Algorithmic Foundations of Genome Graph Construction and Comparison

Yutong Qiu

CMU-CB-23-101
May 2023

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This material is based upon research supported by the National Institute of General Medical Sciences of the National Institutes of Health under award number R01GM122935, the National Human Genome Research Institute of the National Institutes of Health under award number R01HG012470, the National Science Foundation under award number 1937540, and the Carnegie Mellon University School of Computer Science Sansom Graduate Fellowship for computational cancer research.

Keywords: Genome graph, Graph theory, Algorithms, Linear programming, Pangenomics

For every encounter.

Abstract

Pangenomic studies have enabled a more accurate depiction of the human genome landscape. Genome graphs are suitable data structures for analyzing collections of genomes due to their efficiency and flexibility of encoding shared and unique substrings from the population of encoded genomes. Novel challenges arise when genome graphs are applied to thousands of genomes because current genome graph models are insufficient in addressing the questions: (1) How can genome graphs be constructed efficiently that optimize the storage space? (2) How can genome graphs be used to more accurately and more efficiently compare heterogeneous sequences such as cancer genomes or immune repertoires? To answer these questions, we lay algorithmic foundations for genome graph construction and comparison.

The size of a genome graph is crucial to both efficient storage and analysis. However, few genome graph construction methods directly optimize the graph size. By drawing connections to data compression, we develop an algorithmic framework for genome graph construction that prioritizes genome graph size and show that the new framework produces small genome graphs efficiently compared to other genome graph schemes. Our compression-based framework not only removes the dependency on hyper-parameters but also opens up the potential for adapting established compression algorithms to construct better genome graphs.

In many scenarios, such as immune repertoire analysis, we need to quantify the similarity between heterogeneous sets of genomic strings, but the complete strings are unknown due to limitations in sequencing technology. The distance between genome graphs can be used to estimate to the difference between these strings. One important metric is defined as the graph traversal edit distance (GTED). We revisit the complexity of and the previously proposed algorithms for GTED. We prove that GTED is NP-complete and show that the previously proposed algorithms computes a lower bound of GTED. In addition, we propose two correct ILP formulations of GTED and characterize the relationship between GTED and the previous lower bound ILPs. We evaluate the empirical efficiency of solving GTED and its lower bound ILP and show that solving GTED exactly with ILPs is currently not practical on larger genomes.

Genome graphs are often highly expressive and represent more than one string sets, and thus the distance between two graphs using standard graph distances does not always model the actual edit distance between true string sets. To quantify this discrepancy, we formally define genome graph expressiveness as its diameter and use it to bound the deviation of the genome graph distance from string set distances. We produce a more accurate distance measure between (unseen) collections of strings encoded as genome graphs. The new distance measure and its deviation from string set distances are evaluated on simulated human T-cell repertoire sequences and Hepatitis B virus genomes.

Acknowledgments

For countless moments during my life as a Ph.D. student, I felt extremely lucky. Being a Ph.D. student gives me a unique point of view of the world and the opportunity to analyze a part of it. Thanks to everyone I have met, I have the privilege of five years to grow with the research I have been doing.

First and foremost, I would like to thank my Ph.D. advisor, Carl Kingsford. Carl gave me space and time to explore what excites me the most but also guided me so that I am not lost. He helped me become a better researcher by always setting good examples for how to formulate a computational problem, how to learn theories and apply them, how to tell a scientific story and so much more. I have had doubts in my research, and there are moments when I was overwhelmed by the unknown. Carl would always point me in the right direction and lead me to walk out of the mess. I am confident that I can always get support from him. Looking back on the past five years, I can only regret not taking his advice sooner. Thank you for your patience, kindness, and every hint, suggestion, comment and question.

I would like to thank my co-authors Cong Ma and Yihang Shen. Through collaborating with them, I have accomplished what I would have never been able to do alone. It is a pleasure collaborating with you and thank you for sharing your knowledge and skills with me. I would like to also give my thanks to my other past and present group members, Guillaume Marçais, Mohsen Ferdosi, Minh Hoang, Shane Elder, Laura Tung, Natalie Sauerwald, Dan DeBlasio, Heewook Lee, Prashant Pandey, Nox Yu, Gwen Ge, Yuqi Wang, Raehash Shah and Yinjie Gao. The conversations with them have guided me to shape my research. Thank you for all of the helpful advice on my presentations, my drafts, and my roughly formed ideas. I would also like to thank my committee members, Jian Ma, Takis Benos and Adam Phillippy for their service and advice on how to improve my research.

I would like to thank my academic mentors before joining the Ph.D. program. Thank you for exposing me to the world of bioinformatics research and the kindness to accept me as one of your team so I can learn from the best what it means to be a researcher. I would also like to thank my mentors outside the academic settings, especially the encounters during the internships and the hackathons, for grounding me and giving inspiration from the most practical side of pangenomics.

I would like to thank my family, my partner and my peers. My mother, Yuben Nie, has been supporting me to achieve and taught me to work for what I wanted. My partner, Zhuojun Yu, has been a rock for me during the ups and downs and reminding me who I am and what the truly important things are. My peers have always been there to accompany me and assure me that I am not alone in this journey. I am fortunate to have you in my life.

Lastly, I give myself credit for my perseverance and efforts and for seizing the opportunities presented to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. The era of pan-genomics

Pangenomics is the study of collections of genomes. The notion of a pangenome first emerged in the study of microbiomes, where a sample in the microbial environment naturally contains a pool of diverse genomes from different strains of bacteria. Instead of focusing on one strain, or one single genome, researchers investigate the diversity of the genomes collected from an environment or site, which provides insight into the genetic dynamics, drug resistance, and pathogenesis of the microbial communities [28, 92, 98, 154, 157].

Pagenomics has enabled a more accurate depiction of the genomic landscape of the various Eukaryotic organisms as well. With the advancement of sequencing technologies, it has become easier to curate larger and more complex eukaryotic genomes with efforts from 1000 Genomes Project [2], UK BioBank [152], the Cancer Genome Atlas [161], the Human Pangenome Reference Consortium [1] and other large-scale sequencing efforts [24, 82]. The information on variations among individual genomes allows researchers to more accurately identify genetic elements that are unique to each human population [112, 148], understand the mechanism of genomic changes in diseases [58, 90, 104] and recover the evolutionary history of live stocks [168] and plants [11].

Despite the vast amount of data generated, it is challenging to completely catalog all variations among genomes and fully use them to improve the genomic analysis. While pangenomics has been widely acknowledged as an essential approach to integrating the vast amount sequencing data, the methods are mostly *ad hoc* and lack theoretical foundations. In this dissertation, we introduce two foundational algorithmic questions related to the pangenomic approach: pangenome reference construction and pangenome comparisons, and properties of those computational methods that are essential to successful pangenomic analyses. We develop algorithms and theories that address the time- and space-efficiency, the expressiveness of pangenome representation, and the accuracy of pangenome comparison. Combined, this dissertation establishes a solid algorithmic foundation for efficient and accurate pangenome construction and comparisons.

1.2. Pangenome reference construction

The human reference genome is the template for most analyses on a human genome. A collection of sequencing reads, which are fragments of the sample genome, are aligned the reference genome to obtain a more comprehensive view of the complete sample genome. All downstream analyses, such as the variant discovery, the analyses on epigenetic elements, and the characterization of chromosomal organization, depend on sequences aligned to the reference genome. For the past few decades, the structure of the human reference genome has been taken to be mostly a single set of strings that contains information mostly from European ancestry [113]. The linear structure and lack of diversity in its sources have resulted in biases and errors during the process of alignment to the reference genome [10, 99]. For example, due to variations among human individuals, collected sequencing reads may not align to the reference genome properly since they come from non-reference alleles [141]. These unaligned reads are usually discarded even though they contain valuable information about the sample genome. It is therefore imperative that the current reference genome is replaced by a high-quality reference that is representative of genomes of various populations, i.e. a pangenome reference.

A human genome consists of more than 3 billion base pairs, or letters in terms of the string representation of the genome. To conduct pan-genomic analyses on human genomes, we often need to access the thousands, even tens of thousands of genomes that aggregate to a set of strings with a total of hundreds of billions of characters. In this setting, it is no longer viable to represent such strings with the traditional linear structure. Therefore, a new form of pangenome representation needs to be established and new algorithms need to be designed to conduct essential operations on the new representation.

There are two major schemes of pangenome representations: text-based and graph-based. Text-based pangenomes encode a collection of genomes via a compressed data structure for space-efficient storage and/or via an indexing data structure that supports fast substring lookup. Graph-based pangenomes encode a collection of genomes in a graph structure, where nodes or edges of the graph are labeled with substrings from the genomes and the proximity of the substrings are encoded by the graph structure. The two schemes are not mutually exclusive to each other. In fact, many text-based indices have been adapted to graph-based representations.

1.2.1 Text-based pangenome representation

One type of text-based pangenome representation is the compression-based representation that uses ideas from data compression. Compression-based pangenome representations usually have the objective of minimizing the space taken to store a collection of genomes. One key feature of the genomes exploited by the compression methods is that genomes in a pangenome study are usually highly similar to each other. This makes using the relative Lempel-Ziv (RLZ) algorithm [13, 35, 74] advantageous to encode pangenomes. The RLZ algorithm uses a reference string, such as a single consensus reference genome, as the template to compress a collection of genomes by replacing the substrings in the individual genomes that occur in the reference with integers that indicate the location of the substring in the reference. RLZ has achieved a very significant reduction in genome size [36] and has a running time that is linear in the size of the input genomes. Recently, indexing structures have been developed to enable string query [30] and random access [43] on RLZ-compressed strings.

Another class of compression-based representations adapts the Burrows-Wheeler transform [21] and the FM-index [44], which are alignment indices developed for linear strings, to large collections of strings by reducing the size of the indexing structure. Similar to the RLZ algorithms, this class of methods reduce space by merging or eliminating the repetitive sequences. Some examples of this approach are prefix-free parsing methods [17, 114, 134], the r-index [49, 107], the hybrid index [93] and the marker array [108].

Compression-based methods make decisions on which common substrings to merge by optimizing for the size of the compressed strings, which may disrupt the boundaries of biologically meaningful regions on the genomes. To adapt to the biological significance of shared or differing substrings among collections of genomes, alignment-based methods use pair-wise or multi-sequence alignments (MSA) to determine which substrings are shared by the genomes. BWBBLE [63] compresses a MSA into a linear pangenome by using additional letters to represent single nucleotide variations (SNV) at each genomic position and concatenating longer variations among genomes. Representing the pangenome in a linear form allows direct adaption of alignment data structures such as BWT and linear aligners such as BWA [88] and bowtie [77] to string alignment to pangenomes. Another pangenome representation is the elastic degenerate (ED) strings. In ED strings [12, 25], common regions among the genomes are stored once as one string, whereas the divergence in genomes is stored as a block of stacked substrings following the shared region.

The reference bias can also be mitigated by replacing the linear reference genome with a panel of genomes that contain most of the known variations among populations. Reference flow [23] represents the human pangenome by a small number of reference genomes and adapts linear aligners to increase the sensitivity of read mapping.

1.2.2 Graph-based pangenome representation

Graph-based pangenomes are used to build the human pangenome reference [50, 52, 91]. In the graph representation of a pangenome, string labels are added to either nodes and edges that are substrings in the represented pangenome. The strings in the pangenomes are implicitly stored in graphs as walks along edges and the concatenation of string labels along the walks. The differences among genomes are encoded with substructures in the graph where several paths diverge from the paths that encode the shared strings.

The specific definition of the graphs depends on the construction and application of the representation methods, which can be divided into the following categories: (a) variant graphs, (b) multiple-alignment graphs and (c) de Bruijn graphs.

Variant graphs are graphs that are constructed by augmenting variants in the population to the linear reference genome. The primary goal of the variant graphs is to curate and enrich the reference panel of genomes that improve read mapping. Samples that are mapped to the variant graphs can be labeled, which supports haplotyping variants in different samples [42, 148]. Variant graphs can either be constructed by inserting known variants into the reference genome [39, 51, 117, 129], or progressively aligning genomes to the existing genome graph that is initialized with the linear reference genome [89].

Alignment graphs such as the Cactus graphs [118] and founder block graphs [94] are constructed based on multiple-sequence alignments, where matches in the alignment are merged into a common path and differences in the alignment are converted to bubble structures.

De Bruijn graphs are originally used in sequence assembly where the complete genomes are reconstructed from a fragmented set of sequencing reads [123]. De Bruijn graphs contain nodes that have length-k labels that can be directly constructed on a set of sequencing reads instead of a complete genome assembly, which make it more widely applicable to metagenomics. Colors can be added to de Bruijn graphs that label the path representing different samples stored in the graphs [64]. Improvements on colored de Bruijn graph construction [5, 62, 69, 103, 105] and updating [106] has resulted in faster and more compact de Bruijn graphs.

While text-based compression schemes represent the input genomes well, they lack the flexibility to encode variations beyond the input genomes and the structural changes in the human genomes [45] such as translocation, inversion and more complex structural variants such as chromoplexy [139] and chromothripsis [29] that are hallmarks of diseases such as cancer [60]. Graph-based representations have advantages over text-based compression schemes in that they are able to encode such large-scale changes in the genomes by edges between affected regions and represent a population of genomes that contain sequences that are combinations of substrings in the input genomes. The application of graph-based pangenomes is not limited to storing the genomic strings and supporting read alignment, but also to support transcriptomic [142] or epigenomic analyses [56].

In this dissertation, we focus on graph representation of the pangenome and study the construction, comparison, and essential properties to make analyses based on genome graphs successful.

1.3. Pangenome comparisons

By comparing heterogeneous samples collected from different time points and locations, an evolution trajectory can be reconstructed to study the dynamics of microbial genomes, which helps in identifying clinically relevant genes [28, 92, 157], e.g. genes relevant to antibiotic resistance. On the other hand, in human genomes, heterogeneity exists in polymorphic regions such as T-cell and HLA regions. The diversity in genomic sequences in these regions helps maintain a robust immune system that is adaptive to the massive number of foreign antigens [133]. Heterogeneity is also a hallmark in diseases such as cancer, where cells at one location contain multiple different genomes due to elevated mutation rates [104]. Comparing immune repertoires and cancer samples can help to determine better therapeutic strategies and advance understanding of the mechanism of human immune systems [115] and tumor progression [137, 144]. Representing a set of diverse genomic sequences is necessary when comparing heterogeneous samples [27] that contain multiple different genome sequences.

Methods to compare genomes [9] have been developed and improved upon in the past 3 decades and applied in comparative genomics where the evolutionary relationships between organisms are characterized. However, there are much fewer methods to compare pangenomes due to the change in pangenome representation. The challenge in comparing pangenome graphs comes from their combination of strings and graphs. On one hand, graph comparison is already difficult, when it is formulated by the graph edit distance or the graph isomorphism problems [67]. On the other hand, few methods have been developed to study the similarities between strings reconstructed by traversals in graphs. We introduce several schemes of genome comparison methods and graph comparison methods and discuss the existing methods to compare labeled graphs or graph pangenomes.

1.3.1 String comparisons

One way of comparing two genomes is through whole-genome alignment via edit distance computation. The edit distance between two strings can be computed using dynamic programming [111], which is not scalable to larger genomes. The banded alignment, which assumes that the pair of strings are similar, has a threshold on the total number of edit operations and can be computed in linear time [97] in the size of an input string. However, the set of edit operations allowed in the traditional edit distance does not include larger-scale mutations such as duplication, inversion, and translocation. Therefore, approximate local alignment methods [34, 65, 96, 118, 138], have been developed to identify segments of the genomes that are similar to each other and an ordering of such segments to transform one input genome into the other.

A common method to perform approximate local alignment is seed-and-extend. Seeds are usually a set of substrings with a fixed length, or k-mers, in each genomic string that can be identified and matched efficiently. The anchors divide the strings into segments. Local alignment methods, e.g. the Smith-Waterman [149] algorithm, are applied in the "extend" step that chains anchors together. Some examples of whole-genome aligners that use seed-and-extend approaches are Mummer [34, 96], wfmash [57] and minimap2 [87].

Pair-wise genome comparisons can be extended to compare multiple strings simultaneously through multiple-sequence alignment (MSA) [70]. The MSA problem is NP-hard. Therefore, existing MSA methods use heuristics such as the progressive alignment approach, where pairwise alignments were performed that chain multiple sequences into one alignment [9, 33].

Another scheme for comparing two sets of strings is to extract a smaller representation of the input pairs of sets of strings for faster distance computation. For example, Mash [116] computes a Jaccard distance between the MinHash [20] sketches that are scalable to cluster more than 50,000 microbial sequences.

1.3.2 Graph comparisons

The similarity between two general graphs without string labels is traditionally measured by the graph edit distance, which is the cost of changing one graph into another by adding and deleting nodes and edges. It is difficult to describe all differences between two large graphs [125], such as gene regulatory networks or protein interaction networks. Analogous to the heuristics in string alignment, the similarities between graphs (or networks) are computed by either comparing the distribution of global graph signatures such as degree distributions, or the distribution of local graph signatures such as graphlets [164], graph spectra [121, 162]. Recently, local graph signatures are extracted using embeddings based on random walks [55], or deep learning approaches such as the graph neural networks [163].

It is challenging to adapt methods to compare graphs to compare genome graphs because the former emphasizes the similarity between graph structures. In pangenome comparison, the emphasis is on the similarity between the sequences. While the graph structure may contain important information on the differences between genomes caused by structural variants, two genome graphs with different underlying graph structures could represent the same set of genomes.

1.3.3 Labeled graph comparisons

On the other hand, it is not trivial to adapt string alignment problems to genome graph alignment either due to the added complexity of the graph structures. Partial order alignment (POA) algorithms adapt the dynamic programming approach of aligning strings to aligning pairs of acyclic graphs [54], which can be done in polynomial time. On general graphs, however, it is NP-hard even to align a sequence to a labeled graph that contains cycles [66, 73]. Graph traversal edit distance [16] generalizes alignment between strings encoded by genome graphs.

An alternative to whole-graph alignment is to find local similarities between two input graphs. BubbZ [101] and SibeliaZ [102] construct compact de Bruijn graphs from two input genomes, identify the most similar paths in the de Bruijn graphs and conduct alignment between the selected paths. Asgan [124] identifies pairs of synteny paths in de Bruijn graphs that match pairs of unique k-mers from each input graph.

Sketching-based comparisons are also applied in genome graph comparisons. EMDeBruijn [95] compares two microbial communities using the Earth Mover's Distance between k-mer frequencies in each input de Bruijn graph that minimizes both the edit distance between paired k-mers and the length of the shortest path between two k-mers in each graph. A distance between succinct colored de Bruijn graphs is computed using gcBB [131] that compares the Burrows-Wheeler Similarity Distributions of the input graphs.

1.4. Challenges of genome graph construction and analysis

The challenges of pangenomics arise from several properties of the representations: efficiency, expressiveness and accuracy.

1.4.1 Efficiency

Efficiency refers to both the reduced storage space and the faster speed in operations on the pangenome. The two aspects of efficiency are often related. For example, the time to align a query string to the reference string grows linearly with the size of both the query string and the reference string. Therefore, improving space efficiency can lead to an improvement in time efficiency as well. However, there is no agree-upon formal definition of the size of a genome graph. Additionally, the closest analogy of the genome graph size optimization problem, as described in detail in Chapter 2, is the string compression problem, and it is NP-complete to produce a compressed string that minimizes the size [150]. Another challenge in efficiency in pan-genomic analysis arises from genome comparison. While it is straightforward to align strings using the dynamic programming approach [111] and its more efficient variants [97], it is not trivial to adapt the string alignment algorithms to graphs. It is also NP-complete to match a string to a labeled Eulerian graph [73].

1.4.2 Expressiveness

The advantage of the genome graph is that it merges common strings in the input genomes to reduce the size of the pangenome representation and increase the flexibility of the genomes it represents. The merging of common sequences gives rise to a new property of the pangenome, which is its expressiveness. The expressiveness of a pangenome representation can be viewed as the diversity of genomic strings that it can represent. In addition to the founding strings that are used to construct the representation, the representation could encode a more diverse set of strings. Merging sequences results in branched structures that are called bubble structures [120], which increases the number of possible paths significantly and in turn increases the number of strings encoded by the graph. The increase in expressiveness has the merit of encoding possible combinations of subsequences that represent a combination of mutations in the population. Using a small number of founding sequences, we may be able to represent a more diverse population by exploiting the expressiveness property.

1.4.3 Accuracy

One of the primary goals of using the pangenome is to increase the accuracy of sequence alignment. Accuracy is therefore an essential property of the pangenome representation. The genomes used to construct a genome graph are accurately represented if they can be reconstructed by concatenating labels following a trail in the graph. By including more genomic sequences from the human genome, the sequence alignment is more sensitive to sequencing reads sampled from unseen human genomes.

1.4.4 Trade-offs between properties of genome graphs

It is challenging to optimize for all three properties of the genome graph simultaneously. By optimizing one property of the genome graph, we may lose some other properties.

We may lose accuracy when we speed up genome graph construction. For example, minigraph [89] constructs large genome graphs quickly by aligning genomes approximately against each other. The constructed graph may not represent the founding genomes accurately because some small variations between genomic sequences are ignored in the fast, approximate alignment.

Expressiveness of a genome graph increases at the cost of accuracy when we reduce the size of a genome graph by merging common substrings. In addition to the desired increase in the diversity of genomes, combinations of substrings in the founding strings of the genome graph may result in strings that do not belong to the represented population. As a result, sequencing reads from a non-human genome may be mapped to the human pangenome, which results in an incorrect interpretation that the read comes from the human genome. Similarly, when we compare pangenomes, we could obtain results that are based on comparisons of unwanted sequences, which yields inaccurate similarity measures.

It is therefore important to characterize all three aspects of the genome graph, understand how they influence each other and directly co-optimize them.

1.5. Contributions

The previous genome graph construction methods do not directly optimize the space taken by genome graphs and construct graphs for which small size is an intended but incidental byproduct of the construction procedure. In Chapter 2, we optimize the size of the genome graph by drawing the first connection between genome graph construction and string compression. We propose an algorithmic framework to transform between genome graphs and compressed strings, which results in an upper bound on the size of the genome graph constructed in terms of the size of an optimal string compression. To demonstrate that classical compression algorithms can be used to efficiently construct space-efficient genome graphs, we adapt the algorithmic framework to construct graphs based on strings compressed by the relative Lempel-Ziv (RLZ) algorithm, which we call the RLZ graph. The compression-based genome graph construction framework enables us to adapt algorithmic innovations from string compression to pangenome representation and analyses and improve time- and space efficiency.

The computational cost for comparing two genome graphs is prohibitively high. Therefore, a more efficient algorithm is needed for computing distances between genome graphs. In Chapter 3, we revisit the graph traversal edit distance (GTED) and the algorithms to solve it. We point out that the originally proposed algorithms in Boroojeny et al. [16] do not — as they previously claimed — compute GTED. Instead, they compute a lower bound of GTED. Additionally, we show that it is NP-hard to compute GTED, confirming the hardness of genome graph comparison. We introduce another graph comparison problem that compares local structures in the genome graph and we show that it is solved by the previously proposed ILPs. To solve GTED, we propose two new ILP formulations. We evaluate the ILPs for these on short genomic strings and show that it is currently impractical to solve these ILPs on larger genomes. This work points to the direction of approximation algorithms for genome graph comparisons.

Lastly, we study the trade-off between the expressiveness of genome graphs and the accuracy of pangenome comparisons. A genome graph constructed from a heterogeneous sample represents a set of diverse genomes. However, existing genome graph comparison methods are mostly based on assembly graphs that contain single genomes, and the existing distance metrics between genome graphs do not take into account both the differences between complete genomic sequences and the abundances of the genomes. In Chapter 4, we propose Flow-GTED, which is a distance metric based on GTED, that more accurately measures the distance between two genome graphs that contain sets of genomes. We characterize the relationship between FGTED and the distance between the sets of strings that the graphs encode and show that FGTED always underestimates the distance between strings due to the extra expressiveness of genome graphs. We characterize the gap between FGTED and the string set distance using the expressiveness of the genome graph. By using expressiveness as a correction factor, we show how to reduce the discrepancy between FGTED and the string set distance, which results in a more accurate estimate of the distance between pangenomes.

In order to analyze large collections of heterogeneous genomes collectively, it is crucial to effectively leverage the advantages provided by the graph structure. However, studies of necessary genome graph properties and how to fully use them on heterogeneous string sets were missing in previous work. This dissertation fills the gap in genome graph studies by reinforcing the fundamental theories in genome graph construction and comparison.

Chapter 2

Efficient Genome Graph Construction Via Compression Schemes

This chapter was published in ISMB 2021 and Bioinformatics [126] and is joint work with Carl Kingsford. The source code for reproducing the results in this chapter is stored in https://github.com/Kingsford-Group/rlzgraph.

2.1. Introduction

The linear reference genome suffers from reference bias that results in discarding informative reads sequenced from non-reference alleles during alignment [10]. To reduce the reference bias, alternative read alignment approaches that use a set of genomes as the reference have been recently introduced [22, 112]. Genome graphs, due to their compact structure to store the shared regions of highly similar strings, are widely used to represent and analyze a collection of genomes compactly [27, 119, 140].

A genome graph is a labeled directed multi-graph that represents a collection of strings if each string is equal to the concatenation of node labels on a path in the graph. We call such a path a reconstruction path. The size of a genome graph is the space to store the graph structure, which is the set of nodes, edges and node labels.

The size of a genome graph is crucial to the efficiency of operations such as mapping sequencing reads. As shown in [66], the time complexity of mapping a string to a genome graph is directly related to the total number of characters in node labels and the number of edges. The speed of sequence-to-graph mapping can be further improved by a graph index, the size of which is also dependent on the size of the genome graphs [119, 146, 147].

Most of the existing genome graph construction algorithms do not directly optimize the size of the genome graph. Some of these algorithms design graph structures to adapt to specific types of input data, such as read alignment [51, 89, 94, 118], variant calls [39, 51, 129] or raw sequencing reads [64, 89], which are not necessarily optimized. Others only optimize the graph index that stores reconstruction paths based on assumed types of genome graphs, for example, the variation graphs [145, 147] or the colored compacted de Bruijn graphs [5, 7, 62, 103, 106]. As a result, the graphs constructed can be large in terms of both the space taken by the graph

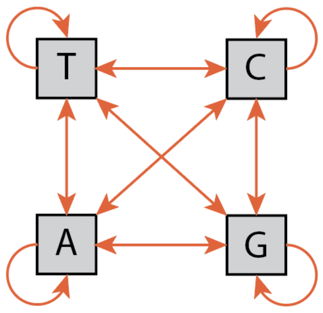


Figure 2.1: A small genome graph, G_4, that contains every genomic string if edges can be used multiple times.

structure or the lengths of the reconstruction paths.

While a small genome graph is desirable, the smallest genome graph may be useless if each edge is allowed to be traversed multiple times. The smallest genome graph is a multi-graph with four nodes, or G_4, whose labels are `A`, `T`, `C` and `G`, respectively (Figure 2.1). The edges in G_4 are two edges directing to and from each pair of nodes and a self-loop on each node. G_4 contains the reconstruction path for any sequence over alphabet {`A`,`T`,`C`,`G`}. However, the length of the reconstruction paths of each string would have lengths equal to the lengths of the string, which undermines the goal of a genome graph to compactly store similar strings.

In order to construct a small genome graph that balances the size of the graph and the lengths of the reconstruction paths, we introduce the definition of a restricted genome graph (Figure 2.2) and formalize the restricted genome graph optimization problem, which seeks to build the smallest restricted genome graph given a collection of strings. We present a genome graph construction

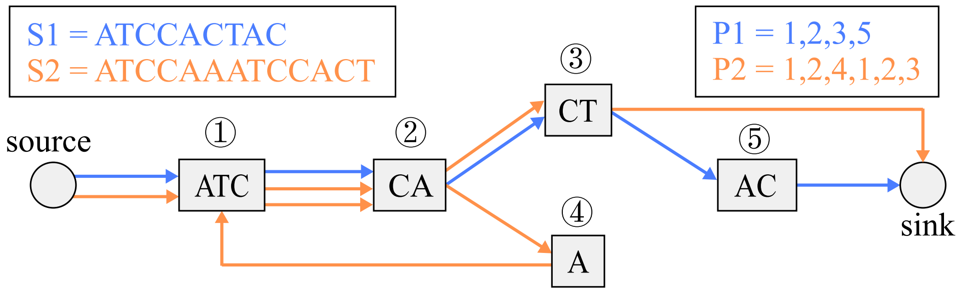


Figure 2.2: An example of a restricted genome graph. The graph stores two strings, S1 and S2. The color of the edges denotes the origin of node adjacencies.

algorithm that directly addresses the restricted genome graph size optimization problem. Optimizing the size of a restricted genome graph is similar to optimizing the space taken by a set of strings, which echoes the external pointer macro (EPM) compression scheme [150]. We introduce a pair of algorithms that transform between the EPM-compressed form and the restricted genome graphs and prove an upper bound on the size of the restricted genome graph constructed given an optimized EPM-compressed form from a set of input sequences. We further reduce the number of nodes and edges by introducing and solving the source assignment problem via integer linear programming (ILP).

As a proof-of-concept that compression-based genome graph construction algorithms produce smaller genome graphs efficiently, we build the RLZ-Graph, which is based on an EPM compression heuristic known as the relative Lempel-Ziv (RLZ) algorithm. The EPM compression problem is NP-complete [151]. Among the approximation heuristics to solve the EPM compression problem, the relative Lempel-Ziv algorithm [74] runs in linear time and achieves good compression ratios on human genomic sequences [35, 36, 43]. The RLZ algorithm is based on the Lempel-Ziv (LZ) algorithm [169]. Although the LZ algorithm achieves a better compression ratio than RLZ, it does not follow the EPM scheme and thus is not applicable in EPM-based genome graph construction.

We evaluate the performance of RLZ-Graph by comparing to the colored compacted de Bruijn graphs (ccdBG) [64]. CcdBG construction methods, similar to the compression-based genome graph construction algorithms, process the input sequences directly without intermediate steps such as alignment or variant calling. In ccdBG, the input sequences are fragmented into preliminary nodes that represent unique strings of length k, or k-mers. Each edge represents the adjacency between two k-mers in the sequences stored. The preliminary nodes with in- and out-degrees equal to 1 on a path are further merged into supernodes. Still, the number of nodes and edges, as well as the number of characters in node labels, in a ccdBG can increase significantly as the number of sequences stored increases. The size of the graph also depends heavily on the parameter k. These factors may offset the effort to efficiently encode the reconstruction path information in the graph indices [5, 7, 62, 103, 106]. Despite the different approaches to build the ccdBG indices, ccdBG construction methods result in the same underlying de Bruijn graph structure. When we compare our algorithm with ccdBG construction algorithms, we only compare the graph structure, which includes nodes, edges and sequences stored in each node.

The performance of RLZ-Graph is examined in multiple aspects. We show that the RLZ-Graph scales well in terms of running time and graph sizes with a large number of genomic sequences by comparing sizes of the RLZ-Graph with the ccdBG constructed by Bifrost [62] from 100 individuals on all chromosomes from the 1000 Genomes Project [2] (Section 2.8.1). Across all chromosomes, the disk space taken to store the graph representation of 100 sequences is on average reduced by 40.7% compared with the ccdBG built under the default settings. Additionally, we examine the performance of RLZ-Graph, Bifrost and VGtoolkit [51] on 32 individuals from the HGSVC dataset [41] that contains more complex structural variants than the 1000 Genomes Project samples (Section 2.8.2). On this dataset, the advantage of RLZ-Graph over ccdBGs persists, and RLZ-Graph constructs genome graphs of similar sizes as variation graphs constructed by VGtoolkit.

Additionally, we evaluate the performance of the ILP solution to the source assignment problem on RLZ-Graphs constructed from *E. coli* genome sequence. We show that the solutions to the source assignment problem reduce the number of nodes by around 8% on 300 *E. coli* genomes [26].

2.2. Definitions

2.2.1 Strings

Let s be a string. $s[b : e]$ denotes a substring starting from position b (inclusively) of s up to position e (inclusively). We assume 0-indexing throughout this chapter. The length of s is denoted by $|s|$. Concatenations of strings $\{s_1, ..., s_n\}$ are denoted by $s' = s_1 \cdot s_2 \cdot ... \cdot s_n$.

