\ \Delta_{i,v} & x=(i-1,\delta), y=(i,v) \quad 1<i\leq m \ \ \& \ \ v\in V \end{cases}$$

Edges $(x,y)\in E_a$ are defined implicitly, as those pairs (x,y) for which ω_a is defined above. $\Delta_{i,v}=\Delta_{sub}$ *if* $q[i]\neq\sigma(v), v\in V$, *and* 0 *otherwise.* Δ_{sub} *denotes the cost of substituting*

$q[i]$ *with* $\sigma(v)$.

Existing definitions of the alignment graph [88, 86] did not include the dummy vertices, and were incomplete. Using the alignment graph, we reformulate the problem of computing an optimal alignment to finding the shortest path in the alignment graph. Even though the alignment graph defined by Amir *et al.* [88] has minor differences, proof in their work can be easily adapted to state the following claim:

Lemma 4.1 (Amir *et al.* [88])**.** *Shortest distance from the source vertex s to the sink vertex t in the alignment graph $G_a(V_a, E_a, \omega_a)$ equals cost of optimal alignment between the query q and the sequence graph $G(V, E, \sigma)$.*

One way of solving the above shortest path problem is to directly apply Dijkstra's algorithm [88, 68]. However, it results in an $O\big(m|V|\log(m|V|) + m|E|\big)$ time algorithm. We next show how to solve the problem in $O(|V| + m|E|)$ time.

Proposed Algorithm

While searching for a shortest path from the source to the sink vertex, we compute the shortest distances from the source to intermediate vertices $V_a \backslash \{s, t\}$ in the alignment graph. An edge from a vertex in layer i is either directed to a vertex in the same layer or to a vertex in the next layer. As a result, the shortest distances to nodes in a layer can be computed once the distances for the previous layer are known. This also makes it feasible to solve for the layers 1 to m, one by one [64]. We use a two-stage strategy to achieve linear $O(|V| + |E|)$ run-time per layer. Before describing the details, we give an outline of the algorithm and its two stages.

Any path from the source vertex to a vertex v in a layer must extend a path ending in the previous layer using either a deletion or a substitution cost weighted edge. Afterwards, a path that ends in the same layer but not at v can be further extended to v using the insertion cost weighted edges if it results in the shortest path to the source. Roughly speaking, the

first stage executes the former task, while the second takes care of the latter. The two stages together are invoked m times during the algorithm until the optimal distances are known for the last layer (Algorithm 4.1). Input to the first stage *InitializeDistance* is an array of the shortest distances of the vertices in previous layer sorted in non-decreasing order. This stage computes the 'tentative' distances of all vertices in the current layer because it ignores the insertion cost weighted edges during the computation. It outputs the sorted tentative distances as an input to the second stage *PropagateInsertion*. The PropagateInsertion stage returns the optimal distances of all vertices in the current layer while maintaining the sorted order for a subsequent iteration.

Algorithm 4.1: Algorithm for sequence to graph alignment

Result: The length of shortest path from s to t

```
1 PreviousLayer = [s];
2 s.distance = 0;
3 for i = 1 to m do        /* Do the computation layer by layer */
4 |   CurrentLayer = [(i, v_1), (i, v_2), ..., (i, v_n), (i, k)];
5 |   x.distance = ∞ ∀x ∈ CurrentLayer;
6 |   InitializeDistance(PreviousLayer,CurrentLayer);
7 |   PropagateInsertion(CurrentLayer);
8 |   PreviousLayer = CurrentLayer;
9 return Min(PreviousLayer.distance);
```

The following are two important aspects of our algorithm. First, we are able to maintain the sorted order of vertices by spending $O(|V|)$ time per layer during the first stage (Lemma 4.2). Secondly, we propagate insertion costs through the edges in $O(|V| + |E|)$ time per layer during the second stage by eluding the need for standard priority queue implementations (Lemmas 4.3-4.5). Both of these features exploit characteristics specific to the alignment graphs.

The InitializeDistance stage We compute tentative distances for each vertex in the current layer by using shortest distances computed for the previous layer (Algorithm 4.2). Because all deletion and substitution cost weighted edges are directed from the previous

Algorithm 4.2: Algorithm to initialize and sort layer before insertion propagation

Result: A sorted layer $CurrentLayer$ with distances initialized using $PreviousLayer$

```
1 Function InitializeDistance(PreviousLayer, CurrentLayer)
2     foreach x ∈ PreviousLayer do
3         foreach y ∈ x.neighbor & y ∈ CurrentLayer do
4             if y.distance > x.distance + ω_a(x, y) then
5                 y.distance = x.distance + ω_a(x, y);
6     Sort(CurrentLayer);
```

layer towards the current, this only requires a straightforward linear $O(|V| + |E|)$ time traversal (line 2-line 5). In addition, we are required to maintain the current layer as per sorted order of distances. Note that vertices in the previous layer are already available in sorted order of their shortest distances from s. A vertex v in the previous layer can assign only three possible distance values ($v.distance, v.distance + \Delta_{sub}$, or $v.distance + \Delta_{del}$) to a neighbor in the current layer. By maintaining three separate lists for each of the three possibilities, we can create the three lists in sorted order and merge them in $O(|V|)$ time. The relative order of vertices in the current layer can be easily determined in linear time by tracking the positions of their distance values in the merged list. As a result, the current layer can be obtained in sorted form in $O(|V|)$ time and $O(|V|)$ space, leading to the following claim.

Lemma 4.2. *Time and space complexity of the sorting procedure in Algorithm 4.2 is* $O(|V|)$.

The PropagateInsertion Stage Note that the tentative distance computed for a vertex is sub-optimal if its shortest path from the source vertex traverses any insertion cost weighted edge in the current layer. One approach to compute optimal distance values is to process vertices in their sorted distance order (minimum first) and update the neighbor vertices, similar to Dijkstra's algorithm. When processing vertex v, the distance of its neighbor should be adjusted such that it is no more than $v.distance + \Delta_{ins}$. Selecting vertices with minimum scores can be achieved using a standard priority queue implementation (e.g., Fi-

bonacci heap); however, it would require $O(|E| + |V| \log |V|)$ time per layer. A key property that can be leveraged here is that all edges being considered in this stage have uniform weights (Δ_{ins}). Therefore, we propose a simpler and faster algorithm using two First-In-First-Out queues (Algorithm 4.3). The first queue q_1 is initialized with sorted vertices in the current layer, and the second queue q_2 is initialized as empty (line 4). The minimum distance vertex is always dequeued from either of the two queues (line 8). As and when distance of a vertex is updated by its neighbor, it is enqueued to q_2 (line 15). Following lemmas establish the correctness and an $O(|E| + |V|)$ time bound for the PropagateInsertion stage in the algorithm.

Algorithm 4.3: Algorithm to propagate insertions in the same layer

```
   Result: A sorted layer CurrentLayer with optimal distance values
1  Function PropagateInsertion(CurrentLayer)
2      x.resolved = false ∀x ∈ CurrentLayer;
3      Queue q1 = ∅, q2 = ∅;
4      q1.Enqueue(CurrentLayer);
5      CurrentLayer = [ ];
6      while q1 ≠ ∅ or q2 ≠ ∅ do
7          q_min = q1.Front() < q2.Front() ? q1 : q2;
8          x = q_min.Dequeue();
9          if x.resolved = false then
10             x.resolved = true;
11             CurrentLayer.Append(x);
12             foreach y ∈ x.neighbor & y.layer = x.layer do
13                 if y.distance > x.distance + Δ_ins then
14                     y.distance = x.distance + Δ_ins;
15                     q2.Enqueue(y);
```

Lemma 4.3. *In each iteration at line 8, Algorithm 4.3 dequeues a vertex with the minimum overall distance in q_1 and q_2.*

Proof. The queue q_1 always maintains its non-decreasing sorted order at the beginning of each loop iteration (line 6) in Algorithm 4.3 as we never enqueue new elements into q_1. We prove by contradiction that q_2 also maintains the order. Maintaining this invariant would immediately imply the above claim. Let i be the first iteration where q_2 lost the order.

Clearly $i > 1$. Because i is the first such iteration, new vertices (say $y_1, y_2, \ldots, y_k$) must have been enqueued to q_2 in the previous iteration (line 15), and the vertex (say x) which caused these additions must have been dequeued (line 8). Note that the distance of all the new vertices, the y_i's, equals $x.distance + \Delta_{ins}$. Therefore, the vertex prior to y_1 (say y_{pre}) must have a distance higher than y_1. However, this leads to a contradiction because if we consider the iteration when y_{pre} was enqueued to q_2, the distance of the vertex that caused addition of y_{pre} could not be higher than the distance of the vertex x. □

Lemma 4.4. *Once a vertex is dequeued in Algorithm 4.3, its computed distance equals the shortest distance from the source vertex.*

Proof. Lemma 4.3 establishes that Algorithm 4.3 processes all vertices that belong to the current layer in sorted order. Therefore, it mimics the choices made by Dijkstra's algorithm [58]. □

Lemma 4.5. *Algorithm 4.3 uses $O(|V| + |E|)$ time and $O(|V|)$ space to compute shortest distances in a layer.*

Proof. Each vertex in the current layer enqueues its updated neighbor vertices into q_2 at most once. Note that distance of a vertex can be updated at most once, therefore the maximum number of enqueue operations into q_2 is $|V|$. In addition, enqueue operations are never performed in q_1. Accordingly, the number of outer loop iterations (line 6) is bounded by $O(|V|)$. The inner loop (line 12) is executed at most once per vertex, therefore the amortized run-time of the inner loop is $O(|V| + |E|)$. □

The above claims yield an $O(m(|V| + |E|))$ time algorithm for aligning the query sequence to sequence graph. Assuming a constant alphabet, we can further tighten the bound to $O(|V| + m|E|)$ by using a simple preprocessing step suggested in [86]. This step transforms the sequence graph by merging all vertices with 0 in-degree into $\leq |\Sigma|$ vertices. As a result, the preprocessing ensures that the count of vertices in the new graph is no more

than $|E| + |\Sigma|$ without affecting the correctness. Summary of the above claims is presented as a following theorem:

Theorem 4.1. *Algorithm algorithm 4.1 computes the optimal cost of aligning a query sequence of length m to graph $G(V, E, \sigma)$ in $O(|V| + m|E|)$ time and $O(|V|)$ space using a linear gap penalty cost function.*

It is natural to wonder whether there exist faster algorithms for solving the sequence to graph alignment problem. As noted in [86], the sequence to sequence alignment problem is a special case of the sequence to graph alignment problem because a sequence can be represented as a directed chain graph with character labels. As a result, existence of either $O(m^{1-\epsilon}|E|)$ or $O(m|E|^{1-\epsilon}), \epsilon > 0$ time algorithm for solving the sequence to graph alignment problem is unlikely because it would also yield a strongly sub-quadratic algorithm for solving the sequence to sequence alignment problem, further contradicting SETH [92].

4.3.2 Affine Gap Penalty

Supporting affine gap penalty functions in the dynamic programming algorithm for sequence to sequence alignment is typically done by using three rather than one scoring matrix [93]. Similarly, the alignment graph can be extended to contain three sub-graphs with substitution, deletion, and insertion cost weighted edges respectively [86]. The edge weights are adjusted for the affine gap penalty model such that a cost for opening a gap is penalized whenever a path leaves the match sub-graph to either the insertion or the deletion sub-graph (Figure 4.2). The properties that were leveraged to design faster algorithm for linear gap penalty functions continue to hold in the new alignment graph. In particular, the sorting still requires linear time during the InitializeDistance stage, and insertion propagation is still executed over uniformly weighted edges in the insertion sub-graph. As a result, the two-stage algorithm can be extended to operate using affine gap penalty function in the same time and space complexity as with the linear gap penalty function.

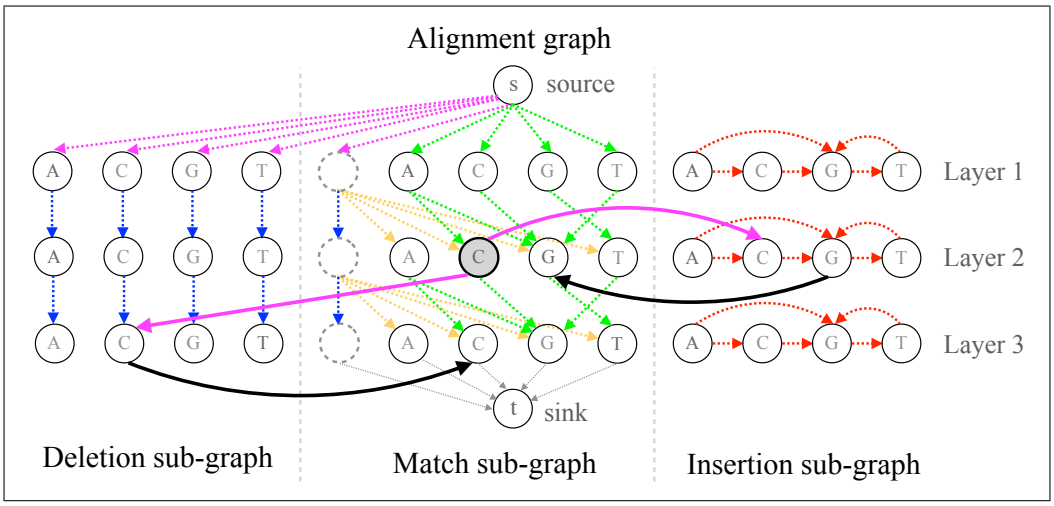

Figure 4.2: An example to illustrate the construction of an alignment graph for sequence to graph alignment using affine gap penalty. The alignment graph now contains three sub-graphs separated by the gray dash lines. The deletion and insertion weighted edges in the alignment graph for linear gap penalty are shifted to deletion sub-graph and insertion sub-graph, respectively. Their weights are also changed to the gap extension penalty. Besides, more edges are added to connect the sub-graphs with each other. For simplicity, we use the highlighted vertex as an example to illustrate how to open a gap and extend it. The weight of magenta colored edges is the sum of gap open penalty and gap extension penalty, and the weight of the black colored edges is 0.

4.4 Summary

In this chapter, we provide a faster polynomial time sequence to graph alignment algorithm that generalizes to linear gap penalty and affine gap penalty functions. The proposed algorithms use elementary data structures, therefore are simple to implement. Overall, the theoretical results presented in this work enhance the fundamental understanding of the problem, and will aid the development of faster tools for mapping to graphs. The alignment problem for sequence graphs is a rich area with several unsolved problems. For the intractable problem variants, development of faster exact and approximate algorithms are fertile grounds for future research. In addition, working towards robust indexing schemes and heuristics that scale to large input graphs and different sequencing technologies remains an important direction.

CHAPTER 5
FAST SEQUENCE TO GRAPH ALIGNMENT USING GRAPH WAVEFRONT ALGORITHM

Though the sequence to graph alignment algorithm we present in the previous chapter is optimal up to subpolynomial factors under SETH [92], it still needs to process the alignment graph layer by layer and visit all the nodes in each layer, which can be slow in practice. This motivates the design of algorithms that can skip some computation in practice when the sequence is highly similar to some walk in the graph.

In this chapter, we propose the graph wavefront alignment algorithm (Gwfa), a new method for aligning a sequence to a sequence graph. Although the worst-case time complexity of Gwfa is the same as the existing algorithms, it is designed to run faster for closely matching sequences, and its runtime in practice often increases only moderately with the edit distance of the optimal alignment. On four real datasets, Gwfa is up to four orders of magnitude faster than other exact sequence-to-graph alignment algorithms. We also propose a graph pruning heuristic on top of Gwfa, which can achieve an additional $\sim$10-fold speedup on large graphs.

The rest of this chapter is organized as follows. In Section 5.1, we review the efforts on accelerating sequence to graph alignment. In Section 5.2, we generalize the previous sequence to character-labeled sequence graph alignment algorithm to work on segment-labeled sequence graphs and then present the Gwfa algorithm together with heuristics to further accelerate it. In Section 5.3, we constructed sequence graphs and evaluated the performance of Gwfa and other sequence to graph alignment methods. Finally, we summarize this work in Section 5.4.

5.1 Related work

The design of the algorithm in this chapter is heavily motivated by the diagonal alignment algorithms for aligning two sequences. Thus, we first reviewed the related literature on pairwise sequence alignment, one of the most fundamental problems in bioinformatics. Two sequences of length N can be aligned by dynamic programming (DP) [94] in $O(N^2)$ time. This quadratic runtime is unaffordable for long sequences, motivating the development of diagonal alignment algorithms [95, 96, 97]. These methods only compute part of the DP matrix within a certain band, thereby reducing the runtime to $O(ND)$ where D is the minimum edit distance between the sequences being aligned. Recently, such methods have been further generalized to work with arbitrary cost linear or affine gap penalty [98, 99, 100, 101], and named wavefront alignment (WFA) as they progressively compute the partial alignments for increasing alignment scores until the best alignment is found. Compared with the standard DP alignment algorithm, these diagonal alignment methods have shown superior performance in practice when the two sequences to align are similar to each other.

As discussed in the previous chapter, aligning sequences to the graph-based structure is the foundation for mapping reads to graph-based reference genomes. This fundamental problem and its variants have been studied by multiple previous works [80, 102, 103, 104, 105]. Besides, this problem has been investigated even earlier in the string literature as approximate pattern matching to hypertext [63, 88, 64]. A summary of these results can be found in previous work [105].

Although several sequence to graph alignment algorithms have been proposed, and some of them are optimal up to subpolynomial improvements on runtime under the Strong Exponential Time Hypothesis [92], they are computationally intensive taking $O(|V| + |q||E|)$ time, where $|V|$ and $|E|$ are the numbers of vertices and edges in the graph and $|q|$ is the query length. This motivated the development of GraphAligner [106] and

Astarix [107, 108], which introduced heuristics to accelerate sequence to graph alignment. GraphAligner has implemented a dynamic banded sequence to graph alignment algorithm, which computes cells in the DP matrix with values less than the minimum cost plus a fixed threshold. As the optimal alignment may not be found within the confines of the band, this approach may report suboptimal alignments. Astarix reformulates the sequence to graph alignment as a shortest path problem and uses the A* algorithm to search for it. The A* search algorithm is an extension of Dijkstra's algorithm, and uses a heuristic function to estimate the distance from the vertex currently being explored to the target vertex for pruning the search space. Though the heuristics still guarantee finding the optimal alignment, extra computation cost is needed to process the query or the graph, which can be avoided.

5.2 Methods

Let Σ denote an alphabet, and x and y be strings over Σ. We use $x[i]$ to denote the i-th character of x, $|x|$ to denote its length, and xy to denote the concatenation of x and y. Let $x[i, j]$ $(1 \leq i \leq j \leq |x|)$ denote $x[i]x[i+1] \dots x[j]$, the substring of x beginning at the i-th position and ending at the j-th position. We assume a constant sized alphabet for DNA sequences, i.e., $\Sigma = \{A, C, G, T\}$.

Definition 5.1 (Sequence graph)**.** *A sequence graph $G(V, E, \sigma)$ is a directed graph with vertices V and edges E. Function $\sigma : V \to \Sigma^*$ labels each vertex $v \in V$ with a string $\sigma(v)$ over the alphabet Σ.*

This definition encompasses commonly used graphs (e.g., de Bruijn graphs, variation graphs) as shown by Jain *et al.* [105]. To simplify notation, we may directly use v to denote $\sigma(v)$ and thus $|v| = |\sigma(v)|$ and $v[i] = \sigma(v)[i]$.

Let $w_{j_1,j_n} = v_{j_1} v_{j_2} \dots v_{j_n}$ $(n \geq 1)$ denote a walk that starts from vertex v_{j_1} and ends at vertex v_{j_n} in G. It spells the sequence $\sigma(w_{j_1,j_n}) = \sigma(v_{j_1})\sigma(v_{j_2}) \dots \sigma(v_{j_n})$. Let $d(x, y)$ denote the edit distance between strings x and y. Then we formulate the problem of optimal sequence to graph alignment as the following.

Definition 5.2 (Optimal global sequence to graph alignment)**.** *Given a query sequence q, a sequence graph $G(V, E, \sigma)$, a start vertex $v_s \in V$ and an end vertex $v_e \in V$ that is reachable from v_s, find a walk $w_{s,e}$ such that $\forall w'_{s,e}$, $d(\sigma(w), q) \leq d(\sigma(w'), q)$.*

This formulation is similar to the global pairwise sequence alignment problem, where the gaps at the beginning or the end of either the query or the target sequence (the walk in the graph) are penalized. Besides, the formulation can be slightly changed to allow gaps at the start or the end of the walk in the sequence graph or perform alignment extension without specifying an end vertex (see Section A.1). These alternative formulations are frequently used to compute the base-level alignments in seed-and-extend methods for read mapping to linear or graphical genomes [35, 40, 106, 109] when the alignment candidate regions are known as a prior from the seeding step.

5.2.1 Sequence to graph alignment algorithm

Rautiainen *et al.* [104] generalized the standard DP for pairwise sequence alignment to formulate a recurrence for the sequence to graph alignment problem. Though there can be cyclic dependencies in their proposed recurrence, they proved that there is only one unique solution for the optimal alignment cost. Moreover, the recurrence can be solved in $O(|V| + |q||E|)$ time by Navarro's algorithm [64]. Note that their problem formulations operate on character labeled vertices, without loss of generality. However, directly allowing string labels results in more compact graph representation as the vertices of sequence graphs are usually labeled with strings (long segments of sequences) in practice (e.g., GFA format at https://github.com/GFA-spec/GFA-spec). Therefore, we first generalize the DP algorithm and propose the following recurrence for sequence alignment to string-labeled sequence graphs.

Let $H_{i,v,j}$ denote the minimum edit distance between a query prefix that ends at its i-th position and a walk sequence that ends at j-th position of (the string at) vertex v, where

$0 \leq i \leq |q|$, $v \in V$, and $0 \leq j \leq |v|$. It can be calculated as:

$$H_{i,v,j} = \min \begin{cases} H_{i-1,v,j} + 1, & i \geq 1 \\ H_{i,v,j-1} + 1, & j \geq 1 \\ H_{i-1,v,j-1} + \Delta_{i,v,j}, & i \geq 1, j \geq 1 \\ H_{i,u,|u|}, & j = 0, \forall u, (u,v) \in E \end{cases} \tag{5.1}$$

where $\Delta_{i,v,j} = 0$ if $q[i] = v[j]$ or 1 otherwise. When searching for optimal global sequence to graph alignment with start vertex v_s and end vertex v_e, the initial condition is $H_{0,v_s,0} = 0$, and the minimum cost is $H_{|q|,v_e,|v_e|}$.

In previous work, Jain *et al.* [105] have shown that a string labeled sequence graph can be converted to an equivalent character labeled sequence graph by splitting the vertex with a string label into a chain of character labeled vertices to compute sequence-to-graph alignment. Similarly, we argue that a character labeled graph can be converted to an equivalent string labeled graph by compacting each chain of character labeled vertices into a single vertex with string label that is the concatenation of the individual character labels, while the in-edges of the first vertex in the chain and the out-edges of the last vertex are reflected in how the new vertex is connected. Given these transformations, the proposed recurrence for sequence alignment to a string labeled graph is essentially equivalent to the recurrence for sequence alignment to a character labeled graph. Therefore, the algorithm proposed by Navarro [64] can also be adapted to solve the recurrence in Equation 5.1, which we show in Section A.2.

5.2.2 Diagonal recurrence

We use (i, v, j) to denote a DP-cell in H, where $0 \leq i \leq |q|$, $v \in V$, and $0 \leq j \leq |v|$. Note that the three-dimensional DP matrix H can be regarded as a set of $|V|$ two-dimensional DP matrices $\{H_v | v \in V\}$ along the vertex dimension. And H_v contains DP-cells $\{(i, v, j) | 0 \leq$

$i \leq |q|, 0 \leq j \leq |v|\}$. Let (v, k) denote a diagonal in H_v. A DP cell (i, u, j) is on diagonal (v, k) if and only if $u = v$ and $k = i - j$. Therefore, DP cell $(k + j, v, j)$ is always on diagonal (v, k). Define

$$\tilde{H}_{d,v,k} = \begin{cases} \max\{j \mid H_{k+j,v,j} = d\}, & \text{if } \exists j \text{ s.t. } H_{k+j,v,j} = d \\ \infty, & \text{otherwise} \end{cases}$$

which is the furthest offset on diagonal (v, k) among DP cells with edit distance d. Importantly, if $\tilde{H}_{d,v,k} = j$, it is always true that $H_{k+j,v,j} = d$, but conversely, if $H_{k+j,v,j} = d$, we only have $j \leq \tilde{H}_{d,v,k}$ because multiple cells on diagonal (v, k) may have the same distance d.

Following the WFA formulation, we calculate $\tilde{H}_{d,v,k}$ by rewriting Equation 5.1 into its diagonal recurrence as Equation 5.2:

$$\begin{aligned} \tilde{J}_{d,v,k} =& \max \begin{cases} \tilde{H}_{d-1,v,k-1} \\ \tilde{H}_{d-1,v,k+1} + 1 \\ \tilde{H}_{d-1,v,k} + 1 \\ 0, \quad \exists u, (u, v) \in E, \tilde{H}_{d,u,k-|u|} = |u| \end{cases} \\ \tilde{H}_{d,v,k} =& j + \mathrm{LCP}\left(q[k + j + 1, |q|], v[j + 1, |v|]\right), j = \tilde{J}_{d,v,k} \end{aligned} \tag{5.2}$$

Here $\mathrm{LCP}(x, y)$ gives the length of the longest common prefix between two strings x and y. The initial condition is $\tilde{J}_{0,v_s,0} = 0$ for global sequence to graph alignment problem, and we aim to find the minimum edit distance $\hat{d}$ such that $\tilde{H}_{\hat{d},v_e,|q|-|v_e|} = |v_e|$, i.e. $H_{|q|,v_e,|v_e|} = \hat{d}$.

Essentially, the first three cases for $\tilde{J}_{d,v,k}$ in Equation 5.2 are equivalent to running the wavefront algorithm within a vertex. The last case corresponds to the last case in Equation 5.1. To see the connection, we note that $\tilde{H}_{d,u,i-|u|} = |u|$ implies $H_{i,u,|u|} = d$ and $\tilde{H}_{d,v,i} = 0$ implies $H_{i,v,0} = d$. When u is a predecessor of v, we carry the computation at the last position of u onto the 0^{th} position of v. The $\mathrm{LCP}(\cdot, \cdot)$ term extends along exact

matches such that we can find the furthest cell in accordance with the definition of $\tilde{H}_{d,v,k}$.

To explain the implementation of Equation 5.2 in the next section, we further define d-wave $\mathcal{W}_d = \{(i, v, j) | H_{i,v,j} = d\}$, which consists of DP cells with the value d. Then $(k + \tilde{H}_{d,v,k}, v, \tilde{H}_{d,v,k}) \in \mathcal{W}_d$ is at the front of the d-wave along diagonal (v, k), and it is called a graph wavefront. The d-wavefront is $\mathcal{F}_d = \{(k+j, v, j) \mid j = \tilde{H}_{d,v,k} < \infty, -|v| \leq k \leq |q|, v \in V\}$.

5.2.3 Graph wavefront algorithm

Our method for computing optimal global sequence to graph alignment using the diagonal formulation is presented in Algorithm 5.1. We assume there is a walk from the start vertex v_s to the end vertex v_e. The algorithm progressively increases the edit distance d. In each iteration, it finds the d-wavefront $\mathcal{F}_d$ with Algorithm 5.2. It then uses Algorithm 5.3 to collect cells in the (d+1)-wave that are adjacent to $\mathcal{F}_d$. This process is repeated until the whole query is aligned to a walk with the last character of v_e as the end (line 8 in Algorithm 5.1).

Algorithm 5.1: Graph wavefront algorithm to find the optimal global sequence to graph alignment

Input: Query sequence q, sequence graph $G = (V, E, \sigma)$, start vertex $v_s \in V$ and end vertex $v_e \in V$.

```
1  function GWFEDITDIST(q, G, v_s, v_e) begin
2      k_e ← |q| − |v_e|
3      H̃_{v_s,0} ← 0
4      Q ← [(v_s, 0)]
5      d ← 0
6      while true do
7          GWFEXTEND(q, G, Q, H̃)
8          if H̃_{v_e,k_e} = |v_e| then
9              return d
10         d ← d + 1
11         GWFEXPAND(q, G, Q, H̃)
```

Specifically, in the iteration for distance d, the part of d-wave adjacent to $\mathcal{F}_{d-1}$ is first extended along each diagonal (v, k) through exact matches (lines 5–8 of Algorithm 5.2). If the end of vertex v is reached, lines 11–14 in Algorithm 5.2 traverse v's neighbors and prepare potential extensions in them. This extension step finds the d-wavefront $\mathcal{F}_d$.

Then lines 8 and 9 in Algorithm 5.1 check $\mathcal{F}_d$ and return the optimal alignment cost when $(|q|, v_e, |v_e|) \in \mathcal{F}_d$. If $(|q|, v_e, |v_e|) \notin \mathcal{F}_d$, Algorithm 5.3 next finds the set of DP cells adjacent to $\mathcal{F}_d$. These cells are part of the (d+1)-wave. They will be extended in the next iteration. We keep track of the graph wavefront in the alignment process by a set and a queue. The set stores the offsets (i.e., $\tilde{H}$) and allows the access of any offset with its diagonal in constant time. This graph wavefront set can be implemented with an array of which the size is the number of diagonals and a special sign to mark the existence of an element in the set. The queue keeps track of the diagonals on which the graph wavefront can be further updated or used to update the graph wavefront on other diagonals. Thus, only the graph wavefront on the diagonals in the queue instead of all the diagonals are processed in each iteration.

Algorithm 5.2: Graph wavefront extension algorithm

Input: Query sequence q, sequence graph $G = (V, E, \sigma)$, queue Q to keep track of diagonals and graph wavefront set $\tilde{H}$.

```
1  function GWFEXTEND(q, G, Q, H̃) begin
2      Q' ← [ ]
3      while Q is not empty do
4          (v, k) ← Q.pop()
5          j ← H̃_{v,k}
6          i ← k + j
7          l ← LCP(q[i + 1, |q|], v[j + 1, |v|])
8          H̃_{v,k} ← j + l
9          Q'.push(v, k)
10         if H̃_{v,k} = |v| then
11             for (v, u) ∈ E do
12                 if H̃_{u,k+|v|} ∉ H̃ then
13                     H̃_{u,k+|v|} ← 0
14                     Q.push(u, k + |v|)
15     Q ← Q'
```

To analyze the runtime of our proposed algorithm, we compute the time spent on updating the graph wavefront and exploring neighbors. After one round of extension and expansion, the graph wavefront on the diagonals in the queue advances at least one DP-cell forward if it has not reached the end. Once the graph wavefront has already reached the last cell on a diagonal, it cannot be updated using the graph wavefront on other diagonals.