2.2.2 Genome Graphs

Definition 1 (Genome graph)**.** *A genome graph $G = (V, E, \ell)$ of a collection of strings $\mathcal{S} = \{s_1, s_2, ..., s_n\}$ is a directed multi-graph with node set V, edge multi-set E, and node labels $\ell(u)$ for each node u. A genome graph of $\mathcal{S}$ contains a collection of paths $\mathcal{P} = \{P_1, P_2, ..., P_n\}$, where $P_i = v_1, v_2, ..., v_{|P_i|}$, such that $s_i = \ell(v_1) \cdot \ell(v_2) \cdot ... \cdot \ell(v_{|P_i|})$ for all $s_i \in \mathcal{S}$. Such paths are called reconstruction paths.*

The size of a genome graph $G = (V, E, \ell)$ is denoted by $size(G)$, which is the space to store the set of nodes, edges and node labels (Section 2.3.1). The number of nodes in the node set V and the number of edges in edge multi-set E are denoted as $|V|$ and $|E|$, respectively. A genome graph may contain parallel edges, where two or more edges are incident to the same pair of nodes, and each edge may be traversed multiple times. We introduce the notion of a restricted genome graph, which limits the number of traversals through each edge.

Definition 2 (Restricted genome graph)**.** *A restricted genome graph is a genome graph with a source and sink node and the restriction that each edge is allowed to be traversed at most once in all reconstruction paths. A source is a node with no incoming edges, which represents the start of all stored sequences. A sink is a node with no outgoing edges, which represents the end of all stored sequences.*

An example of a restricted genome graph is shown in Figure 2.2. Each edge is traversed only once in all reconstruction paths, and parallel edges are present. In a restricted genome graph, if we add edges directing from sink to source, then the concatenation of reconstruction paths for all sequences forms an Eulerian tour. For a restricted genome graph $G = (V, E, \ell)$ and a collection of all reconstruction paths $\mathcal{P} = \{P_1, P_2, .., P_n\}$, we have $|E| = \sum_{P_i \in \mathcal{P}}(|P_i| - 1) + 2n$, where $2n$ edges are the edges directing from source nodes and edges directing to sink nodes.

2.3. Size formulation of restricted genome graphs and EPM-compressed forms

2.3.1 Size of a genome graph

We adopt a natural formulation of the size of a labeled graph, which describes the space to store nodes, edges and node labels. Given a restricted genome graph $G = (V, E, \ell)$ over alphabet Σ, let L be a string that contains every node label as a substring. Each node can be represented as a pointer to L, i.e. $v = (pos, len)$, such that $\ell(v) = L[pos : pos + len - 1]$. Each node takes

$2 \log |L|$ bits to be stored. The graph structure is stored as pairs of adjacent nodes. Each edge takes space $2 \log |V|$ bits. Therefore, the total space taken by a restricted genome graph, denoted by $size(G)$, under this model is:

$$size(G) = |L| \cdot \log |\Sigma| + |V| \cdot 2 \log |L| + |E| \cdot 2 \log |V|. \tag{2.1}$$

We introduce the restricted genome graph optimization problem:

Problem 1 (Restricted genome graph optimization problem)**.** *Given a set of sequences, build a restricted genome graph G such that $size(G)$ is minimized.*

In the above formulation, note that $|E|$ refers to the number of edges including the parallel edges. Solutions to Problem 1 avoid a trivial genome graph solution, that is a multigraph, G_4, with four nodes whose labels are A, T, C, and G, respectively, and edges such that there are at least two edges with different directions between each pair of nodes. Any sequence over the alphabet $\Sigma = \{A, T, C, G\}$ can be reconstructed in G_4 under the definition of a genome graph (Definition 1) since each edge may be traversed more than once. On the other hand, in a restricted genome graph, the number of edges grows as the lengths of reconstruction paths increase. Therefore, minimizing the size of the restricted genome graph achieves a combined objective of a small genome graph and short parsing of the input sequences.

2.3.2 External Pointer Macro (EPM) Compression Scheme

Optimizing the size of a genome graph is similar to optimizing the size of a set of strings, which has been studied as a data compression problem. We review the definition of the external pointer macro (EPM) scheme for data compression [151].

Given an input string, the external pointer macro scheme transform the input string into a set of pointers to a reference string, where each pointer is the compressed representation of a substring that occurs in both the input string and the reference string.

Definition 3 (Pointers in EPM)**.** *Given a reference string R, a pointer $p_i = (pos_i, len_i)$ represents the substring $R[pos_i : pos_i + len_i - 1]$.*

We say that two pointers, $p_i = (pos_i, len_i)$ and $p_j = (pos_j, len_j)$ are equivalent to each other if $R[pos_i : pos_i + len_i - 1] = R[pos_j : pos_j + len_j - 1]$. We refer to the length of a pointer $p_i = (pos_i, len_i)$ as $p_i.len$ and the position of a pointer as $p_i.pos$.

Definition 4 (External pointer macro (EPM) model [151])**.** *Given an alphabet Σ and a string T, a compressed form of string T adopts the EPM if the compressed data follows the form $C = R\#t$, where R is a string over Σ, $t = p_1, p_2...$ is a sequence of pointers that represent substrings in R, $\#$ is a separator symbol that is not in Σ, and T is equal to the string produced by substituting pointers in t by their corresponding substrings.*

An example of a string compressed using an EPM-scheme is shown in Figure 2.3.

The string T may represent a set of strings $S = \{s_1, s_2, ..., s_n\}$ by concatenation, i.e. $T = s_1\$s_2...\$s_n$, where $\$ \neq \#$ and $\$ \notin \Sigma$, where Σ is the alphabet for S. In order to define the end of each string and forbid pointers to cross the boundaries between strings, the alphabet for T is be constructed as $\Sigma' = \Sigma \cup \{\$\}$. Additionally, a \$ character is appended to the end of the reference string.

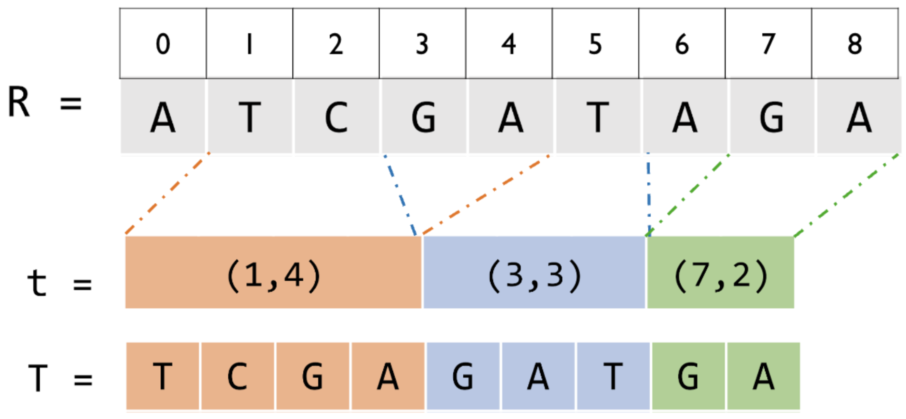


Figure 2.3: An example of external pointer macro scehem where the input string T=`TCGAGATGA`, and the compressed form is $C = R\#t$, where R =`ATCGATAGA` and $t = (1,4)(3,3)(7,2)$.

2.3.3 Size of an EPM-compressed form

We quantify the space taken by an EPM-compressed form $C = R\#t$. The space taken by C, $size(C)$, is the space to store the total number of unique pointers in t, the sequence t and the reference string R. We first encode each unique pointer with a pair of integers, (pos, len), which takes space $2\log|R|$ bits. If there are n unique pointers, t can be stored as a sequence of identifiers of the unique pointers using $|t|\log n$ bits. Therefore, the total space taken by an EPM-compressed form is

$$size(C) = |R| \cdot \log|\Sigma| + |t| \cdot \log n + n \cdot 2\log|R|. \tag{2.2}$$

In Storer and Szymanski [151], the problem of minimizing $size(C)$ given an input string T is shown to be NP-complete.

From equations (2.1) and (2.2), both the restricted genome graph and the EPM-compressed form have a size formulation that has three terms, which are the space taken by a reference string, the space taken by the unique pointers and the space to store the adjacencies between pointers.

In order to reduce the size of the restricted genome graphs (Definition 2), it is natural to borrow ideas from the field of string compression. We introduce two algorithms that transform between genome graphs and compressed strings produced by EPM compression scheme [151].

2.4. Transformation between EPM-compressed forms and genome graphs

2.4.1 EPM-compressed string to genome graph

Given an EPM-compressed form $C = R\#t$ of the original string T, and an alphabet Σ, the genome graph construction algorithm produces a restricted genome graph that stores both R and

R = AAATCG

S = $\underline{\text{AAA}}$ $\underline{\text{AAAT}}$ $\underline{\text{AAATC}}$

Figure 2.4: String S is factored into three pointers given the reference string R. Each underlined substring is represented by a different pointer. According to the naïve algorithm to construct the genome graph, three nodes are created from three pointers.

T.

A naïve algorithm to construct a genome graph is to create a node for each unique pointer in t and add an edge between nodes that represent each pair of pointers $t[i]$ and $t[i+1]$. However, in repetitive sequences such as the human genome, a substring may occur in several pointers and thus may be stored several times redundantly. As shown in Figure 2.4, the substring `AAA` would be stored three times according to the naïve algorithm, which results in excess space spent on storing repetitive content.

Our construction algorithm, introduced below as `two-pass CtoG`, merges the repetitive substrings shared by multiple pointers by grouping pointers by their positions on the reference. `Two-pass CtoG` constructs the genome graph in two passes through t. In the first pass, the algorithm creates nodes by cutting the reference string according to the boundaries of each pointer. In the second pass, the algorithm connects the nodes according to the adjacencies between pointers in the compressed string t.

We first introduce the notion of sources on the reference string, which are all the occurrences of substrings represented by each pointer.

Definition 5 (Source)**.** *A source, (pos_1, len), of a pointer $p = (pos_2, len)$ is an occurrence of $R[pos_2 : pos_2 + len - 1]$ in R. In other words, $R[pos_1 : pos_1 + len - 1] = R[pos_2 : pos_2 + len - 1]$. Each pointer p is associated with a source set $\mathcal{S}_p = \{ss_1, ss_2, ...\}$, where $R[ss_i.pos : ss_i.pos + ss_i.len - 1] = R[p.pos : p.pos + p.len - 1]$ for all $ss_i \in \mathcal{S}_p$.*

Sources are used to refer to the occurrences of a substring on the reference string R and are not stored in the compressed form. Pointers are used to refer to the pair of integers eventually stored in the compressed string t.

Definition 6 (Boundaries of sources and pointers)**.** *The boundaries of a source $s = (pos, len)$ are defined as (b, e), where $b = pos$ and $e = pos + len$. b is the left boundary and e is the right boundary. The similar definition of boundaries applies to pointers.*

Two boundaries, (b_1, e_1) and (b_2, e_2), intersect if and only if $b_1 = b_2$ or $b_1 = e_2$ or $e_1 = b_2$ or $e_1 = e_2$.

First pass. Create a bit vector, $\mathcal{B}$. A bit set at $\mathcal{B}[i]$ indicates that a pointer boundary falls at position i on R. Process t from left to right. For each pointer $p = (pos, len)$, mark its boundaries by setting $\mathcal{B}[pos] = 1$ and $\mathcal{B}[pos + len] = 1$. After t is exhausted, transform $\mathcal{B}$ into a succinct bitvector that supports `rank` operations in constant time (e.g. [68, 130]), where $\texttt{rank}_{\mathcal{B}}(i)$ returns the number of set bits at or before position i in $\mathcal{B}$. We then cut the reference string at positions where a bit is set in $\mathcal{B}$. If $\mathcal{B}[i]$ and $\mathcal{B}[j]$ are the only set bits in the interval $[i : j]$, we create a node $v = (pos, len) = (i, j - i)$ with $\ell(v) = R[i : j - 1]$. Each node can be treated as a pointer whose left and right boundaries are i and j, respectively. Each node is identified using its left boundary, i.e. $\texttt{rank}_{\mathcal{B}}(i)$.

We define the ordering of nodes: $v_i = (pos_i, len_i) \prec v_j = (pos_j, len_j)$ and $i < j$ iff $pos_i < pos_j$, where i and j are the identifiers of v_i and v_j. Since nodes are created by cutting the reference, different nodes will always have different starting positions. Add an edge between each v_i and v_{i+1} for all $i < |V| - 1$. The path $v_1, v_2, ..., v_{|V|}$ represents the reference string R.

Second pass. We process t from left to right again in the second pass to connect nodes.

Create a source and a sink node that represent the start and end of each sequence represented in T. Add an edge to the node whose position corresponds to $t[0]$.

For each pointer $t[i]$, add reference edges between nodes that fall between the range of pos_i and $pos_i + len_i$, i.e. nodes in $\{v_j \mid pos_i \leq v_j.pos < pos_i + len_i\}$. The identifiers of such nodes are consecutive because the nodes are sorted by their starting positions. The range of such node identifiers is between $\texttt{rank}_{\mathcal{B}}(pos_i)$ and $\texttt{rank}_{\mathcal{B}}(pos_i + len_i - 1)$. If a pointer $t[i] = (pos_i, len_i)$ represents the suffix of a sequence, then $R[pos_i + len_i - 1] = \$$. When such pointer is encountered, add an edge from node $v_{\texttt{rank}_{\mathcal{B}}(pos_i + len_i - 1)}$ to sink. Add an edge from the source to node $v_{\texttt{rank}_{\mathcal{B}}(t[i+1].pos)}$.

For each pair of pointers $t[i]$ and $t[i+1]$, we need to connect the nodes that mark the right and left boundaries of $t[i]$ and $t[i+1]$, respectively. Let $t[i] = (pos_i, len_i)$ and $t[i+1] = (pos_{i+1}, len_{i+1})$. We need to find two nodes, $v_m = (pos_m, len_m)$ and $v_n = (pos_n, len_n)$, such that $pos_m + len_m = pos_i + len_i$ and $pos_n = pos_{i+1}$. Since each node is identified by their left boundary, two nodes can be identified by $m = \texttt{rank}_{\mathcal{B}}(pos_i + len_i - 1)$ and $n = \texttt{rank}_{\mathcal{B}}(pos_{i+1})$. Edge (v_m, v_n) then represents the adjacency between $t[i]$ and $t[i+1]$ in t.

Repeat the process until t is exhausted. An example output of the algorithm is shown in Figure 2.5.

The running time of the construction algorithm is $O(|T| + |R|)$. In the first pass, the algorithm passes through each pointer once, and the number of pointers, $|t| \leq |T|$. The number of created nodes is at most $|R|$. In the second pass, the algorithm adds at most $|T|$ reference edges and $|t|$ edges that represent adjacency between pointers. In the actual implementation, the parallel edges are merged. Therefore, during the second pass, we do not need to add reference edges for each pointer. Hence the running time is $O(|t| + |R|)$.

The constructed restricted genome graph stores the set of nodes, edges and node labels. While storing the reconstruction paths is also important, it is a separate challenge from optimizing the graph structure. There has been a line of work that constructs small graph indices to store the reconstruction paths efficiently given any graph structure [145–147]. These indices can also be applied to our genome graph.

There are three types of edges in the produced restricted genome graph: the backward edges, the forward edges and the reference edges. We define the backward edges as edges that direct from v_j to v_i, where $j \geq i$, which include self-loops. We define the forward edges as edges that direct from v_i to v_j, where $i < j - 1$. We define the reference edges as the edges that direct from v_i to v_{i+1}. In other words, reference edges $(v_i = (pos_i, len_i), v_j = (pos_j, len_j))$ connect nodes where the first node's right boundary intersects with the second node's left boundary, i.e. $pos_i + len_i = pos_j$.

We prove the correctness of teh algorithm by showing that the constructed graph is a restricted genome graph that contains reconstruction paths for R and T.

Theorem 1. *Given an EPM-compressed form of string T, $C = R\#t$, the algorithm described above creates a genome graph $G = (V, E, \ell)$ that contains reconstruction paths for R and T.*

Proof. In the second pass of the algorithm, edges are added between the nodes that are the suffix and the prefix of adjacent pointers. Therefore, all pointer adjacencies are represented as edges in the genome graph.

All substrings $R[i : j]$ can be reconstructed from G. If $R[i : j]$ is a substring of a node label, it can be reconstructed from G. If $R[i : j]$ spans two nodes, it spans two nodes connected by a reference edge.

Any substring $T[i : j]$ can be reconstructed from G. Suppose position i lands in the middle of a pointer $p_k = (pos_k, len_k)$, which means that $k \leq i \leq k + len_k - 1$.

1. If $j \leq k + len_k - 1$, which means that $T[i : j]$ is a substring of the string represented by a pointer. Since all pointers in t point to substrings in R and R can be reconstructed from G, a substring of a pointer can be reconstructed.
2. If $j > k + len_k - 1$, which means that $T[i : j]$ spans at least two pointers. From the previous case, we have that $T[i : k + len_k - 1]$ can be reconstructed using nodes and edges in G. Since all adjacencies between two pointers are represented in G, we can apply the analysis to the rest of $T[i : j]$. Therefore, $T[i : j]$ can be reconstructed if it spans more than one pointer.

Finally, we show that the created graph is a restricted genome graph. During the second pass, a reference edge is added for each node adjacency within each pointer, and an edge is added for each pointer adjacency. Therefore, each edge is only used once in all of the reconstruction paths.

$\square$

2.4.2 Genome graph to EPM-compressed form

Given a restricted genome graph $G = (V, E, \ell)$ and a set of reconstruction paths $\mathcal{P}$ that represent strings in $\mathcal{S}$, we present an algorithm, `GtoC`, that produces an EPM-compressed form $C = R\#t$ whose decompression equals string T, which is a concatenation of strings in $\mathcal{S}$.

Produce the reference string R by concatenating the node labels in an arbitrary order. Each node can then be represented as a pointer to R and be denoted as $v_i = (pos_i, len_i)$, where $\ell(v_i) = R[pos_i : pos_i + len_i - 1]$. Assign an identifier to each node such that for v_i and v_j, $i < j$ if $pos_i < pos_j$.

Process all $P \in \mathcal{P}$ by substituting nodes with their pointer representations. If two or more adjacent nodes $v_i, v_{i+1}, ..., v_j$ in P are connected by a reference edge, merge the two nodes into one pointer $p = (pos_i, pos_j + len_j - pos_i)$. Concatenate all processed P, which results in t. The converted sequence of pointers t is then $p_1, p_2, ..., p_{|t|}$, where $|t| \leq \sum_{P \in \mathcal{P}} |P|$.

The converted C satisfies the EPM definition where R is a string over Σ, and t is a sequence of pointers to substrings in R. Since the concatenation of paths in $\mathcal{P}$ spells out T by concatenating

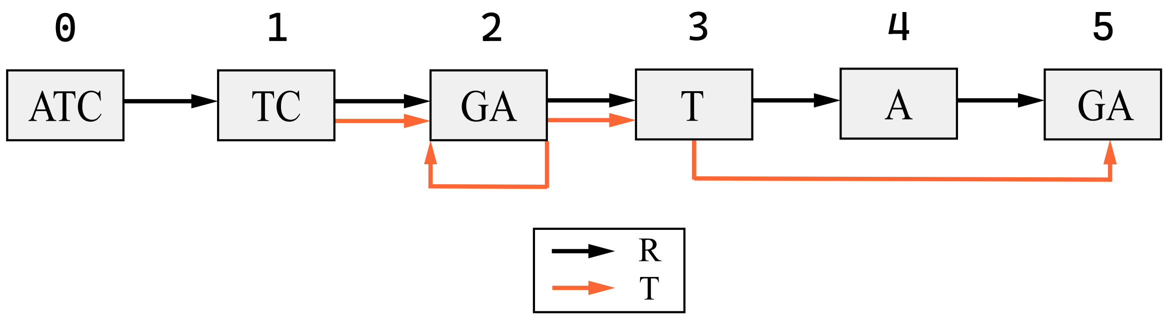


Figure 2.5: The RLZ-Graph of reference R =`ATCGATAGA` and input string T =`TCGAGATGA`. The black path $0, 1, 2, 3, 4, 5$ encodes R, the orange path $1, 2, 2, 3, 5$ encodes T. The parallel edges are shown for the purpose of illustration and are merged in the final graph.

all the labels of nodes on the path, substituting the pointers in t with corresponding substrings reconstructs T.

The running time of the `GtoC` construction algorithm is $O(|V|+\sum_{P\in\mathcal{P}}|P|) = O(|V|+|E|)$.

The size of the produced EPM-compressed form can be further reduced if the reference string R is equal to the shortest superstring that contains all the node labels. While finding the shortest superstring problem is NP-hard [128] when the number of nodes is greater than 2, it may be approached by using approximation algorithms [4, 14, 155].

2.5. Upper-bound on the size of the restricted genome graph and the EPM-compressed form

We show that the size of a restricted genome graph G produced using the `two-pass CtoG` algorithm is bounded by the terms of the input EPM-compressed form C (Lemma 1). In the following proofs, we assume that the input string T represents a concatenated set of m sequences.

Lemma 1. *Given an optimally compressed EPM form $C = R\#t$ of text T, the size of the transformed restricted genome graph $G = (V, E, \ell)$, $size(G)$, according to `two-pass CtoG` in Section 2.4.1 has an upper bound:*

$$\begin{aligned} size(G) \leq\ & |R| \cdot \log|\Sigma| + \min(2n, |R|) \cdot 2\log|R| \\ & + (\min(2n, |R|) \cdot |t| - 1 + 2m) \cdot 2\log(\min(2n, |R|)) \end{aligned} \tag{2.3}$$

where n is the number of unique pointers in t.

Proof. The algorithm introduced in Section 4.1 creates nodes by cutting the reference string R according to the boundaries of pointers. Each node is stored as a pointer (pos, len) to R, which takes $2\log|R|$ bits.

The total number of nodes produced by cutting the reference is $\leq \min(2n, |R|)$. The number of cuts introduced by each unique pointer is ≤ 2. The maximum number of nodes given a reference string R is $|R|$. Therefore, the space to store all the nodes is $\leq \min(2n, |R|) \cdot 2\log|R|$.

The total number of edges, including reference and non-reference edges, in a restricted genome graph is $\leq \min(2n, |R|) \cdot |t| - 1$. After the first pass of `two-pass CtoG`, the interval corresponding to each pointer may be cut into several nodes. Let the average number of nodes contained in each pointer's interval be $a \leq |V| \leq \min(2n, |R|)$. The average number of reference edges within each pointer is then $a - 1$, and the total number of edges within pointers is $(a-1) \cdot |t|$. The total number of edges between pointers is $|t| - 1$. Since T represents m sequences, we have $2m$ additional edges directing to and from the sink and source. Together, the number of edges in the reconstruction path is $a \cdot |t| - 1 \leq \min(2n, |R|) \cdot |t| - 1 + 2m$.

The size of the genome graph is then:

$$\begin{aligned} size(G) &= |R| \cdot \log|\Sigma| + |V| \cdot 2\log|R| + |E| \cdot 2\log|V| \\ &\leq |R| \cdot \log|\Sigma| + \min(2n, |R|) \cdot 2\log|R| \\ &\quad + (\min(2n, |R|) \cdot |t| - 1 + 2m) \cdot 2\log(\min(2n, |R|)). \end{aligned}$$ □

In practice, the graphs are stored such that the parallel edges are merged. We show that the size of the genome graph G' produced by merging the parallel edges in G can also be bounded by the terms of the EPM-compressed form C (Lemma 2).

Lemma 2. *Given a restricted genome graph, $G = (V, E, \ell)$, constructed from an optimally compressed EPM form $C = R\#t$, the size of the genome graph, $G' = (V, E', \ell)$, produced by merging parallel edges in G has an upper bound:*

$$\begin{aligned} size(G') &\leq |R| \cdot \log|\Sigma| + \min(2n, |R|) \cdot 2\log|R| \\ &\quad + (\min(2n, |R|) + |t| - 1 + 2m) \cdot 2\log(\min(2n, |R|)), \end{aligned} \tag{2.4}$$

where n is the number of unique pointers in t.

We show in Lemma 3 that the size of the EPM-compressed form produced by `GtoC` algorithm (Section 2.4.2) is upper-bounded by combined sizes of storing edges, nodes and the node labels restricted genome graph.

Lemma 3. *Given a restricted genome graph $G = (V, E, \ell)$ of a collection of strings $\mathcal{S}$, the size of the transformed EPM-compressed form of the concatenated strings in $\mathcal{S}$, $C = R\#t$ according to `GtoC` described in Section 2.4.2 has an upper bound:*

$$size(C) \leq |R| \cdot \log|\Sigma| + |E| \cdot \log\binom{|V|+1}{2} + 2\binom{|V|+1}{2}\log|R|, \tag{2.5}$$

where R is a string formed by concatenating all node labels.

Proof. Merging parallel edges does not change the number of nodes and the concatenation of node labels.

The number of reference edges in G' is equal to $|V| - 1$, as the nodes are produced by cutting the reference string.

The number of forward and backward edges in G is equal to $|t| - 1$, and the number of forward and backward edges in G' is $\leq |t| - 1$ due to parallel edge merging. According to `two-pass CtoG`, since C is optimal, only a forward or a backward edge can be added for each pair of adjacent pointers in t during the second pass. Suppose two adjacent pointers, $p_1 = (pos_1, len_1)$ and $p_2 = (pos_2, len_2)$, result in a reference edge, which means that $pos_2 = pos_1 + len_1$, the two pointers can be merged into $p_3 = (pos_1, len_1 + len_2)$. Merging two pointers reduces the size of C, which contradicts the assumption that the size of C is optimal.

Together, the space to store all the edges in G' is $\leq (|V| + |t| - 1) \cdot 2 \log \min(2n, R) \leq (\min(2n, |R|) + |t| - 1 + 2m) \cdot 2 \log \min(2n, |R|)$.

Therefore, the size of the genome graph G' after merging the parallel edges in G is:

$$\begin{aligned}|G'| &= |R| \cdot \log|\Sigma| + |V| \cdot 2\log|R| + |E'| \cdot 2\log|V| \\ &\leq |R| \cdot \log|\Sigma| + \min(2n, |R|) \cdot 2\log|R| \\ &+ (\min(2n, |R|) + |t| - 1 + 2m) \cdot 2\log(\min(2n, |R|)). \qquad \square\end{aligned}$$

The pair of algorithms do not produce an optimal genome graph or optimal EPM-compressed form. Still, given an optimal input, the algorithms achieve results that are bounded by the original terms in the input. We further improve the transformation from EPM-compressed form to genome graph by addressing the source assignment problem.

2.6. Source assignment problem

In an EPM-compressed form $C = R\#t$, each pointer may be associated with a substring that occurs several times in R. We name such occurrences as sources. A source (pos_i, len_i) is assigned to a pointer p if $p = (pos_i, len_i)$.

In the EPM formulation, assigning different sources to a pointer does not change the size of the compressed string. However, the assignment of sources may change the number of nodes. According to the `two-pass CtoG` algorithm, the number of cuts made in the reference is equal to the number of distinct pointer boundaries. Therefore, the choice of sources is directly related to the number of nodes in the graph. An example is illustrated in Figures 2.6 and 2.5. The last phrase, $(7, 2)$, is associated with two sources, $(3, 2)$ and $(7, 2)$. If we assign $(3, 2)$ to the phrase, which is different from the case in Figure 2.6, the number of nodes created will be 5. Otherwise, 6 nodes will be created as in Figure 2.5.

Given an EPM-compressed form and the set of sources corresponding to each pointer, if we can assign sources such that the total number of unique pointer boundaries is minimized, we can reduce the size of the created graph. We formulate the source assignment problem and present an integer linear programming (ILP) solution for the optimal source assignment during genome graph construction.

Problem 2 (Source assignment problem). *Given a collection of sources sets $\mathcal{S} = \{S_1, S_2, ..., S_n\}$, where S_i denotes the set of sources for a unique pointer i, find a set of sources S' such that for all S_i, $S_i \cap S' \neq \emptyset$ and $|\bigcup_{s_m \in S'}\{b_m, e_m\}|$ is minimized, where b_m, e_m are boundaries of source m.*

In this problem, we choose one source for each pointer such that the union of boundaries $\{b_m, e_m\}$ of each chosen source $s_m = (pos_m, len_m)$ is minimized. As a reminder, $b_m = pos_m$ and $e_m = pos_m + len_m$. For convenience, we denote the union of boundaries in a source set S by $\bigcup_B\{S\}$, which is equivalent to $\bigcup_{s_m \in S}\{b_m, e_m\}$.

The formulation of the source assignment problem is similar to the hitting set problem in that it chooses the minimum number of positions to hit every pointer. However, the objective is indirectly related to the number of the chosen sources, and the sources and pointers are defined in a string context. The hardness of the source assignment problem is open due to these differences from the setting of the hitting set problem. Still, the similarities to the hitting set problem lead to the formulation of an integer linear programming (ILP) solution.

2.6.1 Integer linear programming formulation

The objective of the ILP is to minimize the number of cuts made in the reference, where each cut is made at the boundaries of chosen sources. For each chosen source $s = (pos_i, len_i)$, a cut is placed at positions pos_i and $pos_i + len_i$, which are left and right boundaries of s.

We first construct a set of integers I that is the union of all source boundaries. Create a binary variable x_p for each $p \in I$. x_p is set to one if a cut is made at position p.

We create a binary variable y_{s_i} for each source $s_i = (pos_i, len_i)$ that indicates whether the source is chosen. We create a constraint (Inequality 2.7) that at least one source is chosen from each set. We create another pair of constraints (Inequality 2.8) that ensures that if a source is chosen, two cuts are made at its left (pos_i) and right ($pos_i + len_i$) boundaries. This leads to the ILP:

$$\min \sum_{p \in I} x_p \tag{2.6}$$

$$\text{subject to} \quad \sum_{s_j \in S_i} y_{s_j} \geq 1 \qquad \forall S_i \in \mathcal{S} \tag{2.7}$$

$$y_{s_j} \leq \min\{x_{pos_j}, x_{pos_j + len_j}\} \tag{2.8}$$

$$x_p, y_{s_j} \in \{0, 1\} \tag{2.9}$$

2.6.2 Pruning to reduce the number of sources

In practice, a pointer with a short length may correspond to a large number of sources. For example, a pointer with length one may correspond to $|R|/4$ sources, where R is the reference string and when the alphabet size is 4. This could result in a huge number of variables in the ILP formulation and would hinder its practicality significantly.

To address this, we preprocess the sources as follows. If a source does not intersect with any other sources of different pointers, we eliminate the source from the source set unless it is the only source of a pointer. We name the eliminated sources isolated sources. Removing such sources does not affect the optimality of the solution.

Lemma 4. *If a set of sources, S, that satisfies the constraints of the source assignment problem, includes an isolated source s, it is possible to find a set of sources S' with equal or lower objective value that does not include s.*

Proof. Let the pointer for the isolated source be p and the source set of p be S_p. Since s is an isolated source, there must be at least another source s' in S_p. If s' also does not intersect with

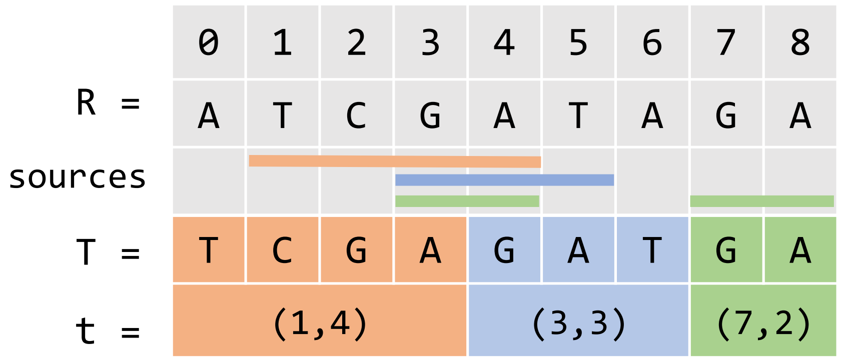


Figure 2.6: An example of RLZ factorization. The top row is the indices of characters in the strings. R is the reference string, T is the input string and t is a sequence of phrases resulted from RLZ factorization. Colored line segments on the third row represent the sources associated with phrases with the same color.

any other sources in S, $S' = |\bigcup_B \{(S \setminus s) \cup s'\}| = |\bigcup_B \{S\}|$. Otherwise, if s' intersects with some sources in S, this means that the union of source boundaries is reduced by at least 1 if we replace s with s', i.e. $S' = |\bigcup_B \{(S \setminus s') \cup s\}| \leq |\bigcup_B \{S\}| - 1$. Therefore, excluding all isolating sources during preprocessing does not affect the optimality of the solution. $\square$

2.7. Relative Lempel-Ziv Graph

As a proof-of-concept that constructing a genome graph using a compression scheme results in small graphs, we implement the graph construction algorithm based on an EPM compression scheme algorithm, relative Lempel-Ziv.

2.7.1 Relative Lempel-Ziv Algorithm

Given a reference string R, the relative Lempel-Ziv (RLZ) algorithm [74], greedily produces a compressed form of R.