Algorithm 5.3: Graph wavefront expansion algorithm

Input: Query sequence q, sequence graph $G = (V, E, \sigma)$, queue Q to keep track of diagonals and graph wavefront set $\tilde{H}$.

function GWFEXPAND($q, G, Q, \tilde{H}$) **begin**
  $\tilde{H}' \leftarrow \emptyset$
  $Q' \leftarrow [\,]$
  while Q *is not empty* **do**
    $(v, k) \leftarrow Q.\text{pop}()$
    $i \leftarrow k + \tilde{H}_{v,k}$
    if $i < |q|$ **then**
      if $\tilde{H}'_{v,k+1} \notin \tilde{H}'$ **then**
        if $\tilde{H}_{v,k+1} \notin \tilde{H}$ **or** $\tilde{H}_{v,k+1} < \tilde{H}_{v,k}$ **then**
          $\tilde{H}'_{v,k+1} \leftarrow \tilde{H}_{v,k}$
          $Q'.\text{push}(v, k+1)$
      else
        $\tilde{H}'_{v,k+1} \leftarrow \max\{\tilde{H}'_{v,k+1}, \tilde{H}_{v,k}\}$
    if $\tilde{H}_{v,k} < |v|$ **then**
      if $\tilde{H}'_{v,k-1} \notin \tilde{H}'$ **then**
        if $\tilde{H}_{v,k-1} \notin \tilde{H}$ **or** $\tilde{H}_{v,k-1} < \tilde{H}_{v,k} + 1$ **then**
          $\tilde{H}'_{v,k-1} \leftarrow \tilde{H}_{v,k} + 1$
          $Q'.\text{push}(v, k-1)$
      else
        $\tilde{H}'_{v,k-1} \leftarrow \max\{\tilde{H}'_{v,k-1}, \tilde{H}_{v,k} + 1\}$
    if $i < |q|$ **and** $\tilde{H}_{v,k} < |v|$ **then**
      if $\tilde{H}'_{v,k} \notin \tilde{H}'$ **then**
        $\tilde{H}'_{v,k} \leftarrow \tilde{H}_{v,k} + 1$
        $Q'.\text{push}(v, k)$
      else
        $\tilde{H}'_{v,k} \leftarrow \max\{\tilde{H}'_{v,k}, \tilde{H}_{v,k} + 1\}$
  for $(v, k) \in Q'$ **do**
    $\tilde{H}_{v,k} \leftarrow \tilde{H}'_{v,k}$
  $Q \leftarrow Q'$

Thus the number of updates on each diagonal is bounded by the length of the diagonal, which means the total number of updates is bounded by the total length of the diagonals as $O(|q| \sum_{v \in V} |v|)$.

Next, we compute the time spent on neighbor exploration (lines 10-14 in Algorithm 5.2). As mentioned above, on each diagonal, the graph wavefront only reaches the end of the diagonal once. So the neighbors of each vertex are explored only once in the extension step when the graph wavefront reaches the end of the diagonal of the vertex. We divide all diagonals into $|q|$ sets, $K_1, \ldots, K_{|q|}$, where K_i contains diagonals $\{(v, i - |v|) | v \in V\}$. As K_i has one diagonal for each vertex, the cost to explore the neighbors of vertices that have corresponding diagonals in K_i is $O(|E|)$. Thus the total time spent on neighbor exploration is $O(|q||E|)$. Therefore, the overall runtime is $O(|q|(\sum_{v \in V} |v| + |E|))$. Since exact and approximate sequence matching to graphs are equally hard [110], our proposed algorithm is optimal up to subpolynomial factors under SETH [92]. The memory usage is $O(|q||V| + \sum_{v \in V} |v|)$ since both the size of the queue to keep track of the diagonals and the graph wavefront set size are bounded by the total number of diagonals.

Note that Algorithm 5.1 and Algorithm 5.2 can also be slightly adapted to solve other sequence to graph alignment problem variants, which we show in Section A.1.

5.2.4 Graph wavefront pruning

The size of the d-wave increases with edit distance d. While some promising components on the d-wavefront $\mathcal{F}_d$ are advancing towards the solution, many other components on $\mathcal{F}_d$ significantly fall behind those promising ones. Based on this observation, we propose a graph wavefront pruning heuristic (Algorithm 5.4) to eliminate those unpromising graph wavefront components, thereby accelerating the alignment process. This algorithm is similar to the wavefront reduction heuristic used by WFA, and it may miss the optimal alignment.

The pruning algorithm first finds the maximum sum of aligned query length and aligned graph walk length $a_{\max}$ from all the DP-cells in the graph wavefront. Then each graph wavefront is rechecked to see whether it is left behind too much. If its sum of aligned query length and aligned graph walk length is too short compared with $a_{\max}$, the graph wavefront will be dropped. Since promising graph wavefronts usually start to appear after several iterations, the check is only performed when the offsets of the graph wavefront are already large enough to avoid overhead. The pruning can be performed before the graph wavefront expansion step (between line 10 and 11 in Algorithm 5.1), which would reduce the size of the graph wavefront set to expand.

Algorithm 5.4: Graph wavefront pruning algorithm

Input: Max allowed difference a_{diff} from the max sum of aligned query length and aligned graph walk length, queue Q to keep track of diagonals, graph wavefront set $\tilde{H}$, and aligned graph walk length set $\tilde{W}$.

```
1  function GWFPRUNE(a_diff, Q, H̃, W̃) begin
2      a_max ← 0
3      for (v, k) ∈ Q do
4          i ← k + H̃_{v,k}
5          a ← i + W̃_{v,k}
6          a_max ← max(a_max, a)
7      if a_max > a_diff then
8          H̃' ← ∅
9          Q' ← [ ]
10         while Q is not empty do
11             (v, k) ← Q.pop()
12             i ← k + H̃_{v,k}
13             a ← i + W̃_{v,k}
14             if a_max − a < a_diff then
15                 H̃'_{v,k} ← H̃_{v,k}
16                 Q'.push(v, k)
17         for (v, k) ∈ Q' do
18             H̃_{v,k} ← H̃'_{v,k}
19         Q ← Q'
```

5.2.5 Graph alignment walk traceback

We further present Algorithm 5.5 and Algorithm 5.6 to compute the optimal alignment walk in the graph. We slightly modified Algorithm 5.1 and Algorithm 5.2 to save traceback

information during the graph extension process (Section A.3). Then the alignment walk can be computed with the traceback information. Algorithm 5.5 is used to record the traceback information when exploring the neighbors of the vertices. It saves information of the previous vertex that leads to the extension to the current vertex. After the minimum edit distance is found, we can start from the last vertex in the alignment walk and iteratively trace vertices recorded in the extension process (Algorithm 5.6).

Algorithm 5.5: Method to add a piece of traceback information during the alignment

Input: Traceback information array T, the vertex v to record traceback information, the previous traceback information index t in T and a hash map M to avoid duplicate traceback information.

```
function GwfPushTrace(v, t, T, M) begin
    if M_{v,t} ∉ M then
        M_{v,t} ← T.size()
        T.push(v, t)
    return M_{v,t}
```

Algorithm 5.6: Graph alignment walk traceback algorithm

Input: Traceback information array T and the walk end index t_e in T.

```
function GwfTraceback(t_e, T) begin
    W ← [ ]
    t ← t_e
    while t ≥ 0 and T[t].v ≥ 0 do
        W.push(T[t].v)
        t ← T[t].t
    Reverse W
    return W
```

5.3 Results

We implemented our proposed method and termed it Gwfa, made available at https://github.com/lh3/gwfa. In the following sections, we establish benchmark datasets and demonstrate the advantages of Gwfa empirically compared with other methods.

5.3.1 Experimental setup

Benchmark data sets

To evaluate the performance of Gwfa, we obtained four sequence graphs around complement component 4 (C4), leukocyte receptor complex (LRC), and major histocompatibility complex (MHC) loci in the human genome (Table 5.1).

Table 5.1: List of benchmark sequence graphs.

Graph	Type	Region	Total segment length (bp)	# vertices	# edges
G1	Cyclic	C4	42,036	1,531	2,073
G2	Cyclic	LRC	1,294,511	48,097	67,008
G3	Cyclic	MHC	5,951,398	232,508	320,009
G4	Acyclic	MHC	5,476,947	1,144	1,608

These three loci play crucial biological roles and are enriched with polymorphisms. The numerous variations at these loci increases the complexity of the graphs which poses a great challenge to alignment. As a result, these loci are often used to benchmark graphical genome tools [111, 112, 29, 113]. Specifically, MHC is one of the most critical regions for infection and autoimmunity in the human genome and is crucial in adaptive and innate immune responses [114]. MHC haplotypes are highly polymorphic, i.e., various alleles are present across individuals in the population. C4 genes are located in MHC and encode proteins involved in the complement system. The LRC locus contains many genetic variations and comprises genes that can regulate immune responses [115].

To construct G1, G2 and G3, we used ODGI [113] to extract the subgraphs around the C4 (chr6:31,972,057–32,055,418 on GRCh38), LRC (chr19:54,528,888–55,595,686) and MHC (chr6:29,000,000–34,000,000) loci from the PGGB human pangenome graphs released by the Human Pangenome Reference Consortium (HPRC). The PGGB graph was built from GRCh38, CHM13 [27], and the phased contig assemblies of 44 diploid individuals. These altogether encode variations from 90 human haplotypes. To construct G4, we used gfatools (https://github.com/lh3/gfatools) to extract the MHC region from the mini-

graph graph [29] built from the same set of assemblies. G4 contains fewer vertices because the minigraph graph only encodes $\geq$50 bp structural variations, while the PGGB graph additionally encodes SNPs and short INDELs in all input samples. The details and command lines to build these sequence graphs can be found in Section A.4.

As HG002, HG005, and NA19240 diploid assemblies were excluded when building the human pangenome graphs, these six haplotypes provide ideal query sequences for the alignment evaluation. We ran "minimap2 -x asm20" to align the C4, LRC, and MHC regions of GRCh38 to HG002, HG005, and NA19240 haplotypes, and retrieved the corresponding regions from the six haplotypes. We excluded LRC on the HG002 maternal haplotype because it was not fully assembled.

Hardware and software

We ran all experiments on a compute node with dual Intel Xeon Gold 6226 CPU (2.70 GHz) processors equipped with 128 GB main memory. The time and memory usage of each run was measured.

We ran Gwfa both in its exact mode and approximate mode with the pruning heuristic, which is controlled by the a_{diff} parameter. We set this parameter to 20,000 for the G1–G3 dataset and 30,000 for the G4 dataset. Setting a_{diff} to 20,000 for the G4 dataset would miss the optimal alignment. We also evaluated Gwfa with traceback to output the walk corresponding to the optimal alignment identified.

Sequence-to-graph alignment can be formulated as a shortest path problem [105]. In this work, we implemented this idea on top of Dijkstra's algorithm. We also generalized Navarro's algorithm for string-labeled graphs and provided an efficient implementation (Algorithm A.3). Both implementations are available at https://github.com/haowenz/sgat. They serve as a baseline and help to verify the correctness of our graph wavefront algorithm implementation.

For further comparative evaluation, we considered GraphAligner [106] and As-

tarix [107], two recently developed tools for accelerating sequence to graph alignment. Like Gwfa, both of these can also work with cyclic sequence graphs. GraphAligner implements the bit-parallel algorithm [104], which guarantees to finding an optimal alignment, as well as a seed-and-extend heuristic method, which is fast in practice. We included both in our evaluation (see Section A.5 for more details about running the tools). Astarix could in theory find minimum edit distance with option 'align-optimal -G 1'. However, Astarix either reported errors or could not finish any of the experiments and was thus excluded.

To investigate the performance gap between aligning a query to a target sequence and a sequence graph, we also ran Edlib [116] and WFA2 [99] to align the C4 regions of HG002, HG005, and NA19240 to the GRCh38 C4 region. The parameters for running these tools are listed in Section A.6. We compared these two tools with the sequence to graph alignment tools mentioned above on aligning queries to a linear sequence graph built from the GRCh38 C4 region (Section A.7).

5.3.2 Runtime comparison

Table 5.2 shows the runtime of each method. When aligning queries to G1, Gwfa and Gwfa-pruning were several orders of magnitude faster than all other sequence to graph alignment algorithms. This is expected since G1 is built from 90 human haplotypes and encodes all their variants, which leads to relatively small edit distances for all the alignments. Both Gwfa and Gwfa-pruning were able to skip the exploration of many DP-cells, which made them significantly faster than other methods. Due to the same reason, our Dijkstra's algorithm implementation was faster than other methods except for Gwfa and Gwfa-pruning on aligning queries to G1. In addition, all the methods were able to report optimal alignments between the queries and sequence graph G1.

On the G2 dataset, Gwfa-pruning was 9 times to 5 orders of magnitude faster than other methods. Gwfa in its exact mode was even faster than the GraphAligner heuristic on the three query sequences with relatively small edit distances. While the GraphAligner

Table 5.2: Runtimes (in seconds) of the methods to align queries to real sequence graphs. Dijkstra's algorithm and generalized Navarro's algorithm cannot finish in 24 hours when mapping queries to the MHC graphs. The LRC region of HG002.2 was not fully assembled. The GraphAligner heuristic would fragment the alignment into multiple pieces when its runtime is marked with *.

Haplotype	Edit distance	Gwfa	Gwfa pruning	Dijkstra's algorithm	Navarro's algorithm	GraphAligner bitvector	GraphAligner heuristic
Alignments to G1							
HG002.1	22	**<0.01**	**<0.01**	0.14	17.40	4.23	1.31
HG002.2	22	**<0.01**	**<0.01**	0.14	17.40	4.24	0.18
HG005.1	21	**<0.01**	**<0.01**	0.10	17.40	4.24	0.18
HG005.2	19	**<0.01**	**<0.01**	0.07	17.40	4.24	0.19
NA19240.1	19	**<0.01**	**<0.01**	0.07	15.90	3.87	0.16
NA19240.2	20	**<0.01**	**<0.01**	0.06	15.90	3.85	0.16
Alignments to G2							
HG002.1	65	1.20	**0.13**	760.13	10,285.00	1,186.55	14.12
HG005.1	81	1.87	**0.15**	1189.67	10,146.00	1,161.97	12.99
HG005.2	174	5.42	**0.24**	3340.85	10,045.00	1,140.04	5.99
NA19240.1	452	13.91	**0.62**	9,944.00	10,184.00	1,134.08	*5.58
NA19240.2	469	10.73	**0.51**	6,476.00	9,702.00	1,110.03	9.80
Alignments to G3							
HG002.1	331	45.52	**5.40**	-	-	19,389.98	24.25
HG002.2	268	16.29	**1.76**	-	-	17,594.77	29.24
HG005.1	300	23.78	**0.91**	-	-	17,754.32	26.48
HG005.2	298	20.90	**0.69**	-	-	19,229.14	22.31
NA19240.1	743	68.84	**2.07**	-	-	17,679.07	*32.06
NA19240.2	885	132.38	**2.06**	-	-	18,215.37	30.39
Alignments to G4							
HG002.1	43,280	868.94	**7.70**	-	-	15,008.00	*31.35
HG002.2	34,790	453.07	**4.39**	-	-	14,733.00	*29.42
HG005.1	42,124	849.21	**6.97**	-	-	15,059.50	*315.57
HG005.2	41,271	824.15	**6.75**	-	-	15,061.50	*28.68
NA19240.1	42,677	709.35	**7.43**	-	-	15,017.50	*29.63
NA19240.2	31,564	346.96	**2.90**	-	-	15,109.50	*31.39

heuristic was faster on the other two queries, it fragmented the alignment of the NA19240.1 haplotype and failed to report the expected alignment.