Definition 7 (Phrase)**.** *Given a reference string R and a string T, let pointer $p = (pos, len)$ represent the substring $R[pos : pos + len - 1]$ which equals $T[pos' : pos' + len - 1]$ for some position pos'. Then p is a phrase if p is right-maximal, i.e. if $R[pos : pos + len] \neq T[pos' : pos' + len]$.*

The relative Lempel-Ziv (RLZ) algorithm, proposed by Kuruppu et al. [74], runs in linear time and achieves good compression ratios with genomic sequences. The RLZ algorithm takes a reference string R as input and parses the input string T greedily from left to right. At position i in T, the RLZ algorithm substitutes the longest prefix of $T[i : |T| - 1]$ that matches a substring in R with a phrase. Let the length of the phrase be len. After substitution, the RLZ algorithm skips to position $i + len$ in T and repeats the substitution process until T is exhausted. The process of phrase production is called RLZ factorization. In some analysis of the RLZ algorithm, the reference string is generated from the set of input strings [48]. Nevertheless, the RLZ algorithm given a reference string remains the same.

The definitions introduced above are demonstrated in Figure 2.6, where R is the reference string and T is the input string to the RLZ algorithm. RLZ factors T into a sequence of three phrases, shown as t. The compressed form of the input string T is $C = R\#t$. Each phrase is associated with some sources that are represented as line segments in the figure. For example, the last phrase, $(7, 2)$, replaces the substring $T[7 : 8]$. It also corresponds to two sources in R: $(3, 2)$ and $(7, 2)$, which are represented by the green line segments in R. The left and right boundaries of phrase $(7, 2)$ are $(b = 7, e = 8)$ in T. Source $(3, 2)$ intersects with sources $(1, 4)$ and $(3, 3)$. However, sources $(1, 4)$ and $(3, 3)$ do not intersect with each other.

2.7.2 Implementation of the RLZ-graph construction framework

RLZ factorization in this manuscript is done on the compressed suffix array in the SDSL C++ library [53]. We apply the `two-pass CtoG` algorithm described in Section 2.4.1 to construct a RLZ-Graph. We merge the parallel edges in the implementation as it is the common practice in genome graph storage.

An example of RLZ-Graph is shown in Figure 2.5. The RLZ-Graph is constructed based on the RLZ factorization in Figure 2.6, where the reference string is R=`ATCGATAGA`, the input string is T=`TCGAGATGA` and the factored phrase sequence is $t = (1, 4), (3, 3), (7, 2)$. The nodes are produced by segmenting R according to the boundaries of sources assigned to phrases in t.

In the implementation of RLZ-Graph, we build a bi-directed graph where each node can be traversed in forward and reverse directions. For each node $v = (pos, len)$, pos is referred to as the head of the node and $pos + len$ is referred to as the tail. If a node is traversed in reverse direction, its label is denoted as $\hat{\ell}(v)$, which is equal to the reverse complement of $\ell(v)$. This technique is useful in genomic sequences that underwent structural variations such as inversions, where the entire genomic segment is replaced by its reverse complement due to a double-strand break. During the construction of the RLZ-Graph, we use a modified reference sequence R by concatenating the reference genome of the organism of interest with its reverse complement. Before the source assignment step, we mark each source as reversed if it is located on the reverse complement half of R and translate its boundary positions to the forward half. After the source assignment step, we mark a pointer as reversed if it is assigned a reversed source. When we add edges, if we encounter a reverse pointer $p = (pos, len)$, we add an edge directing to the tail of the node $v_i = (pos_i, len_i)$ and an edge directing from the head of the node $v_j = (pos_j, len_j)$, where $pos_i = pos$ and $pos_j + len_j = pos + len$.

2.7.3 Different source assignment heuristics

Aside from the ILP solution to the source assignment problem (Section 2.6), sources are chosen by other heuristics in literature regarding RLZ factorization [75]. Specifically, from the source set corresponding to a phrase, the leftmost source on the reference string is chosen (Left), or the lexicographically smallest source is chosen (Lex). A source $s_i = (pos_i, len_i)$ is to the left of source $s_j = (pos_j, len_j)$, or $s_i <_{left} s_j$, if $pos_i < pos_j$. A source s_i is lexicographically smaller than s_j, or $s_i <_{lex} s_j$, if $R[pos_i : |R| - 1] < R[pos_j : |R| - 1]$ given a reference string R.

In the implementation of RLZ-Graph, the phrase is assigned to the lexicographically smallest source during RLZ factorization. After that, we re-assign the leftmost source to each phrase in order to construct a smaller genome graph, which is the default behavior of the RLZ-Graph software. We evaluate the performance of various source assignment heuristics to reduce the number of nodes in the graph in the following section.

2.8. Experimental results

We ran all our experiments on a server with 24 cores (48 threads) of two Intel Xeon E5 2690 v3 @ 2.60GHz and 377 GB of memory. The system was running Ubuntu 18.04 with Linux kernel 4.15.0.

In this section, we compare the size of the colored compacted de Bruijn graphs [64] and variation graphs [51] with that of RLZ-Graphs on human genomic sequences. There are many genome graph construction methods. However, we are mainly concerned with the methods that are based on complete genome sequences as input and generate genome graphs that guarantee exact reconstruction of input sequences. Specifically, we focus on comparing to colored compacted de Bruijn graphs. While there have been many graph construction algorithms for building colored de Bruijn graphs, the graph structure of ccdBG remains the same in these algorithms despite the different approaches to store the reconstruction paths as identifiers in each node.

The comparisons made in this section only concern the nodes, edges and node labels.

2.8.1 Performance of RLZ-Graph compared to the colored compacted de Bruijn graphs

We use Bifrost [62] to construct the ccdBG. The genome graphs constructed include nodes, labels of nodes and edges, and are stored in graphical fragment assembly (GFA) format [86]. In a GFA file, the nodes of a graph are stored as a list of pairs of node identifiers and labels, and edges are stored as a list of pairs of node identifiers. Same as the RLZ-Graph, the graph constructed by Bifrost is bi-directed and does not contain parallel edges. The RLZ-Graph produced in this section does not use the ILP solution to assign sources due to time and memory concerns. Instead, we adopt the leftmost heuristic, where the leftmost source is assigned to each phrase.

We build the graphs on all human chromosomes and show the results on chromosome 1 in this section (see Figures 2.10–2.12 for the rest of the chromosomes). The genomes we use are from the 1000 Genomes Project phase 3 [2]. In each experiment, we randomly choose 5, 25, 50, 75 and 100 samples and generate their genomic sequences on all chromosomes using the consensus command from bcftools [85]. We construct the ccdBG with Bifrost and RLZ-Graph using the sample sequences and the reference hg37. Hg37 is also used as the reference string during RLZ factorization. We vary the k-mer sizes used for Bifrost and report the sizes of graphs with $k = 31$, 63 and 127. The default choice of k of Bifrost is 31. We repeat each experiment 5 times.

As shown in Figure 2.7, we compare the graph size in different aspects. From 5 sequences up to 100 sequences, the graph produced by RLZ-Graph is smaller than the graph produced by Bifrost with different choices of k under all measures in the figure. The number of total characters

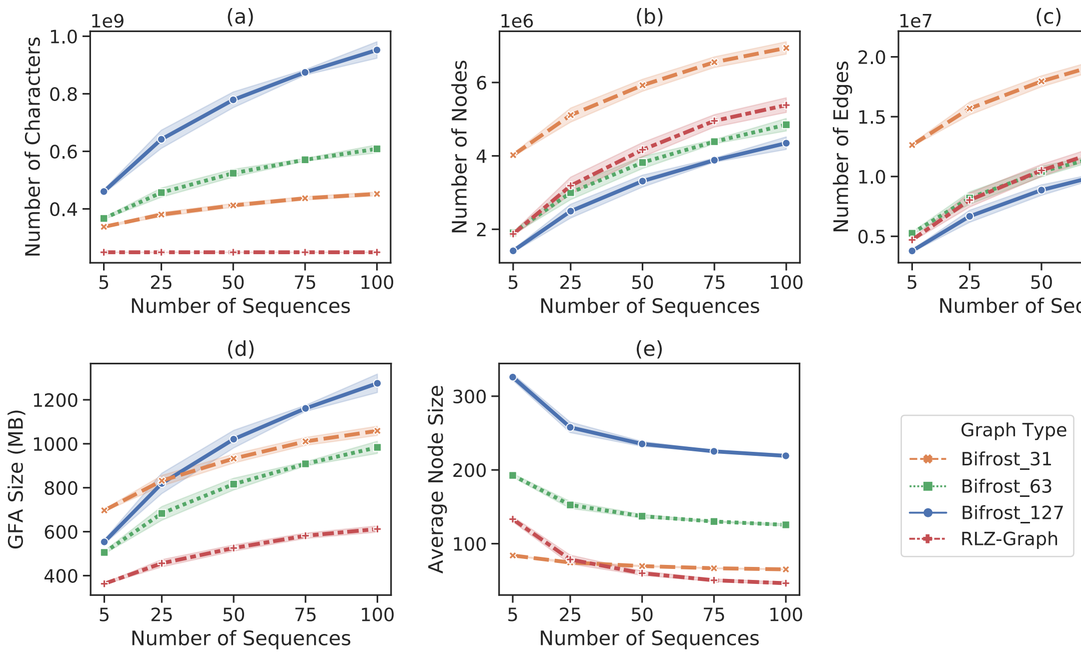


Figure 2.7: Comparison between RLZ-Graph and ccDBG constructed by Bifrost with $k = 31$, 63 and 127 on human chromosome 1 sequences. (a) Total number of characters in the node labels. (b) Number of nodes. (c) Number of edges. (d) Size of the GFA file that stores the graph structure and node labels. The shaded region represents the standard deviation across 5 experiments and each data point in the plots represents the mean across 5 experiments.

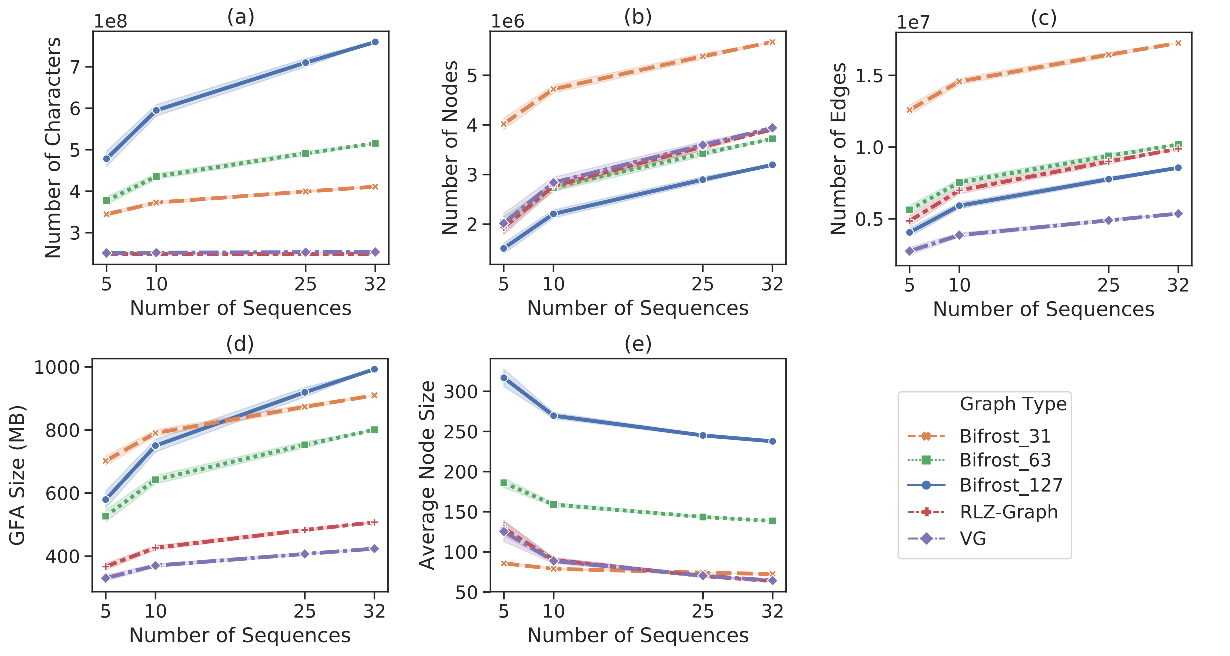


Figure 2.8: Comparison across RLZ-Graph, ccDBG constructed by Bifrost and variation graph constructed by VGtoolkit. (a) The total number of characters in the node labels. (b) The number of nodes. (c) The number of edges. (d) Size of the GFA file that stores the graph structure and node labels. (e) The average number of characters in node labels. The shaded region represents the standard deviation across 5 experiments and each data point in the plots represents the mean across 5 experiments.

in the concatenated node labels are constant in the RLZ-Graph regardless of the increase of the number of sequences because nodes are produced by cutting a reference string (Figure 2.7(a)). At 100 sequences, the GFA file that stores the RLZ-Graph is 37% smaller than the GFA file storing the colored de Bruijn graph produced by Bifrost with $k = 63$ and is 42.2% smaller when $k = 31$ (Figure 2.7(d)). When k is smaller, the ccdBG becomes impractical for human chromosomes in terms of running time (Figures 2.10-2.12). Therefore, we do not include the results of graphs constructed with smaller k values in Figure 2.7.

2.8.2 Comparison between ccdBGs, variation graphs and RLZ-Graphs on HGSVC data

In addition to SNPs, the Human Genome Structural Variants Consortium (HGSVC) dataset [41] provides the set of large-scale insertion, deletion, inversion and translocation events. To evaluate RLZ-Graph on a more comprehensive set of variants, we ran RLZ-Graph, Bifrost and VG-toolkit [51] on the HGSVC dataset, which contains 32 samples from people in various populations.

While VGtoolkit supports genome graph construction given genomic sequences as input, it constructs the graph by iteratively aligning sequences to the graph [145–147], which can be inefficient when the lengths and the number of sequences are large. Therefore, we construct variation graphs based on the set of variants as input using VGtoolkit.

In each experiment, we randomly choose 5, 10, 25, 32 samples and generate their sequences using the consensus command from bcftools. We construct the ccdBG with Bifrost and RLZ-Graph using the sample sequences and the reference hg38. We vary the k-mer sizes used for Bifrost and report the size of graphs with $k = 31, 63$ and 127. The sizes of graphs on disk are evaluated again using the size of the GFA file. VG format is converted to GFA using `vg view` command. We repeat each experiment 5 times.

As shown in Figure 2.8, the sizes of the graphs constructed by Bifrost and RLZ-Graph are similar to those built from the 1000 Genomes Project (Figure 4). RLZ-Graph constructs graphs that have similar sizes as variation graphs, which shows that RLZ-Graph does not depend on preprocessing steps to construct small genome graphs on full-length genomic sequences.

2.8.3 Running time and peak memory used by Bifrost and RLZ-Graph on 1000 Genome Project dataset

The average wall-clock running time and resident set size (RSS) of RLZ-Graph and Bifrost [62] with $k = 31,\ 63$ and 127 on chromosome 1 are reported in Table 2.1 and 2.2. It takes RLZ-Graph around 2.5 hours to build a graph with 100 chromosome 1 sequences. The running time includes the time to do RLZ factorization. In all experiments, Bifrost is run in parallel in 20 threads while RLZ-Graph is run in a single thread.

The RLZ-Graph implementation is not optimized and not parallelized compared to the implementation of Bifrost. Still, the running time of RLZ-Graph is on a similar scale compared to Bifrost. While RLZ-Graph is not optimized for memory usage, the peak memory, measured by the resident set size (RSS), used by RLZ-Graph grows linearly in the number of input sequences. A future direction would be to improve the implementation of RLZ-Graph by parallelizing the RLZ factorization step.

When $k = 15$, the size of the GFA file that stores the ccdBG is 15 gigabytes for 5 sequences of chromosome 1 and the running time of Bifrost is around 8 hours, while the running time is 396 seconds for $k = 31$. Both the size of the graph and the running time is impractical compared to other k values. When $k = 3$, the size of the GFA file is 4.2 kilobytes for 5 sequences of chromosome 1 with 32 nodes and 127 edges, and the running time of Bifrost is around 2.5 hours. Although the graph is small, it is similar to the G_4 solution to the genome graph size optimization problem, where the length of the reconstruction path is approximately the same as the original string.

2.9. Performance of various source assignment heuristics on *E. coli* genomes

In the EPM formulation, assigning different sources to a pointer does not change the size of the compressed string, but may affect the number of nodes.

Number of sequences	5	25	50	75	100
RLZ-Graph time (s)	1031	2707	4758	6872	9136
Bifrost k=31 time (s)	396	656	986	1542	1852
Bifrost k=63 time (s)	280	510	793	1126	1332
Bifrost k=127 time (s)	412	733	1098	1443	1744

Table 2.1: Average wall-clock running time of RLZ-Graph and Bifrost with different k values on chromosome 1 sequences.

Number of sequences	5	25	50	75	100
RLZ-Graph RSS (MB)	8562	13316	19258	25918	33762
Bifrost k=31 RSS (MB)	4450	4716	7510	5019	7550
Bifrost k=63 RSS (MB)	6904	6961	7079	7528	7567
Bifrost k=127 RSS (MB)	6949	7059	7655	7727	7803

Table 2.2: Average resident set size of RLZ-Graph and Bifrost with different k values on chromosome 1 sequences.

Aside from the ILP solution to the source assignment problem, sources are chosen by other heuristics in literature regarding RLZ factorization [75]. Specifically, from the source set corresponding to a phrase, the leftmost source on the reference string is chosen (Left), or the lexicographically smallest source is chosen (Lex). A source $s_i = (pos_i, len_i)$ is to the left of source $s_j = (pos_j, len_j)$, or $s_i <_{left} s_j$, if $pos_i < pos_j$. A source s_i is lexicographically smaller than s_j, or $s_i <_{lex} s_j$, if $R[pos_i : |R| - 1] < R[pos_j : |R| - 1]$ given a reference string R. We compare the number of nodes eliminated by various source assignment heuristics on *E. coli* genomes.

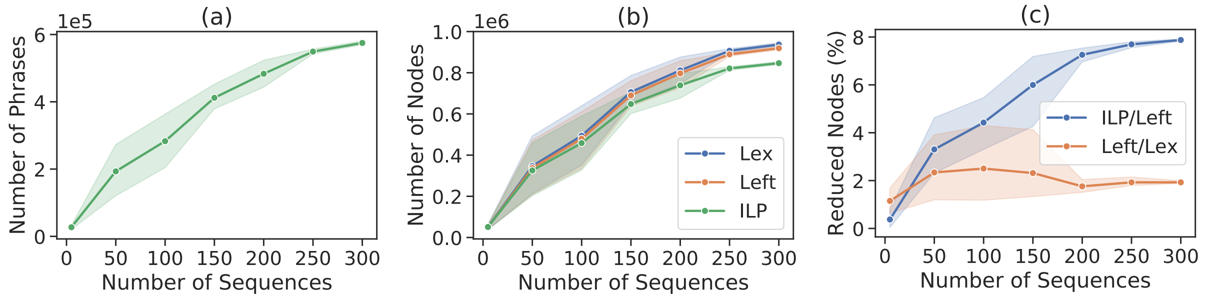


Figure 2.9: Performance of heuristics solving the source assignment problem. (a) The number of phrases. (b) The number of nodes. (c) Percentage of nodes was reduced using the leftmost heuristic and the ILP solution during the source assignment step. The shaded area in the plots represents the standard deviation across 5 experiments and each data point in the plots represents the mean across 5 experiments. Lex: lexicographical heuristic. Left: leftmost heuristic. ILP: ILP solution.

We obtain 300 genomic assemblies of *E. coli* O157 strain from Genbank [26]. In each experiment, we randomly permute the 300 sequences and construct the RLZ-Graph on the first 50, 100, 150, 200, 250, 300 sequences. The first sequence in the randomly permuted 300 sequences is used as the reference string. We repeat each experiment 5 times.

In Figure 2.9(a), we show the rate at which the number of phrases produced by the RLZ factorization increases as the number of sequences increases. In Figure 2.9(b), we show the number of nodes produced due to different source assignment strategies. The ILP solution has the best performance and results in the fewest nodes. The percentage of reduced nodes is around 8% for 300 *E. coli* sequences. As the number of sequences increases, the ILP solution is able to eliminate more nodes compared to the heuristic that always chooses the leftmost source (Figure 2.9(c)). The percentage of eliminated nodes is calculated as $1 - (|V|_{ILP}/|V|_{Left})$ and $1 - (|V|_{Left}/|V|_{Lex})$, respectively.

Solving the source assignment problem prior to graph construction reduces the number of nodes by around 8%. Although it is a relatively small percentage, when dealing with very large genome graphs, it translates into substantial space-savings.

2.10. Discussion

We define the restricted genome graph and formalize the restricted genome graph size optimization problem. The optimization problem balances both the size of the graph structure and the length of the reconstruction paths of sequences stored in the graph, which is similar to the string compression problem. Inspired by the similarity, we present a pair of algorithms that bridge genome graph construction and the external pointer macro model. We prove an upper bound on the size of the genome graph that is constructed based on an optimally compressed string from the EPM model. One key advantage of our graph construction algorithm is that the total number of characters stored in the graph is always equal to the size of the reference string regardless of the number of sequences stored in the graph. The constant number of characters stored in the graph keeps the space taken by the graph small. Further, since the number of nodes and edges are derived from an already compressed representation of strings, the number of nodes and the number of edges remain small.

Equivalent choices made by data compression algorithms may affect the size of the genome graph differently (Section 2.9). We address this discrepancy by solving the source assignment problem, which is not limited to the relative Lempel-Ziv algorithm but can be applied to any EPM-compressed form to reduce the number of nodes and edges. The NP-completeness of the source assignment problem is still open.

As a proof-of-concept that compression-based genome graph construction algorithms can produce small genome graphs, we implement RLZ-Graph based on the relative Lempel-Ziv algorithm [74]. We show that RLZ-Graph can reduce the size of the graph significantly on disk compared to the colored compacted de Bruijn graph.

RLZ-Graph does not depend on hyperparameters or preprocessing steps to construct genome graphs on full-length genomic sequences. The choice of k-mer sizes is important in de Bruijn graph construction as it significantly affects the size of the graph. RLZ-Graph removes this dependence on the choice of k and produces practical graphs with a smaller size that is scalable to the entire human genome. On the other hand, RLZ-Graph produces graphs with similar sizes to VGtoolkit, even when the genome sequences are not processed by variant callers or sequence aligners.

While existing genome graph indexing methods (e.g. [145–147]) can be applied to RLZ-Graphs for downstream genomics analysis, such as alignment and variant calling, it may be more efficient to use an index specialized for RLZ factorization. There has been a line of work focusing on fast sequence query given a string compressed by the RLZ algorithm [40, 47, 110]. It is possible to extend these text indices to graph indices that enable faster sequence queries.

This work is an initial investigation into the connection between genome graph construction and string compression. We show that by using compression algorithms, we can build small genome graphs efficiently, which opens up the possibilities in future research in adapting other data compression schemes to genome graph construction.

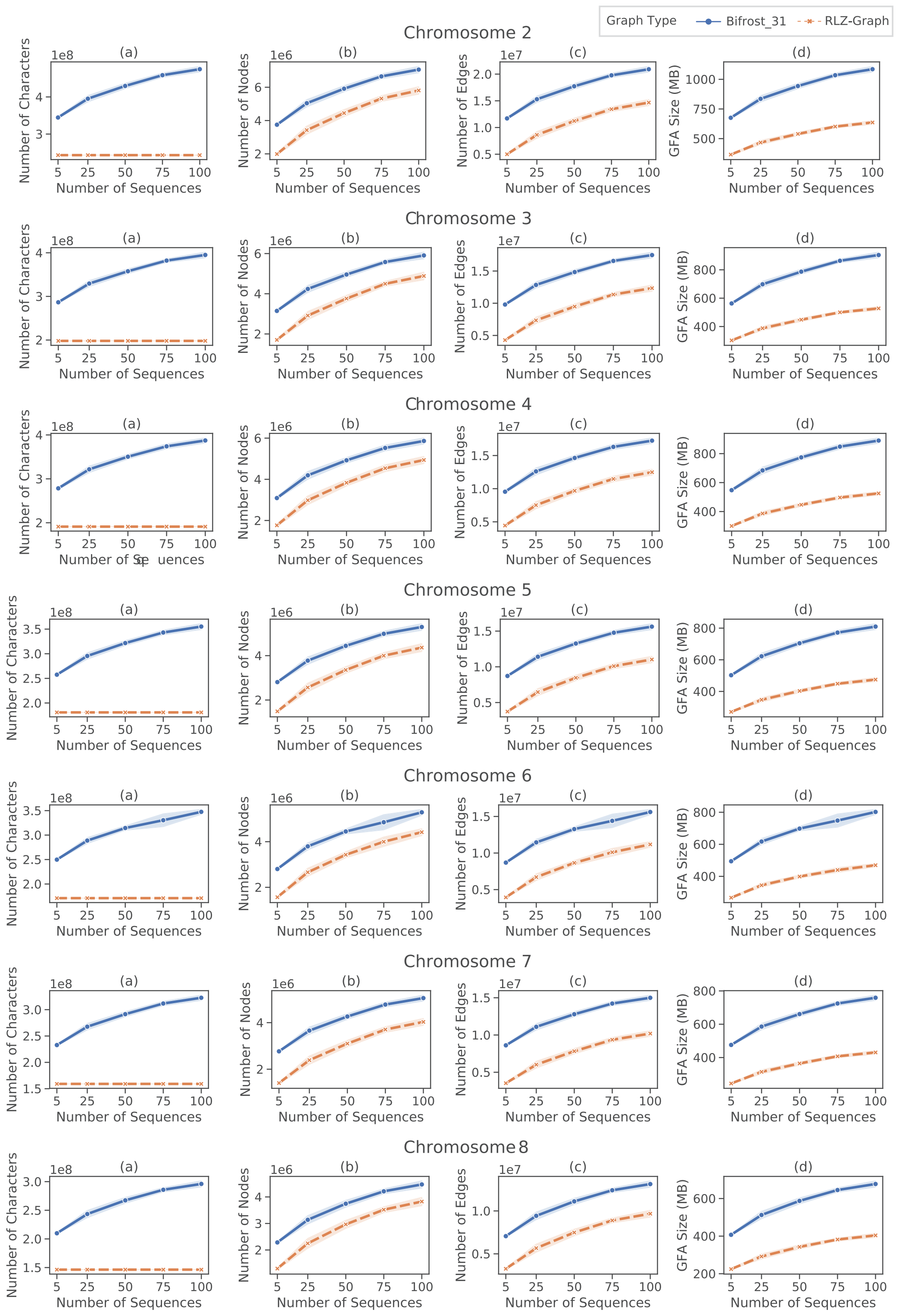


Figure 2.10: Comparison between RLZ-Graph and ccdBG constructed by Bifrost with $k = 31$ on human chromosomes 2–8. (a) Total number of characters in the node labels. (b) Number of nodes. (c) Number of edges. (d) Size of GFA file that stores the graph structure and node labels.

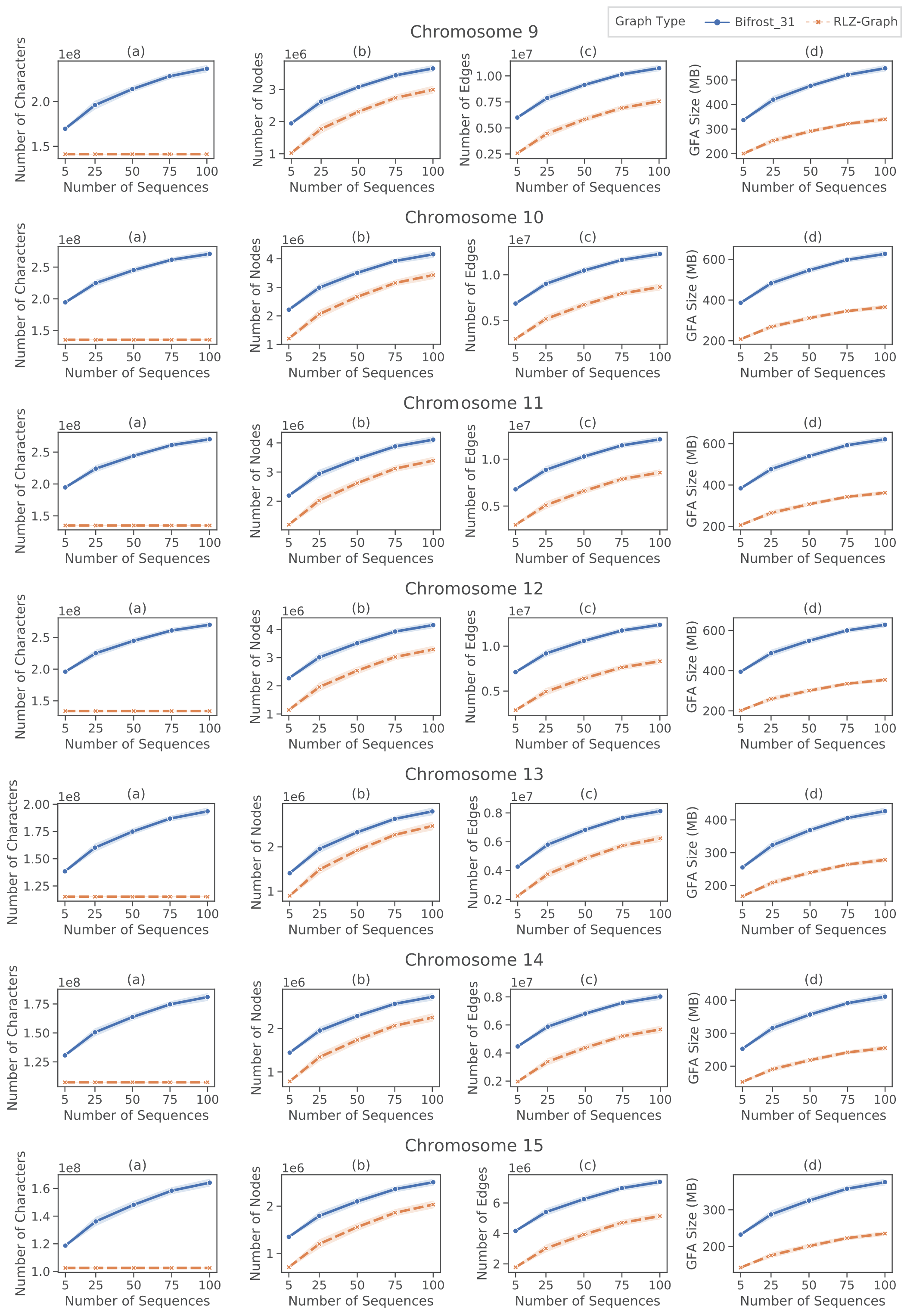


Figure 2.11: Comparison between RLZ-Graph and ccdBG constructed by Bifrost with $k = 31$ on human chromosomes 9–15. (a) Total number of characters in the node labels. (b) Number of nodes. (c) Number of edges. (d) Size of GFA file that stores the graph structure and node labels.