On the much larger G3 dataset, both Dijkstra's and Navarro's algorithms could not finish in 24 hours. Gwfa-pruning was the fastest. Gwfa in its exact mode generally has comparable performance to the GraphAligner in its heuristic mode. GraphAligner heuristic again fragmented the alignment of the NA19240.1 haplotype.

As G4 only encodes large variants, the edit distances of alignments on G4 are much larger than those on other datasets. Thus, the speed of Gwfa decreased. Although the GraphAligner heuristic was faster than Gwfa, it failed to report complete alignments for all query sequences. The Gwfa pruning heuristic was the fastest and could report the optimal distances.

When the optimal alignment walk was traced, there was only a mild increase in runtime for Gwfa or Gwfa-pruning. When aligning queries to G1, the runtime with alignment walk traceback was still less than 0.01s. On average, the runtime increased by 13.6% and 3.8% for Gwfa and Gwfa-pruning to trace the optimal alignment walk on G2, and 14.5% and 7.8% on G3 respectively. For alignment traceback on G4, the runtime increased less than 2% for Gwfa with or without pruning mainly because G4 has fewer vertices and edges than other graphs.

5.3.3 Memory usage

Table 5.3 shows the memory usage of the tools. When aligning queries to sequence graphs, Gwfa-pruning used 2-83 times less memory than other methods. Besides, compared with other methods, Gwfa without pruning also used comparable or less memory. The alignment walk traceback process of Gwfa and Gwfa-pruning on G1 needs an extra 6.8% memory on average. The average memory usage increased by 49.1% and 15.8% for Gwfa and Gwfa-pruning to trace back the alignment walk on G2, and 75.2% and 18.9% on G3, respectively, as the problem size is larger. Nevertheless, Gwfa-pruning still used the least memory to

align any queries to G2 or G3 and trace the alignment. For alignments on G4, the memory usage of Gwfa with or without pruning only increased by less than 3% on average. This is because G4 has fewer vertices than the other graphs, which results in a smaller traceback stack size. The results consistently point to small memory footprint needed by our proposed methods.

Table 5.3: Memory usage (MB) of the methods to align queries to real sequence graphs. Dijkstra's algorithm and generalized Navarro's algorithm cannot finish in 24 hours when mapping queries to the MHC graphs. The LRC region of HG002.2 was not fully assembled. The GraphAligner heuristic would fragment the alignment into multiple pieces when its runtime is marked with *.

Haplotype	Edit distance	Gwfa	Gwfa pruning	Dijkstra's algorithm	Navarro's algorithm	GraphAligner bitvector	GraphAligner heuristic
Alignments to G1							
HG002.1	22	3.3	**2.7**	191.1	165.4	7.0	53.6
HG002.2	22	3.5	**2.5**	190.8	165.4	7.0	51.1
HG005.1	21	3.5	**2.5**	186.4	165.4	7.0	51.2
HG005.2	19	**2.2**	**2.2**	184.1	165.4	7.0	51.5
NA19240.1	19	**2.2**	**2.2**	184.1	165.4	7.0	50.2
NA19240.2	20	2.7	**2.5**	184.1	165.4	7.0	48.0
Alignments on G2							
HG002.1	65	211.4	**37.4**	2,364.4	213.3	164.9	476.0
HG005.1	81	214.9	**37.4**	3,463.8	213.3	164.6	459.3
HG005.2	174	212.4	**37.4**	8,691.5	213.3	164.6	503.5
NA19240.1	452	230.6	**42.9**	23,995.3	213.3	164.7	*455.8
NA19240.2	469	244.6	**37.4**	19,185.0	213.3	164.5	446.3
Alignments on G3							
HG002.1	331	975.7	**442.9**	-	-	769.4	1251.9
HG002.2	268	824.9	**224.8**	-	-	769.0	1388.5
HG005.1	300	912.6	**178.1**	-	-	769.2	1587.6
HG005.2	298	946.7	**158.0**	-	-	769.2	1232.8
NA19240.1	743	960.0	**235.6**	-	-	769.2	*1352.6
NA19240.2	885	934.6	**220.8**	-	-	769.4	1566.3
Alignments on G4							
HG002.1	43,280	439.1	**32.8**	-	-	889.6	*1,122.8
HG002.2	34,790	400.0	**27.0**	-	-	889.2	*1,041.0
HG005.1	42,124	421.2	**27.1**	-	-	889.4	*1,063.1
HG005.2	41,271	441.2	**32.5**	-	-	889.4	*971.7
NA19240.1	42,677	425.7	**37.0**	-	-	889.4	*1,012.6
NA19240.2	31,564	397.8	**36.8**	-	-	889.7	*1,113.4

5.4 Summary

Sequence-to-graph alignment is the foundation of various pan-genomics applications. However, it is computationally expensive when the sequence is long or the graph is large, as the worst-case runtime is the product of these. Previous algorithms could not fully leverage the similarity information between the sequence and its optimal alignment walk in the graph to reduce runtime while preserving optimality. In this work we proposed Gwfa, a novel sequence-to-graph alignment algorithm, and a heuristic to accelerate the alignment further. We demonstrated empirically the superior performance of Gwfa over other sequence-to-graph alignment algorithms on various input queries and graphs.

CHAPTER 6

VALIDATING PAIRED-END READ MAPPINGS IN GRAPH GENOMES

Paired-end Illumina sequencing is a commonly used sequencing platform in genomics, where the paired-end distance constraints allow the disambiguation of repeats. Many recent works have explored provably good index-based and alignment-based strategies for mapping individual reads to graphs. However, validating distance constraints efficiently over graphs is not trivial, and existing sequence to graph mappers rely on heuristics.

In this chapter, we introduce a mathematical formulation of the problem and provide a new algorithm to solve it exactly. We take advantage of the high sparsity of reference graphs and use sparse matrix-matrix multiplications (SpGEMM) to build an index that can be queried efficiently by a mapping algorithm for validating the distance constraints. The algorithm's effectiveness is demonstrated using real reference graphs, including a human MHC variation graph and a pan-genome de-Bruijn graph built using genomes of 20 B. *anthracis* strains. While the one-time indexing time can vary from a few minutes to a few hours using our algorithm, answering a million distance queries takes less than a second.

The rest of this chapter is organized as follows. In Section 6.1, we reviewed several heuristics developed for mapping paired-end reads to graphs. In Section 6.2, we formulated the paired-end read mapping validation problem and discuss the limitation of a few trivial solutions to this problem. In Section 6.4, we present our index-based algorithm and analyze the complexity of it. In Section 6.5, we test our proposed algorithm on real reference graphs, and demonstrate the algorithm is fast in practice. We then summarize this work in Section 6.6.

6.1 Related work

Designing provably good algorithms for approximate sequence matching to graphs, using both index-based and alignment-based approaches, remains an active research area. A few recent works have investigated extending Burrows-Wheeler-Transform-based indexing to sequence DAGs [85] and de-Bruijn graphs [117, 118, 119]. Similarly, there exist studies that have explored extension of the classic sequence-to-sequence alignment routines to graphs [120, 103, 64, 86]. In the previous two chapters, we also proposed two sequence to graph alignment algorithms. However, a general string to graph pattern matching formulation is only good for mapping single-end reads or single-molecule sequencing reads, and does not account for pairing information.

Paired-end sequencing provides information about the relative orientation and genomic distance between the two reads in a pair. When reads originate from repetitive regions, this information is valuable for pruning large number of false candidates [121]. Popular short read mapping tools for linear references, e.g., BWA-mem [122] and Bowtie2 [36], therefore, enforce these constraints in a read pair to guide the selection of the true mapping locus. Using a linear reference, calculating gap between two mapping locations is just a simple subtraction operation. However, it still remains unclear how to efficiently validate the constraints using large non-linear graph-based references and read sets.

Several sequence to graph aligners have been developed in recent years to map reads to variation graphs [123, 32, 111, 124, 125, 126, 127], de-Bruijn graphs [84, 83, 82] and splicing graphs [74, 128]. Readers are referred to review articles, e.g., [129, 31] for an expanded list of the tools. Among these tools, Graph-Aligner [126], vg [32], deBGA [82], HISAT2 [124] and HLA-PRG [123] support paired-end read mapping. However, all of these use heuristics to measure the observed insert size between the two reads in a pair, mainly due to lack of associated provably-good graph-based algorithms. A popular heuristic adopted by the tools is to do the computation while assuming a linear ordering of vertices

(e.g., topological order). However, it can produce misleading results in complex variation-rich graph regions.

6.2 Problem Formulation

Sequence graph is typically defined as a directed graph with either string or character labeled vertices because converting one form into the other is straightforward. In this chapter, we focus on the char-labeled sequence graph defined as Definition 4.1. In addition, commonly used graph formats, such as de-Bruijn graphs, bi-directional de-Bruijn graphs, overlap graphs or variation graphs can be converted into sequence graphs with at most a constant factor increase in vertex or edge set sizes. While single-end read alignments can be judged by their alignment scores alone, a valid paired-end read alignment over a sequence graph should satisfy the expected paired-end distance constraints and orientation. As insert size can vary within a range, let d_1 and d_2 denote the minimum and maximum allowed values of the inner distance between the reads within a pair (see Figure 6.1).

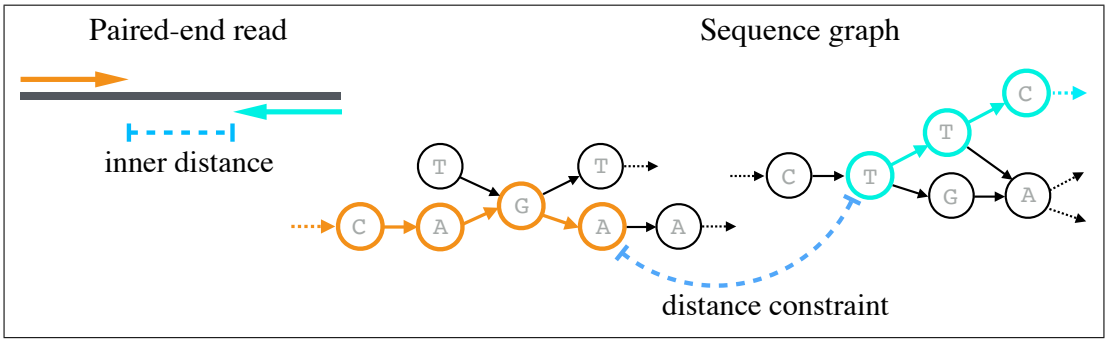

Figure 6.1: Visualizing distance constraints while mapping paired-end reads to sequence graphs.

Definition 6.1. *Paired-end Validation Problem: Suppose two reads r_1, r_2 in a pair are independently mapped to a sequence graph $G(V, E)$ using their positive and negative strands respectively. Let v_1 be the vertex where a walk to which r_1 (+ve strand) is mapped ends, and v_2 be the vertex where a walk to which r_2 (-ve strand) is mapped starts; then we refer*

to this pair of walks as a valid paired-end read mapping if and only if there exists a walk from v_1 to v_2 of length $d \in [d_1, d_2]$.

The above problem definition is based on the assumption that a fragment, from which a read pair is sequenced, can align to any (cyclic or acyclic) path in the input sequence graph. In this work, we focus on designing an efficient algorithm that can quickly answer the above path queries for any two given vertices in the graph. Typically, there are multiple mapping candidates to evaluate for each read pair, especially if a read is sequenced from repetitive regions of a genome. In addition, a typical read set in a genomic study may contain millions or billions of reads. Therefore, validating the distance queries quickly using an appropriate indexing scheme is desirable.

6.3 Related Problems in Graph Theory

Computing all-pairs shortest paths in $G(V, E)$ may help to identify true-positives or true-negatives, but only for those vertex pairs whose shortest distance $\geq d_1$. No conclusion can be drawn when the shortest distance between two vertices is $< d_1$, as a valid walk need not be the shortest path. In addition, computing all-pairs shortest paths is expensive, and may not provide the desired scalability [58]. If $d_1 = d_2$, the formulated problem becomes a special case of the exact-path length problem [130], with all edge weights set to 1. The exact-path length problem determines if a path of a specified distance exists between two vertices in a weighted graph. An extension of this problem, referred to as the gap-filling problem [131], has been explored in the context of genome assembly using paired-end or mate pair read sets. Although the exact-path length problem has been shown to be $\mathcal{NP}$-complete [130], we will demonstrate a simple and practical polynomial-time algorithm for our problem with unweighted edges. Finally, if $d_1 = 0$ and $d_2 = |V|$, then our problem is equivalent to determining transitive closure of a graph [132]. In our case, however, we expect $d_2 \ll |V|$.

Our approach is based on an indexing strategy where we pre-compute a boolean index

matrix, which has a 1 for each vertex pair that satisfies the distance constraints (Section 6.4). Computing the index requires polynomial operations, and paired-end distance queries can be computed quickly using index lookups during the read mapping process. Before describing the algorithm, we first discuss a trivial pseudo-polynomial time algorithm to solve the paired-end distance validation problem. It is based on a well-known algorithm used to solve the intractable subset-sum problem.

A Pseudo-polynomial Time Algorithm

The problem of validating distance constraints between two vertices can be solved using dynamic programming. Assume $s \in V$ is the source vertex from where we need to query walks of length $d \in [d_1, d_2]$. For a vertex $v \in V$, let $a(v, l)$ be a boolean value which is true if and only if there is a walk of length l from source s to v. Then, the following recurrence solves the problem:

$$a(v, 0) = 1 \text{ if } v = s \text{ and } 0 \text{ otherwise,}$$
$$a(v, l) = \bigvee_{(u,v) \in E} a(u, l - 1)$$

Solving the above recurrence requires filling a $|V| \times d_2$ table in column-wise order. The distance constraint from the source vertex s to $t \in V$ is satisfied if and only if $a(t, l) = 1$ for any $l \in [d_1, d_2]$. Note that it is sufficient to store two columns in memory to fill the table, and an additional column to track the final result. The algorithm is summarized as a lemma below.

Lemma 6.1. *There exists an $O(d_2|E|)$ time and $O(|V|)$ space algorithm that decides existence of a walk of length $d \in [d_1, d_2]$ from one vertex to another in $G(V, E)$.*

The time complexity of the above algorithm is significantly high, as it requires $O(d_2|E|)$ time to validate distance constraints from a fixed source vertex. With some optimizations however, the above algorithm can be accelerated. As observed by Salmela *et al.* [131] in

the context of gap-filling problem, we expect $d_2 \ll |V|$, therefore, it should be possible to compute a sub-graph containing vertices within $\leq d_2/2$ distance from v_1 or v_2, before solving the recurrence. While this strategy was shown to be effective for gap-filling between assembled contigs, the count of vertex pairs to evaluate during read mapping process is expected to be significantly higher for large read sets. Reference genomes (e.g., GRCh38 for human genome) or graphs are static, or evolve slowly, in genomic analyses. As such, it is desirable to use an index-based strategy, where we pay a one-time cost to build an index, and validate the paired-end distance constraints quickly.

6.4 An Index-based Polynomial-time Algorithm

In the following, we describe our index construction and querying algorithm. Given a sequence graph in the form of a boolean adjacency matrix, the index construction procedure uses boolean matrix additions and multiplications. As we will note later, the worst-case time of building our index is polynomial in the input size, but still computationally prohibitive to handle real data instances. Subsequently, we will show how to exploit sparsity in graphs to accelerate the computation. The construction algorithm relies on the following boolean matrix operations.

Definition 6.2. *Boolean matrix operations: Let A and B be two boolean $n \times n$ matrices. The standard boolean matrix operations are evaluated in the following way:*

Addition	$C = A \vee B$	$C_{ij} = A_{ij} \vee B_{ij}$
Multiplication	$C = A \cdot B$	$C_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge B_{kj}$
Power	$C = A^k$	$C = \underbrace{A \cdot A \cdot \ldots A}_{k \text{ times}}$

The boolean matrix addition and multiplication can also be performed using the standard matrix addition and multiplication, respectively. This is done by adjusting the non-

zero values in output matrix to 1. Next, we define index matrix $\mathcal{T}$, built using the adjacency matrix of the input graph and the distance parameters d_1 and d_2. Lemma 6.2 and 6.3 include its correctness proof and worst-case construction time complexity.

Definition 6.3. *Let Adj be the $|V| \times |V|$-sized boolean adjacency matrix associated with graph $G(V, E)$. Define index matrix $\mathcal{T} = Adj^{d_1} \cdot (Adj \vee I)^{d_2 - d_1}$, where I is an identity matrix.*

Lemma 6.2. *$\mathcal{T}[i, j] = 1$ if and only if there exists a walk of length $d \in [d_1, d_2]$ from vertex v_i to vertex v_j.*

Proof. Note that $Adj^k[i, j] = 1$ if and only if there is a walk of length k from vertex v_i to v_j. To validate the paired-end distance constraints, we require $Adj^{d_1} \vee Adj^{d_1+1} \vee \ldots \vee Adj^{d_2}$.

$$\begin{aligned} \bigvee_{i=d_1}^{d_2} Adj^i &= Adj^{d_1} \cdot (\bigvee_{i=0}^{d_2-d_1} Adj^i) \\ &= Adj^{d_1} \cdot (Adj \vee I)^{d_2-d_1} \end{aligned}$$

□

Lemma 6.3. *If multiplying two square matrices of dimension $|V| \times |V|$ requires $O(|V|^\omega)$ time, $\omega \in \mathbb{R}$, then computing the index matrix requires $O(|V|^\omega \cdot \log(d_2))$ time and $O(|V|^2)$ space.*

Proof. Matrix addition uses $O(|V|^2)$ operations. Computing A^k requires $O(|V|^\omega \log k)$ operations. Therefore, computing the index matrix requires $O(|V|^2 + |V|^\omega(1 + \log d_1 + \log(d_2 - d_1)))$ operations. As $\omega \geq 2$ and $d_2 \geq d_1$, this simplifies to $O(|V|^\omega \cdot \log(d_2))$ time. □

The current best algorithm to compute general matrix multiplication requires $O(|V|^{2.37})$ time [133]. Using the general matrix storage format, querying for the distance constraints

between a vertex pair requires a simple $O(1)$ lookup. However, general matrix multiplication solvers require at least quadratic time and space (in terms of $|V|$), which does not scale to real graph instances. We next propose an alternate approach to build the index matrix that exploits sparsity in sequence graphs.

6.4.1 Exploiting Sparsity in Sequence Graphs

Typically, sequence graphs representing variation or assembly graphs have large diameter and high sparsity, with edge to vertex ratio close to 1 [78]. As $d_2 \ll |V|$ in practice, we also expect our final index matrix to be sparse. As a result, we propose using SpGEMM (sparse matrix-matrix multiplication) operations to build the index. Below, we briefly recall the algorithm and matrix storage format used for SpGEMM. Subsequently, we shed light on the construction time and size of the index using this approach. We borrow standard notations typically used to discuss SpGEMM algorithms. Let $nnz(A)$ denote number of non-zero values in matrix A. During boolean matrix multiplication $A \cdot B$, let $bitops(AB)$ indicate the count of non-zero bitwise-AND operations (i.e., $1 \wedge 1$), assuming Definition 6.2.

Working with Sparse Matrices

Storage: During SpGEMM, the input and output matrices are stored in a sparse format, such that the space is primarily used for non-zeros. Compressed Sparse Row (CSR) is a classic data structure for this purpose [134]. In CSR format, a boolean matrix $A_{n\times n}$ can be represented by using two arrays: the first array ptr of size $n+1$ contains row pointers, and the second array $cols$ of size $O(nnz(A))$ contains column indices of each non-zero entry in A, starting from the first row to the last. The row pointers are essentially offsets within the second array, such that the range $\big[cols[ptr[i]], cols[ptr[i+1]]\big)$ lists column indices in row i. By default, CSR format does not require the indices of a row to be sorted. However, the sorted order will be useful in our index storage to enable fast querying. Therefore, we use 'sorted-CSR' format in our application.

Remark 6.1. *Storing a matrix $A_{n\times n}$ in sorted-CSR format requires $\Theta(n + nnz(A))$ space.*

Remark 6.2. *Given a sequence graph $G(V, E)$ as an array of edge tuples, transforming its adjacency information into sorted-CSR format takes $\Theta(|V| + |E|)$ time using count sort [135].*

Multiplication (SpGEMM): SpGEMM algorithms limit their operation count to just non-zero multiplications and additions required to compute the product, as the remaining entries are guaranteed to be 0. Most of the sequential and high-performance parallel algorithms for SpGEMM, including in MATLAB [136], are based on Gustavson's algorithm [135]. The algorithm can take input matrices A and B in sorted-CSR format and produce the output matrix $C = A \cdot B$ in the same format. In this algorithm, a row of matrix C, i.e., $C[i, :]$ is computed as a linear combination of the rows κ of B for which $A[i, \kappa] \neq 0$. The complexity result from Gustavson's work is listed as the following lemma.

Lemma 6.4. *The time complexity to multiply two sparse matrices $A_{n\times n}$ and $B_{n\times n}$ using Gustavson's algorithm is $\Theta(n + nnz(A) + bitops(AB))$.*

Indexing Time and Storage Complexity

Computing the index (Definition 6.3) requires several SpGEMM operations. As such, it is hard to derive a tight bound on the complexity, as runtime and index size depend on non-zero structure of the input sequence graph. However, it is important to get an insight into how the different parameters, e.g., $|V|, d_1, d_2$ may affect them. To address this, we derive a practically useful lower-bound on the complexity.

Consider the chain graph $G'(V', E')$ associated with a longest path in $G(V, E)$, $V' \subset V, E' \subset E$. We claim that the time needed to index $G(V, E)$ is either the same or worse than indexing its chain $G'(V', E')$ (Lemma 6.5). Subsequently, we compute the time complexity for indexing the chain using our SpGEMM-based algorithm. The rationale for analyzing the chain graph is (a) non-zero structure of a chain is simple and well-defined for computing

time complexity, and (b) sequence graphs are expected to have 'near-linear' topology in practice, therefore the derived lower-bounds will be a useful indication of the true costs.

Lemma 6.5. *The time requirement for indexing a graph $G(V, E)$ using SpGEMM is either the same or higher than indexing the chain graph $G'(V', E')$ associated with its longest path.*

Proof. Note that G' is a sub-graph of G, which will also reflect in their adjacency matrices. The above lemma is based on the following simple observation. Suppose $A, B, A', B', \delta_1, \delta_2$ are boolean square matrices, such that, $A = A' \vee \delta_1$ and $B = B' \vee \delta_2$. Then, using Gustavson's algorithm, multiplying A and B requires at least as much time as required for multiplying A' and B'. In addition, the product $(A \cdot B)$ is of the form $(A' \cdot B') \vee \delta_3$, where δ_3 is a boolean matrix. For each SpGEMM executed while computing the index, we can use this argument to support the claim. □

Lemma 6.6. *Computing the index for $G(V, E)$ using SpGEMM requires $\Omega(|V'|((d_2 - d_1)^2 + \log d_1))$ time.*

Proof. Let Adj' be the adjacency matrix associated with $G'(V', E')$. To prove the above claim, it is useful to visualize the structure of Adj' (Figure 6.2). Define a constant $k \ll |V'|$. For simplicity, assume d_1 and $d_2 - d_1$ are powers of 2. Throughout the index computation, the time required by Gustavson's SpGEMM algorithm is dictated by $bitops$. Following Lemma 6.4, multiplying Adj'^k to Adj'^k requires $\Theta(|V'|)$ time. In addition, multiplying $(Adj' \vee I)^k$ with $(Adj' \vee I)^k$ requires $\Theta(|V'|(k^2))$ time. Therefore, we need $\Theta(|V'|(\underbrace{1 + 1 + \ldots + 1}_{\log d_1 \text{ times}}))$ time to compute Adj'^{d_1}, and $\Theta(|V'|(1+2^2+4^2+\ldots+(d_2-d_1)^2))$ time to compute $(Adj' \vee I)^{d_2-d_1}$. The final multiplication between Adj'^{d_1} and $(Adj' \vee I)^{d_2-d_1}$ uses $\Theta(|V'|(d_2 - d_1))$ time. All these operations add up to $\Theta(|V'|((d_2 - d_1)^2 + \log d_1))$ time. This argument, and Lemma 6.5 suffice to support the claim. □

Remark 6.3. *The index size is dictated by count of non-zeros in the final output matrix.*

Using similar arguments as above, it can be shown that the index storage for graph $G(V, E)$ requires $\Omega(|V'|(d_2 - d_1 + 1))$ space.

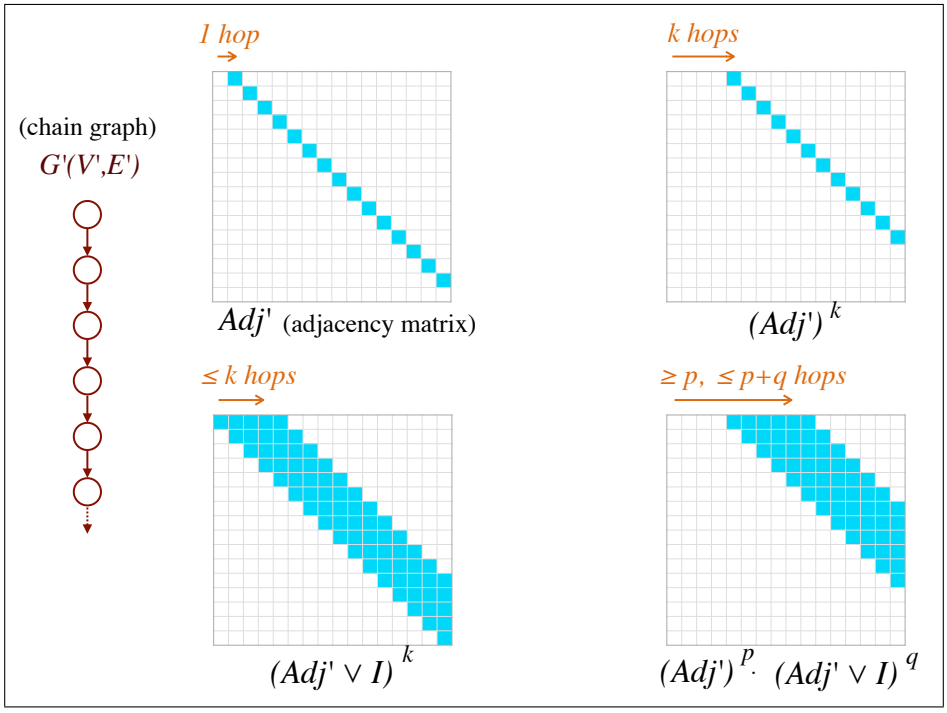

Figure 6.2: Visualizing non-zero structure of adjacency matrix of a chain graph. We also show how the structure changes after exponentiation. This is useful to count $bitops$ during SpGEMM.

Querying the Index

Querying for a value in a sorted-CSR formatted index is trivial. The lookup procedure for $\mathcal{T}[i, j]$ requires a binary search among the non-zeros of row i. Let $maxRownnz(\mathcal{T})$ be the maximum row size, i.e., maximum number of non-zero entries in a row of index matrix $\mathcal{T}$. After computing the index, deciding the existence of a path of length $d \in [d_1, d_2]$ between two vertices in $G(V, E)$ requires $O(\log maxRownnz(\mathcal{T}))$ time.

6.5 Results

We implemented our algorithm, referred to here as PairG, in C++. The source code is available at https://github.com/ParBLiSS/PairG. We conducted our evaluation using an Intel

Xeon CPU E5-2680 v4, equipped with 28 physical cores and 256 GB main memory. In our implementation, we utilize KokkosKernels [137], an open-source parallel library for basic linear algebra (BLAS) routines. This library does not provide explicit support for boolean matrix operations, so we used integer matrix operations instead, while rounding the non-zero output values to one. We leveraged multi-threading support in KokkosKernels, and allowed it to use 28 threads during execution. Our benchmark data sets, summarized below, consist of cyclic and acyclic graphs built using publicly available real data. We tested indexing and querying performance using PairG for various values of distance constraints. These choices were motivated by the typical insert sizes used for Illumina paired-end sequencing. We demonstrate that PairG can index graphs with more than a million vertices in a reasonable time. Once the index is built, it can answer a million distance constraint queries in a fraction of a second.

6.5.1 Datasets

We generated seven sequence graphs, four acyclic (G1-G4) and three cyclic (G5-G7) (see Table 6.2). The first four sequence graphs are variation graphs built using human genome segments (GRCh37) and variant files from the 1000 Genomes Project (Phase 3) [4]. We used vg [32] for building these graphs. The human genomic regions considered in our evaluation are mitochondrial DNA (mtDNA), BRCA1 gene, the killer cell immunoglobulin-like receptors (LRC_KIR), and the major histocompatibility complex (MHC). The sizes of these regions range from 16.6 kilobases (mtDNA) to 5.0 megabases (MHC) in the human genome. De Bruijn Graph (DBG) is another popular format to represent 'pan-genome' of a species. DBG is also a good candidate structure to test our algorithm on more complex graphs. We built DBGs with k-mer length 25 using whole-genome sequences of one (G5), five (G6), and twenty (G7) B. *anthracis* strains, using SplitMEM [138]. Strain ids and sizes of these genomes are listed in Table 6.1.

We tested PairG using three different ranges of distance constraints, associated with

Table 6.1: List of 20 *Bacillus anthracis* strains used to build the sequence graphs G5-G7. We used the first strain in G5, the first five strains in G6, and all the 20 strains in G7.

Strain id	Assembly id	Size (Mbp)
Ames	GCA_000007845.1	5.23
delta Sterne	GCA_000742695.1	5.23
Cvac02	GCA_000747335.1	5.23
Parent1	GCA_001683095.1	5.23
PR09-1	GCA_001683255.1	5.23
Sterne	GCA_000008165.1	5.23
CNEVA-9066	GCA_000167235.1	5.49
A1055	GCA_000167255.1	5.37
A0174	GCA_000182055.1	5.29
Sen2Col2	GCA_000359425.1	5.17
SVA11	GCA_000583105.1	5.49
95014	GCA_000585275.1	5.34
Smith 1013	GCA_000742315.1	5.29
2000031021	GCA_000742655.1	5.33
PAK-1	GCA_000832425.1	5.40
Pasteur	GCA_000832585.1	5.23
PR06	GCA_001683195.1	5.23
55-VNIIVViM	GCA_001835485.1	5.35
Sterne 34F2	GCA_002005265.1	5.39
UT308	GCA_003345105.1	5.37

Table 6.2: Directed sequence graphs used for evaluation. In these graphs, each vertex is labeled with a DNA nucleotide. Four acyclic graphs are derived from segments of human genome and variant files from the 1000 Genomes Project (Phase 3). Three cyclic graphs are de Bruijn graphs built using whole-genome sequences of *Bacillus anthracis* strains, with k-mer length 25.