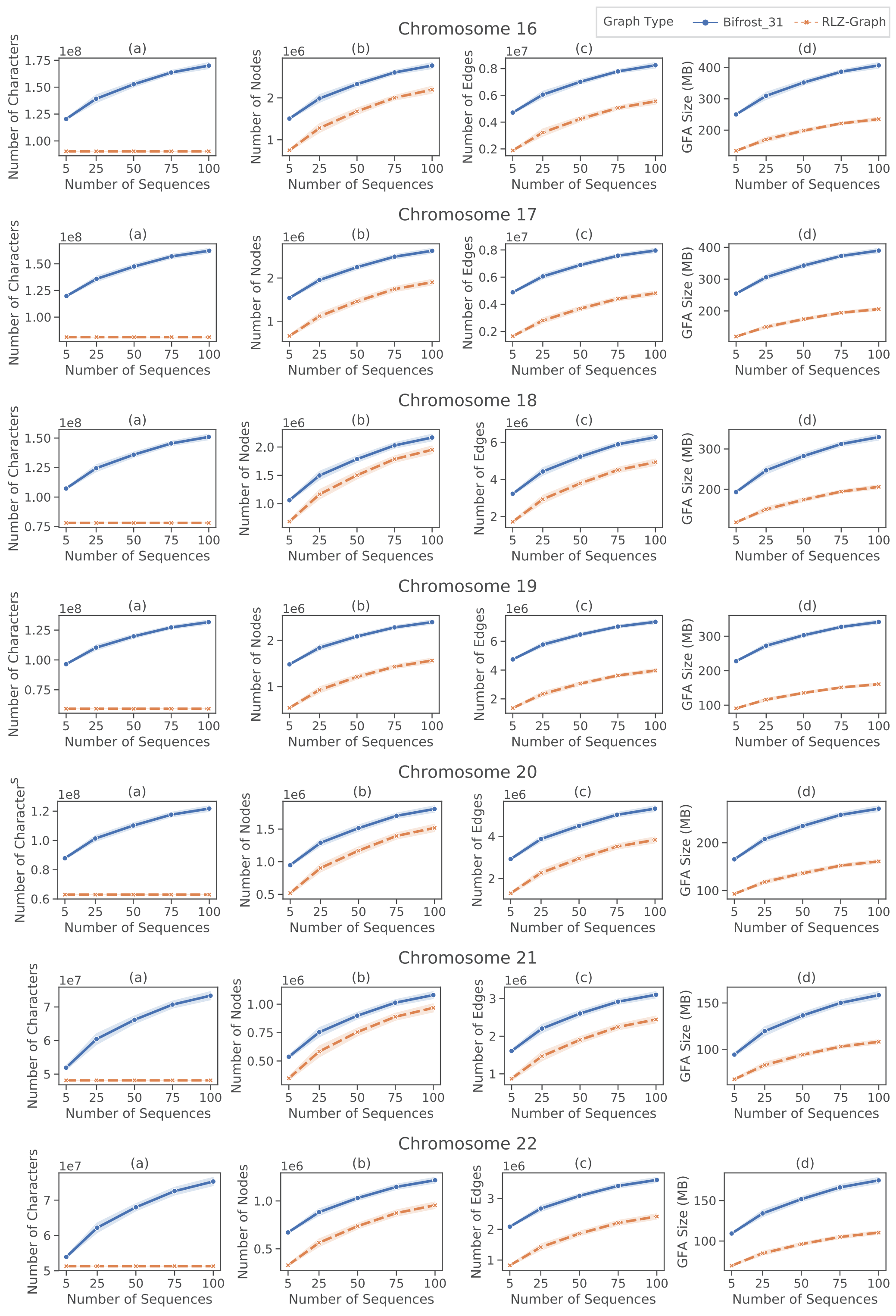


Figure 2.12: Comparison between RLZ-Graph and ccdBG constructed by Bifrost with $k = 31$ on human chromosomes 16–22. (a) Total number of characters in the node labels. (b) Number of nodes. (c) Number of edges. (d) Size of GFA file that stores the graph structure and node labels.

Chapter 3

The Complexity of and Algorithms for Genome Graph Comparison

This chapter is joint work with Yihang Shen and Carl Kingsford. The code to reproduce the results in this chapter can be found in https://github.com/Kingsford-Group/gtednewilp.

3.1. Introduction

Graph traversal edit distance (GTED) [16] is an elegant measure of the similarity between the strings represented by edge-labeled Eulerian graphs. For example, given two de Bruijn assembly graphs [123], computing GTED between them measures the similarity between two genomes without the computationally intensive and possibly error-prone process of assembling the genomes. Using an approximation of GTED between assembly graphs of Hepatitis B viruses, Boroojeny et al. [16] group the viruses into clusters consistent with their taxonomy. This can be extended to inferring phylogeny relationships in metagenomic communities or comparing heterogeneous disease samples such as cancer. There are several other methods to compute a similarity measure between strings encoded by two assembly graphs [95, 101, 102, 124]. GTED has the advantage that it does not require prior knowledge on the type of the genome graph or the complete sequence of the input genomes. The input to the GTED problem is two unidirectional, edge-labeled Eulerian graphs, which are defined as:

Definition 8 (Unidirectional, edge-labeled Eulerian Graph)**.** *A unidirectional, edge-labeled Eulerian graph is a connected directed graph $G = (V, E, \ell, \Sigma)$, with node set V, edge multi-set E, constant-size alphabet Σ, and single-character edge labels $\ell : E \to \Sigma$, such that G contains an Eulerian trail that traverses every edge $e \in E$ exactly once. The unidirectional condition means that all edges between the same pair of nodes are in the same direction.*

Such graphs arise in genome assembly problems (e.g. the de Bruijn subgraphs). Computing GTED is the problem of computing the minimum edit distance between the two most similar strings represented by Eulerian trails each input graph.

Problem 3 (Graph Traversal Edit Distance (GTED) [16])**.** *Given two unidirectional, edge-*

labeled Eulerian graphs G_1 and G_2, compute

$$\mathrm{GTED}(G_1, G_2) \triangleq \min_{\substack{t_1 \in \mathit{trails}(G_1) \\ t_2 \in \mathit{trails}(G_2)}} \mathit{edit}(\mathit{str}(t_1), \mathit{str}(t_2)). \tag{3.1}$$

Here, $\mathit{trails}(G)$ is the collection of all Eulerian trails in graph G, $\mathit{str}(t)$ is a string constructed by concatenating labels on the Eulerian trail $t = (e_0, e_1, \ldots, e_n)$, and $\mathit{edit}(s_1, s_2)$ is the edit distance between strings s_1 and s_2.

Boroojeny et al. [16] claim that GTED is polynomially solvable by proposing an integer linear programming (ILP) formulation of GTED and arguing that the constraints of the ILP make it polynomially solvable. This result, however, conflicts with several complexity results on string-to-graph matching problems. Kupferman and Vardi [73] show that it is NP-complete to determine if a string exactly matches an Eulerian tour in an edge-labeled Eulerian graph. Additionally, Jain et al. [66] show that it is NP-complete to compute an edit distance between a string and strings represented by a labeled graph if edit operations are allowed on the graph. On the other hand, polynomial-time algorithms exist to solve string-to-string alignment [111] and string-to-graph alignment [66] when edit operations on graphs are not allowed.

We resolve the conflict among the results on complexity of graph comparisons by revisiting the complexity of and the proposed solutions to GTED. We prove that computing GTED is NP-complete by reducing from the HAMILTONIAN PATH problem, reaching an agreement with other related results on complexity. Further, we point out with a counter-example that the optimal solution of the ILP formulation proposed by Boroojeny et al. [16] does not solve GTED.

We give two ILP formulations for GTED. The first ILP has an exponential number of constraints and can be solved by subtour elimination iteratively [31, 38]. The second ILP has a polynomial number of constraints and shares a similar high-level idea of the global ordering approach [38] in solving the TRAVELING SALESMAN problem [100].

While the optimal solution to ILP proposed in Boroojeny et al. [16] does not solve GTED, it does compute a lower bound to GTED. We characterize the cases when GTED is equal to this lower bound. In addition, we point out that solving this ILP formulation finds a minimum-cost matching between closed-trail decompositions in the input graphs, which may be used to compute the similarity between repeats in the genomes. Boroojeny et al. [16] claim their proposed ILP formulation is solvable in polynomial time by arguing that the constraint matrix of the linear relaxation of the ILP is always totally unimodular. We show that this claim is false by proving that the constraint matrix is not always totally unimodular and showing that there exists optimal fractional solutions to its linear relaxation.

We evaluate the efficiency of solving ILP formulations for GTED and its lower bound on simulated genomic strings and show that it is impractical to compute GTED on larger genomes.

In summary, we revisit two important problems in genome graph comparisons: Graph Traversal Edit Distance (GTED). We show that GTED is NP-complete, and provide the first correct ILP formulations for it. We also show that the ILP formulation proposed by [16] is a lower bound to GTED. We evaluate the efficiency of the ILPs for GTED and its lower bound on genomic sequences. These results provide solid algorithmic foundations for continued algorithmic innovation on the task of comparing genome graphs and point to the direction of approximation heuristics.

3.2. GTED is NP-complete

3.2.1 Conflicting results on computational complexity of GTED and string-to-graph matching

The natural decision versions of all of the computational problems described in this chapter are clearly in NP. Under the assumption that P $\neq$ NP, the results on the computational complexity of GTED and string-to-graph matching claimed in Boroojeny et al. [16] and Kupferman and Vardi [73], respectively, cannot be both true.

Kupferman and Vardi [73] show that the problem of determining if an input string can be spelled by concatenating edge labels in an Eulerian trail in an input graph is NP-complete. We call this problem EULERIAN TRAIL EQUALING WORD. We show in Theorem 2 that we can reduce ETEW to GTED, and therefore if GTED is polynomially solvable, then ETEW is polynomially solvable.

Problem 4 (Eulerian Trail Equaling Word [73]). *Given a string $s \in \Sigma^*$, an edge-labaled Eulerian graph G, find an Eulerian trail t of G such that $str(t) = s$.*

Theorem 2. *If* GTED $\in$ *P then* ETEW $\in$ *P.*

Proof. Let $\langle s, G \rangle$ be an instance of ETEW. Construct a directed, acyclic graph (DAG), C, that has only one path. Let the path in C be $P = (e_1, \ldots, e_{|s|})$ and the edge label of e_i be $s[i]$. Clearly, C is a unidirectional, edge-labeled Eulerian graph, P is the only Eulerian trail in C, and $str(P) = s$.

For the graph $G = (V_G, E_G, \ell_G, \Sigma)$ from the ETEW instance, which may not be unidirectional, create another graph G' that contains all of the nodes and edges in G except the anti-parallel edges. Let $\Sigma_{G'} = \Sigma \cup \{\epsilon\}$, where ϵ is a character that is not in Σ. For each pair of anti-parallel edges (u, v) and (v, u) in G, add four edges $(u, w_1), (w_1, v), (v, w_2), (w_2, u)$ by introducing new vertices w_1, w_2 to G'. Let $\ell_{G'}(u, w_1) = \ell_G(u, v)$ and $\ell_{G'}(w_2, u) = \ell_G(v, u)$. Let $\ell_{G'}(w_1, v) = \ell_{G'}(v, w_2) = \epsilon$ for every newly introduced vertex. G' has at most twice the number of edges as G and is Eulerian and unidirectional.

Define the cost of changing a character from a to b cost(a, b) for $a, b \in \Sigma \cup \{-\}$ to be 0 if $a = b$ and 1 otherwise. "$-$" is the gap character indicating an insertion or a deletion. Define cost(a, ϵ) with $a \in \Sigma$ to be 1. Define cost$(-, \epsilon)$ to be 0.

Use the (assumed) polynomial-time algorithm for GTED to ask whether GTED$(C, G') \leq 0$ under edit distance Σ. If yes, then let (s_1, s_2) be the 0-cost alignment of the strings spelled out by the trails in C and G', respectively. The non-gap characters of s_1 must spell out s since there is only one Eulerian trail in C. Because the alignment cost is 0, any $-$ (gap) characters in s_1 must be aligned with ϵ characters in s_2 and any non-gap characters in s_1 must be aligned to the same character in s_2. The trail in G' that spells s_2 can be transformed to a trail that spells s_3 by collapsing the edges with ϵ character labels, and $s_3 = s_1$.

If GTED$(C, G') > 0$, G must not contain an Eulerian trail that spells s. Otherwise, such a trail could be extended to a trail introducing some ϵ characters that could be aligned to s with zero cost by aligning gaps with ϵ characters. $\square$

Hence, an (assumed) polynomial-time algorithm for GTED solves ETEW in polynomial time. This contradicts Theorem 6 of Kupferman and Vardi [73] of the NP-completeness of ETEW (under P $\neq$ NP).

3.2.2 Reduction from Hamiltonian Path to GTED

We resolve the contradiction by showing that GTED is NP-complete.

Theorem 3. GTED *is NP-complete.*

Proof. We reduce from the HAMILTONIAN PATH problem, which asks whether a directed, simple graph G contains a path that visits every vertex exactly once. Here simple means no self-loops or parallel edges. Let $\langle G = (V, E)\rangle$ be an instance of HAMILTONIAN PATH, with $n = |V|$ vertices. The reduction is almost identical to that presented in Kupferman and Vardi [73], and from here until noted later in the proof the argument is identical except for the technicalities introduced to force unidirectionality (and another minor change described later). The first step is to construct the Eulerian closure of G, which is defined as $G' = (V', E')$ where

$$V' = \{v^{in}, v^{out} : v \in V\} \cup \{w\}, \tag{3.2}$$

and E' is the union of the following sets of edges and their labels:

- $E_1 = \{(v^{in}, v^{out}) : v \in V\}$, labeled `a`,
- $E_2 = \{(u^{out}, v^{in}) : (u, v) \in E\}$, labeled `b`,
- $E_3 = \{(v^{out}, v^{in}) : v \in V\}$, labeled `c`,
- $E_4 = \{(v^{in}, u^{out}) : (u, v) \in E\}$, labeled `c`,
- $E_5 = \{(u^{in}, w) : u \in V\}$, labeled `c`,
- $E_6 = \{(w, u^{in}) : u \in V\}$, labeled `b`.

Since G' is connected and every outgoing edge in G' has a corresponding antiparallel incoming edge, G' is Eulerian. It is not unidirectional, so we further create G'' from G' by adding dummy nodes to each pair of antiparallel edges and labelling the length-2 paths so created with `x#`, where `x` is the original label of the split edge (`a`, `b`, or `c`) and `#` is some new symbol (shared between all the new edges). We call these length-2 paths introduced to achieve unidirectionality "split edges".

We now argue that G has a Hamiltonian path iff G'' has an Eulerian trail that spells out

$$q = \mathtt{a\#}(\mathtt{b\#a\#})^{n-1}(\mathtt{c\#})^{2n-1}(\mathtt{c\#b\#})^{|E|+1}. \tag{3.3}$$

If such an Eulerian trail exists, then the trail starts with spelling the string $\mathtt{a\#}(\mathtt{b\#a\#})^{n-1}$, which corresponds to a Hamiltonian trail in G since it visits exactly n "vertex split edges" (type E_1, labeled `a#`) and each vertex split edge can be used only once (since it is an Eulerian trail). Further, successively visited vertices must be connected by an edge in G since those are the only `b#` split edges in G'' (except those leaving w, but w must not be involved in spelling out $\mathtt{a\#}(\mathtt{b\#a\#})^{n-1}$, since entering w requires using a split edge labeled `c#`).

For the other direction, if a G has a Hamiltonian path $v_1, \ldots, v_n$, then walking that sequence of vertices in G'' will spell out $\mathtt{a\#}(\mathtt{b\#a\#})^{n-1}$. This path will cover all E_1 edges and the E_2 edges that are on the Hamiltonian path. Retracing the path so far in reverse will use $2n - 1$ split edges labeled c#, consuming the $(\texttt{c\#})^{2n-1}$ term in q and covering all nodes' reverse vertex edges E_3 (since the path is Hamiltonian). The reverse path also covers the E_4 edges corresponding to reverse Hamiltonian path edges. Our Eulerian trail is now "at" node v_1^{in}.

What remains is to complete the Eulerian walk covering (a) edges and their antiparallel counterparts corresponding to edges in G that were not used in the Hamiltonian path, and (b) the edges adjacent to node w. To do this, define pred(v) be the vertices u in G for which edge (u, v) exists and u is not the predecessor of v along the Hamiltonian path. For each $u \in$ pred(v_1), traverse the split edge labeled c# to u^{out} then traverse the forward split edge labeled b# back to v_1^{in}. This results in a string $(\texttt{c\#b\#})^{|\text{pred}(v_1)|}$. Once the predecessors of v_1 are exhausted, traverse the split edge labeled c# from v_1^{in} into node w and then traverse the split edge labeled b# to v_2^{in}. This again generates a c#b# string. Repeat the process, covering the edges of v_2's predecessors and returning to w to move to the next node along the Hamiltonian path for each node $v_3, \ldots, v_n$. After covering the predecessors of v_n^{in}, go to v_1^{in} through the remaining edges in E_5 and E_6, (v_n^{in}, w) and (w, v_1^{in}), which completes the Eulerian tour. This covers all the edges of G''. The word spelled out in this last section of the Eulerian trail is a sequence of repetitions of c#b#, with one repetition for each edge that is not in the Hamiltonian path ($|E| - n + 1$) and all of the edges in E_5 and E_6 for entering and leaving each node ($2n$), with a total of $|E| + 1$ repetitions, which is the final $(\texttt{c\#b\#})^{|E|+1}$ term in q.

This ends the slight modification of the proof in Kupferman and Vardi [73], where the differences are (a) the introduction of the # characters and (b) using the exponent $|E| + 1$ of the final part of q instead of $|E| + n + 1$ as in Kupferman and Vardi [73] since we create w-edges only to v^{in} vertices. (This second change has no material effect on the proof, but reduces the length of the string that must be matched.)

Now, given an instance $\langle G = (V, E)\rangle$ of HAMILTONIAN PATH, with $n = |V|$ vertices, we construct G'' as above (obtaining a unidirectional Eulerian graph) and create graph C that only represents string q. Note that $|\Sigma| = 4$ and G'' and C can be constructed in polynomial time. GTED$(G'', C) = 0$ if and only if an Eulerian path in G'' spells out q, since there can be no indels or mismatches. By the above argument, an Eulerian tour that spells out q exists if and only if G has a Hamiltonian path. □

3.3. Revisiting the correctness of the proposed ILP solutions to GTED

In this section, we revisit two proposed ILP solutions to GTED by Boroojeny et al. [16] and show that the optimal solution to these ILP is not always equal to GTED.

3.3.1 Alignment graph

The previously proposed ILP formulations for GTED are based on the alignment graph constructed from the input graphs. The high-level concept of an alignment graph is similar to the dynamic programming matrix for the string-to-string alignment problem [111].

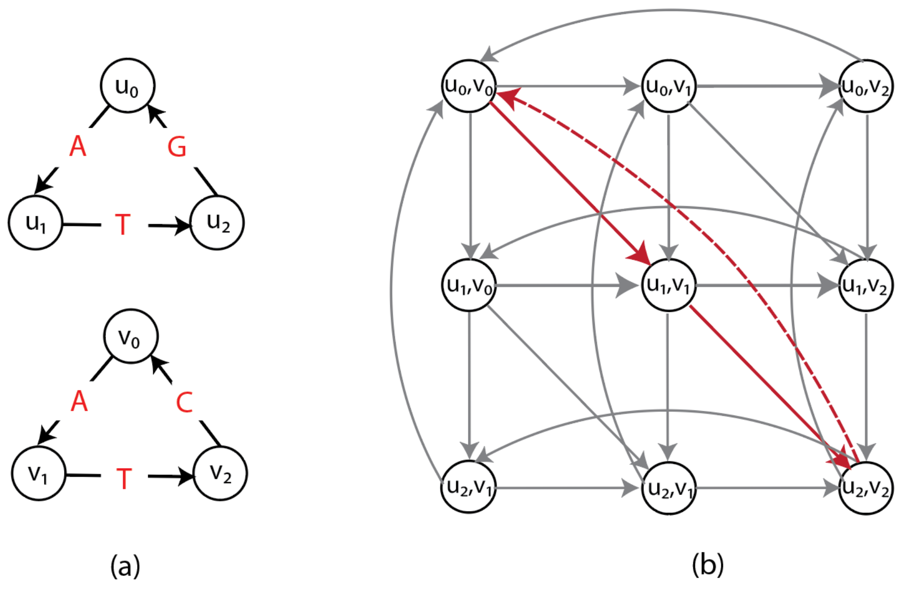


Figure 3.1: (a) An example of two edge-labeled Eulerian graphs G_1 (top) and G_2 (bottom). (b) The alignment graph $\mathcal{A}(G_1, G_2)$. The cycle with red edges is the path corresponding to GTED(G_1, G_2). Red solid edges are matches with cost 0 and red dashed-line edge is mismatch with cost 1.

Definition 9 (Alignment graph)**.** *Let G_1, G_2 be two unidirectional, edge-labeled Eulerian graphs. The* alignment graph $\mathcal{A}(G_1, G_2) = (V, E, \delta)$ *is a directed graph that has vertex set $V = V_1 \times V_2$ and edge multi-set E that equals the union of the following:*
Vertical edges $[(u_1, u_2), (v_1, u_2)]$ *for* $(u_1, v_1) \in E_1$ *and* $u_2 \in V_2$,
Horizontal edges $[(u_1, u_2), (u_1, v_2)]$ *for* $u_1 \in V_1$ *and* $(u_2, v_2) \in E_2$,
Diagonal edges $[(u_1, u_2), (v_1, v_2)]$ *for* $(u_1, v_1) \in E_1$ *and* $(u_2, v_2) \in E_2$.
Each edge is associated with a cost by the cost function $\delta : E \to \mathbb{R}$.

Each diagonal edge $e = [(u_1, v_1), (u_2, v_2)]$ in an alignment graph can be projected to (u_1, v_1) and (u_2, v_2) in G_1 and G_2, respectively. Similarly, each vertical edge can be projected to one edge in G_1, and each horizontal edge can be projected to one edge in G_2.

We define the edge projection function π_i that projects an edge from the alignment graph to an edge in the input graph G_i. We also define the path projection function Π_i that projects a trail in the alignment graph to a trail in the input graph G_i. For example, let a trail in the alignment graph be $p = (e_1, e_2, \ldots, e_m)$, and $\Pi_i(p) = (\pi_i(e_1), \pi_i(e_2), \ldots, \pi_i(e_m))$ is a trail in G_i.

An example of an alignment graph is shown in Figure 3.1(b). The horizontal edges correspond to gaps in strings represented by G_1, vertical edges correspond to gaps in strings represented by G_2, and diagonal edges correspond to the matching between edge labels from the two graphs. In the rest of this paper, we assume that the costs for horizontal and vertical edges are 1, and the costs for the diagonal edges are 1 if the diagonal edge represents a mismatch and 0 if it is a match. The cost function δ can be defined to capture the cost of matching between edge labels or inserting gaps. This definition of alignment graph is also a generalization of the alignment graph used in string-to-graph alignment [66].

3.3.2 The first previously proposed ILP for GTED

Lemma 1 in Boroojeny et al. [16] provides a model for computing GTED by finding the minimum-cost trail in the alignment graph. We reiterate it here for completeness.

Lemma 5 ([16])**.** *For any two edge-labeled Eulerian graphs G_1 and G_2,*

$$\begin{aligned} \mathrm{GTED}(G_1, G_2) = \text{minimize}_c \quad & \delta(c) \\ \text{subject to} \quad & c \textit{ is a trail in } \mathcal{A}(G_1, G_2), \\ & \Pi_i(c) \textit{ is an Eulerian trail in } G_i \textit{ for } i = 1, 2, \end{aligned} \tag{3.4}$$

where $\delta(c)$ is the total edge cost of c, and $\Pi_i(c)$ is the projection from c to G_i.

An example of such a minimum-cost trail is shown in Figure 3.1(b). Boroojeny et al. [16] provide the following ILP formulation and claim that it is a direct translation of Lemma 5:

$$\underset{x\in\mathbb{N}^{|E|}}{\text{minimize}} \quad \sum_{e\in E} x_e \delta(e) \tag{3.5}$$

$$\text{subject to} \quad Ax = 0 \tag{3.6}$$

$$\sum_{e\in E} x_e I_i(e, f) = 1 \quad \text{for } i = 1, 2 \text{ and for all } f \in E_i \tag{3.7}$$

$$A_{ue} = \begin{cases} -1 & \text{if } e = (u, v) \in E \text{ for some vertex } v \in V \\ 1 & \text{if } e = (v, u) \in E \text{ for some } u \in V \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

Here, E is the edge set of $\mathcal{A}(G_1, G_2)$. A is the negative incidence matrix of size $|V| \times |E|$, and $I_i(e, f)$ is an indicator function that is 1 if edge e in E projects to edge f in the input graph G_i (and 0 otherwise). We define the domain of each x_e to include all non-negative integers. However, due to constraints (3.7), the values of x_e are limited to either 0 or 1. We describe this ILP formulation with the assumption that both input graphs have closed Eulerian trails, which means that each node has equal numbers of incoming and outgoing edges. We discuss the cases when input graphs contain open Eulerian trails in Section 3.4.

The ILP in (3.5)-(3.8) allows the solutions to select disjoint cycles in the alignment graph, and the projection of edges in these disjoint cycles does not correspond to a single string represented by either of the input graphs. We show that the ILP in (3.5)-(3.8) does not solve GTED by giving an example where the objective value of the optimal solution to the ILP in (3.5)-(3.8) is not equal to GTED.

Construct two input graphs as shown in Figure 3.2(a). Specifically, G_1 spells circular permutations of `TTTGAA` and G_2 spells circular permutations of `TTTAGA`. It is clear that
$\mathrm{GTED}(G_1, G_2) = 2$ under Levenshtein edit distance. On the other hand, as shown in Figure 3.2(a), an optimal solution in $\mathcal{A}(G_1, G_2)$ contains two disjoint cycles with nonzero x_e values that have a total edge cost equal to 0. This solution is a feasible solution to the ILP in (3.5)-(3.8). It is also an optimal solution because the objective value is zero, which is the lower bound on

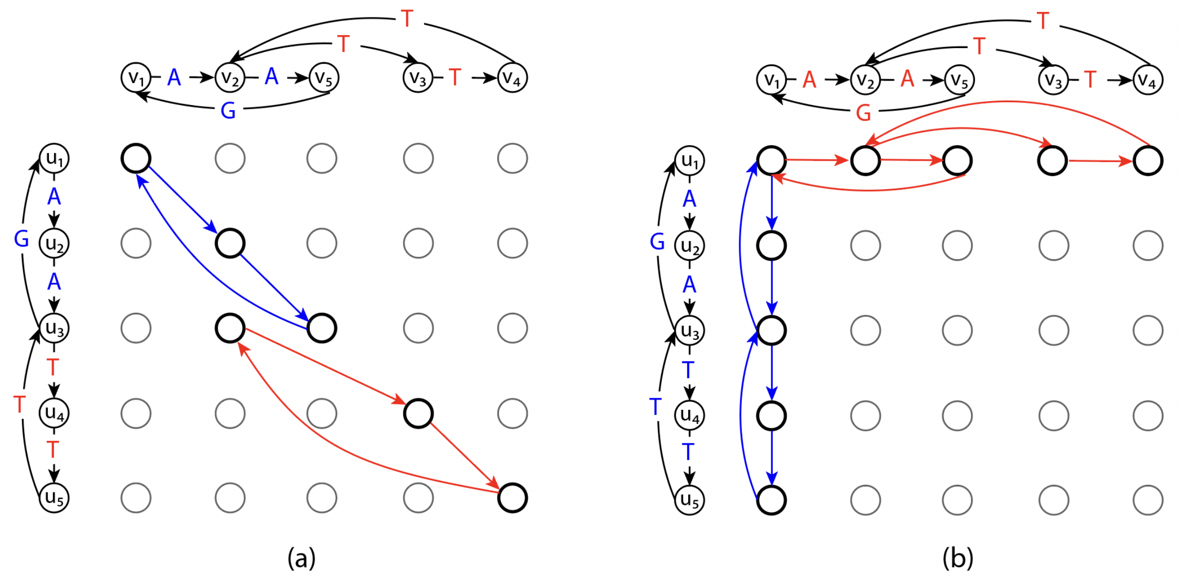


Figure 3.2: (a) The subgraph in the alignment graph induced by an optimal solution to the ILP in (3.5)-(3.8) and the ILP in (3.11)-(3.12) with input graphs on the left and top. The red and blue edges in the alignment graph are edges matching labels in red and blue font, respectively, and are part of the optimal solution to the ILP in (3.5)-(3.8). The cost of the red and blue edges are zero. (b) The subgraph induced by x^{init} with $s_1 = u_1$ and $s_2 = v_1$ according to the ILP in (3.11)-(3.12). The rest of the edges in the alignment graph are omitted for simplicity.

the ILP in (3.5)-(3.8). This optimal objective value, however, is smaller than GTED(G_1, G_2). Therefore, the ILP in (3.5)-(3.8) does not solve GTED since it allows the solution to be a set of disjoint components.

3.3.3 The second previously proposed ILP formulation of GTED

We describe the second proposed ILP formulation of GTED by Boroojeny et al. [16]. Following Boroojeny et al. [16], we use simplices, a notion from geometry, to generalize the notion of an edge to higher dimensions. A k-simplex is a k-dimensional polytope which is the convex hull of its $k+1$ vertices. For example, a 1-simplex is an undirected edge, and a 2-simplex is a triangle. We use the orientation of a simplex, which is given by the ordering of the vertex set of a simplex up to an even permutation, to generalize the notion of the edge direction [109, p. 26]. We use square brackets $[\cdot]$ to denote an oriented simplex. For example, $[v_0, v_1]$ denotes a 1-simplex with orientation $v_0 \to v_1$, which is a directed edge from v_0 to v_1, and $[v_0, v_1, v_2]$ denotes a 2-simplex with orientation corresponding to the vertex ordering $v_0 \to v_1 \to v_2 \to v_0$. Each k-simplex has two possible unique orientations, and we use the signed coefficient to connect their forms together, e.g. $[v_0, v_1] = -[v_1, v_0]$.

For each pair of graphs G_1 and G_2 and their alignment graph $\mathcal{A}(G_1, G_2)$, we define an oriented 2-simplex set $T(G_1, G_2)$ which is the union of:

- $[(u_1, u_2), (v_1, u_2), (v_1, v_2)]$ for all $(u_1, v_1) \in E_1$ and $(u_2, v_2) \in E_2$, or
- $[(u_1, u_2), (u_1, v_2), (v_1, v_2)]$ for all $(u_1, v_1) \in E_1$ and $(u_2, v_2) \in E_2$,

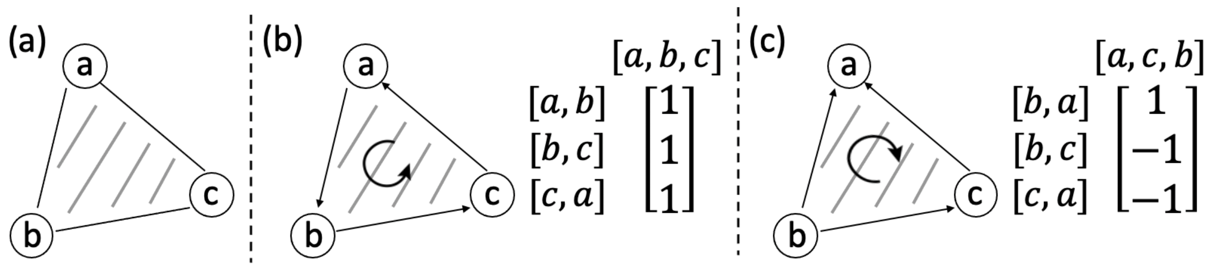


Figure 3.3: (a) A graph that contains an unoriented 2-simplex with three unoriented 1-simplices. (b), (c) The same graph with two different ways of orienting the simplices and the corresponding boundary matrices.

We use the boundary operator [109, p. 28], denoted by ∂, to map an oriented k-simplex to a sum of oriented $(k-1)$-simplices with signed coefficients.

$$\partial[v_0, v_1, \ldots, v_k] = \sum_{i=0}^{p} (-1)^i [v_0, \ldots, \hat{v_i}, \ldots, v_k], \tag{3.9}$$

where $\hat{v_i}$ denotes the vertex v_i is to be deleted. Intuitively, the boundary operator maps the oriented k-simplex to a sum of oriented $(k-1)$-simplices such that their vertices are in the k-simplex and their orientations are consistent with the orientation of the k-simplex. For example, when $k = 2$, we have:

$$\partial[v_0, v_1, v_2] = [v_1, v_2] - [v_0, v_2] + [v_0, v_1] = [v_1, v_2] + [v_2, v_0] + [v_0, v_1]. \tag{3.10}$$

We reiterate the second ILP formulation proposed in Boroojeny et al. [16]. Given an alignment graph $\mathcal{A}(G_1, G_2) = (V, E, \delta)$ and the oriented 2-simplex set $T(G_1, G_2)$,

$$\begin{aligned} \underset{x \in \mathbb{N}^{|E|}, y \in \mathbb{Z}^{|T(G_1,G_2)|}}{\text{minimize}} \quad & \sum_{e \in E} x_e \delta(e) \\ \text{subject to} \quad & x = x^{init} + [\partial] y \end{aligned} \tag{3.11}$$

Entries in x and y correspond to 1-simplices and 2-simplices in E and $T(G_1, G_2)$, respectively. $[\partial]$ is a $|E| \times |T(G_1, G_2)|$ boundary matrix where each entry $[\partial]_{i,j}$ is the signed coefficient of the oriented 1-simplex (the directed edge) in E corresponding to x_i in the boundary of the oriented 2-simplex in $T(G_1, G_2)$ corresponding to y_j. The index i, j for each 1-simplex or 2-simplex is assigned based on an arbitrary ordering of the 1-simplices in E or the 2-simplices in $T(G_1, G_2)$. An example of the boundary matrix is shown in Figure 3.3. $\delta(e)$ is the cost of each edge. $x^{init} \in \mathbb{R}^{|E|}$ is a vector where each entry corresponds to a 1-simplex in E with $|E_1| + |E_2|$ nonzero entries that represent one Eulerian trail in each input graph. x^{init} is a feasible solution to the ILP. Let s_1 be the source of the Eulerian trail in G_1, and s_2 be the sink of the Eulerian trail in G_2. Each entry in x^{init} is defined by

$$x_e^{init} = \begin{cases} 1 & \text{if } e = [(u_1, s_2), (v_1, s_2)] \text{ or } e = [(s_1, u_2), (s_1, v_2)], \\ 0 & \text{otherwise.} \end{cases} \tag{3.12}$$

If the Eulerian trail is closed in G_i, s_i can be any vertex in V_i. An example of x^{init} is shown in Figure 3.2(b).