Id	Graph	# vertices	# edges	Type
G1	mitochondrial-DNA	21,038	26,842	acyclic
G2	BRCA1	83,268	85,422	
G3	LRC_KIR	1,108,769	1,154,046	
G4	MHC	5,138,913	5,318,639	
G5	B. *anthracis* (1 strain)	5,174,432	5,175,000	cyclic
G6	B. *anthracis* (5 strains)	10,360,296	10,363,334	
G7	B. *anthracis* (20 strains)	11,237,067	11,253,437	

three insert-size configurations. Assuming a typical sequencing scenario of insert-size configuration as 300 bp and read length 100 bp, the inner distance between paired-end reads should equal 100 bp (i.e., insert size minus twice the read length). To allow for sufficient variability, we tested PairG using $d_1 = 0, d_2 = 250$. Similarly, for insert-size configurations of 500 bp and 700 bp, we tested PairG using inner distance limits $(d_1 = 150, d_2 = 450)$ and $(d_1 = 350, d_2 = 650)$, respectively. There may be insert size configurations where allowing read overlaps may also be necessary; this can be handled trivially by computing a second matrix with desirable limit on the overlap length.

6.5.2 Index construction

We report wall-clock time, memory-usage, and index size (nnz) using various graphs and distance constraints in Table 6.3. We note large variation in performance for various graph sizes and complexity. Time, memory, and index size increase almost linearly with increasing graph size. This is also expected based on our theoretical analysis (Section 6.4.1). The specified range of distance constraints $[d_1, d_2]$ can also affect performance. The index size for graphs G1-G5 remains almost uniform for the three different constraints. This should be because the first five graphs have linear chain-like topology, where the performance should be dictated by the gap $(d_2 - d_1)$ according to our analysis. The graphs G1-G4 were built by augmenting the variations (substitutions, indels) on the reference sequence, and the fifth graph uses a single strain. On the other hand, index size varies with distance constraints for the last two graphs, especially G7. We expect G7 to have significantly more branching due to complex structural variations and repeats. Another important contributing factor is that DBGs collapse repetitive k-mers into a single vertex, causing large deviation from the linear topology. As a result, it takes time ranging from 1.3 hours to 17 hours for graph G7.

For larger graphs, we expect the index size (nnz) to increase with graph size. In sorted-CSR storage format, we use 4 bytes for each non-zero, which can become prohibitively large at the scale of complete human genome. However, we expect substantial room for

Table 6.3: Performance measured in terms of wall-clock time and memory-usage for building index matrix using all input graphs and different distance constraints. *nnz* represents number of non-zero elements in the index matrix, to indicate its size. Our implementation uses 4 bytes to store each non-zero of a matrix in memory.

Id	Distance constraints								
	[0 - 250]			[150 - 450]			[350 - 650]		
	Time	Mem	*nnz*	Time	Mem	*nnz*	Time	Mem	*nnz*
G1	0.2s	0.1G	6.8M	0.4s	0.2G	7.8M	0.4s	0.2G	7.6M
G2	0.4s	0.3G	21M	0.9s	0.5G	26M	0.9s	0.5G	26M
G3	5.6s	3.8G	0.3B	12s	6.2G	0.3B	12s	6.2G	0.3B
G4	25s	17G	1.3B	53s	28G	1.6B	53s	28G	1.6B
G5	25s	17G	1.3B	54s	28G	1.6B	54s	28G	1.7B
G6	52s	35G	2.8B	2m	60G	3.6B	2m	60G	4.2B
G7	1.3h	56G	4.6B	7.2h	118G	8.9B	17.2h	129G	14B

compressing the final index, and plan to explore it in the future. Memory-usage of a matrix operation (addition or multiplication) is dictated by the size of the associated input and output matrices. As a result, memory required for the index construction appears to increase proportionally with the index size (Table 6.3).

6.5.3 Querying Performance

While indexing is a one-time routine for a sequence graph, we would need to query the index numerous times during the mapping process. Querying for a vertex pair requires a simple and fast lookup in the index (Section 6.4.1). As read mapping locations are expected to be uniformly distributed over the graph, we tested the querying performance by generating a million random vertex pairs $(u, v), u, v \in [1, |V|]$. For all the seven graphs, querying a million vertex pairs finished in less than a second (Table 6.4). Even though the majority of randomly generated queries result in a 'no' answer, this aspect has insignificant effect on the query performance. Our results implicate that distance constraints can be validated exactly without additional overhead on the mapping time, which is similar to the case of mapping reads to a reference sequence.

We also compared our index-based algorithm using a breadth first search (BFS)-based heuristic. Using this heuristic, we answer a query of a pair of vertices (u, v) as true if and

Table 6.4: Time to execute a million queries using all the graphs and distance constraints. Each query is a random pair of vertices in the graph.

Id	Distance constraints		
	[0 - 250]	[150 - 450]	[350 - 650]
	Time (sec)		
G1	0.1	0.1	0.1
G2	0.2	0.2	0.2
G3	0.4	0.4	0.5
G4	0.5	0.5	0.5
G5	0.4	0.5	0.5
G6	0.4	0.5	0.5
G7	0.5	0.5	0.6

only if v is reachable within d_2 distance from u. Accordingly, BFS initiated from vertex u is terminated if we find vertex v, or after we have explored all vertices up to depth d_2. This heuristic does not require a pre-computed index. However, we find that querying time using our index-based algorithm is faster by two to three orders of magnitude. For the randomly generated query set, fraction of results that agree between the two approaches varied from 98% to 100%. The above heuristic yields incorrect result for a vertex pair (u, v) when all the possible paths that connect u to v have length $< d_1$.

6.6 Summary

In this chapter, we formulated the Paired-end Validation Problem, required for paired-end read mapping on sequence graphs. We proposed the first provably good and practically useful exact algorithm for solving this problem. The proposed algorithm builds on top of existing SpGEMM algorithms, to exploit the sparsity and large diameter characteristics of sequence graphs. Our experiments indicate that index construction time is affected by the size and topology of the sequence graph, as well as the desired distance constraints. The querying time is less than a second for answering a million distance queries using all test cases.

CHAPTER 7
CONCLUSIONS

The advancement of sequencing technologies poses concomitant challenges to the development of computational methods for sequencing data processing. In this dissertation, we present efficient methods that can significantly reduce the runtime of read mapping, an essential step in sequencing data analysis, and expand its applications to newer technologies and reference protocols. The methods presented in the dissertation can take various types of reads as input and map them to either linear or graph-based reference genomes.

Specifically, we first developed Chromap to map short reads generated by chromatin profiling protocols. We demonstrated the empirically superior performance of Chromap on mapping ChIP-seq, Hi-C, and scATAC-seq profiles compared with other general-purpose aligners. Chromap also incorporates other short read preprocessing steps such as adapter trimming, alignment deduplication, and barcode correction, which ease users' burden in installing and tuning multiple tools to process chromatin profiling data.

Subsequently, we developed Sigmap to map nanopore raw reads for real-time targeted sequencing. Unlike other mapping methods that convert raw reads to sequences, Sigmap is the first method that fully works in signal space, which is promising for completely base-calling-free nanopore sequencing data analysis. The faster mapping speed might also support real-time targeted sequencing on ONT sequencing devices with more pores and higher sequencing throughput.

Furthermore, we designed two algorithms to align sequences to sequence graphs, which is the key for mapping reads to graph-based reference genomes. When arbitrary score linear/affine gap functions are used, we proposed an algorithm that improves the existent sequence to graph alignment algorithms on time complexity. Moreover, we developed a method termed Gwfa (Graph wave front alignment) to find the optimal global sequence-

to-graph alignment with unit edit cost and present heuristics to accelerate the alignment process further. We showed that Gwfa was orders of magnitude faster than other sequence-to-graph alignment algorithms on various input query sequences and graphs built from highly polymorphic regions that play essential roles in the human immune systems.

Finally, we presented the first mathematical formulation of the paired-end read mapping validation problem and proposed an exact algorithm to solve it. The method is designed to leverage the high sparsity of reference genome graphs and build an index that can be queried efficiently by mapping algorithms to validate distance constraints. We showed that our algorithm could validate a million paired-end read distance constraints on real reference genome graphs in less than a second, which is effective for practical usage.

We believe open-source tools would greatly facilitate biological and medical research. Thus, we implemented the methods presented in this dissertation and made them available as open-source software with the following links:

- Chromap: https://github.com/haowenz/chromap
- Sigmap: https://github.com/haowenz/sigmap
- SGA, Gwfa: https://github.com/haowenz/SGA, https://github.com/lh3/gwfa
- PairG: https://github.com/ParBLiSS/PairG

Appendices

APPENDIX A
FAST SEQUENCE TO GRAPH ALIGNMENT USING GRAPH WAVEFRONT ALGORITHM

A.1 Sequence to graph alignment problem variants

We extend the definition of a walk in the sequence graph by associating a start offset so and an end offset eo with the start vertex v_{j_1} and end vertex v_{j_n} in the walk w, respectively. And we use $w_{j_1,j_n}(so, eo)$ to denote a walk that skips a prefix and a suffix in the start vertex and end vertex respectively, which spells $\sigma(w_{j_1,j_n}(so, eo)) = \sigma(v_{j_1})[so, |v_{j_1}|]\sigma(v_{j_2})[1, |v_{j_2}|] \dots \sigma(v_{j_n})[1, eo]$.

Definition A.1 (Optimal global sequence to graph extension)**.** *Given a query sequence q, a sequence graph $G(V, E, \sigma)$ and a start vertex $v_s \in V$, find an end vertex v_e and a walk $w_{s,e}(1, \cdot)$ such that $\forall w'_{s,\cdot}(1, \cdot)$, $d(\sigma(w), q) \leq d(\sigma(w'), q)$.*

This formulation is similar to the global extension (or prefix, SHW) problem for pairwise sequence alignment. Gaps at the end of the target sequence (or the walk in the sequence graph) are not penalized. We can slightly modify our proposed algorithms to solve this problem variant in Algorithm A.1 and Algorithm A.2.

Definition A.2 (Optimal semi-global sequence to graph extension)**.** *Given a query sequence q, a sequence graph $G(V, E, \sigma)$ and a start vertex $v_s \in V$, find an end vertex v_e with an end offset eo and a walk $w_{s,e}(so, eo)$ such that $\forall w'_{s,\cdot}(., .)$, $d(\sigma(w), q) \leq d(\sigma(w'), q)$.*

This formulation is similar to the semi-global extension (or infix, HW) problem for pairwise sequence alignment. Gaps at the start and the end of the target sequence (or the walk in the sequence graph) are not penalized. This problem can be solved by setting $\tilde{H}_{v_s,j_s} = 0$ for $0 \leq j_s \leq |\sigma(v_s)|$ and push all these diagonals into the queue on line 4 of Algorithm A.1.

Algorithm A.1: Graph wavefront algorithm to find the optimal global sequence to graph extension

Input: Query sequence q, sequence graph $G = (V, E, \sigma)$, start vertex $v_s \in V$.

function GWFEDITDIST(q, G, v_s, v_e) **begin**
  $k_e \leftarrow |q| - |\sigma(v_e)|$
  $\tilde{H}_{v_s,0} \leftarrow 0$
  $Q \leftarrow [(v_s, 0)]$
  $d \leftarrow 0$
  $terminate \leftarrow false$
  while true do
    GWFEXTEND$(q, G, Q, \tilde{H}, terminate)$
    if $terminate$ **then**
      return d
    $d \leftarrow d + 1$
    GWFEXPAND$(q, G, Q, \tilde{H})$

Algorithm A.2: Graph wavefront extension algorithm

Input: Query sequence q, sequence graph $G = (V, E, \sigma)$, queue Q to keep track of diagonals, graph wavefront set $\tilde{H}$.

function GWFEXTEND$(q, G, Q, \tilde{H}, terminate)$ **begin**
  $Q' \leftarrow [\,]$
  while Q *is not empty* **do**
    $(v, k) \leftarrow Q.pop()$
    $j \leftarrow \tilde{H}_{v,k}$
    $i \leftarrow k + j$
    $l \leftarrow LCP(q[i + 1, |q|], \sigma(v)[j + 1, |\sigma(v)|])$
    $\tilde{H}_{v,k} \leftarrow j + l$
    $Q'.push(v, k)$
    $i \leftarrow i + l$
    if $i = |q|$ **then**
      $terminate \leftarrow true$
      return
    if $\tilde{H}_{v,k} = |\sigma(v)|$ **then**
      for $(v, u) \in E$ **do**
        if $\tilde{H}_{u,k+|\sigma(v)|} \notin \tilde{H}$ **then**
          $\tilde{H}_{u,k+|\sigma(v)|} \leftarrow 0$
          $Q.push(u, k + |\sigma(v)|)$
  $Q \leftarrow Q'$

A.2 Generalized Navarro's algorithm

We present the generalized Navarro's algorithm that can find sequence alignments to a segment-labeled graph in Algorithm A.3.

Algorithm A.3: Generalized Navarro's algorithm to find the optimal global sequence to graph alignment

Input: Query sequence q, sequence graph $G = (V, E, \sigma)$, start vertex $v_s \in V$ and end vertex $v_e \in V$.

function NAVARROALGORITHM(q, G, v_s, v_e) **begin**
  $C_{0,v_s,0} = 0$
  for $i \leftarrow 0$ *to* $|q|$ **do**
    for $v \in V$ **do**
      if $i \geq 1$ **then**
        for $j \leftarrow 1$ *to* $|\sigma(v)|$ **do**
          $C_{i,v,j} \leftarrow \min\{C_{i-1,v,j} + 1, C_{i-1,v,j-1} + \Delta_{i,v,j}\}$
      for $(v, u) \in E$ **do**
        $C_{i,u,0} \leftarrow \min\{C_{i,u,0}, C_{i,v,|\sigma(v)|}\}$
    for $v \in V$ **do**
      PROPAGATE(v, C_i)
  return $C_{|q|,v_e,|\sigma(|v_e|)|}$
function PROPAGATE(v, C_i) **begin**
  for $j \leftarrow 1$ *to* $|\sigma(v)|$ **do**
    if $C_{i,v,j} > C_{i,v,j-1} + 1$ **then**
      $C_{i,v,j} \leftarrow C_{i,v,j-1} + 1$
    else
      break
  for $(v, u) \in E$ **do**
    if $C_{i,u,0} > C_{i,v,|\sigma(v)|} + 1$ **then**
      $C_{i,u,0} \leftarrow C_{i,v,|\sigma(v)|} + 1$
      PROPAGATE(u, C_i)

A.3 Sequence to graph alignment traceback

We show how to adapt our proposed algorithm to trace the optimal alignment walk in Algorithm A.4 and Algorithm A.5.

Algorithm A.4: Graph wavefront algorithm to find the optimal global sequence to graph alignment and trace back the alignment walk

Input: Query sequence q, sequence graph $G = (V, E, \sigma)$, start vertex $v_s \in V$ and end vertex $v_e \in V$.

function GWFEDITDIST(q, G, v_s, v_e) **begin**
  $k_e \leftarrow |q| - |\sigma(v_e)|$
  $\tilde{H}_{v_s,0} \leftarrow 0$
  $\tilde{T}_{v_s,0} \leftarrow -1$
  $Q \leftarrow [(v_s, 0)]$
  $d \leftarrow 0$
  $T \leftarrow [\,]$
  $M \leftarrow \emptyset$
  while true do
    GWFEXTEND($q, G, Q, \tilde{H}, \tilde{T}, T, M$)
    if $\tilde{H}_{v_e,k_e} \geq |\sigma(v_e)|$ **then**
      break
    $d \leftarrow d + 1$
    GWFEXPAND($q, G, Q, \tilde{H}$)
  GWFTRACEBACK($\tilde{T}_{d,v_e,|\sigma(v_e)|}, T$)
  return d

Algorithm A.5: Graph wavefront extension algorithm that saves traceback information

Input: Query sequence q, sequence graph $G = (V, E, \sigma)$, queue Q to keep track of diagonals, graph wavefront set $\tilde{H}$, traceback information $\tilde{T}, T, M$.

function GWFEXTEND($q, G, \tilde{H}, \tilde{T}, T, M$)) **begin**
  $Q' \leftarrow [\,]$
  while Q *is not empty* **do**
    $(v, k) \leftarrow Q.pop()$
    $j \leftarrow \tilde{H}_{v,k}$
    $i \leftarrow k + j$
    $l \leftarrow LCP(q[i + 1, |q|], \sigma(v)[j + 1, |\sigma(v)|])$
    $\tilde{H}_{v,k} \leftarrow j + l$
    $Q'.push(v, k)$
    if $\tilde{H}_{v,k} = |\sigma(v)|$ **then**
      for $(v, u) \in E$ **do**
        if $\tilde{H}_{u,k+|\sigma(v)|} \notin \tilde{H}$ **then**
          $\tilde{H}_{u,k+|\sigma(v)|} \leftarrow 0$
          $\tilde{T}_{u,k+|\sigma(v)|} \leftarrow$ GWFPUSHTRACE($u, \tilde{T}_{v,k}, T, M$)
          $Q.push(u, k + |\sigma(v)|)$
  $Q \leftarrow Q'$

A.4 Details on sequence graph construction

The scripts to construct the sequence graphs and retrieve the query sequences are available at https://github.com/haowenz/gwfa_evaluation. We also made the queries and graphs available on Zenodo with doi 10.5281/zenodo.6622036 for download.

A.5 Versions and parameters to run sequence to graph alignment tools

GraphAligner v1.0.15 was used as GraphAligner-heuristic, and "-x vg" was the parameter. The PaperExperiments branch from GraphAligner GitHub repo (https://github.com/maickrau/GraphAligner/tree/PaperExperiments) was used as GraphAligner-bitvector, which was also used in the evaluation in the GraphAlinger bit-vector algorithm [104].

A.6 Versions and parameters of the pairwise sequence alignment tools

For Edlib, we used version 1.2.7 and parameters '-m SHW'. For WFA2, version 2.1 was used, and the parameters were set using the code as follows.

```
wavefront_aligner_attr_t attributes =
   wavefront_aligner_attr_default;
attributes.heuristic.strategy = wf_heuristic_none;
attributes.distance_metric = edit;
attributes.alignment_scope = compute_score;
attributes.alignment_form.span = alignment_endsfree;
attributes.alignment_form.pattern_begin_free = 0;
attributes.alignment_form.pattern_end_free = 0;
attributes.alignment_form.text_begin_free = 0;
attributes.alignment_form.text_end_free = min(text_ks->seq.l,
   pattern_ks->seq.l / 2;
```

A.7 Comparison with pairwise sequence alignment tools

To investigate the performance gap between aligning a query to a target sequence and a sequence graph, we constructed a linear graph with one vertex, and the C4 region of GRCh38 is the vertex segment. Note that the total segment length of the linear graph is much larger than G1, though they both represent the C4 region. This is because a large segmental duplication is collapsed as a cycle in G1, which reduces its total segment length. Table A.1 shows the runtime of the methods to align the queries to the linear genome or the sequence graphs.

We observed that running Gwfa and Gwfa-pruning to align the C4 regions of HG002 and HG005 to the linear C4 sequence graph was as fast as running WFA2 and even faster than running Edlib to perform pairwise sequence alignment of the C4 regions, which means our implementation is efficient. However, Gwfa and Gwfa-pruning were slower than Edlib and WFA2 on aligning C4 regions of NA19240 haplotypes and GRCh38. This was caused by a structural variant in the C4 regions of NA19240 haplotypes, which leads to a high edit distance and a large graph wavefront set. Since the range of graph wavefront sets in general cannot be easily maintained by only one interval as how the range of wavefront set for pairwise sequence alignment is represented, Gwfa as a sequence to graph alignment method would cost more computing time on this special use case.

Besides, when aligning queries to the linear C4 sequence graph, Gwfa and Gwfa-pruning were orders of magnitude faster than other sequence to graph alignment algorithms, except for the case when aligning NA19240 C4 regions to the linear C4 sequence graph. Though GraphAligner heuristic was faster than Gwfa or Gwfa-prunning in this case, the alignments reported by GraphAligner heuristic were broken into several pieces with alternative alignments due to the repeats in the C4 region. In constrast, all the other sequence to graph alignment methods were able to report the optimal alignments.

Table A.2 shows the memory usage of the tools. Gwfa-pruning used the least amount

of memory except the case when aligning the C4 regions of NA19240 haplotypes. The reason is the same as the reason for the longer runtime, which is the high edit distance of the alignments between the C4 regions of these two haplotypes and GRCh38.

Table A.1: Runtime (in seconds) of the methods to align queries to the linear sequence graphs.

Haplotype	Edit distance	Edlib	WFA2	Gwfa	Gwfa-pruning	Dijkstra's algorithm	Navarro's algorithm	GraphAligner bitvector	GraphAligner heuristic
HG002.1	89	0.03	**<0.01**	**<0.01**	**<0.01**	3.06	24.84	3.56	0.57
HG002.2	90	0.03	**<0.01**	**<0.01**	**<0.01**	3.08	24.82	3.56	0.15
HG005.1	24	0.01	**<0.01**	**<0.01**	**<0.01**	0.54	24.85	3.73	0.22
HG005.2	86	0.01	**<0.01**	**<0.01**	**<0.01**	2.97	24.85	3.73	0.13
NA19240.1	6,454	**0.15**	**0.15**	0.73	0.81	633.76	23.37	3.43	0.20
NA19240.2	6,673	**0.16**	**0.16**	0.78	0.83	740.73	23.28	3.41	0.18

Table A.2: Memory usage (MB) of the methods to align queries to the linear sequence graphs.

Haplotype	Edit distance	Edlib	WFA2	Gwfa	Gwfa-pruning	Dijkstra's algorithm	Navarro's algorithm	GraphAligner bitvector	GraphAligner heuristic
HG002.1	89	3.9	3.4	2.0	**1.5**	459.3	164.6	77.2	44.0
HG002.2	90	3.9	3.4	2.0	**1.5**	460.0	164.6	77.2	43.8
HG005.1	24	3.9	3.4	**1.3**	**1.3**	244.7	164.6	77.1	41.8
HG005.2	86	3.9	3.4	2.0	**1.5**	453.6	164.6	77.1	44.3
NA19240.1	6,454	3.9	**3.4**	9.3	9.3	26,912.3	164.6	77.1	36.6
NA19240.2	6,673	3.9	**3.4**	9.3	9.3	32,344.7	164.6	77.1	38.1

REFERENCES

[1] E. Lander *et al.*, "Initial sequencing and analysis of the human genome," *Nature*, vol. 409, no. 6822, pp. 860–921, 2001.

[2] J. C. Venter *et al.*, "The sequence of the human genome," *science*, vol. 291, no. 5507, pp. 1304–1351, 2001.

[3] A. Rhie *et al.*, "Towards complete and error-free genome assemblies of all vertebrate species," *Nature*, vol. 592, no. 7856, pp. 737–746, 2021.

[4] 1. G. P. Consortium *et al.*, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, p. 68, 2015.

[5] J. Shendure *et al.*, "Dna sequencing at 40: Past, present and future," *Nature*, vol. 550, no. 7676, pp. 345–353, 2017.

[6] T. Wang *et al.*, "The human pangenome project: A global resource to map genomic diversity," *Nature*, vol. 604, no. 7906, pp. 437–446, 2022.

[7] A. Barski *et al.*, "High-resolution profiling of histone methylations in the human genome," *Cell*, vol. 129, no. 4, pp. 823–837, 2007.

[8] J. D. Buenrostro, P. G. Giresi, L. C. Zaba, H. Y. Chang, and W. J. Greenleaf, "Transposition of native chromatin for fast and sensitive epigenomic profiling of open chromatin, dna-binding proteins and nucleosome position," *Nature methods*, vol. 10, no. 12, pp. 1213–1218, 2013.

[9] E. Lieberman-Aiden *et al.*, "Comprehensive mapping of long-range interactions reveals folding principles of the human genome," *science*, vol. 326, no. 5950, pp. 289–293, 2009.

[10] P. J. Farnham, "Insights from genomic profiling of transcription factors," *Nature Reviews Genetics*, vol. 10, no. 9, pp. 605–616, 2009.

[11] J. D. Buenrostro, B. Wu, H. Y. Chang, and W. J. Greenleaf, "Atac-seq: A method for assaying chromatin accessibility genome-wide," *Current protocols in molecular biology*, vol. 109, no. 1, pp. 21–29, 2015.

[12] J. R. Dixon *et al.*, "Topological domains in mammalian genomes identified by analysis of chromatin interactions," *Nature*, vol. 485, no. 7398, pp. 376–380, 2012.

[13] S. S. Rao *et al.*, "A 3d map of the human genome at kilobase resolution reveals principles of chromatin looping," *Cell*, vol. 159, no. 7, pp. 1665–1680, 2014.

[14] J. D. Buenrostro *et al.*, "Single-cell chromatin accessibility reveals principles of regulatory variation," *Nature*, vol. 523, no. 7561, pp. 486–490, 2015.

[15] K. H. Miga *et al.*, "Telomere-to-telomere assembly of a complete human x chromosome," *Nature*, vol. 585, no. 7823, pp. 79–84, 2020.

[16] D. R. Garalde *et al.*, "Highly parallel direct rna sequencing on an array of nanopores," *Nature methods*, vol. 15, no. 3, p. 201, 2018.

[17] J. T. Simpson, R. E. Workman, P. Zuzarte, M. David, L. Dursi, and W. Timp, "Detecting dna cytosine methylation using nanopore sequencing," *Nature methods*, vol. 14, no. 4, pp. 407–410, 2017.

[18] F. J. Rang, W. P. Kloosterman, and J. de Ridder, "From squiggle to basepair: Computational approaches for improving nanopore sequencing read accuracy," *Genome biology*, vol. 19, no. 1, p. 90, 2018.

[19] J. Quick *et al.*, "Real-time, portable genome sequencing for ebola surveillance," *Nature*, vol. 530, no. 7589, pp. 228–232, 2016.

[20] M. Wang *et al.*, "Nanopore targeted sequencing for the accurate and comprehensive detection of sars-cov-2 and other respiratory viruses," *Small*, vol. 16, no. 32, p. 2 002 169, 2020.

[21] T. Gilpatrick *et al.*, "Targeted nanopore sequencing with cas9-guided adapter ligation," *Nature biotechnology*, vol. 38, no. 4, pp. 433–438, 2020.

[22] H. S. Edwards, R. Krishnakumar, A. Sinha, S. W. Bird, K. D. Patel, and M. S. Bartsch, "Real-time selective sequencing with rubric: Read until with basecall and reference-informed criteria," *Scientific Reports*, vol. 9, no. 1, pp. 1–11, 2019.

[23] A. Payne, N. Holmes, T. Clarke, R. Munro, B. J. Debebe, and M. Loose, "Readfish enables targeted nanopore sequencing of gigabase-sized genomes," *Nature Biotechnology*, pp. 1–9, 2020.

[24] S. Kovaka, Y. Fan, B. Ni, W. Timp, and M. C. Schatz, "Targeted nanopore sequencing by real-time mapping of raw electrical signal with uncalled," *Nature Biotechnology*, pp. 1–11, 2020.

[25] K. Shafin *et al.*, "Nanopore sequencing and the shasta toolkit enable efficient de novo assembly of eleven human genomes," *Nature biotechnology*, vol. 38, no. 9, pp. 1044–1053, 2020.

[26] 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*, vol. 18, no. 2, pp. 170–175, 2021.

[27] S. Nurk *et al.*, "The complete sequence of a human genome," *Science*, vol. 376, no. 6588, pp. 44–53, 2022.

[28] "Computational pan-genomics: Status, promises and challenges," *Briefings in bioinformatics*, vol. 19, no. 1, pp. 118–135, 2018.

[29] H. Li, X. Feng, and C. Chu, "The design and construction of reference pangenome graphs with minigraph," *Genome biology*, vol. 21, no. 1, pp. 1–19, 2020.

[30] J. M. Eizenga *et al.*, "Pangenome graphs," *Annual review of genomics and human genetics*, vol. 21, pp. 139–162, 2020.

[31] B. Paten, A. M. Novak, J. M. Eizenga, and E. Garrison, "Genome graphs and the evolution of genome inference," *Genome research*, vol. 27, no. 5, pp. 665–676, 2017.

[32] E. Garrison *et al.*, "Variation graph toolkit improves read mapping by representing genetic variation in the reference," *Nature biotechnology*, 2018.

[33] J. Sirén *et al.*, "Pangenomics enables genotyping of known structural variants in 5202 diverse genomes," *Science*, vol. 374, no. 6574, abg8871, 2021.

[34] E. P. Consortium *et al.*, "An integrated encyclopedia of dna elements in the human genome," *Nature*, vol. 489, no. 7414, p. 57, 2012.

[35] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[36] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, pp. 357–359, 2012.

[37] H. Li *et al.*, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[38] *Picard toolkit*, https://broadinstitute.github.io/picard/, 2019.

[39] Y. Zhang *et al.*, "Model-based analysis of chip-seq (macs)," *Genome biology*, vol. 9, no. 9, pp. 1–9, 2008.

[40] H. Li, "Minimap2: Pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[41] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biological sequence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004.

[42] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[43] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM (JACM)*, vol. 46, no. 3, pp. 395–415, 1999.

[44] M. Holtgrewe, "Mason: A read simulator for second generation sequencing data," 2010.

[45] Y. Yan, N. Chaturvedi, and R. Appuswamy, "Accel-align: A fast sequence mapper and aligner based on the seed–embed–extend method," *BMC bioinformatics*, vol. 22, no. 1, pp. 1–20, 2021.

[46] G. Yu, L.-G. Wang, and Q.-Y. He, "Chipseeker: An r/bioconductor package for chip peak annotation, comparison and visualization," *Bioinformatics*, vol. 31, no. 14, pp. 2382–2383, 2015.

[47] J. Dekker *et al.*, "The 4d nucleome project," *Nature*, vol. 549, no. 7671, pp. 219–226, 2017.

[48] T. Yang *et al.*, "Hicrep: Assessing the reproducibility of hi-c data using a stratum-adjusted correlation coefficient," *Genome research*, vol. 27, no. 11, pp. 1939–1949, 2017.

[49] C. Wang *et al.*, "Integrative analyses of single-cell transcriptome and regulome using maestro," *Genome biology*, vol. 21, no. 1, pp. 1–28, 2020.