3.3.4 Equivalence between two ILPs proposed by Boroojeny et al.

The analysis provided by Boroojeny et al. [16] states that the LP relaxation of the ILP in (3.5)-(3.8) does not always yield integer solutions, but the LP relaxation of the ILP in (3.11)-(3.12) always yields integer solutions. This suggests that the two LP relaxations have difference feasibility regions for x. We show that these two LP relaxations are actually equivalent in Theorem 4. Further, we show that the ILP in (3.5)-(3.8) and the ILP in (3.11)-(3.12) are also equivalent. Since the ILP in (3.5)-(3.8) does not solve for $\mathrm{GTED}(G_1, G_2)$ as shown in 3.3.2, we conclude that the ILP in (3.11)-(3.12) also does not solve $\mathrm{GTED}(G_1, G_2)$.

Theorem 4. *Given two unidirectional, edge-labeled Eulerian graphs G_1, G_2, the feasibility region of x in the LP relaxation of the ILP in* (3.11)-(3.12) *is the same as the feasibility region of x in the LP relaxation of the ILP in* (3.5)-(3.8).

Let $\mathcal{A}(G_1, G_2) = (V, E, \delta)$ be the alignment graph of $G_1 = (V_1, E_1, \ell_1, \Sigma_1)$ and $G_2 = (V_2, E_2, \ell_2, \Sigma_2)$, and let $T(G_1, G_2)$ be its two-simplex set. First, we have the following result:

Lemma 6. *Let $[y_i] \in \mathbb{R}^{|T(G_1,G_2)|}$ be a vector such that the j-th entry of $[y_i]$, $[y_i]_j$ is equal to 0 for all $j \neq i$. The vector $x' = x + [\partial][y_i]$ satisfies the constraints* (3.6)-(3.7) *if the vector x satisfies the constraints* (3.6)-(3.7).

Proof. Let $\sigma_i \in T(G_1, G_2)$ be the 2-simplex corresponding to the entry i of $[y_i]$. Based on the construction of $T(G_1, G_2)$, σ_i has two forms: $[(u_1, u_2), (v_1, u_2), (v_1, v_2)]$ or $[(u_1, u_2), (u_1, v_2), (v_1, v_2)]$. Without loss of generality, we assume $\sigma_i = [(u_1, u_2), (v_1, u_2), (v_1, v_2)]$. We can prove this lemma by using the same way when $\sigma_i = [(u_1, u_2), (u_1, v_2), (v_1, v_2)]$. Since

$$\partial\sigma_i = [(u_1, u_2), (v_1, u_2)] + [(v_1, u_2), (v_1, v_2)] - [(u_1, u_2), (v_1, v_2)],$$

We have

$$[\partial][y_i] = [y_i]_i[x_{e_1}] + [y_i]_i[x_{e_2}] - [y_i]_i[x_{e_3}],$$

where $e_1 = [(u_1, u_2), (v_1, u_2)]$, $e_2 = [(v_1, u_2), (v_1, v_2)]$, $e_3 = [(u_1, u_2), (v_1, v_2)]$, and $[x_e] \in \mathbb{R}^{|E|}$ is a vector such that all the entries are 0 except that the one corresponding to edge e is 1. we also let $[x_v] \in \mathbb{R}^{|V|}$ be a vector such that all the entries are 0 except that the one corresponding to vertex v is 1. Therefore, we have

$$Ax' = Ax + [y_i]_i[x_{v_2}] - [y_i]_i[x_{v_1}] + [y_i]_i[x_{v_3}] - [y_i]_i[x_{v_2}] - [y_i]_i[x_{v_3}] + [y_i]_i[x_{v_1}] = Ax,$$

where $v_1 = (u_1, u_2)$, $v_2 = (v_1, u_2)$, and $v_3 = (v_1, v_2)$. Hence, x' satisfies the constraints (3.6) if x satisfies the constraints (3.6).

In addition, since $\sum_{e \in E} x'_e I_i(e, f) = \sum_{e \in E} x_e I_i(e, f) + [y_i]_i I_i(e_1, f) + [y_i]_i I_i(e_2, f) - [y_i]_i I_i(e_3, f)$, and:

- $I_1(e_1, (u_1, v_1)) = 1$ and $I_i(e_1, f) = 0$ for other $f \in G_i$,
- $I_2(e_2, (u_2, v_2)) = 1$ and $I_i(e_2, f) = 0$ for other $f \in G_i$,
- $I_1(e_3, (u_1, v_1)) = 1$, $I_2(e_3, (u_2, v_2)) = 1$, and $I_i(e_3, f) = 0$ for other $f \in G_i$,

we have:

- $[y_i]_i I_1(e_1, (u_1, v_1)) + [y_i]_i I_1(e_2, (u_1, v_1)) - [y_i]_i I_1(e_3, (u_1, v_1)) = [y_i]_i + 0 - [y_i]_i = 0$,
- $[y_i]_i I_2(e_1, (u_2, v_2)) + [y_i]_i I_2(e_2, (u_2, v_2)) - [y_i]_i I_2(e_3, (u_2, v_2)) = 0 + [y_i]_i - [y_i]_i = 0$,
- $[y_i]_i I_i(e_1, f) + [y_i]_i I_i(e_2, f) - [y_i]_i I_i(e_3, f) = 0 + 0 - 0 = 0$ for any other $i = 1, 2$ and $f \in E_i$.

Therefore, $\sum_{e \in E} x'_e I_i(e, f) = \sum_{e \in E} x_e I_i(e, f)$, meaning that x' satisfies the constraints (3.7) if x satisfies the constraints (3.7). □

With Lemma 6, we prove that any feasible solution of x in (3.11) is a feasible solution of (3.5)-(3.8). First, it is easy to check that x^{init} satisfies the constraints (3.6)-(3.7). For each feasible solution of x in (3.11), since $x = x^{init} + [\partial]y = x^{init} + \sum_i [\partial][y_i]$, by iteratively using Lemma 6, we get that x satisfies the constraints (3.6)-(3.7). Since $x_e \geq 0$ for all $e \in E$ is a constraint existing in both linear relaxations, x is a feasible solution of (3.5)-(3.8).

We now show that any feasible solution of (3.5)-(3.8) is a feasible solution of (3.11). Let x be a feasible solution of (3.5)-(3.8). We show that x is also a feasible solution of (3.11) by proving that x can be converted to x^{init} in (3.11) via the boundary operator ∂. First, if there is a diagonal edge $e = [(u_1, u_2), (v_1, v_2)]$ in E such that $x_e > 0$, then it can be replaced by the horizontal edge $e_h = [(u_1, u_2), (u_1, v_2)]$ followed by the vertical edge $e_v = [(u_1, v_2), (v_1, v_2)]$ by using one boundary operation on the 2-simplex $[(u_1, u_2), (u_1, v_2), (v_1, v_2)]$. Hence, x can be converted to a new vector x', such that $x'_e = 0$, $x'_{e_h} = x_{e_h} + x_e$, $x'_{e_v} = x_{e_v} + x_e$, and all the other entries in x' are the same as those in x. It is easy to check that x' is also a feasible solution of (3.5)-(3.8). Therefore, without loss of generality, we assume x to be a vector such that all the entries corresponding to diagonal edges in $\mathcal{A}(G_1, G_2)$ are zero.

We then prove that any x can be converted to x^{init} in (3.11) via the boundary operator. Let the source and the sink node of x in $\mathcal{A}(G_1, G_2)$ be (s_1^1, s_1^2) and (s_2^1, s_2^2), where s_1^i is the source node of G_i and s_2^i is the sink node of G_i. When the Eulerian trail is closed (meaning that it is an Eulerian tour) in G_i, we let $s_1^i = s_2^i$ be an arbitrary vertex in V_i. x^{init} can be seen as a trail (tour) in $\mathcal{A}(G_1, G_2)$ that starts from (s_1^1, s_1^2), walks along an Eulerian trail of G_2 via all the horizontal edges P_h,

$$P_h = \{[(s_1^1, s_1^2), (s_1^1, v_1^2)], [(s_1^1, v_1^2), (s_1^1, v_2^2)], \ldots, [(s_1^1, v_{i-1}^2), (s_1^1, v_i^2)], [(s_1^1, v_i^2), (s_1^1, s_2^2)]\},$$

and then walks along an Eulerian trail of G_1 via all the vertical edges P_v,

$$P_v = \{[(s_1^1, s_2^2), (v_1^1, s_2^2)], [(v_1^1, s_2^2), (v_2^1, s_2^2)], \ldots, [(v_{j-1}^1, s_2^2), (v_j^1, s_2^2)], [(v_j^1, s_2^2), (s_2^1, s_2^2)]\},$$

until the sink node (s_2^1, s_2^2). Here $\{s_1^2, v_1^2, v_2^2, \ldots, v_{i-1}^2, v_i^2, s_2^2\}$ is an Eulerian trail of G_2 and $\{s_1^1, v_1^1, v_2^1, \ldots, v_{i-1}^1, v_i^1, s_2^1\}$ is an Eulerian trail of G_1. We use $P_0 = \{P_h, P_v\}$ to denote the trail from (s_1^1, s_1^2) to (s_2^1, s_2^2) that is the concatenation of P_h and P_v. It is easy to see that each edge in P_0 is unique.

As shown in Qiu and Kingsford [127], x is a flow of $\mathcal{A}(G_1, G_2)$ with the additional constraints (3.7). Therefore, according to the flow decomposition theorem [3, p. 80], x can be decomposed into a finite set of weighted paths in $\mathcal{A}(G_1, G_2)$ from (s_1^1, s_1^2) to (s_2^1, s_2^2), which is denoted as $\{(p_1, w_1^p), \ldots, (p_n, w_n^p)\}$, and a finite set of weight cycles in $\mathcal{A}(G_1, G_2)$, which is denoted as $\{(c_1, w_1^c), \ldots, (c_m, w_m^c)\}$. Each path or cycle only contains horizontal and vertical edges.

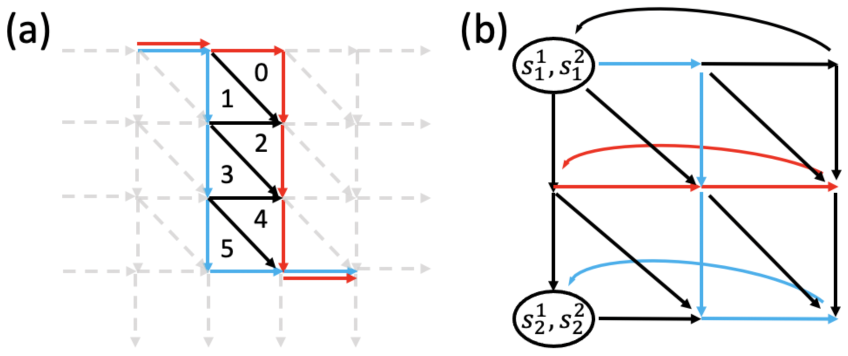


Figure 3.4: (a) An example of converting three vertical edges followed by one horizontal edge (blue line) to one horizontal edge followed by three vertical edges (red line). It can be done by doing boundary operations on 2-simplices labeled from 0 to 5. (b) An example of a cycle path (red line) and its auxiliary trail (blue line).

For path i, we use a vector $x^{p,i}$ to represent (p_i, w_i^p),

$$x_e^{p,i} = \begin{cases} w_i^p & \text{if } e \in p_i \\ 0 & \text{otherwise,} \end{cases} \tag{3.13}$$

By using the boundary operator, each path p_i can actually be converted to a new trail p_i' such that each edge in p_i' is also an edge in P_0. To prove this, we consider the following two cases:

- If p_i walks along all the horizontal edges followed by all the vertical edges, then every edge in p_i is an edge in P_0. To see that, let e be an horizontal edge in p_i, since p_i starts from (s_1^1, s_1^2), e has the form $[(s_1^1, v), (s_1^1, v')]$ where $[v, v'] \in E_2$. Since P_h corresponds to the Eulerian trail of G_2, for each $[v, v'] \in E_2$, we have $[(s_1^1, v), (s_1^1, v')] \in P_h$. Therefore $e \in P_0$. We can use the same way to prove $e \in P_0$ when e is a vertical edge. Note that in this case, the number of horizontal edges or vertical edges can be zero.
- If not, then we let $p_i = \{e_1^i, e_2^i, \dots, e_m^i\}$, and let e_t^i be the vertical edge with the smallest index t. There exists an integer k ($k \geq 1$) such that $\{e_t^i, e_{t+1}^i, \dots, e_{t+k-1}^i\}$ are all vertical edges and e_{t+k}^i is an horizontal edge. We denote each vertical edge $e_{t+w}^i \in \{e_t^i, e_{t+1}^i, \dots, e_{t+k-1}^i\}$ as $[(v_w, v_t), (v_{w+1}, v_t)]$ and denote e_{t+k}^i as $[(v_k, v_t), (v_k, v_{t+1})]$. It is easy to see that when $w = 0$, $v_w = s_1^1$. By using the boundary operator, this subpath $\{e_t^i, e_{t+1}^i, \dots, e_{t+k-1}^i, e_{t+k}^i\}$ can be replaced by another subpath with one horizontal edge $[(s_1^1, v_t), (s_1^1, v_{t+1})]$ followed by k vertical edges:

  $$\{[(s_1^1, v_{t+1}), (v_1, v_{t+1})], [(v_1, v_{t+1}), (v_2, v_{t+1})], \dots, [(v_{k-1}, v_{t+1}), (v_k, v_{t+1})].$$

  Now we have a new path, denoted as p_i^1, in which the smallest index of the vertical edges becomes $t + 1$. Figure 3.4(a) shows an example, in which the blue line represents the subpath of p_i and the red line represents the new subpath in p_i^1.

  To create a new vector that represents p_i^1, we first create a zero vector $y^{p,i,1} \in \mathbb{R}^{|T(G_1,G_2)|}$, and from $w = 0$ to $w = k - 1$, we iteratively update $y^{p,i,1}$ via the following equations:

$$y^{p,i,1}_\sigma = \begin{cases} y^{p,i,1}_\sigma - w^p_i & \text{if } \sigma = [(v_w, v_t), (v_{w+1}, v_t), (v_{w+1}, v_{t+1})] \\ y^{p,i,1}_\sigma + w^p_i & \text{if } \sigma = [(v_w, v_t), (v_w, v_{t+1}), (v_{w+1}, v_{t+1})] \\ 0 & \text{otherwise.} \end{cases} \tag{3.14}$$

The vector $x^{p,i,1} = x^{p,i} + [\partial]y^{p,i,1}$ is the one that represents p^1_i.

Since the length of p_i is finite, by doing such a transformation a finite number of times, we can convert p_i to a new path p'_i such that p'_i walks along all the horizontal edges first followed by all the vertical edges, therefore each edge in p'_i is also an edge in P_0. We use the vector $\hat{x}^{p,i}$ to represent p'_i, $\hat{x}^{p,i} = x^{p,i} + [\partial] \sum_{j=1}^{q} y^{p,i,j}$ where q is the number of transformations. Apperantly, $\hat{x}^{p,i}_e = 0$ when $e \notin P_0$. Let $y^{p,i} = \sum_{j=1}^{q} y^{p,i,j}$, we have $\hat{x}^{p,i} = x^{p,i} + [\partial]y^{p,i}$.

For cycle i, we also use a vector $x^{c,i}$ to represent (c_i, w^c_i),

$$x^{c,i}_e = \begin{cases} w^c_i & \text{if } e \in c_i \\ 0 & \text{otherwise,} \end{cases} \tag{3.15}$$

Let (v, v') be an arbitrary chosen node in c_i, we construct a trail p^i_{aux} that passes (v, v') as follows:

- From (s^1_1, s^2_1), walk along P_h until the node (s^1_1, v'). It corresponds to a part of an Eulerian trail of G_2.
- From (s^1_1, v'), walk along an Eulerian trail of G_1 to (s^1_2, v'). It must passes the node (v, v').
- From (s^1_2, v'), walk along the remaining part of the Eulerian trail of G_2 to the node (s^1_2, s^2_2).

Figure 3.4(b) shows an example, in which the blue line represents p^i_{aux} and the red line represents c_i.

We use $x^{aux,i}$ to denote the vector representing p^i_{aux}. The combination of c_i and p^i_{aux}, represented by the vector $x^{c,i} + x^{aux,i}$ creates a new trail (may have repeated edges) from (s^1_1, s^2_1) to (s^1_2, s^2_2): (1) walk along p^i_{aux} from (s^1_1, s^2_1) to (v, v'), (2) walk along c_i from (v, v') to itself, and (3) walk along the remaining part of p^i_{aux} from (v, v') to (s^1_2, s^2_2). By using the same way as we described above, each $c_i + p^i_{aux}$ or p^i_{aux} can be converted to a new trail in which each edge is also an edge in P_0. We use $\hat{x}^{c,i}$ or $\hat{x}^{aux,i}$ to represent the new trail accordingly, therefore, we have $\hat{x}^{c,i} = x^{c,i} + x^{aux,i} + [\partial]y^{c,i}$ and $\hat{x}^{aux,i} = x^{aux,i} + [\partial]y^{aux,i}$. Likewise, $\hat{x}^{c,i}_e = \hat{x}^{aux,i}_e = 0$ when $e \notin P_0$.

We define a new vector $\hat{x}$ such that:

$$\begin{aligned}
\hat{x} &= \sum_{i=1}^{n} \hat{x}^{p,i} + \sum_{j=1}^{m} \hat{x}^{c,j} - \hat{x}^{aux,j} \\
&= \sum_{i=1}^{n} x^{p,i} + [\partial] y^{p,i} + \sum_{j=1}^{m} x^{c,j} + x^{aux,j} + [\partial] y^{c,j} - \left(\sum_{j=1}^{m} x^{aux,j} + [\partial] y^{aux,j} \right) \\
&= \sum_{i=1}^{n} x^{p,i} + \sum_{j=1}^{m} x^{c,j} + [\partial] \left(\sum_{i=1}^{n} y^{p,i} + \sum_{j=1}^{m} y^{c,j} - \sum_{j=1}^{m} y^{aux,j} \right) \\
&= x + [\partial] \left(\sum_{i=1}^{n} y^{p,i} + \sum_{j=1}^{m} y^{c,j} - \sum_{j=1}^{m} y^{aux,j} \right).
\end{aligned}$$

Therefore, $\hat{x}$ is a vector converted from x via boundary operations. $\hat{x}$ is equal to x^{init} because:

1. $\hat{x}_e = 0$ when $e \notin P_0$ since $\hat{x}^{p,i}_e = \hat{x}^{c,i}_e = \hat{x}^{aux,i}_e = 0$ when $e \notin P_0$ for each i.
2. As we have proved above, the boundary operator preserves the constraints (3.6)-(3.7). Therefore, $\hat{x}$ satisfies the constraints (3.6)-(3.7) since x is a feasible solution of (3.5)-(3.8). Combined with the first point, we have that $\hat{x}_e = 1$ if $e \in P_0$ and $\hat{x}_e = 0$ otherwise, meaning that $\hat{x} = x^{init}$.

Hence, for each feasible solution x of (3.5)-(3.8), we have:

$$\begin{aligned}
x &= x^{init} - [\partial] \left(\sum_{i=1}^{n} y^{p,i} + \sum_{j=1}^{m} y^{c,j} - \sum_{j=1}^{m} y^{aux,j} \right) \\
&= x^{init} + [\partial] \left(- \sum_{i=1}^{n} y^{p,i} - \sum_{j=1}^{m} y^{c,j} + \sum_{j=1}^{m} y^{aux,j} \right),
\end{aligned}$$

meaning that x is also a feasible solution of (3.11).

We proved that the feasibility region of x in (3.11) is the same as the feasibility region of x in (3.5)-(3.8), and since the objective functions of these two linear relaxations are the same, the optimal solutions of them are equal.

By employing the same approach and taking into account that if all edge weights in a flow network are non-negative integers, the flow decomposition theorem guarantees that the network can be decomposed into a finite set of weighted paths and cycles, each with positive integer weight, we can prove that the ILP in (3.5)-(3.8) and the ILP in (3.11)-(3.12) are also equivalent.

Based on the proof, we can conclude that the way to index the vertices or edges in the alignment graph, or the 2-simplices in $T(G_1, G_2)$, will not affect the equivalence result. Additionally, different choices of orientations for the 2-simplices in $T(G_1, G_2)$ will also not impact the equivalence result. This is because for any two sets $T(G_1, G_2)$ and $T'(G_1, G_2)$ containing the same 2-simplices with the same indices but different orientations, if (x, y) is a feasible solution of the ILP in (3.11)-(3.12) (or its relaxation) that corresponds to $T(G_1, G_2)$, then (x, y') is a feasible solution of the ILP in (3.11)-(3.12) (or its relaxation) that corresponds to $T'(G_1, G_2)$, where $y_i = y'_i$ when $\sigma_i \in T(G_1, G_2)$ has the same orientation as $\sigma'_i \in T'(G_1, G_2)$, and $y_i = -y'_i$ when

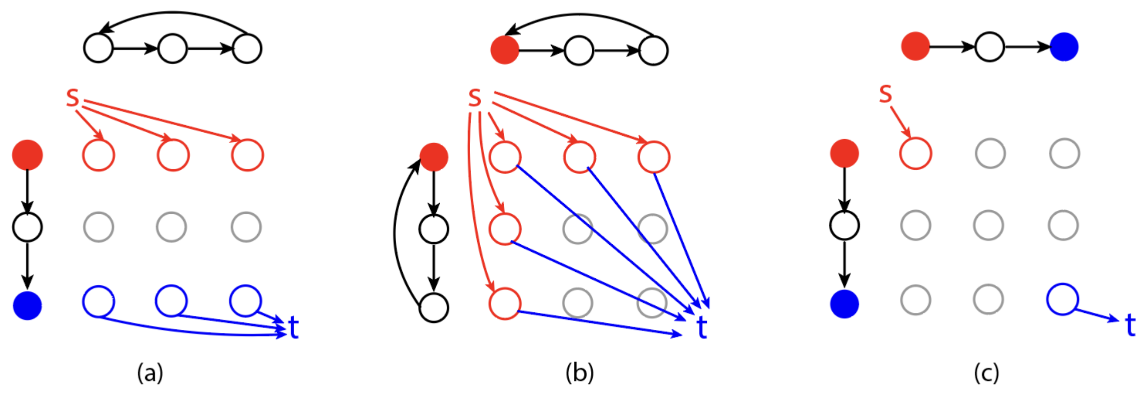


Figure 3.5: Modified alignment graphs based on input types. (a) G_1 has open Eulerian trails while G_2 has closed Eulerian trails. (b) Both G_1 and G_2 have closed Eulerian trails. (c) Both G_1 and G_2 have open Eulerian trails. Solid red and blue nodes are the source and sink nodes of the graphs with open Eulerian trails. "s" and "t" are the added source and sink nodes. Colored edges are added alignment edges directing from and to source and sink nodes, respectively.

$\sigma_i \in T(G_1, G_2)$ has the opposite orientation to $\sigma'_i \in T'(G_1, G_2)$. Therefore, it is acceptable to specify a particular orientation for each 2-simplex when defining $T(G_1, G_2)$.

3.4. New ILP solutions to GTED

To ensure that our new ILP formulations are applicable to input graphs regardless of whether they contain an open or closed Eulerian trail, we add a source node s and a sink node t to the alignment graph. Figure 3.5 illustrates three possible cases of input graphs.

1. If only one of the input graphs has closed Eulerian trails, wlog, let G_1 be the input graph with open Eulerian trails. Let a_1 and b_1 be the start and end of the Eulerian trail that have odd degrees. Add edges $[s, (a_1, v_2)]$ and $[(b_1, v_2), t]$ to E for all nodes $v_2 \in V_2$ (Figure 3.5(a)).
2. If both input graphs have closed Eulerian trails, let a_1 and a_2 be two arbitrary nodes in G_1 and G_2, respectively. Add edges $[s, (a_1, v_2)]$, $[s, (v_1, a_2)]$, $[(a_1, v_2), t]$ and $[(v_1, a_2), t]$ for all nodes $v_1 \in V_1$ and $v_2 \in V_2$ to E (Figure 3.5(b)).
3. If both input graphs have open Eulerian trails, add edges $[s, (a_1, a_2)]$ and $[t, (b_1, b_2)]$, where a_i and b_i are start and end nodes of the Eulerian trails in G_i, respectively (Figure 3.5(c)).

According to Lemma 5, we can solve GTED(G_1, G_2) by finding a single trail in $\mathcal{A}(G_1, G_2)$ that satisfies the projection requirements. This is equivalent to finding a s-t trail in $\mathcal{A}(G_1, G_2)$ that satisfies constraints:

$$\sum_{(u,v)\in E} x_{uv} I_i((u,v), f) = 1 \quad \text{for all } (u,v) \in E, f \in G_i,\ u \neq s,\ v \neq t, \tag{3.16}$$

where $I_i(e, f) = 1$ if the alignment edge e projects to f in G_i. An optimal solution to GTED in the alignment graph must start and end with the source and sink node because they are connected to all possible starts and ends of Eulerian trails in the input graphs.

Since a trail in $\mathcal{A}(G_1, G_2)$ is a flow network, we use the following flow constraints to enforce the equality between the number of in- and out-edges for each node in the alignment graph except the source and sink nodes.

$$\sum_{(s,u)\in E} x_{su} = 1 \tag{3.17}$$

$$\sum_{(v,t)\in E} x_{vt} = 1 \tag{3.18}$$

$$\sum_{(u,v)\in E} x_{uv} = \sum_{(v,w)\in E} x_{vw} \quad \text{for all } v \in V \tag{3.19}$$

Constraints (3.16) and (3.19) are equivalent to constraints (3.7) and (3.6), respectively. Therefore, we rewrite the ILP in (3.5)-(3.8) in terms of the modified alignment graph.

$$\begin{aligned} \underset{x\in\mathbb{N}^{|E|}}{\text{minimize}} \quad & \sum_{e\in E} x_e \delta(e) \\ \text{subject to} \quad & \text{constraints (3.16)–(3.19).} \end{aligned} \tag{lower bound ILP}$$

As we show in Section 3.3.2, constraints (3.16)-(3.19) do not guarantee that the ILP solution is one trail in $\mathcal{A}(G_1, G_2)$, thus allowing several disjoint covering trails to be selected in the solution and fails to model GTED correctly. We show in Section 3.5 that the solutions to this ILP is a lower bound to GTED.

According to Lemma 1 in Dias et al. [38], a subgraph of a directed graph G with source node s and sink node t is a s-t trail if and only if it is a flow network and every strongly connected component (SCC) of the subgraph has at least one edge outgoing from it. Thus, in order to formulate an ILP for the GTED problem, it is necessary to devise constraints that prevent disjoint SCCs from being selected in the alignment graph. In the following, we describe two approaches for achieving this.

3.4.1 Enforcing one trail in the alignment graph via constraint generation

Section 3.2 of Dias et al. [38] proposes a method to design linear constraints for eliminating disjoint SCCs, which can be directly adapted to our problem. Let $\mathcal{C}$ be the collection of all strongly connected subgraphs of the alignment graph $\mathcal{A}(G_1, G_2)$. We use the following constraint to enforce that the selected edges form one s-t trail in the alignment graph:

$$\text{If} \sum_{(u,v)\in E(C)} x_{uv} = |E(C)|, \text{then} \sum_{(u,v)\in \varepsilon^+(C)} x_{uv} \geq 1 \quad \text{for all } C \in \mathcal{C}, \tag{3.20}$$

where $E(C)$ is the set of edges in the strongly connected subgraph C and $\varepsilon^+(C)$ is the set of edges (u, v) such that u belongs to C and v does not belong to C. $\sum_{(u,v)\in E(C)} x_{uv} = |E(C)|$

indicates that C is in the subgraph of $\mathcal{A}(G_1, G_2)$ constructed by all edges (u, v) with positive x_{uv}, and $\sum_{(u,v)\in\varepsilon^+(C)} x_{uv} \geq 1$ guarantees that there exists an out-going edge of C that is in the subgraph.

We use the same technique as Dias et al. [38] to linearize the "if-then" condition in (3.20) by introducing a new variable β for each strongly connected component:

$$\sum_{(u,v)\in E(C)} x_{uv} \geq |E(C)|\beta_C \quad \text{for all } C \in \mathcal{C} \tag{3.21}$$

$$\sum_{(u,v)\in E(C)} x_{uv} - |E(C)| + 1 - |E(C)|\beta_C \leq 0 \quad \text{for all } C \in \mathcal{C} \tag{3.22}$$

$$\sum_{(u,v)\in\varepsilon^+(C)} x_{uv} \geq \beta_C \quad \text{for all } C \in \mathcal{C} \tag{3.23}$$

$$\beta_C \in \{0, 1\} \quad \text{for all } C \in \mathcal{C} \tag{3.24}$$

To summarize, given any pair of unidirectional, edge-labeled Eulerian graphs G_1 and G_2 and their alignment graph $\mathcal{A}(G_1, G_2) = (V, E, \delta)$, $\text{GTED}(G_1, G_2)$ is equal to the optimal solution of the following ILP formulation:

$$\begin{aligned} \underset{x\in\{0,1\}^{|E|}}{\text{minimize}} \quad & \sum_{e\in E} x_e \delta(e) \\ \text{subject to} \quad & \text{constraints (3.16)–(3.19) and} \\ & \text{constraints (3.21)–(3.24).} \end{aligned} \tag{exponential ILP}$$

This ILP has an exponential number of constraints as there is a set of constraints for every strongly connected subgraph in the alignment graph. To solve this ILP more efficiently, we can use the procedure similar to the iterative constraint generation procedure in Dias et al. [38]. Initially, solve the ILP with only constraints (3.16)-(3.19). Create a subgraph, G', induced by edges with positive x_{uv}. For each disjoint SCC in G' that does not contain the sink node, add constraints (3.21)-(3.24) for edges in the SCC and solve the new ILP. Iterate until no disjoint SCCs are found in the solution (Algorithm 1).

3.4.2 A compact ILP for GTED with polynomial number of constraints

In the worst case, the number of iterations to solve (exponential ILP) via constraint generation is exponential. As an alternative, we introduce a compact ILP with only a polynomial number of constraints. The intuition behind this ILP is that we can impose a partially increasing ordering on all the edges so that the selected edges forms a s-t trail in the alignment graph. This idea is similar to the Miller-Tucker-Zemlin ILP formulation of the TRAVELLING SALESMAN problem (TSP) [100].

We add variables d_{uv} that are constrained to provide a partial ordering of the edges in the s-t trail and set the variables d_{uv} to zero for edges that are not selected in the s-t trail. Intuitively, there must exist an ordering of edges in a s-t trail such that for each pair of consecutive edges (u, v) and (v, w), the difference in their order variable d_{uv} and d_{vw} is 1. Therefore, for each

Algorithm 1: Iterative constraints generation algorithm to solve (exponential ILP)

```
Input Two unidirectional, edge-labeled Eulerian graphs and their alignment graph
$\mathcal{C} \leftarrow \emptyset$
while true do
    Solve the ILP (exponential ILP) with $\mathcal{C}$
    if the ILP variables $x_{uv}$ induce a strongly connected component $C$ not satisfying (3.20) then
        $\mathcal{C} = \mathcal{C} \cup \{C\}$
    else
        return the optimal ILP value and the corresponding optimal solution $x$
    end if
end while
```

node v that is not the source or the sink, if we sum up the order variables for the incoming edges and outgoing edges respectively, the difference between the two sums is equal to the number of selected incoming/outgoing edges. Lastly, the order variable for the edge starting at source is 1, and the order variable for the edge ending at sink is the number of selected edges. This gives the ordering constraints as follows:

$$\text{If } x_{uv} = 0, \text{ then } d_{uv} = 0 \quad \text{for all } (u,v) \in E \tag{3.25}$$

$$\sum_{(v,w)\in E} d_{vw} - \sum_{(u,v)\in E} d_{uv} = \sum_{(v,w)\in E} x_{vw} \quad \text{for all } v \in V \setminus \{s,t\} \tag{3.26}$$

$$\sum_{(s,u)\in E} d_{su} = 1 \tag{3.27}$$

$$\sum_{(v,t)\in E} d_{vt} = \sum_{(u,v)\in E} x_{uv} \tag{3.28}$$

We enforce that all variables $x_e \in \{0,1\}$ and $d_e \in \mathbb{N}$ for all $e \in E$.

The "if-then" statement in Equation (3.25) can be linearized by introducing an additional binary variable y_{uv} for each edge [19, 38]:

$$-x_{uv} - |E| y_{uv} \leq -1 \tag{3.29}$$

$$d_{uv} - |E|(1 - y_{uv}) \leq 0 \tag{3.30}$$

$$y_{uv} \in \{0,1\}. \tag{3.31}$$

Here, y_{uv} is an indicator of whether $x_{uv} \geq 0$. The coefficient $|E|$ is the number of edges in the alignment graph and also an upper bound on the ordering variables. When $y_{uv} = 1$, $d_{uv} \leq 0$, and y_{uv} does not impose constraints on x_{uv}. When $y_{uv} = 0$, $x_{uv} \geq 1$, and y_{uv} does not impose constraints on d_{uv}.

We prove the correctness of the compact ILP by showing that the ordering constraints (3.26) correctly forbids disjoint strongly connected components from being selected in the ILP solution.

Lemma 7. *Let x_e and d_e be ILP variables. Let G' be a subgraph of $\mathcal{A}(G_1, G_2)$ that is induced by edges with $x_e = 1$. If x_e and d_e satisfy constraints* (3.16)-(3.28) *for all $e \in E$, G' is connected with one trail from s to t that traverses each edge in G' exactly once.*

Proof. We prove the lemma in 2 parts: (1) all nodes except s and t in G' have an equal number of in- and out-edges, (2) G' contains only one connected component.

The first statement holds because the edges of G' form a flow from s to t, and is enforced by constraints (3.19).

We then show that G' does not contain isolated subgraphs that are not reachable from s or t. Due to constraints (3.19), the only possible scenario is that the isolated subgraph is strongly connected. Suppose for contradiction that there is a strongly connected component, C, in G' that is not reachable from s or t.

The sum of the left hand side of constraints (3.26) over all vertices in C is

$$\sum_{v \in C} \Big(\sum_{(u,v) \in C} d_{uv} - \sum_{(v,w) \in C} d_{vw} \Big) = \sum_{v \in C} \sum_{(u,v) \in C} d_{uv} - \sum_{v \in C} \sum_{(v,w) \in C} d_{vw} \tag{3.32}$$

$$= \sum_{(u,v) \in E(C)} d_{uv} - \sum_{(v,w) \in E(C)} d_{vw} = 0. \tag{3.33}$$

However, the right-hand side of the same constraints is always positive. Hence we have a contradiction. Therefore, G' has only one connected component. □

Due to Lemma 5 and Lemma 7, given input graphs G_1 and G_2 and the alignment graph $\mathcal{A}(G_1, G_2)$, GTED(G_1, G_2) is equal to the optimal objective of

$$\begin{aligned} \underset{x \in \{0,1\}^{|E|}}{\text{minimize}} \quad & \sum_{e \in E} x_e \delta(e) \\ \text{subject to} \quad & \text{constraints (3.16)–(3.19),} \\ & \text{constraints (3.26)–(3.28)} \\ & \text{and constraints (3.29)–(3.31).} \end{aligned} \tag{compact ILP}$$

3.5. A lower bound on GTED

While the (lower bound ILP) and the ILP in (3.11)-(3.12) do not solve GTED, the optimal solution to these ILPs is a lower bound of GTED. It also solves an interesting variant of GTED (Section 3.5.1), which is a local similarity measure between two genome graphs. We call this variant as Closed-trail Cover Traversal Edit Distance (CCTED).

3.5.1 Closed-trail Cover Traversal Edit Distance

We introduce a variant of GTED, the Closed-trail Cover Traversal Edit Distance. We show in this section that the lower bound ILP solves CCTED when the two input graphs have closed Eulerian trails.

We first introduce the min-cost item matching problem between two multi-sets. Let two multi-sets of items be S_1 and S_2, and, wlog, let $|S_1| \leq |S_2|$. Let $c : (S_1 \cup \{\epsilon\}) \times S_2 \to \mathbb{N}$ be the cost of matching either an empty item ϵ or an item in S_1 with an item in S_2. Given S_1, S_2 and the cost function c, min-cost matching problem finds a matching, $\mathcal{M}_c(S_1, S_2)$, such that each item in $S_1 \cup \{\epsilon\}^{|S_2|-|S_1|}$ is matched with exactly one item in S_2 and the total cost of the matching, $\sum_{(s_1,s_2)\in\mathcal{M}_c(S_1,S_2)} c(s_1, s_2)$, is minimized.

The min-cost item matching problem is similar to the Earth Mover's Distance defined in [122], except that only integral units of items can be matched and the cost of matching an empty item with another item is not constant. Similar to the Earth Mover's Distance, the min-cost item matching problem can be computed using the ILP formulation of the min-cost max-flow problem [127, 135]. When the cost is the edit distance, the cost to match ϵ with a string is equal to the length of the string.

Define traversal edit distance, $edit_t(t_1, t_2)$ as the edit distance between the strings constructed from a pair of trails t_1 and t_2. In other words, $edit_t(t_1, t_2) = edit(\text{str}(t_1), \text{str}(t_2))$.

Problem 5 (Closed-Trail Cover Traversal Edit Distance (CCTED))**.** *Given two unidirectional, edge-labeled Eulerian graphs G_1 and G_2 with closed Eulerian trails, compute*

$$\text{CCTED}(G_1, G_2) \triangleq \min_{\substack{C_1\in CC(G_1),\\ C_2\in CC(G_2)}} \sum_{(t_1,t_2)\in\mathcal{M}_{edit_t}(C_1,C_2)} edit(str(t_1), str(t_2)), \tag{3.34}$$

Here, $CC(G)$ denotes the collection of all possible sets of edge-disjoint, closed trails in G, such that every edge in G belongs to exactly one of these trails. Each element of $CC(G)$ can be interpreted as a cover of G using such trails. $\mathcal{M}_{edit_t}(C_1, C_2)$ is a min-cost matching between two covers using the traversal edit distance as the cost.

CCTED is likely a more suitable metric comparisons between genomes that undergo large-scale rearrangements. This analogy is to the relationship between the synteny block comparison [124] and the string edit distance computation, where the former is more often used in interspecies comparisons and in detecting segmental duplications [18, 158, 159], and the latter is more often seen in intraspecies comparisons.

Following similar ideas as Lemma 5, we can compute CCTED by finding a set of closed trails in the alignment graph such that the total cost of alignment edges are minimized, and the projection of all edges in the collection of selected trails is equal to the multi-set of input graph edges.

Lemma 8. *For any two edge-labeled Eulerian graphs G_1 and G_2,*

$$\text{CCTED}(G_1, G_2) = \underset{C}{\textit{minimize}} \quad \sum_{c\in C} \delta(c) \tag{3.35}$$

$$\textit{subject to} \quad C \textit{ is a set of closed trails in } \mathcal{A}(G_1, G_2),$$

$$\bigcup_{e\in C} \Pi_i(e) = E_i \quad \textit{for } i = 1, 2, \tag{3.36}$$

where C is a collection of trails and $\delta(c)$ is the total cost of edges in trail c.

Proof. Given any pair of covers $C_1 \in \text{CC}(G_1)$ and $C_2 \in \text{CC}(G_2)$ and their min-cost matching based on the edit distance $\mathcal{M}_{\text{edit}_t}(C_1, C_2)$, we can project each pair of matched closed trailed to a closed trail in the alignment graph. For a matching between a trail and the empty item ϵ, we can project it to a closed trail in the alignment graph with all vertical edges if the trail is from G_1 or horizontal edges if the trail is from G_2. The total cost of the projected edges must be greater than or equal to the objective (3.35). On the other hand, every collection of trails C that satisfy constraints (3.36) can be projected to a cover in each of the input graphs, and $\sum_{c\in C}\delta(c) \geq CCTED(G_1, G_2)$. Hence equality holds. □

3.5.2 Correctness of the ILP formulation for CCTED

We show that the ILP in (3.5)-(3.8) proposed by Boroojeny et al. [16] solves CCTED.

Theorem 5. *Given two input graphs G_1 and G_2, the optimal objective value of the ILP in* (3.5)-(3.8) *based on $\mathcal{A}(G_1, G_2)$ is equal to* $\mathrm{CCTED}(G_1, G_2)$.

Proof. As shown in the proof of Lemma 8, any pair of edge-disjoint, closed-trail covers in the input graph can be projected to a set of closed trails in $\mathcal{A}(G_1, G_2)$, which satisfied constraints (3.6)-(3.8). The objective of this feasible solution, which is the total cost of the projected closed trails, equals CCTED. Therefore, $\mathrm{CCTED}(G_1, G_2)$ is greater than or equal to the objective of the ILP in (3.5)-(3.8).

Conversely, we can transform any feasible solutions of the ILP in (3.5)-(3.8) to a pair of covers of G_1 and G_2. We can do this by transforming one closed trail at a time from the subgraph of the alignment graph, $\mathcal{A}'$ induced by edges with ILP variable $x_{uv} = 1$. Let c be a closed trail in $\mathcal{A}'$. Let $c_1 = \Pi_1(c)$ and $c_2 = \Pi_2(c)$ be two closed trails in G_1 and G_2 that are projected from c. We can construct an alignment between str(c_1) and str(c_2) from c by adding match or insertion/deletion columns for each match or insertion/deletion edges in c accordingly. The cost of the alignment is equal to the total cost of edges in c by the construction of the alignment graph. We can then remove edges in c from the alignment graph and edges in c_1 and c_2 from the input graphs, respectively. The remaining edges in $\mathcal{A}'$ and G_1 and G_2 still satisfy the constraints (3.6)-(3.8). Repeat this process and we get a total cost of $\sum_{e\in E} x_e\delta(e)$ that aligns pairs of closed trails that form covers of G_1 and G_2. This total cost is greater than or equal to $\mathrm{CCTED}(G_1, G_2)$. □

3.5.3 The relationship between CCTED and GTED

Let the variables in an optimal solution to (lower bound ILP) be x^{opt} and the optimal objective value be c^{opt}. Since the constraints for (lower bound ILP) is a subset of (exponential ILP), and two ILPs have the same objective function, $c^{opt} \leq \mathrm{GTED}(G_1, G_2)$ for any pair of graphs.

Moreover, when the solution to (lower bound ILP) forms only one connected component, the optimal value of (lower bound ILP) is equal to GTED.

Theorem 6. *Let $\mathcal{A}'(G_1, G_2)$ be the subgraph of $\mathcal{A}(G_1, G_2)$ induced by edges $(u, v) \in E$ with $x^{opt}_{uv} = 1$ in the optimal solution to* (lower bound ILP). *There exists $\mathcal{A}'(G_1, G_2)$ that has exactly one connected component if and only if $c^{opt} = \mathrm{GTED}(G_1, G_2)$.*

Proof. We first show that if $c^{opt} = \mathrm{GTED}(G_1, G_2)$, then there exists $\mathcal{A}'(G_1, G_2)$ that has one connected component. A feasible solution to (exponential ILP) is always a feasible solution to (lower bound ILP), and since $c^{opt} = \mathrm{GTED}(G_1, G_2)$, an optimal solution to (exponential ILP) is also an optimal solution to (lower bound ILP), which can induce a subgraph in the alignment graph that only contains one connected component.

Conversely, if x^{opt} induces a subgraph in the alignment graph with only one connected component, it satisfies constraints (3.21)-(3.24) and therefore is feasible to the ILP for GTED (exponential ILP). Since $c^{opt} \leq \text{GTED}(G_1, G_2)$, this solution must also be optimal for $\text{GTED}(G_1, G_2)$. □

In practice, we may estimate GTED approximately by the solution to (lower bound ILP). As we show in Section 3.7, the time needed to solve (lower bound ILP) is much less than the time needed to solve GTED. However, in adversarial cases, c^{opt} could be zero but GTED could be arbitrarily large. We can determine if the c^{opt} is a lower bound on GTED or exactly equal to GTED by checking if the subgraph induced by the solution to (lower bound ILP) has multiple connected components.

3.6. Characterizations of the ILP in (3.11)-(3.12)

Boroojeny et al. [16] propose the ILP in (3.11)-(3.12), and we show in Section **??** that the x variables in this ILP have the same feasible region as the x variables in lower bound ILP. However, Boroojeny et al. [16] argue that the linear programming relaxation of the ILP in (3.11)-(3.12) always yields integer optimal solutions, and therefore the ILP in (3.11)-(3.12) can be solved in polynomial time. We provide a counterexample where the ILP in (3.11)-(3.12) yields fractional optimal solutions with fractional variable values. Additionally, we show that the constraint matrix of the LP relaxation of the ILP in (3.11)-(3.12) is not totally unimodular given most non-trivial input graphs.

3.6.1 The linear relaxation of the ILP in (3.11)-(3.12) does not always yield integer solutions

3.6.1.1 $[\partial]$ is not necessarily totally unimodular

A linear programming formulation always yields integer solutions if its constraint matrix is totally unimodular, which means that all of its square submatrices have determinants of 0, -1 or 1 [37]. To show that the constraint matrix of the LP relaxation of the ILP in (3.11)-(3.12) is not totally unimodular, we first write the LP in standard form.

In a standard form of an LP, all variables are greater than or equal to 0. Since y vectors in the LP relaxation of the ILP in (3.11)-(3.12) can contain negative entries, we decompose it into $y^+ - y^-$. Given alignment graph $\mathcal{A}(G_1, G_2) = (V, E, \delta)$ and $T(G_1, G_2)$, we can now write the standard form of the LP in (3.11)-(3.12) as

$$\begin{aligned} \underset{x\in\mathbb{R}^{|E|},y^+,y^-\in\mathbb{R}^{|T(G_1,G_2)|}}{\text{minimize}} \quad & \sum_{e\in E} x_e \delta(e) \\ \text{subject to} \quad & [I, -[\partial], [\partial]]\,[x, y^+, y^-]^\top = x^{init} \\ & x, y^+, y^- \geq 0. \end{aligned} \tag{3.37}$$

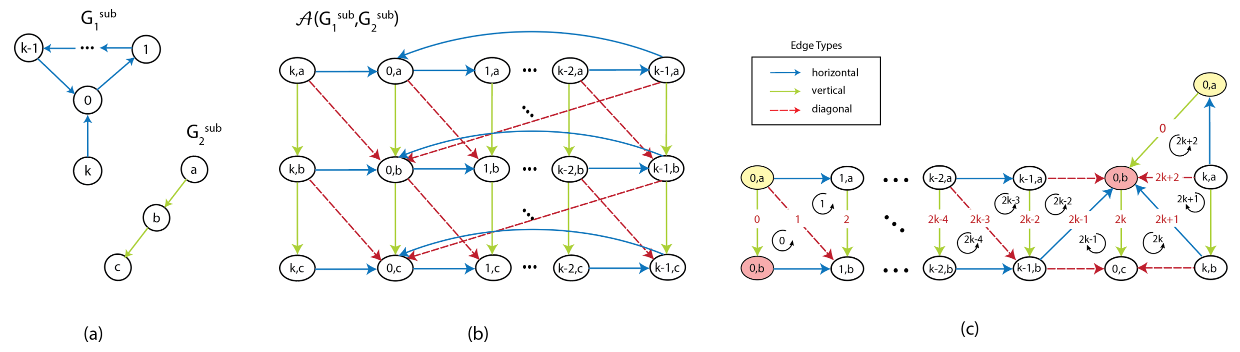


Figure 3.6: (a) Subgraphs G_1^{sub} and G_2^{sub} of input graphs G_1 and G_2. Dots represent a path from node 1 to $k-1$ with middle nodes omitted. (b) The alignment graph $\mathcal{A}(G_1^{sub}, G_2^{sub})$ with different edges labeled with colors. (c) A subgraph of the alignment graph in (b) with edges and triangles numbered. Dots represent horizontal and diagonal edges omitted. The same vertices that are repeated in (c) are marked with yellow and red filling colors.

Hence the constraint matrix of the LP relaxation is $A = [I, -[\partial], [\partial]]$. According to the characteristics of a totally unimodular matrix [136, p. 280] A is not totally unimodular if $[\partial]$ is not totally unimodular. We show that $[\partial]$ is not TU when the input graphs satisfy the constraints given in the following theorem.

Theorem 7. *Given two unidirectional, edge-labeled Eulerian graphs G_1 and G_2 where $|E_1| \geq 2$ and $|E_2| \geq 2$, the boundary matrix $[\partial]$ constructed from $\mathcal{A}(G_1, G_2) = (V, E, \delta)$ and $T(G_1, G_2)$ is not totally unimodular if there is a vertex $v \in V_1$ or V_2 such that there are at least 3 unique edges in E_1 or E_2 that are incident to v. Here, unique edges are edges that connect to v at one end but have different endpoints at the other end.*

Proof. To prove that the boundary matrix is not TU, we only need to show that it is not TU under one specific chosen orientation for 1- and 2-simplices, as well as one specific chosen set of indices for 1- and 2-simplices. This is because changing the orientations or indices of 1-simplices in E or 2-simplices in $T(G_1, G_2)$ corresponds to permuting rows and columns of $[\partial]$ or multiplying rows and columns of $[\partial]$ by -1, which preserves the total unimodularity [136, p. 280].

Without loss of generality, let $v_0 \in V_1$ be a node that is incident to at least 3 unique edges. Since G_1 is an Eulerian graph, v must be part of a cycle C in G_1. Also, there must exist another node v_k and an edge between v_0 and v_k in either direction, such that the edge between v_0 and v_k is not contained in cycle C (Figure 3.6(a)). Suppose the number of nodes in the cycle is k ($k \geq 3$ due to the unidirectionality constraint), and let the cycle $C = v_0, v_1, \ldots, v_{k-1}$. Since a specific choice of 1-simplex orientations does not affect the total unimodularity of the boundary matrix, we assume the edge between v_0 and v_k is $[v_k, v_0]$ without loss of generality. We use $G_1^{sub} = (V_1^{sub}, E_1^{sub})$ to denote the subgraph with $V_1^{sub} = \{v_0, \ldots, v_{k-1}, v_k\}$ and $E_1^{sub} = \{[v_i, v_{i+1}] : i \in \{0, 1, \ldots k-2\}\} \cup \{[v_k, v_0]\}$. Since $|E_2| \geq 2$ and G_2 is a connected graph, there exist two consecutive, directed edges in G_2. We use $G_2^{sub} = (V_2^{sub}, E_2^{sub})$ to denote the subgraph of G_2 with $V_2^{sub} = \{v_a, v_b, v_c\}$ and $E_2^{sub} = \{[v_a, v_b], [v_b, v_c]\}$. The alignment graph $\mathcal{A}(G_1^{sub}, G_2^{sub})$ is formed with G_1^{sub} and G_2^{sub} and is a subgraph of $\mathcal{A}(G_1, G_2)$, therefore, each subgraph of $\mathcal{A}(G_1^{sub}, G_2^{sub})$ is also a subgraph of $\mathcal{A}(G_1, G_2)$. Similarly, the 2-simplex set $T(G_1^{sub}, G_2^{sub})$ is a subset of $T(G_1, G_2)$.

We extract a sequence of 2-simplices (Figure 3.6(c)), T_c, from $T(G_1^{sub}, G_2^{sub})$ via following steps:

1. Extract all oriented 2-simplices $[(v_i, v_a), (v_i, v_b), (v_{i+1}, v_b)]$ and $[(v_i, v_a), (v_{i+1}, v_a), (v_{i+1}, v_b)]$ for $0 \leq i \leq k-2$ from $T(G_1^{sub}, G_2^{sub})$.
   Flip the orientations of $[(v_i, v_a), (v_{i+1}, v_a), (v_{i+1}, v_b)]$ for all $0 \leq i \leq k-2$, obtaining $[(v_i, v_a), (v_{i+1}, v_b), (v_{i+1}, v_a)]$. Use σ_{2i} to denote $[(v_i, v_a), (v_i, v_b), (v_{i+1}, v_b)]$, and σ_{2i+1} to denote $[(v_i, v_a), (v_{i+1}, v_b), (v_{i+1}, v_a)]$.
2. Add to the sequence another five oriented 2-simplices from $T(G_1^{sub}, G_2^{sub})$ in the order as specified: $\sigma_{2k-2} = [(v_{k-1}, v_a), (v_{k-1}, v_b), (v_0, v_b)]$, $\sigma_{2k-1} = [(v_{k-1}, v_b), (v_0, v_b), (v_0, v_c)]$, $\sigma_{2k} = [(v_k, v_b), (v_0, v_b), (v_0, v_c)]$, $\sigma_{2k+1} = [(v_k, v_a), (v_k, v_b), (v_0, v_b)]$ and finally $\sigma_{2k+2} = [(v_k, v_a), (v_0, v_a), (v_0, v_b)]$.

In total, we extract a sequence of $(2k+3)$ oriented 2-simplices, $T_c = \{\sigma_0, \sigma_1, \dots, \sigma_{2k+2}\}$, such that σ_i and $\sigma_{i+1 \bmod (2k+3)}$ share one edge. The extracted 2-simplices and their orientations as well as all shared edges are shown in Figure 3.6(c). We flip the orientations of $[(v_i, v_a), (v_{i+1}, v_a), (v_{i+1}, v_b)]$ solely to ensure that the submatrix constructed below has a simple form, which makes it easier to compute the determinant.

Based on T_c, we obtain M_1, a $(2k+3) \times (2k+3)$ submatrix of $[\partial]$ where each roll corresponds to a shared edge and each column corresponds to a 2-simplex in T_c. The entry values of M_1 are the signed coefficients of each selected 1-simplex from the boundaries of selected 2-simplices.

$$M_1 = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & \dots & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & \dots & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \dots & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & -1 & -1 \end{bmatrix}$$

The determinant of M_1 is:

$$
\begin{aligned}
&\det M_1 \\
&= \det \begin{bmatrix}
-1 & 1 & \dots & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & \dots & 0 & 0 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & \dots & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \dots & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & \dots & 0 & -1 & 1 & 0 & 0 & 0 \\
0 & 0 & \dots & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & \dots & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & \dots & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & -1
\end{bmatrix}
- \det \begin{bmatrix}
1 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 1 & \dots & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & \dots & 0 & 0 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & \dots & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \dots & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & \dots & 0 & -1 & 1 & 0 & 0 & 0 \\
0 & 0 & \dots & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & \dots & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & \dots & 0 & 0 & 0 & 0 & 1 & 1
\end{bmatrix} \\
&= (-1)^{2k-2} \times (-1) - 1^{2k+2} = -2.
\end{aligned}
$$

Since the determinant of M_1 is -2, and M_1 is a submatrix of $[\partial]$, $[\partial]$ is not totally unimodular. □

The minimal pair of input graphs that satisfy the conditions in Theorem 7 is a graph with one 3-node cycle and one additional edge incident to the cycle and an acyclic, connected graph with three nodes. In practice, most non-trivial edge-labeled Eulerian graphs satisfy these conditions.

According to the definitions in Dey et al. [37], the subgraph used to construct M_1 in the above proof (Figure 3.6(c)) is a Möbius subcomplex, and M_1 is a $(2k+3)$-Möbius cycle matrix (MCM). Theorem 7 also establishes that there may exist a Möbius subcomplex in an alignment graph, which corrects the false claim made in Lemma 2 in [16].

Theorem 2 in Boroojeny et al. [16] attempts to employ a more algebraic approach to attempt to demonstrate that $[\partial]$ is TU by establishing that the alignment graph is a Möbius-free product space. However, the property of being Möbius-free globally does not imply the absence of Möbius subcomplexes locally. As we show in Theorem 7, although the alignment graph $\mathcal{A}(G_1^{sub}, G_2^{sub})$ is homotopically equivalent to the one-dimensional circle, which is Möbius-free, it still contains a Möbius subcomplex.

3.6.1.2 The LP yields optimal fractional solutions.

The fact that $[\partial]$ is not totally unimodular does not guarantee that the LP in (3.11)-(3.12) has a fractional optimal objective value. In this section, we prove that the LP in (3.11)-(3.12) does not always yield integer optimal solutions by constructing a specific example with a fractional optimal objective value.

Theorem 8. *The LP in* (3.5)-(3.8) *and the LP in* (3.11)-(3.12) *do not always yield optimal integer solutions.*

We prove the above theorem by giving an example where the LP in (3.5)-(3.8) yields a fractional optimal solution. Since by Theorem 4, two LPs are equivalent, it follows that the LP in (3.11)-(3.12) also yields the same fractional optimal solution.

Construct G_1 and G_2 such that their edges and edge labels are equal to the ones specified in Figure 3.7(a). Let the edge multi-set of $\mathcal{A}(G_1, G_2)$ be E. We assign an edge cost to 0 if

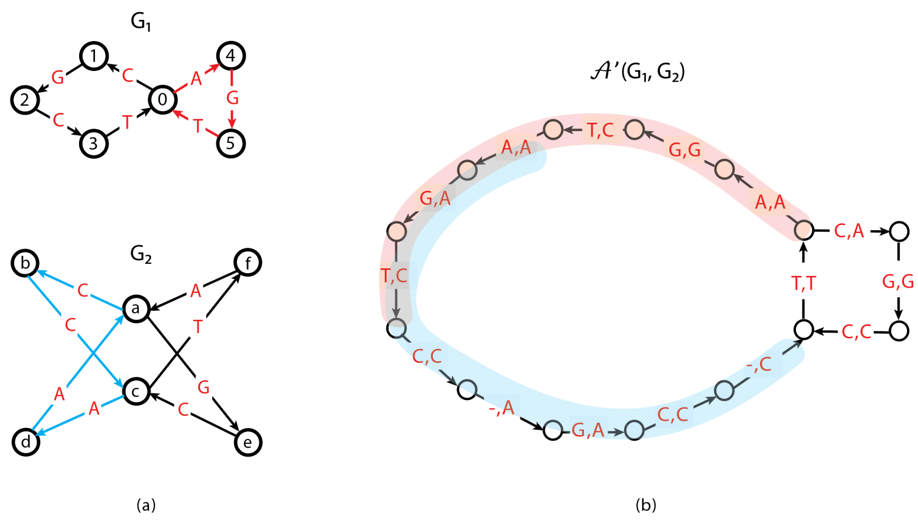


Figure 3.7: An example of a fractional optimal solution to the LP in (3.11)-(3.12) and the LP in (3.5)-(3.8). (a) A pair of input graphs to the LP in (3.11)-(3.12) and the LP in (3.5)-(3.8). Letters in red are edge labels. (b) A subgraph of $\mathcal{A}(G_1, G_2)$ that is induced by alignment edges with non-zero weights (blue font) in an optimal solution to the LPs. The letters in red show the matching between the edge labels or between edge labels and gaps.

the edge matches two equal characters and 1 otherwise. Construct vector $x^* \in \mathrm{R}^{|E|}$ and set entries corresponding to edges in Figure 3.7(b) to 0.5 except edge $[(v_3, v_c), (v_0, v_f)]$ to which the corresponding entry is set to 1. Set the rest of the entries of x^* to 0.

Lemma 9. *x^* is an optimal solution to the LP in* (3.5)-(3.8) *constructed with $\mathcal{A}(G_1, G_2)$ and $T(G_1, G_2)$.*

Proof. We prove the optimality of x^* via complementary slackness. We first write the LP in (3.5)-(3.8) in standard form.

$$\begin{aligned} \underset{x \in \mathbb{R}^{|E|}}{\text{minimize}} \quad & \sum_{e \in E} \delta_e x_e \\ \text{subject to} \quad & Ax = b \\ & x_e \geq 0 \quad \text{for all } e \in E. \end{aligned} \tag{3.38}$$

Here, δ is a vector of size $|E|$ where each entry is cost of edge e. The constraint matrix A of the primal LP (3.38) has $|E|$ columns and $|V| + |E_1| + |E_2| = m$ rows, where V is the vertex set of $\mathcal{A}(G_1, G_2)$, and E_1 and E_2 are edge multi-sets of the input graphs. The first $|V|$ rows correspond to the constraints specified in (3.6). The rest of the rows correspond to the constraints in (3.7) that enforce the projected multi-set of edges to be equal to the multi-set of edges in each input graph. Since the input graphs both contain Eulerian tours, the vector b has size m, where the first $|V|$ entries are zeroes and the rest of the entries are 1s.

We write the dual form of LP (3.38) as follows.

$$\begin{aligned} \underset{y \in \mathbb{R}^m}{\text{maximize}} \quad & \sum_{j=1}^{m} b_j y_j \\ \text{subject to} \quad & A^\top y \leq \delta. \end{aligned} \tag{3.39}$$

Let the objective value of LP (3.38) given a x as input is obj_x^p, and the objective value of LP (3.39) given a y as input is obj_y^d. To show that x^* is an optimal solution to the LP in (3.5)-(3.8), we need to show that there exists a feasible solution to the dual LP, y^*, that satisfies the complementary slackness conditions and that $\text{obj}_{y^*}^d = \text{obj}_{x^*}^p$.

Since each alignment edge has two endpoints and is projected to at most one edge in each graph, there are at most 4 non-zero entries in each column of A. The variables in y of the dual form can be interpreted in three parts. Each of the first $|V|$ entries of y can be assigned to each vertex in the alignment graph, and the next $|E_1|$ entries can be assigned to edges in G_1 and the last $|E_2|$ entries can be assigned to edges in G_2. There are $|E|$ constraints in the dual LP, and the e-th constraint can be assigned to one edge in the alignment graph has cost δ_e. Therefore, each constraint that is assigned to a horizontal or a vertical edge can be written as

$$y_{v_e^{out}} - y_{v_e^{in}} + y_{e_i} \leq \delta_e, \tag{3.40}$$

where $i = 1$ if e is a horizontal edge, and $i = 2$ if e is a vertical edge. $y_{v_e^{in}}$ and $y_{v_e^{out}}$ are the y entries that are assigned to the vertices that are the start and end of edge e, and y_{e_i} are the y entries that assigned to the $\pi_i(e)$.

Similarly, each constraint that is assigned to a diagonal edge is

$$y_{v_e^{in}} - y_{v_e^{out}} + y_{e_1} + y_{e_2} \leq \delta_e. \tag{3.41}$$

We can verify that x^* is a feasible solution of the primal form (3.38) by checking if constraints (3.6)-(3.7) are satisfied. The primal objective value can be computed in a straightforward way, and we can obtain $\text{obj}_{x^*}^p = 3.5$.

According to complementary slackness conditions, since $x_e^* > 0$ for edges shown in Figure 3.7(b), the corresponding constraints in the dual LP (3.39) must be tight, meaning that the equality must hold in these constraints. The rest of the dual constraints could have non-zero slacks.

Let the subgraph of $\mathcal{A}(G_1, G_2)$ shown in Figure 3.7(b) be A'. Denote the cycle that traverses from $[(0, f), (4, a)]$ to $[(3, c), (0, f)]$ be C' and the 4-node cycle that traverses $((0, f), (1, a), (2, e), (3, c))$ be C''. Denote the concatenation of two cycles with C. The projected cycle from C to G_1 is

$$C_1 = (v_0, v_4, v_5, v_0, v_4, v_5, v_0, v_1, v_2, v_3, v_0, v_1, v_2, v_3, v_0). \tag{3.42}$$

The projected cycle from C to G_2 is

$$C_2 = (v_f, v_a, v_e, v_c, v_d, v_a, v_b, v_c, v_d, v_a, v_b, v_c, v_f, v_a, v_e, v_c, v_f). \tag{3.43}$$

Sum up all the constraints that are assigned edge e where $x_e^* > 0$. Since these edges form a cycle, we get:

$$\begin{aligned}
& \sum_{e \in C} \left(y_{v_e^{out}} - y_{v_e^{in}}\right) + 2\Big(\sum_{e_1 \in C_1} y_{e_1} + \sum_{e_2 \in C_2} y_{e_2}\Big) && (3.44)\\
= \quad & 0 + 2\Big(\sum_{e_1 \in C_1} y_{e_1} + \sum_{e_2 \in C_2} y_{e_2}\Big) && (3.45)\\
= \quad & \sum_{e \in C} \delta_e = 7, && (3.46)\\
\Rightarrow \quad & \sum_{e_1 \in C_1} y_{e_1} + \sum_{e_2 \in C_2} y_{e_2} = 3.5. && (3.47)
\end{aligned}$$

The summed edge cost is 7 as there are 7 edges that are either mismatch edges or vertical edges.

All y entries that correspond to vertices are free variables and are in every constraint. After fixing the y variables that satisfy constraints (3.47), the rest of the y variables can be set to satisfy the dual cosntraint. We now obtain y^* which is a feasible solution to the dual LP.

The only entries in y^* that could have non-zero dual costs are those that correspond to edges in E_1 and E_2. Since these corresponding dual costs are all 1,

$$\text{obj}^d_{y^*} = \sum_{e_1 \in C_1} y_{e_1} + \sum_{e_2 \in C_2} y_{e_2} = 3.5 = \text{obj}^p_{x^*}.$$

□

Since the costs of alignment graph edges are all integers, the fact that the LP in (3.11)-(3.12) and the LP in (3.5)-(3.8) yield fractional optimal objective values mean that they must yield fractional solutions and assign fractional values to entries in x. Theorem 8 follows. Since the LP in (3.11)-(3.12) yields fractional solutions and GTED is always an integer, solving the LP in (3.11)-(3.12) does not solve GTED.