[50] J. M. Granja *et al.*, "Archr is a scalable software package for integrative single-cell chromatin accessibility analysis," *Nature genetics*, vol. 53, no. 3, pp. 403–411, 2021.

[51] M. Loose, S. Malla, and M. Stout, "Real-time selective sequencing using nanopore technology," *Nature methods*, vol. 13, no. 9, pp. 751–754, 2016.

[52] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM (JACM)*, vol. 52, no. 4, pp. 552–581, 2005.

[53] M. J. Chaisson and G. Tesler, "Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): Application and theory," *BMC bioinformatics*, vol. 13, no. 1, p. 238, 2012.

[54] I. Sović, M. Šikić, A. Wilm, S. N. Fenlon, S. Chen, and N. Nagarajan, "Fast and sensitive mapping of nanopore sequencing reads with graphmap," *Nature communications*, vol. 7, no. 1, pp. 1–11, 2016.

[55] F. J. Sedlazeck *et al.*, "Accurate detection of complex structural variations using single-molecule sequencing," *Nature methods*, vol. 15, no. 6, pp. 461–468, 2018.

[56] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[57] D.-T. Lee and C. Wong, "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees," *Acta Informatica*, vol. 9, no. 1, pp. 23–29, 1977.

[58] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[59] R. Han, S. Wang, and X. Gao, "Novel algorithms for efficient subsequence searching and mapping in nanopore raw signals towards targeted sequencing," *Bioinformatics*, vol. 36, no. 5, pp. 1333–1343, 2020.

[60] M. I. Abouelhoda and E. Ohlebusch, "Chaining algorithms for multiple genome comparison," *Journal of Discrete Algorithms*, vol. 3, no. 2-4, pp. 321–341, 2005.

[61] Y. Li, S. Wang, C. Bi, Z. Qiu, M. Li, and X. Gao, "Deepsimulator1.5: A more powerful, quicker and lighter simulator for nanopore sequencing," *Bioinformatics*, vol. 36, no. 8, pp. 2578–2580, 2020.

[62] D. E. Miller *et al.*, "Targeted long-read sequencing resolves complex structural variants and identifies missing disease-causing variants," *bioRxiv*, 2020.

[63] U. Manber and S. Wu, "Approximate string matching with arbitrary costs for text and hypertext," in *Advances In Structural And Syntactic Pattern Recognition*, World Scientific, 1992, pp. 22–33.

[64] G. Navarro, "Improved approximate pattern matching on hypertext," *Theoretical Computer Science*, vol. 237, no. 1-2, pp. 455–463, 2000.

[65] N. Nguyen *et al.*, "Building a pan-genome reference for a population," *Journal of Computational Biology*, vol. 22, no. 5, pp. 387–401, 2015.

[66] A. Dilthey, C. Cox, Z. Iqbal, M. R. Nelson, and G. McVean, "Improved genome inference in the MHC using a population reference graph," *Nature genetics*, vol. 47, no. 6, p. 682, 2015.

[67] H. P. Eggertsson *et al.*, "Graphtyper enables population-scale genotyping using pangenome graphs," *Nature genetics*, vol. 49, no. 11, p. 1654, 2017.

[68] D. Antipov, A. Korobeynikov, J. S. McLean, and P. A. Pevzner, "Hybridspades: An algorithm for hybrid assembly of short and long reads," *Bioinformatics*, vol. 32, no. 7, pp. 1009–1015, 2015.

[69] R. R. Wick, L. M. Judd, C. L. Gorrie, and K. E. Holt, "Unicycler: Resolving bacterial genome assemblies from short and long sequencing reads," *PLoS computational biology*, vol. 13, no. 6, e1005595, 2017.

[70] S. Garg, M. Rautiainen, A. M. Novak, E. Garrison, R. Durbin, and T. Marschall, "A graph-based approach to diploid genome assembly," *Bioinformatics*, vol. 34, no. 13, pp. i105–i114, 2018.

[71] L. Salmela and E. Rivals, "Lordec: Accurate and efficient long read error correction," *Bioinformatics*, vol. 30, no. 24, pp. 3506–3514, 2014.

[72] J. R. Wang, J. Holt, L. McMillan, and C. D. Jones, "Fmlrc: Hybrid long read error correction using an FM-index," *BMC bioinformatics*, vol. 19, no. 1, p. 50, 2018.

[73] H. Zhang, C. Jain, and S. Aluru, "A comprehensive evaluation of long read error correction methods," *bioRxiv*, p. 519 330, 2019.

[74] S. Beretta, P. Bonizzoni, L. Denti, M. Previtali, and R. Rizzi, "Mapping RNA-seq data to a transcript graph via approximate pattern matching to a hypertext," in *International Conference on Algorithms for Computational Biology*, Springer, 2017, pp. 49–61.

[75] A. Kuosmanen, T. Paavilainen, T. Gagie, R. Chikhi, A. Tomescu, and V. Mäkinen, "Using minimum path cover to boost dynamic programming on DAGs: Co-linear chaining extended," in *International Conference on Research in Computational Molecular Biology*, Springer, 2018, pp. 105–121.

[76] W. P. Rowe and M. D. Winn, "Indexed variation graphs for efficient and accurate resistome profiling," *Bioinformatics*, vol. 1, p. 8, 2018.

[77] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to DNA fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[78] A. M. Novak *et al.*, "Genome graphs," *bioRxiv*, p. 101 378, 2017.

[79] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl_2, pp. ii79–ii85, 2005.

[80] C. Lee, C. Grasso, and M. F. Sharlow, "Multiple sequence alignment using partial order graphs," *Bioinformatics*, vol. 18, no. 3, pp. 452–464, 2002.

[81] L. Huang, V. Popic, and S. Batzoglou, "Short read alignment with populations of genomes," *Bioinformatics*, vol. 29, no. 13, pp. i361–i370, 2013.

[82] B. Liu, H. Guo, M. Brudno, and Y. Wang, "Debga: Read alignment with de bruijn graph-based seed and extension," *Bioinformatics*, vol. 32, no. 21, pp. 3224–3232, 2016.

[83] A. Limasset, B. Cazaux, E. Rivals, and P. Peterlongo, "Read mapping on de bruijn graphs," *BMC bioinformatics*, vol. 17, no. 1, p. 237, 2016.

[84] M. Heydari, G. Miclotte, Y. Van de Peer, and J. Fostier, "Browniealigner: Accurate alignment of illumina sequencing data to de bruijn graphs," *BMC bioinformatics*, vol. 19, no. 1, p. 311, 2018.

[85] J. Sirén, N. Välimäki, and V. Mäkinen, "Indexing graphs for path queries with applications in genome research," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 11, no. 2, pp. 375–388, 2014.

[86] M. Rautiainen and T. Marschall, "Aligning sequences to general graphs in O(V+ mE) time," *bioRxiv*, p. 216 127, 2017.

[87] K. Vaddadi, K. Tayal, R. Srinivasan, and N. Sivadasan, "Sequence alignment on directed graphs," *Journal of Computational Biology*, 2018.

[88] A. Amir, M. Lewenstein, and N. Lewenstein, "Pattern matching in hypertext," *Journal of Algorithms*, vol. 35, no. 1, pp. 82–99, 2000.

[89] C. Thachuk, "Indexing hypertext," *Journal of Discrete Algorithms*, vol. 18, pp. 113–122, 2013.

[90] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[91] E. W. Myers, *An overview of sequence comparison algorithms in molecular biology*. University of Arizona. Department of Computer Science, 1991.

[92] A. Backurs and P. Indyk, "Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)," in *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, ACM, 2015, pp. 51–58.

[93] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.

[94] 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*, vol. 48, no. 3, pp. 443–453, 1970.

[95] E. W. Myers, "Ano (nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.

[96] E. Ukkonen, "Algorithms for approximate string matching," *Information and control*, vol. 64, no. 1-3, pp. 100–118, 1985.

[97] G. M. Landau and U. Vishkin, "Fast parallel and serial approximate string matching," *Journal of algorithms*, vol. 10, no. 2, pp. 157–169, 1989.

[98] H. Xin, J. Kim, S. Nahar, C. Alkan, and O. Mutlu, "Leap: A generalization of the landau-vishkin algorithm with custom gap penalties," *bioRxiv*, p. 133 157, 2017.

[99] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, "Fast gap-affine pairwise alignment using the wavefront algorithm," *Bioinformatics*, vol. 37, no. 4, pp. 456–463, 2021.

[100] J. M. Eizenga and B. Paten, "Improving the time and space complexity of the wfa algorithm and generalizing its scoring," *bioRxiv*, 2022.

[101] S. Marco-Sola, J. M. Eizenga, A. Guarracino, B. Paten, E. Garrison, and M. Moreto, "Optimal gap-affine alignment in o (s) space," *bioRxiv*, 2022.

[102] D. Antipov, A. Korobeynikov, J. S. McLean, and P. A. Pevzner, "Hybridspades: An algorithm for hybrid assembly of short and long reads," *Bioinformatics*, vol. 32, no. 7, pp. 1009–1015, 2016.

[103] V. N. S. Kavya, K. Tayal, R. Srinivasan, and N. Sivadasan, "Sequence alignment on directed graphs," *Journal of Computational Biology*, vol. 26, no. 1, pp. 53–67, 2019.

[104] M. Rautiainen, V. Mäkinen, and T. Marschall, "Bit-parallel sequence-to-graph alignment," *Bioinformatics*, vol. 35, no. 19, pp. 3599–3607, 2019.

[105] C. Jain, H. Zhang, Y. Gao, and S. Aluru, "On the complexity of sequence-to-graph alignment," *Journal of Computational Biology*, vol. 27, no. 4, pp. 640–654, 2020.

[106] M. Rautiainen and T. Marschall, "Graphaligner: Rapid and versatile sequence-to-graph alignment," *Genome biology*, vol. 21, no. 1, pp. 1–28, 2020.

[107] P. Ivanov, B. Bichsel, H. Mustafa, A. Kahles, G. Rätsch, and M. Vechev, "Astarix: Fast and optimal sequence-to-graph alignment," in *International Conference on Research in Computational Molecular Biology*, Springer, 2020, pp. 104–119.

[108] P. Ivanov, B. Bichsel, and M. Vechev, "Fast and optimal sequence-to-graph alignment guided by seeds," in *International Conference on Research in Computational Molecular Biology*, Springer, 2022, pp. 306–325.

[109] H. Zhang *et al.*, "Fast alignment and preprocessing of chromatin profiles with chromap," *Nature communications*, vol. 12, no. 1, pp. 1–6, 2021.

[110] M. Equi, R. Grossi, V. Mäkinen, A. Tomescu, *et al.*, "On the complexity of string matching for graphs," in *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[111] C. Jain, S. Misra, H. Zhang, A. Dilthey, and S. Aluru, "Accelerating sequence alignment to graphs," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2019, pp. 451–461.

[112] C. Jain, H. Zhang, A. Dilthey, and S. Aluru, "Validating paired-end read alignments in sequence graphs," *Leibniz international proceedings in informatics*, vol. 143, 2019.

[113] A. Guarracino, S. Heumos, S. Nahnsen, P. Prins, and E. Garrison, "Odgi: Understanding pangenome graphs," *bioRxiv*, 2021.

[114] R. Horton *et al.*, "Gene map of the extended human mhc," *Nature Reviews Genetics*, vol. 5, no. 12, pp. 889–899, 2004.

[115] A. D. Barrow and J. Trowsdale, "The extended human leukocyte receptor complex: Diverse ways of modulating immune responses," *Immunological reviews*, vol. 224, no. 1, pp. 98–123, 2008.

[116] M. Šošić and M. Šikić, "Edlib: A c/c++ library for fast, exact sequence alignment using edit distance," *Bioinformatics*, vol. 33, no. 9, pp. 1394–1395, 2017.

[117] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de bruijn graphs," in *International Workshop on Algorithms in Bioinformatics*, Springer, 2012, pp. 225–235.

[118] M. D. Muggli *et al.*, "Succinct colored de bruijn graphs," *Bioinformatics*, vol. 33, no. 20, pp. 3181–3187, 2017.

[119] J. Sirén, "Indexing variation graphs," in *2017 Proceedings of the ninteenth workshop on algorithm engineering and experiments (ALENEX)*, SIAM, 2017, pp. 13–27.

[120] C. Jain, H. Zhang, Y. Gao, and S. Aluru, "On the complexity of sequence to graph alignment," in *Research in Computational Molecular Biology*, Cham: Springer International Publishing, 2019, pp. 85–100.

[121] S. Canzar and S. L. Salzberg, "Short read mapping: An algorithmic tour," *Proceedings of the IEEE*, vol. 105, no. 3, pp. 436–458, 2015.

[122] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint arXiv:1303.3997*, 2013.

[123] A. Dilthey, P.-A. Gourraud, A. J. Mentzer, N. Cereb, Z. Iqbal, and G. McVean, "High-accuracy HLA type inference from whole-genome sequencing data using population reference graphs," *PLoS computational biology*, vol. 12, no. 10, e1005151, 2016.

[124] D. Kim, J. M. Paggi, and S. Salzberg, "Hisat-genotype: Next generation genomic analysis platform on a personal computer," *BioRxiv*, p. 266 197, 2018.

[125] T. O. Mokveld, J. Linthorst, Z. Al-Ars, and M. Reinders, "Chop: Haplotype-aware path indexing in population graphs," *bioRxiv*, 2018.

[126] G. Rakocevic *et al.*, "Fast and accurate genomic analyses using genome graphs," Nature Publishing Group, Tech. Rep., 2019.

[127] M. Rautiainen, V. Mäkinen, and T. Marschall, "Bit-parallel sequence-to-graph alignment," *Bioinformatics*, Mar. 2019.

[128] L. Denti, R. Rizzi, S. Beretta, G. Della Vedova, M. Previtali, and P. Bonizzoni, "Asgal: Aligning RNA-Seq data to a splicing graph to detect novel alternative splicing events," *BMC bioinformatics*, vol. 19, no. 1, p. 444, 2018.

[129] Computational Pan-Genomics Consortium, "Computational pan-genomics: Status, promises and challenges," *Briefings in bioinformatics*, vol. 19, no. 1, pp. 118–135, 2016.

[130] M. Nykänen and E. Ukkonen, "The exact path length problem," *Journal of Algorithms*, vol. 42, no. 1, pp. 41–53, 2002.

[131] L. Salmela, K. Sahlin, V. Mäkinen, and A. I. Tomescu, "Gap filling as exact path length problem," *Journal of Computational Biology*, vol. 23, no. 5, pp. 347–361, 2016.

[132] E. Nuutila, *Efficient transitive closure computation in large digraphs.* Finnish Academy of Technology, 1998.

[133] F. Le Gall, "Powers of tensors and fast matrix multiplication," in *Proceedings of the 39th international symposium on symbolic and algebraic computation*, ACM, 2014, pp. 296–303.

[134] A. Buluç, J. Gilbert, and V. B. Shah, "Implementing sparse matrices for graph algorithms," in *Graph Algorithms in the Language of Linear Algebra*, SIAM, 2011, pp. 287–313.

[135] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[136] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in MATLAB: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[137] M. Deveci, C. Trott, and S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2017, pp. 693–702.

[138] S. Marcus, H. Lee, and M. C. Schatz, "Splitmem: A graphical algorithm for pan-genome analysis with suffix skips," *Bioinformatics*, vol. 30, no. 24, pp. 3476–3483, 2014.