3.7. Empirical evaluation of the ILP formulations for GTED and its lower bound

3.7.1 Implementation of the ILP formulations

We implement the algorithms and ILP formulations for (exponential ILP), (compact ILP) and (lower bound ILP). In practice, the multi-set of edges of each input graph may contain many duplicates of edges that have the same start and end vertices due to repeats in the strings. We reduce the number of variables and constraints in the implemented ILPs by merging the edges that share the same start and end nodes and record the multiplicity of each edge. Each x variable is no longer binary but a non-negative integer that satisfy the modified projection constraints (3.16):

$$\sum_{(u,v) \in E} x_{uv} I_i((u,v), f) = M_i(f) \quad \text{for all } (u,v) \in E,\ f \in G_i, u \neq s, v \neq t, \tag{3.48}$$

where $M_i(f)$ is the multiplicity of edge f in G_i. Let C be the strongly connected component in the subgraph induced by positive x_{uv}, now $\sum_{(u,v)\in E(C)} x_{uv}$ is no longer upper bounded by $|E(C)|$. Therefore, the constraints (3.22) is changed to

$$\sum_{(u,v)\in E(C)} x_{uv} - |E(C)| + 1 - W(C)\beta_C \leq 0 \quad \text{for all } C \in \mathcal{C}, \tag{3.49}$$

$$W(C) = \sum_{(u,v)\in E(C)} \max\left(\sum_{f\in G_1} M_1(f) I_1((u,v),f), \sum_{f\in G_2} M_2(f) I_2((u,v),f)\right),$$

where $W(C)$ is the maximum total multiplicities of edges in the strongly connected subgraph in each input graph that is projected from C.

Likewise, constraints (3.30) that set the upper bounds on the ordering variables also need to be modified, as the upper bound of the ordering variable d_{uv} for each edge no longer represents the order of one edge but the sum of orders of copies of (u,v) that are selected, which is at most $|E|^2$. Therefore, constraints (3.30) are changed to

$$d_{uv} - |E|^2(1 - y_{uv}) \leq 0. \tag{3.50}$$

The rest of the constraints remain unchanged.

We ran all our experiments on a server with 48 cores (96 threads) of Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz and 378 GB of memory. The system was running Ubuntu 18.04 with Linux kernel 4.15.0. We solve all the ILP formulations and their linear relaxations using the Gurobi solver [59] using 32 threads.

3.7.2 GTED on simulated TCR sequences

We construct 20 de Bruijn graphs with $k = 4$ using 150-character sequences extracted from the V genes from the IMGT database [80]. We solve the linear relaxation of (compact ILP), (exponential ILP) and (lower bound ILP) and their linear relaxation on all 190 pairs of graphs. We do not show results for solving (compact ILP) for GTED on this set of graphs as the running time exceeds 30 minutes on most pairs of graphs.

To compare the time to solve the ILP formulations when GTED is equal to the optimal objective of (lower bound ILP), we only include 168 out of 190 graphs where GTED is equal to the lower bound. On average, it takes 26 seconds wall-clock time to solve (lower bound ILP), and 71 seconds to solve (exponential ILP) using the iterative algorithm. On average, it takes 9 seconds to solve the LP relaxation of (compact ILP) and 1 second to solve the LP relaxation of (lower bound ILP). The time to construct the alignment graph for all pairs is less than 0.2 seconds. The distribution of wall-clock running time is shown in Figure 3.8(a). The time to solve (exponential ILP) and (lower bound ILP) is generally positively correlated with the GTED values (Figure 3.8(b)). On average, it takes 7 iterations for the iterative algorithm to find the optimal solution that induces one strongly connected subgraph (Figure 3.8(c)).

In summary, it is fastest to compute the lower bound of GTED. Computing GTED exactly by solving the proposed ILPs on genome graphs of size 150 is already time consuming. When the sizes of the genome graphs are fixed, the time to solve for GTED and its lower bound increases

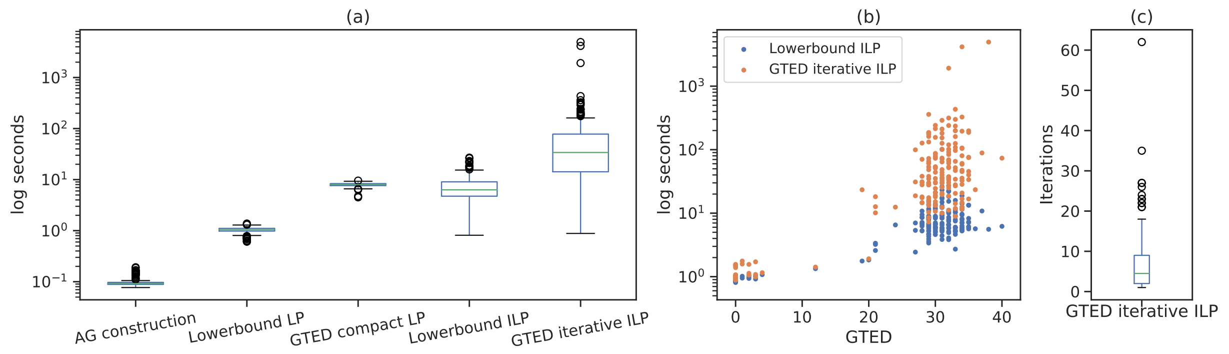


Figure 3.8: (a) The distribution of wall-clock running time for constructing alignment graphs, solving the ILP formulations for GTED and its lower bound, and their linear relaxations on the log scale. (b) The relationship between the time to solve (lower bound ILP), (exponential ILP) iteratively and GTED. (c) The distribution of the number of iterations to solve exponential ILP. The box plots in each plot show the median (middle line), the first and third quantiles (upper and lower boundaries of the box), the range of data within 1.5 inter-quantile range between Q1 and Q3 (whiskers), and the outlier data points.

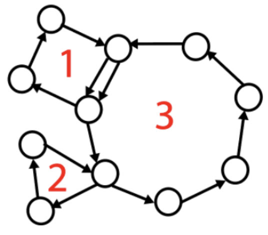
as GTED between the two genome graphs increases. In the case where GTED is equal to its lower bound, the subgraph induced by some optimal solutions of (lower bound ILP) contains more than one strongly connected component. Therefore, in order to reconstruct the strings from each input graph that have the smallest edit distance, we generally need to obtain the optimal solution to the ILP for GTED. In all cases, the time to solve the (exponential ILP) is less than the time to solve the (compact ILP).

3.7.3 GTED on difficult cases

Repeats, such as segmental duplications and translocations [32, 90] in the genomes increase the complexity of genome comparisons. We simulate such structures with a class of graphs that contain n simple cycles of which $n-1$ peripheral cycles are attached to the n-th central cycle at either a node or a set of edges (Figure 3.9(a)). The input graphs in Figure 3.2 belong to this class of graphs that contain 2 cycles. This class of graphs simulates the complex structural variants in disease genomes or the differences between genomes of different species.

We generate pairs of 3-cycle graphs with varying sizes and randomly assign letters from {A, T, C, G} to edges. We compute the lower bound of GTED and GTED using (lower bound ILP) and (compact ILP), respectively. We denote the lower bound of GTED computed by solving (lower bound ILP) as GTED$_l$. We group the generated 3-cycle graph pairs based on the value of $(\text{GTED} - \text{GTED}_l)$ and select 20 pairs of graphs randomly for each $(\text{GTED} - \text{GTED}_l)$ value ranging from 1 to 5. The maximum number of edges in all selected graphs is 32.

We show the difficulty of computing GTED using the iterative algorithm on the 100 selected pairs of 3-cycle graphs. We terminate the ILP solver after 20 minutes. As shown in Figure 3.9, as the difference between GTED and GTED$_l$ increases, the wall-clock time to solve (exponential ILP) for GTED increases faster than the time to solve (compact ILP) for GTED. For pairs on graphs with $(\text{GTED} - \text{GTED}_l) = 5$, on average it takes more than 15



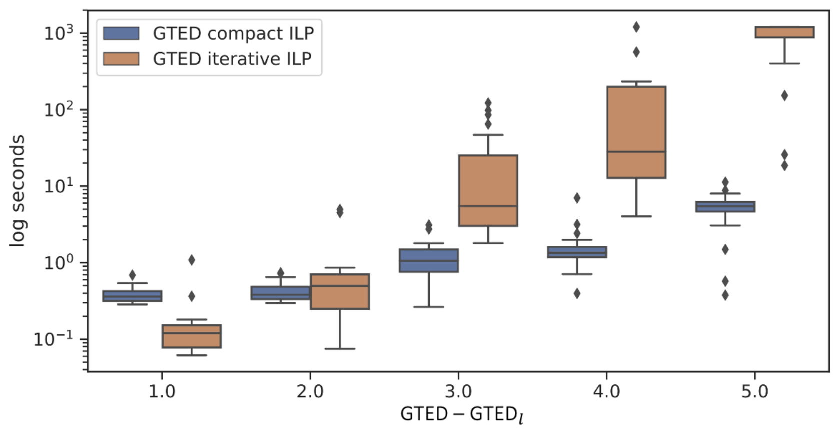


Figure 3.9: (a) An example of a 3-cycle graph. Cycle 1 and 2 are attached to cycle 3. (b) The distribution of wall-clock time to solve the compact ILP and the iterative exponential ILP on 100 pairs of 3-cycle graphs.

minutes to solve (exponential ILP) with more than 500 iterations. On the other hand, it takes an average of 5 seconds to solve (compact ILP) for GTED and no more than 1 second to solve for the lower bound. The average time to solve each ILP is shown in Table 3.1.

In summary, on the class of 3-cycle graphs introduced above, the difficulty to solve GTED via the iterative algorithm increases rapidly as the gap between GTED and GTED_l increases. Although (exponential ILP) is solved more quickly than (compact ILP) for GTED when the sequences are long and the GTED is equal to GTED_l (Section 3.7.2), (compact ILP) may be more efficient when the graphs contain overlapping cycles such that the gap between GTED and GTED_l is larger.

GTED - GTED$_l$	**lower bound ILP runtime (s)**	**GTED iterative runtime (s)**	**Iterations**	**GTED compact runtime (s)**
1.0	0.06	0.17	3.55	0.39
2.0	0.05	0.87	13.00	0.43
3.0	0.08	25.41	67.60	1.24
4.0	0.07	205.59	179.10	1.70
5.0	0.08	943.68	502.85	5.37

Table 3.1: The average wall-clock time to solve lower bound ILP, exponential ILP, compact ILP and the number of iterations for pairs of 3-cycle graphs for each $\text{GTED} - \text{GTED}_l$.

3.8. Conclusion

We point out the contradictions in the result on the complexity of labeled graph comparison problems and resolve the contradictions by showing that GTED, as opposed to the results in Boroojeny et al. [16], is NP-complete. On one hand, this makes GTED a less attractive measure for comparing graphs since it is unlikely that there is an efficient algorithm to compute the measure. On the other hand, this result better explains the difficulty of finding a truly efficient algorithm for computing GTED exactly. In addition, we show that the previously proposed ILP of GTED [16] does not solve GTED and give two new ILP formulations of GTED.

While the previously proposed ILP of GTED does not solve GTED, it solves for a lower bound of GTED. Further, we characterize the LP relaxation of the ILP in (3.11)-(3.12) and show that, contrary to the results in Boroojeny et al. [16], the LP in (3.11)-(3.12) does not always yield optimal integer solutions.

As shown previously [16, 127], it takes more than 4 hours to solve (lower bound ILP) for graphs that represent viral genomes that contain ≈ 3000 bases with a multi-threaded LP solver. Likewise, we show that computing GTED using either (exponential ILP) or (compact ILP) is already slow on small genomes, especially on pairs of genomes that are different due to segmental duplications and translations. The empirical results show that it is currently impossible to solve GTED or its lower bound directly using this approach for bacterial- or eukaryotic-sized genomes on modern hardware. The results here should increase the theoretical interest in GTED along the directions of heuristics or approximation algorithms as justified by the NP-hardness of finding GTED.

Chapter 4

Expressiveness of Genome Graphs and its Effect on Genome Graph Comparison

This chapter was published in ISMB 2022 and Bioinformatics [127] and is joint work with Carl Kingsford. The code to reproduce results in this chapter can be found at https://github.com/Kingsford-Group/gtedemedtest.

4.1. Introduction

Intra-sample heterogeneity describes the phenomenon where a genomic sample contains a diverse set of genomic sequences. A heterogeneous string set is a set of strings where each string is assigned a weight representing its abundance in the set. Computing the distance between heterogeneous string sets is essentially computing the distance between two distributions of strings. We formulate the problem of heterogeneous sample comparison as the heterogeneous string set comparison problem.

This problem can be used to compare samples that contain different sets of genomic sequences. For example, cancer samples are clustered based on differences in their genomic and transcriptomic features [104, 167] into cancer subtypes that correlate with patient survival rates. The dissimilarities between T-cell receptor (TCR) sequences are computed between individuals to study immune responses [15]. Different compositions of these sequences result in different clinical outcomes such as response to treatment.

We point out that the Earth Mover's Distance (EMD) [135], or the Wasserstein distance [160], with edit distance as the ground metric is an elegant metric to compare a pair of heterogeneous string sets. Given two distributions of items and a cost to transform one item into another, EMD computes the total cost of transforming one distribution into another. The EMD was initially used in computer vision to compare distributions of pixel values in images [84] and later adapted to natural language processing [76]. It has also been used to approximate the distance between two genomes [95] by computing the distance between two distributions of k-mers. To compare heterogeneous string sets, when the strings and their distributions are known, we use edit distance as the cost to transform one string to another. We refer to this as the Earth Mover's Edit Distance (EMED).

In practice, the complete strings of interest and their abundances are often unknown, and these strings are only observed as fragmented sequencing reads. It is impossible to exactly compute EMED between the true sets of complete strings from the sequencing reads only.

The challenges posed by incomplete observed sequences can be alleviated by representing the string set using a graph structure. Multiple types of genome graphs have been introduced [6, 39, 51, 62, 64, 79, 89, 103, 118, 119]. For our purposes, a genome graph is a directed multigraph with labeled nodes and weighted edges, along with a source and a sink node. A string is spelled by a source-to-sink path, or s-t path, if it is equal to the concatenation of node labels on the path. We say that a genome graph represents a string set if the union of paths that spells each string in the set is equal to the graph. In other words, a string set can be spelled by a decomposition of the genome graph.

There are several methods that compute the distance between genome graphs [16, 102, 124]. Among those, Graph Traversal Edit Distance (GTED) [16] is a general measure that can be applied to genome graphs and does not rely on the type of genome graphs nor the knowledge of the true string sets. Given two genome graphs, GTED is the minimum edit distance between two strings spelled by Eulerian trails in each graph.

However, applying GTED on genome graphs representing heterogeneous string sets may overestimate the similarity between these string sets for two reasons. First, Eulerian trails in genome graphs that represent heterogeneous string sets spells concatenated strings in the string set. As a result, since GTED is the distance between Eulerian trails, it measures the edit distance between the concatenated strings instead of the distance between the string sets. Specifically, computing the edit distance between concatenated strings may align the prefix of one string to the suffix of another string without appropriate penalties. Thus, as we show in this chapter, GTED always underestimates the actual distance between the heterogeneous string sets. We address this challenge by proposing a variant of GTED, called Flow-GTED, which is the minimum earth mover's edit distance between the string sets spelled by flow decomposition of input graphs.

Second, and more significantly, both FGTED and GTED are the edit distance between the two string sets represented by each genome graph that are most similar to each other. However, a genome graph that is constructed from sequencing fragments typically is able to represent more than one set of strings [72, 120]. As a genome graph merges shared sequences into the same node, it creates chains of bubble structures [165] that result in an exponential number of possible paths, and these paths spell a much more diverse collection of strings than the original set. We call the degree to which a genome graph encodes a larger set of strings than the true underlying set the "expressiveness" of a genome graph. Due to the expressiveness of a genome graph, the Eulerian cycles found by GTED may not spell the true set of strings and the computed distance may be far from the true distance between string sets used to construct the graphs (Figure 4.1(a)).

We prove both that FGTED is always larger than or equal to GTED, and that FGTED is always less than or equal to the EMED between true sets of strings.

However, FGTED and GTED can be quite far from the EMED. To resolve this discrepancy between FGTED and EMED, we define the collection of strings that can be represented by the genome graph as its string set universe, and genome graph expressiveness as the diameter of its string set universe (SUD), which is the maximum EMED between two string sets that can be represented by the graph (Figure 4.1(b)).

Using diameters, we are able to upper-bound the deviation of FGTED from EMED. Addi-

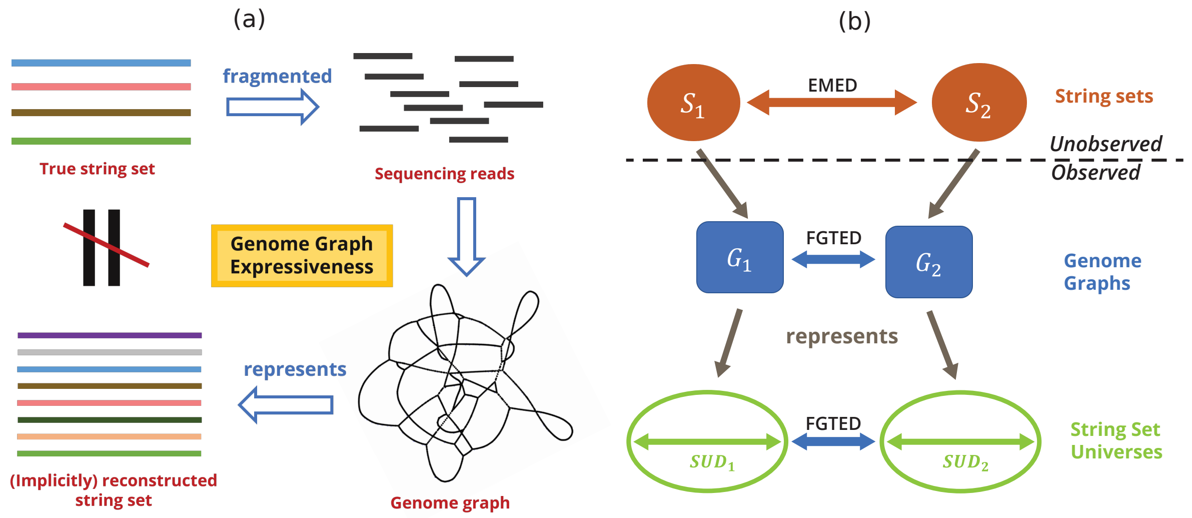


Figure 4.1: (a) Genome graph expressiveness results in inexact representations of true string sets. (b) Overview of part of theoretical contributions.

tionally, we are able to correct FGTED and more accurately estimate the true string set distance empirically. On simulated TCR sequences, we reduce the average deviation of FGTED from EMED by more than 300%, and increase the correlation between the true and estimated string set distances by 20%. On Hepatitis B virus genomes, we reduce the average deviation by more than 250%.

These results provide the first connection between comparisons of genome graphs that encode multiple sequences and a natural string distance and provide the first formalization of the expressiveness of genome graphs. Additionally, they provide a practical method to estimate and reduce discrepancy between genome graph distances and string set distances.

4.2. Preliminary Concepts

4.2.1 Strings

Definition 10 (Heterogeneous string set)**.** *A heterogeneous string set* $\mathcal{S} = \{(w_1, s_1), \dots, (w_n, s_n)\}$ *contains a set of strings, where each string* s_i *is assigned a weight* $w_i \in [0, 1]$ *that indicates the abundance of* s_i *in* $\mathcal{S}$*. We say that the total weight of* $\mathcal{S}$ *is* $\sum_{i\in[1,n]} w_i = 1$.

Here, edit(s_1, s_2) is the minimum cost to transform s_1 into s_2 under edit distance [83]. The set of operations that transforms s_1 to s_2 can be written as an alignment between s_1 and s_2, or $A = align(s_1, s_2)$. The i-th position in A is denoted by $A[i] = \begin{bmatrix} c_{(1,i)} \\ c_{(2,i)} \end{bmatrix}$, where $c_{(a,i)}$ is either a gap character "-" or a character in s_a.

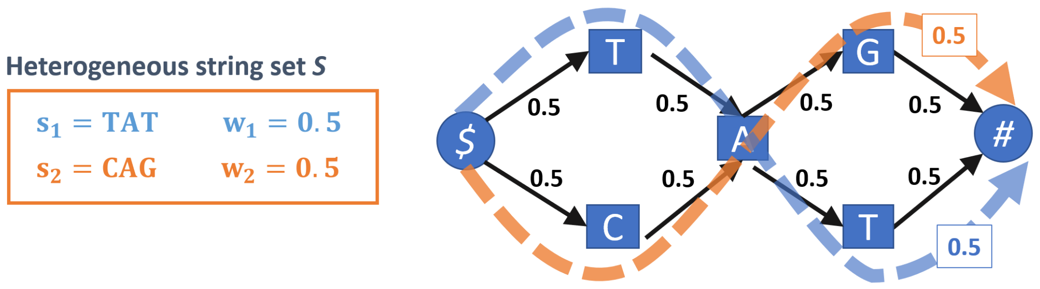


Figure 4.2: An example of a valid flow graph and flow decomposition. The heterogeneous string set $\mathcal{S} = \{(\texttt{TAT}, 0.5), (\texttt{CAG}, 0.5)\}$ is represented by the graph on the right. Each edge in the graph has a capacity/flow of 0.5. The blue and orange dashed arrows represent two paths in a flow decomposition in the graph.

4.2.2 Flow Networks

Definition 11 (Valid flow network)**.** *A directed graph $\mathcal{G} = (V, E, w)$, where $w(e)$ is the weight of each edge, is a valid flow network if there exists a source s and sink node t such that the graph contains exactly one strongly connected component and that*

$$\textit{(Flow conservation)} \sum_{(u,v)\in E} w(u,v) = \sum_{(v,w)\in E} w(v,w) \quad \forall v \in V, v \neq s, v \neq t,$$

$$\textit{(Total capacity)} \sum_{(s,u)\in E} w(s,u) = \sum_{(v,t)\in E} w(v,t) > 0.$$

Definition 12 (s-t Flow decomposition)**.** *A s-t flow decomposition of a valid flow graph $\mathcal{G}$, denoted as $D(\mathcal{G})$, is a collection of s-t paths and their weights $\mathcal{P} = \{(w_1, p_1), \ldots, (w_n, p_n)\}$, where $p_i = \big((s, u_1), \ldots, (u_m, t)\big)$ is an ordered sequence of edges in $\mathcal{G}$, such that:*

$$\textit{(Flow coverage)} \sum_{p_i \in \mathcal{P}} O(e,i) \cdot w_i = w(e) \quad \forall e \in \mathcal{G},$$

where $O(e, i)$ is equal to the number of occurrences of edge e in path p_i.

A valid flow network (Figure 4.2) has at least one way of flow decomposition according to the flow decomposition theorem [3] and typically has more than one flow decomposition. Let the set of all possible flow decomposition of $\mathcal{G}$ be $\mathcal{D}_{\mathcal{G}}$.

4.2.3 Genome Graphs

There are many variants of genome graphs used for various purposes and in various settings. Here, we introduce the definition of genome graphs we will use.

Definition 13 (Genome graph)**.** *A genome graph $\mathcal{G} = (V, E, l, w)$ is a valid flow network with node set V, edge set E, node labels $l(u)$ for each $u \in V$ and edge weights $w(e)$ for each $e \in E$. A genome graph contains a source node s and a sink node t, and $l(s) =$“\$”, $l(t) =$“#”,* *where \$ and # are special characters that do not appear in any string set considered in the scope of this chapter.*

Define operator $S(\cdot)$ that transforms a set of paths in a genome graph $\mathcal{G}$ to a set of strings by concatenating the node labels on each path. $S(\mathcal{P}) = \{(\text{concat}(p), w(p)) \mid p \in \mathcal{P}\}$ is a heterogeneous string set where the weight of each string is equal to the weight of the path that spells the string.

Definition 14 (String set represented by a genome graph)**.** *A genome graph $\mathcal{G}$ represents a string set $\mathcal{S}$ if there exists a decomposition $D(\mathcal{G}) \in \mathcal{D}_{\mathcal{G}}$, such that $S(D(\mathcal{G})) = \mathcal{S}$.*

We use $\mathcal{G} = G(\mathcal{S})$ to denote when $\mathcal{G}$ represents $\mathcal{S}$.

Definition 15 (String set universe represented by a genome graph)**.** *The string set universe $SU(\mathcal{G})$ of a genome graph $\mathcal{G}$ is the collection of heterogeneous string sets that can be represented by $\mathcal{G}$. Formally, $SU(\mathcal{G}) = \{S(D) \mid D \in \mathcal{D}_{\mathcal{G}}\}$.*

4.2.4 Alignment Graph

We use the same definition for the alignment graphs as in Chapter 3. For the sake of completeness, we reiterate the definition here.

Definition 16 (Alignment graph)**.** *Let $G_1,\ G_2$ be two unidirectional, edge-labeled Eulerian graphs. The* alignment graph *$\mathcal{A}(G_1, G_2) = (V, E, \delta)$ is a directed graph that has vertex set $V = V_1 \times V_2$ and edge multi-set E that equals the union of the following:*
Vertical edges $[(u_1, u_2), (v_1, u_2)]$ *for* $(u_1, v_1) \in E_1$ *and* $u_2 \in V_2$,
Horizontal edges $[(u_1, u_2), (u_1, v_2)]$ *for* $u_1 \in V_1$ *and* $(u_2, v_2) \in E_2$,
Diagonal edges $[(u_1, u_2), (v_1, v_2)]$ *for* $(u_1, v_1) \in E_1$ *and* $(u_2, v_2) \in E_2$.
Each edge is associated with a cost by the cost function $\delta : E \to \mathbb{R}$.

Definition 17 (Projection function)**.** *Define the projection function as $P_{(\mathcal{G},\mathcal{H})}(e) = E'$ that maps an edge e from graph $\mathcal{G}$ to a set of edges E' in graph $\mathcal{H}$. The projection function maps an edge in the alignment graph to the edges in the input graphs that are matched together by that edge. It also maps an edge in one of the input graphs to a set of edges in the alignment graph where it is matched with other edges in another input graph. Specifically:*
Projection from alignment graph to one of the input graphs is defined by

$$P_{(AG,\mathcal{G}_i)}((u_1, u_2), (v_1, v_2)) = \{(u_i, v_i)\},\ i \in \{1, 2\}.$$

Projection from one of the input graphs to alignment graph is defined by

$$P_{(\mathcal{G}_1,AG)}((u_1, v_1)) = \{e \mid e = ((u_1, u_2), (v_1, v_2)) \in E_{AG}\},$$
$$P_{(\mathcal{G}_2,AG)}((u_2, v_2)) = \{e \mid e = ((u_1, u_2), (v_1, v_2)) \in E_{AG}\}.$$

Given a set of paths $\mathcal{P}$ in AG, we use $P_{(AG,\mathcal{G}_i)}(\mathcal{P})$ to denote the projection of $\mathcal{P}$ onto $\mathcal{G}_i$, where $P_{(AG,\mathcal{G}_i)}(\mathcal{P}) = \{(P_{(AG,\mathcal{G}_i)}(e) \mid e \in p) \mid p \in \mathcal{P}\}$.

For convenience, we define that $f_i(D(AG)) = S(P_{(AG,\mathcal{G}_i)}(D(AG)))$, which is the set of strings spelled by a path decomposition in AG that is projected onto $\mathcal{G}_i$.

4.2.5 Graph Traversal Edit Distance (GTED)

Graph Traversal Edit Distance (GTED), proposed by Boroojeny et al. [16], is a distance between two labeled graphs which are assumed to be Eulerian graphs. Given a genome graph in our definition, we add an edge directing from sink to source with a weight equal to the sum of edge weights that are directed from the source node in order to make an Eulerian graph.

We reiterate the definition of GTED from Chapter 3.

Problem 6 (Graph Traversal Edit Distance (GTED) [16])**.** *Given two unidirectional, edge-labeled Eulerian graphs G_1 and G_2, compute*

$$\text{GTED}(G_1, G_2) \triangleq \min_{\substack{t_1 \in \mathit{trails}(G_1) \\ t_2 \in \mathit{trails}(G_2)}} \mathit{edit}(\mathit{str}(t_1), \mathit{str}(t_2)). \tag{4.1}$$

Here, $\mathit{trails}(G)$ is the collection of all Eulerian trails in graph G, $\mathit{str}(t)$ is a string constructed by concatenating labels on the Eulerian trail $t = (e_0, e_1, \ldots, e_n)$, and $\mathit{edit}(s_1, s_2)$ is the edit distance between strings s_1 and s_2.

According to Lemma 5, GTED can be computed by finding a s-t trail in the alignment graph whose projection is equal to the input graphs that has a minimum cost.

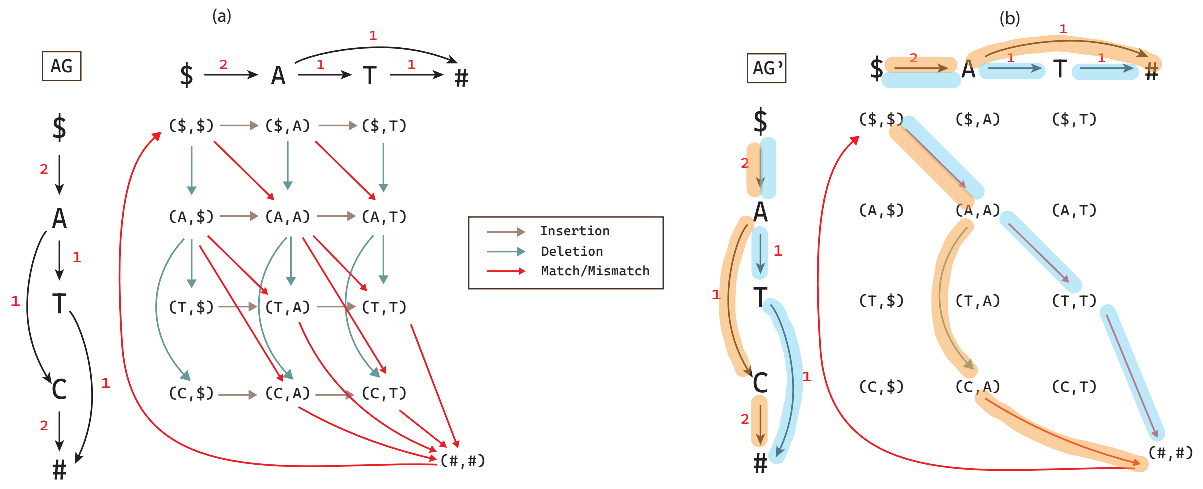


Figure 4.3: (a) An alignment graph AG between $\mathcal{G}_1$ (vertical) and $\mathcal{G}_2$ (horizontal). Insertion, deletion and match/mismatch edges are labeled with different colors. (b) AG' after removing all the edges with zero flow in a solution to FGTED$(\mathcal{G}_1, \mathcal{G}_2)$. Edges in $\mathcal{G}_1$ and $\mathcal{G}_2$ that are highlighted with matching colors are projections from edges in AG' to $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively. Path $(\$, A, T, \#) \in \mathcal{G}_1$ is aligned to $(\$, A, T, \#) \in \mathcal{G}_2$ and path $(\$, A, C, \#) \in \mathcal{G}_1$ is aligned to $(\$, A, \#) \in \mathcal{G}_2$. The weights on AG and AG' edges are omitted for simplicity.

4.3. Traversal edit distance between two pangenome graphs

GTED was originally used to compare genome graphs that are assumed to contain single genomes. It is therefore intuitive that each string represented by the genome graph is spelled with an Eulerian cycle. This property follows the property of assembly graphs [123]. When the genome graph represents more than one string, finding a string spelled by an Eulerian cycle c in the graph is equivalent to finding a concatenation of a permutation of strings in a string set. When aligning two Eulerian cycles, c_1 and c_2, from input graphs, the boundaries between strings are ignored and the prefix of one string may be aligned to the suffix of another string with no cost. However, such alignment is not allowed when we compare sets of strings.

4.3.1 Earth Mover's Edit Distance

To find a distance between two heterogeneous string sets, we need to take into account not only the distance between pairs of strings, but also the abundance of each string in the set. When we compare two heterogeneous string sets, we are essentially comparing two distributions of strings. Therefore, we propose using the Earth Mover's Distance (EMD) as a natural distance measure, which is also known as the Wasserstein distance [156] in the literature and is related to optimal transport.

Given two distributions of items (here, strings) and a cost function that quantifies the cost of transforming one item into another, the EMD between the two distributions is the minimum cost to transform one distribution into another. Computing EMD can be viewed as a transportation problem that finds a many-to-many mapping between two sets of items and minimizes the total cost of the mapping [135, 160].

EMD was used to compare histograms that are normalized to having the same support [135]. In the case where two sets have different total weights, the items that are not matched are dropped without penalty.

To use EMD with the edit distance and compare sets of strings, we allow strings to not be matched with any string and add a penalty for when a string is not matched. The Earth Mover's Edit Distance (EMED) is equal to the cost of the normalized min-cost matching between two heterogeneous string sets defined below. The normalized min-cost matching is similar to the min-cost matching introduced in Section 3.5.1.

Given two heterogeneous string sets, $\mathcal{S}_1$ and $\mathcal{S}_2$, and the edit distance cost function edit$(\cdot,\cdot)$, we add an empty string, ϵ, with weight 0 to each string set to produce $\mathcal{S}_1'$ and $\mathcal{S}_2'$, respectively. The normalized min-cost matching problem finds a matching, $\mathcal{M} = \mathcal{M}_{\text{edit}}(\mathcal{S}_1, \mathcal{S}_2)$, which is a matrix of size $(|\mathcal{S}_1| + 1) \times (|\mathcal{S}_2| + 1)$, such that

$$\begin{aligned} \mathcal{M} = \underset{\mathcal{M}_{\text{edit}}(\mathcal{S}_1,\mathcal{S}_2)}{\operatorname{argmin}} & \sum_{\substack{s_1 \in \mathcal{S}_1' \\ s_2 \in \mathcal{S}_2'}} \text{edit}(s_i, s_j) \cdot \mathcal{M}_{i,j} \\ \text{subject to} \quad & \sum_{i=1}^{|\mathcal{S}_1|} \mathcal{M}_{i,j} = w_j \quad \text{for all } 1 \leq j \leq |\mathcal{S}_2|, \\ & \sum_{j=1}^{|\mathcal{S}_2|} \mathcal{M}_{i,j} = w_i \quad \text{for all } 1 \leq j \leq |\mathcal{S}_1|. \end{aligned} \tag{4.2}$$

Equations (4.2) define a mapping between $\mathcal{S}_1'$ and $\mathcal{S}_2'$ that contain the added empty strings. The normalized EMED allows but does not enforce matching between a nonempty string with an empty string. The edit distance between an empty string and a nonempty string s is equal to the length of s.

EMED between two string sets can be computed in polynomial time by solving the LP relaxation of the ILP formulation (4.2).

4.3.2 Flow Graph Traversal Edit Distance

We propose a variant of GTED so that it is equal to the minimum EMED between string sets represented by each genome graph.

Definition 18 (Flow-GTED). *Given two genome graphs $\mathcal{G}_1$ and $\mathcal{G}_2$,*

$$\text{FGTED}(G_1, G_2) = \min_{\substack{D_1 \in D_{\mathcal{G}_1} \\ D_2 \in D_{\mathcal{G}_2}}} \quad \textit{EMED}(S(D_1), S(D_2))$$

4.3.3 FGTED is NP-complete

We show that FGTED is NP-complete using a similar argument as in the proof of the NP-completeness of GTED in chapter 3 (Theorem 3).

Theorem 9. *FGTED is NP-complete.*

Proof. Let $G = (v, E)$ be an instance of the HAMILTONIAN CYCLE problem. Let $n = |V|$ be the number of vertices in G. Construct the Eulerian closure of G and split the anti-parallel edges. Let the new graph be $G' = (V', E')$. Attach a source s and a sink node t to an arbitrary node v_1^{in} by adding edge (s, v_1^{in}) and (v_1^{in}, t) with labels $\mathtt{s}$ and $\mathtt{t}$, respectively.

Construct a string q, such that

$$q = \mathtt{sa\#}(\mathtt{b\#a\#})^{n-1}(\mathtt{c\#})^{2n-1}(\mathtt{c\#b\#})^{|E|+1}\mathtt{t}. \tag{4.3}$$

Create a graph Q that only contains one path with labels on the edges of the path that spell the string q. The union of the set of trails in any flow decomposition of G' is equal to a set of Eulerian trails, $\mathcal{E}$, that starts at s and ends at t. All Eulerian trails in $\mathcal{E}$ are also closed Eulerian trails of $G' \setminus \{s, t\}$ that starts and ends at v_1^{in}.

Using the same line of argument in the proof of Theorem 3, an Eulerian trail in G' that spells q is equivalent to a Hamilton Cycle in G. In addition, $\text{FGTED}(Q, G') = 0$ if and only if all Eulerian trails in $\mathcal{E}$ spell out q. Therefore, if $\text{FGTED}(Q, G') = 0$, then there is a Hamiltonian Cycle in G. Otherwise, then there must not exist a Hamiltonian Cycle in G. □

4.3.4 FGTED reduced to GTED with a modified alignment graph

We show that FGTED can be computed by using GTED as a subroutine and propose the following algorithm for solving FGTED.

Given two input graphs G_1 and G_2, we add an edge from sink to source to each input graph so that they are Eulerian. Let $\mathcal{A}(G_1, G_2)$ be the alignment graph constructed based on the two input genome graphs G_1, G_2. Let s and t be the source and sink of the alignment graph. Modify the edge costs such that the cost of aligning the sink character "#" with any other character and the cost of introducing gaps right before the edge with the sink character is infinity.

Suppose a solver solves GTED by finding the minimum-cost s-t trail in $\mathcal{A}(G_1, G_2)$. Let the trail found by the GTED solver be $\mathcal{A}'$, which is a subgraph of $\mathcal{A}$. Let $\mathcal{A}^*$ be $\mathcal{A}'$ after removing the sink-to-source edge.

Theorem 10. *The total cost of edges in $\mathcal{A}^*$ is equal to* $\text{FGTED}(\mathcal{G}_1, \mathcal{G}_2)$.

We prove Theorem 10 by showing that each s-t path in the flow decomposition of $D(\mathcal{A}^*)$ is equal to the edit distance between two strings represented by two s-t paths in each of the input graphs.

Figure 4.3(a) gives an example of the alignment graph built from two input graphs using the proposed cost function. Let the sink nodes in $\mathcal{G}_1$ and $\mathcal{G}_2$ be t_1 and t_2, and the source nodes be s_1 and s_2, respectively. After removing all the alignment edges with infinite costs, there is an edge to the alignment node (t_1, t_2) in AG if and only if there exists an edge (u_1, t_1) in $\mathcal{G}_1$ and an edge (u_2, t_2) in $\mathcal{G}_2$. The only incoming edge to (s_1, s_2) is $(s, (s_1, s_2))$ and $((t_1, t_2), (s_1, s_2))$, and the only outgoing edge from (t_1, t_2) is $((s_1, s_2), t)$ and $((t_1, t_2), (s_1, s_2))$. We refer to the edge $((t_1, t_2), (s_1, s_2))$ as the sink-to-source edge in the alignment graph in the rest of this chapter.

Lemma 10. *Given an s-t path $p \in D(\mathcal{A}^*)$, let $s_1 = S(P_{(\mathcal{A}^*, \mathcal{G}_1)}(p))$ be the string spelled by projecting p onto $\mathcal{G}_1$, and $s_2 = S(P_{(\mathcal{A}^*, \mathcal{G}_2)}(p))$. Then for any $p \in D(\mathcal{A}^*)$,*

$$\sum_{e \in p} \mathit{cost}(e) = \mathit{edit}(s_1, s_2).$$

Proof. We prove in two directions.

($\geq$ direction) We construct $A = \mathit{align}(s_1, s_2)$ from p. For each $e = ((u_1, u_2), (v_1, v_2)) \in p$: (1) if $u_1 = v_1$, add $c = \begin{bmatrix} - \\ l(v_2) \end{bmatrix}$ to A, (2) if $u_2 = v_2$, add $c = \begin{bmatrix} l(v_1) \\ - \end{bmatrix}$ to A, (3) else, add $c = \begin{bmatrix} l(v_1) \\ l(v_2) \end{bmatrix}$ to A. By definition of an alignment graph, $\mathit{cost}(e) = \mathit{cost}(c)$ in for all e, and therefore

$$\mathit{cost}(A) = \sum_{c \in A} \mathit{cost}(c) = \sum_{e \in p} \mathit{cost}(e).$$

Since edit distance minimizes the cost of edit operations, $\mathit{cost}(A) = \mathit{cost}(p) \geq \text{ED}(s_1, s_2)$.

($\leq$ direction) We construct p' from $A^* = \mathit{align}(s_1, s_2)$ such that $\mathit{cost}(A^*) = \text{ED}(s_1, s_2)$. The procedure is similar as above — for each pair of adjacent entries in A^*, add corresponding edge to p'. Then $\mathit{cost}(p') = \mathit{cost}(A^*) = \text{ED}(s_1, s_2)$.

Let $\mathcal{A}' = \mathcal{A}^* \setminus p \cup p'$. Both p and p' can be found in $\mathcal{A}$, and both p and p' can be constructed by the alignment of the same pair of strings. Therefore, $\mathcal{A}'$ is also a valid flow network and a feasible solution to GTED. Since $\mathcal{A}^*$ is the optimal solution to GTED, $\mathit{cost}(\mathcal{A}^*) \leq \mathit{cost}(\mathcal{A}')$, and

$$\begin{aligned}
&\mathit{cost}(AG^*) - \mathit{cost}(AG') \leq 0 \\
&\Rightarrow w(p) \cdot (\mathit{cost}(p) - \mathit{cost}(p')) \leq 0 \\
&\Rightarrow w(p) \cdot (\mathit{cost}(p) - \text{ED}(s_1, s_2)) \leq 0 \\
&\Rightarrow \mathit{cost}(p) \leq \text{ED}(s_1, s_2).
\end{aligned}$$

□

We then show that for any arbitrary s-t flow decomposition of $\mathcal{A}^*$, the earth mover's edit distance between string sets reconstructed from the decomposition is equal to the sum of edge costs in $\mathcal{A}^*$.

Lemma 11. *Given $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$ obtained from any decomposition $D(AG^*) \in \mathcal{D}_{\mathcal{A}^*}$,*

$$\textit{EMED}(\mathcal{S}_1^*, \mathcal{S}_2^*) = \textit{cost}(\mathcal{A}^*),$$

where $\textit{cost}(\mathcal{A}^*)$ is sum of edge costs in the solution alignment graph to GTED.

Proof. We prove in two directions.

($\leq$ direction) We construct a mapping M between strings in $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$ from the decomposition $D(\mathcal{A}^*)$, where $M(s_i, s_j)$ is the portion of $s_i \in \mathcal{S}_1$ and $s_j \in \mathcal{S}_2$ that are aligned. For each $p \in D(\mathcal{A}^*)$, we obtain s_1 and s_2 as strings constructed from projections of p onto $\mathcal{G}_1$ and $\mathcal{G}_2$ and increment the weight of mapping $M(s_1, s_2)$ by $w(p)$. After iterating through all paths in $D(\mathcal{A}^*)$, the cost of M is

$$\begin{aligned} \textit{cost}(M) &= \sum_{(s_1,s_2)\in M} M(s_1, s_2) \cdot \text{ED}(s_1, s_2) \\ &= \sum_{p\in D(\mathcal{A}^*)} w(p) \cdot \textit{cost}(p) = \textit{cost}(\mathcal{A}^*). \end{aligned}$$

M is also a feasible solution to the LP formulation of EMED (4.2). Since EMED minimizes the cost of mapping between $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$, EMED$(\mathcal{S}_1^*, \mathcal{S}_2^*) \leq \textit{cost}(\mathcal{A}^*)$.

($\geq$ direction) We construct a valid flow network, $\mathcal{A}'$ using an optimal solution to EMED$(\mathcal{S}_1^*, \mathcal{S}_2^*)$. For each pairing (s_i, s_j) for $s_i \in \mathcal{S}_1$ and $s_j \in \mathcal{S}_2$, we obtain its weight w and cost c from the EMED solution. Let $A = \textit{align}(s_i, s_j)$ be an optimal alignment under edit distance, and $\textit{cost}(A) = c$. We then add a path corresponding to A with weight w in $\mathcal{A}'$. This follows the same procedure in the proof of Lemma 10. After adding all paths, we obtain $\mathcal{A}'$ with $\textit{cost}(\mathcal{A}') =$ EMED$(\mathcal{S}_1^*, \mathcal{S}_2^*)$. Since $\textit{cost}(\mathcal{A}^*)$ is minimized by GTED, $\textit{cost}(\mathcal{A}') \geq \textit{cost}(\mathcal{A}^*) \Rightarrow$ EMED$(\mathcal{S}_1^*, \mathcal{S}_2^*) \geq \textit{cost}(\mathcal{A}^*)$. □

Lemma 11 provides a transformation algorithm between optimal solutions to EMED and solutions to GTED under modified edge costs related to sink characters. Using Lemma 11, we can show that the EMED between $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$ constructed from any decomposition in $\mathcal{A}^*$ is equal to the decompositions of $\mathcal{G}_1$ and $\mathcal{G}_2$ that are closest in terms of EMED.

Lemma 12. *Given $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$ obtained from any decomposition $D(\mathcal{A}^*) \in \mathcal{D}_{\mathcal{A}^*}$,*

$$\textit{EMED}(\mathcal{S}_1^*, \mathcal{S}_2^*) = \min_{\substack{D_1\in\mathcal{D}_{\mathcal{G}_2},\\ D_2\in\mathcal{D}_{\mathcal{G}_2}}} \textit{EMED}\big(S(D_1), S(D_2)\big)$$

Proof. In Lemma 11, $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$ can be constructed from decomposing $\mathcal{G}_1$ and $\mathcal{G}_2$. Suppose for contradiction that there exists a decomposition that constructs string sets $\mathcal{S}_1'$ and $\mathcal{S}_2'$, such that EMED$(\mathcal{S}_1^*, \mathcal{S}_2^*) >$ EMED$(\mathcal{S}_1', \mathcal{S}_2')$. Following the procedure in the proof of Lemma 11, we can construct a feasible solution to GTED with a cost equal to EMED$(\mathcal{S}_1', \mathcal{S}_2')$, which is less than $\textit{cost}(\mathcal{A}^*) =$ EMED$(\mathcal{S}_1^*, \mathcal{S}_2^*)$. This contradicts the assumption that GTED minimizes $\textit{cost}(\mathcal{A}^*)$. □

According to Lemma 12 and definition of FGTED, the proposed algorithm to solve FGTED via GTED is correct.

4.4. The Relationship Between GTED, FGTED and EMED

4.4.1 FGTED is Always Smaller Than or Equal to GTED

Theorem 11. $\text{GTED}(\mathcal{G}_1, \mathcal{G}_2) \leq \text{FGTED}(\mathcal{G}_1, \mathcal{G}_2)$ *for any pair of genome graphs* $\mathcal{G}_1, \mathcal{G}_2$.

Proof. Since computing FGTED uses an alignment graph that is a subgraph of the alignment graph without modification, any solution to FGTED can be transformed into a solution in GTED. Since $GTED(\mathcal{G}_1, \mathcal{G}_2)$ minimizes the total edge costs, the theorem is true. □

Observing that we can do flow decomposition in both the FGTED solution and input genome graphs, we will show in this section that FGTED can be bounded by EMED between decompositions in input genome graphs and in the alignment graph solutions.

Theorem 12. *Given two sets of strings $\mathcal{S}_1$ and $\mathcal{S}_2$, and genome graphs representing these string sets, $\mathcal{G}_1 = G(\mathcal{S}_1)$ and $\mathcal{G}_2 = G(\mathcal{S}_2)$,*

$$\begin{aligned} 0 &\leq \textit{EMED}(\mathcal{S}_1, \mathcal{S}_2) - \textit{FGTED}(\mathcal{G}_1, \mathcal{G}_2) && (4.4)\\ &\leq \min_{\substack{D(\mathcal{A}^*) \in \mathcal{D}_{\mathcal{A}^*} \\ \mathcal{S}_1^* = f_1(D(\mathcal{A}^*)) \\ \mathcal{S}_2^* = f_2(D(\mathcal{A}^*))}} \big(\textit{EMED}(\mathcal{S}_1, \mathcal{S}_1^*) + \textit{EMED}(\mathcal{S}_2, \mathcal{S}_2^*)\big) && (4.5) \end{aligned}$$

where $\mathcal{A}^*$ is the solution obtained from $\text{FGTED}(\mathcal{G}_1, \mathcal{G}_2)$.

The proof of this theorem is completed in two parts. The first inequality is shown in Section 4.4.2 and the second is proven in Section 4.4.3. Since FGTED computes a distance that is larger than GTED between the same pair of genome graphs (Theorem 11), Theorem 12 also shows that FGTED always estimates the distance between true string sets more accurately than GTED.

4.4.2 FGTED is Always Less Than or Equal to EMED

Lemma 13. *Given heterogeneous string sets $\mathcal{S}_1$ and $\mathcal{S}_2$ and genome graphs representing these string sets, $\mathcal{G}_1 = G(\mathcal{S}_1)$ and $\mathcal{G}_2 = G(\mathcal{S}_2)$, $\textit{FGTED}(\mathcal{G}_1, \mathcal{G}_2) \leq \textit{EMED}(\mathcal{S}_1, \mathcal{S}_2)$.*

Proof. According to the definition of FGTED, FGTED is equal to flow decomposition in $\mathcal{D}_{\mathcal{G}_1}$ and $\mathcal{D}_{\mathcal{G}_2}$ that minimizes the EMED between them. Since $\mathcal{S}_1$ and $\mathcal{S}_2$ can be constructed from a flow decomposition in $\mathcal{D}_{\mathcal{G}_1}$ and $\mathcal{D}_{\mathcal{G}_2}$, respectively, this lemma is true. □

4.4.3 Genome Graph Expressiveness

A genome graph typically can represent more than one set of strings. We name the collection of string sets representable by a genome graph the *string set universe* of that genome graph, or $SU(\mathcal{G})$. FGTED is equal to the minimum EMED between two sets of strings in the string set universe of $\mathcal{G}$ that are the closest in the metric space of EMED. We define the expressiveness of a genome graph as the diameter of its string set universe, which is the maximum EMED between the string sets in $SU(\mathcal{G})$.

Definition 19 (String Set Universe Diameter (SUD))**.** *Given a genome graph $\mathcal{G}$,*

$$SUD(\mathcal{G}) = \max_{\mathcal{S}_a, \mathcal{S}_b \in SU(\mathcal{G})} EMED(\mathcal{S}_a, \mathcal{S}_b)$$

4.4.3.1 String Set Universe Diameter as an Upper Bound on Deviation of FGTED from EMED

The string set universe diameter gives one measure of the size of $SU(G)$, and it can also be used to characterize the deviation of GTED from EMED.

Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be the true string sets that are represented by $\mathcal{G}_1$ and $\mathcal{G}_2$. Recall that $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$ are string sets obtained from a decomposition $D(\mathcal{A}^*)$, and that
EMED$(\mathcal{S}_1^*, \mathcal{S}_2^*)$ = FGTED$(G(\mathcal{S}_1), G(\mathcal{S}_2))$. From Theorem 12, we have that EMED$(\mathcal{S}_1, \mathcal{S}_2) \geq$ EMED$(\mathcal{S}_1^*, \mathcal{S}_2^*)$. We can bound the deviation of EMED$(\mathcal{S}_1^*, \mathcal{S}_2^*)$ from EMED$(\mathcal{S}_1, \mathcal{S}_2)$ using triangle inequalities.

Lemma 14. *Given string sets $\mathcal{S}_1$ and $\mathcal{S}_2$ and genome graphs $\mathcal{G}_1 = G(\mathcal{S}_1)$ and $\mathcal{G}_2 = G(\mathcal{S}_2)$,*

$$\begin{aligned} &EMED(\mathcal{S}_1, \mathcal{S}_2) - FGTED(\mathcal{G}_1, \mathcal{G}_2) \\ &\leq \min_{\substack{D(AG^*) \in \mathcal{D}_{AG^*} \\ \mathcal{S}_1^* = f_1(D(AG^*)) \\ \mathcal{S}_2^* = f_2(D(AG^*))}} \big(EMED(\mathcal{S}_1, \mathcal{S}_1^*) + EMED(\mathcal{S}_2, \mathcal{S}_2^*)\big), \end{aligned} \tag{4.6}$$

where AG^* is the solution obtained from FGTED$(\mathcal{G}_1, \mathcal{G}_2)$.

Proof. Both edit distance and EMD are metrics [83, 135], which means that triangle inequality holds for EMED between strings. Therefore, for any string sets $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$,

$$\begin{aligned} \text{EMED}(\mathcal{S}_1, \mathcal{S}_1^*) + \text{EMED}(\mathcal{S}_1^*, \mathcal{S}_2) \geq \text{EMED}(\mathcal{S}_1, \mathcal{S}_2) \\ \text{EMED}(\mathcal{S}_2, \mathcal{S}_2^*) + \text{EMED}(\mathcal{S}_1^*, \mathcal{S}_2^*) \geq \text{EMED}(\mathcal{S}_1^*, \mathcal{S}_2) \end{aligned}$$

Combining two inequalities, we have

$$\begin{aligned} &\text{EMED}(\mathcal{S}_1^*, \mathcal{S}_2^*) = \text{FGTED}(\mathcal{G}_1, \mathcal{G}_2) \geq \text{EMED}(\mathcal{S}_1, \mathcal{S}_2) - (\text{EMED}(\mathcal{S}_1, \mathcal{S}_1^*) + \text{EMED}(\mathcal{S}_2, \mathcal{S}_2^*)) \\ \Rightarrow\quad &\text{EMED}(\mathcal{S}_1, \mathcal{S}_2) - \text{FGTED}(\mathcal{G}_1, \mathcal{G}_2) \leq \text{EMED}(\mathcal{S}_1, \mathcal{S}_1^*) + \text{EMED}(\mathcal{S}_2, \mathcal{S}_2^*). \end{aligned} \tag{4.7}$$

The above inequality (4.7) holds for any string sets $\mathcal{S}_1^*$ and $\mathcal{S}_2^*$. To give a tight upper bound on the deviation, we take the minimum over all possible pairs of string sets constructed from decomposing $\mathcal{A}^*$ that yields inequality (4.6). □

Lemma 14 proves the second inequality of Theorem 12 thus completing the proof for Theorem 12 with Lemma 13.

The upper-bound found in Lemma 14 can be used as a factor that evaluates the pair-wise expressiveness of two genome graphs. While a genome graph may represent a large universe of string sets, as long as the true string set is close to the "best" string set in the pair-wise comparison, the deviation of FGTED from EMED is small. We define this upper bound as the String Universe Co-Expansion Factor (SUCEF), which can be used to evaluate the discrepancy between FGTED and EMED.

Definition 20 (String Universe Co-Expansion Factor (SUCEF))**.**

$$\mathit{SUCEF}(\mathcal{S}_1, \mathcal{S}_2, \mathcal{G}_1, \mathcal{G}_2) = \min_{\substack{D(AG^*) \in \mathcal{D}_{AG^*} \\ \mathcal{S}_1^* = f_1(D(AG^*)) \\ \mathcal{S}_2^* = f_2(D(AG^*))}} \big(\mathit{EMED}(\mathcal{S}_1, \mathcal{S}_1^*) + \mathit{EMED}(\mathcal{S}_2, \mathcal{S}_2^*)\big),$$

where $\mathcal{A}^*$ is the solution to FGTED$(\mathcal{G}_1, \mathcal{G}_2)$.

On the other hand, finding SUCEF not only requires knowledge of true string sets $\mathcal{S}_1$ and $\mathcal{S}_2$, but SUCEF is also a pair-dependent measure that needs to be calculated for every pair of string sets and corresponding genome graphs. In order to characterize the effect of the expressiveness of individual genome graphs, we introduce another upper bound on the deviation of FGTED from EMED using the string set universe diameters.

The sum of string set universe diameters of two genome graphs is an upper bound on SUCEF of these graphs and any two sets of strings they represent.

Lemma 15. *Given two genome graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ and two sets of strings $\mathcal{S}_1$ and $\mathcal{S}_2$ they represent,*

$$\begin{aligned} \mathit{EMED}(\mathcal{S}_1, \mathcal{S}_2) - \mathit{FGTED}(\mathcal{G}_1, \mathcal{G}_2) &\leq \mathit{SUCEF}(\mathcal{S}_1, \mathcal{S}_2, \mathcal{G}_1, \mathcal{G}_2) \\ &\leq \mathit{SUD}(\mathcal{G}_1) + \mathit{SUD}(\mathcal{G}_2). \end{aligned}$$

Proof. Both $\mathcal{S}_1$ and $\mathcal{S}_1^*$ are represented by $\mathcal{G}_1$ and belong to SU$(\mathcal{G}_1)$. Therefore, by definition of string set universe diameter, EMED$(\mathcal{S}_1, \mathcal{S}_1^*) \leq$ SUD$(\mathcal{G}_1)$ as the diameter maximizes the distance between any pair of strings represented by the genome graph. The same holds for EMED$(\mathcal{S}_2, \mathcal{S}_2^*) \leq$ SUD$(\mathcal{G}_2)$. $\square$

Using Lemma 15, we can bound the deviation of FGTED from EMED using the expressiveness of individual genome graphs even when we do not have the knowledge of ground truth string sets. In practice, we can construct genome graphs using known sequences from the species of interest and form a training set. Using the training set, we can learn the relationship between SUDs and the deviation of FGTED from EMED, and then empirically estimate the anticipated discrepancy between FGTED and EMED. In the following sections, we show that we can improve FGTED using SUDs to obtain reduced anticipated deviation from EMED and stronger correlation with EMED.

4.5. Correcting the discrepancy between FGTED and EMED empirically

4.5.1 Estimating String Set Universe Diameters

The string set universe diameter of a genome graph can be estimated by sampling flow decompositions of the graph. To sample a flow decomposition, we first sample one s-t path. At each node u, we choose the neighbor v with the highest edge weight $w(u, v)$ with probability 0.5 and randomly choose a neighbor otherwise. After sampling a path, we send flow that is equal to the minimum edge weight on that path and produce the residual graph by subtracting the flow from edge weights on that s-t path. We repeat this process on the residual graph until all edge weights are zero. This process assumes that the input genome graphs are acyclic to ensure all edge capacities (weights) are satisfied because otherwise there might be isolated cycles in the graph that is not reachable from the source node. If a genome graph is cyclic, e.g. de Bruijn graphs, string sets from $\text{SU}(\mathcal{G}_1)$ can be obtained by sampling Eulerian cycles in the genome graph, and each string in the string set is obtained by segmenting the sampled Eulerian cycle at source and sink nodes. After sampling 50 pairs of flow decompositions, we construct string sets from sampled flow decompositions and calculate pairwise EMED. We then obtain the highest pairwise EMED and use it as the estimated diameter.

4.5.2 Correcting FGTED Using String Set Universe Diameters

Using the sum of SUDs, we empirically estimate the deviation of FGTED from EMED with a linear regression model. We denote the deviation of FGTED from EMED by $\text{deviation}(\mathcal{S}_1, \mathcal{S}_2, \mathcal{G}_1, \mathcal{G}_2)$, which is computed as $|\text{EMED}(\mathcal{S}_1, \mathcal{S}_2) - \text{FGTED}(G(\mathcal{S}_1), G(\mathcal{S}_2))|$. The linear regression model, LR, has the following form:

$$\begin{aligned}\text{deviation}(\mathcal{S}_1, \mathcal{S}_2, \mathcal{G}_1, \mathcal{G}_2) &= a \cdot \big(\text{SUD}(\mathcal{G}_1) + \text{SUD}(\mathcal{G}_2)\big) + b \\ &= LR\big(\text{SUD}(\mathcal{G}_1) + \text{SUD}(\mathcal{G}_2)\big),\end{aligned}$$

where a is the coefficient of the model and b is the intercept. The fitted model will minimize the mean squared error between predicted deviation and true deviation in the training set.

The corrected FGTED for each pair of graphs is calculated using the learned linear regression model as follows.

$$\text{correctedFGTED}(\mathcal{G}_1, \mathcal{G}_2) = \text{FGTED}(\mathcal{G}_1, \mathcal{G}_2) + LR\big(\text{SUD}(\mathcal{G}_1) + \text{SUD}(\mathcal{G}_2)\big)$$

The deviation of corrected FGTED from EMED has the same form as the deviation of uncorrected FGTED from EMED.

4.5.3 Data

We evaluate the use of string set universe diameters on two sequence sets:

1. **Simulated T-Cell Receptor (TCR) Repertoire.** We simulate 50 sets of TCR sequences and assign weights to each sequence using reference gene sequences of V, D and J genes from Immunogenetics (IMGT) V-Quest sequence directory [81]. The number of sequences in each set varies from 2 to 5. We then generate 225 pairs of TCR string sets. Each TCR sequence is about 300 base pairs long. See Supplementary Materials for detailed simulation process.
2. **Hepatitis B Virus (HBV) Genomes.** We collect 9 sets of HBV genomes from three hosts — humans, bats and ducks — from the NCBI virus database [61]. We build 36 pairs of HBV string sets. See Supplementary Materials for detailed string set construction process.

4.5.4 Procedures to generate data sets

	Group 1	Group 2	Group 3	Group 4	Group 5
TRB_V	5	34	63	92	121
TRB_J	5	8	11	14	15
TRB_D	3	3	3	3	3

Table 4.1: The number of unique V, J and D gene sequences in each reference gene group.

4.5.4.1 Synthetic Sets of T-cell Receptor Sequences

We construct five reference gene groups by sampling reference sequences obtained from the IMGT database [81] that represent varied diversities of V, D, J gene repertoires. The number of sequences in each group is shown in Table 4.1. We construct five TCR sequence groups, and each group of TCR sequences are constructed using genes from one of the five reference gene groups. To generate each TCR sequence, we randomly select a V, D and J gene from corresponding gene group, and randomly introduce $m \in \{1, 3, 5, 8, 10\}$ single-nucleotide mutations to each sequence at random locations. This step is to simulate recombination and occurrences of junction single nucleotide polymorphisms (SNPs). 500 sequences are generated in each TCR sequence group.

We construct 50 immune repertoires in five groups. Each repertoire group are constructed using simulated TCR sequences from corresponding TCR sequence group. Within each group, each sequence set contains 2–10 sequences with randomly assigned weights that sum to 100. 45 string set pairs are generated within each group.

4.5.4.2 Heterogeneous sets of Hepatitis B Virus genomes

We obtain 30 HBV genomes from each of three host species — human, bat, duck — from the NCBI virus database [61]. We construct 3 string sets for each host species. For each string set, we randomly select 5 HBV genomes from one host and randomly assign a weight to each string so that the sum of string weights in each set is equal to 100.

We construct a partial order MSA graph on each string set [78]. We first conduct multiple sequence alignment (MSA) for each string set using Clustal Omega [143]. Then for each column of the MSA, we create a node for each unique character and add an edge between two nodes if the characters in node labels are adjacent in the input strings at that column. For each consecutive stretch of gap characters, no nodes are created, but an edge is added between flanking columns of the stretch of gaps. We also create a source node and a sink node that are connected to nodes representing the first and last characters of the input strings. The MSA graphs created in this process are all acyclic. We compute FGTED on MSA graphs by adding sink-to-source edges.

We also construct a de Bruijn graph [123] with k-mer size equal to 4 on TCR sequence sets, which we refer to as dBG4 in the following sections. This k-mer size is reasonable as compared to the average lengths of TCR sequences which is 350 base pairs and allows us to experiment with graphs that are expected to have higher expressiveness. In dBG4, each node corresponds to a k-mer, $S[i : i + k]$, where S is a string from the ground truth string set, $\mathcal{S}$. Each edge corresponds to the overlap between two k-mers, $S[i : i + k]$ and $S[i + 1 : i + k + 1]$ for any $S \in \mathcal{S}$. In order to construct the alignment graph, we process the de Bruijn graphs such that each node represents one character. We add a source node and a sink node to each dBG4 and connect them to nodes that represent the first and the last character in each string, respectively.

4.5.5 Estimation of FGTED and GTED using LP relaxations of CCTED

As noted in Chapter 3, GTED can be solved using the ILP formulations (compact ILP) and (exponential ILP). However, it is impossible to compute GTED exactly within a reasonable time on graphs that represent strings of lengths of more than 100 characters. On the other hand, while the value of CCTED between two graphs can be much smaller than GTED, when the solution of CCTED has only one component, CCTED is equal to GTED.

In the following sections, we estimate FGTED and GTED by the LP relaxations of the ILP in (3.5)-(3.8) that computes a lower bound on CCTED. We validate that the solutions to the LP relaxation of CCTED in all of the input graph pairs have one connected component.

4.5.6 FGTED Deviates from EMED as the Expressiveness of the Genome Graph Increases

We compute EMED and FGTED on string set pairs and genome graph pairs. The alignment graphs are constructed using one thread, which on average takes 6 seconds for dBG4s, 8 seconds for each MSA graph on TCR sequences, 9.43 minutes for each MSA graph on HBV genomes. Optimization for LP with 10 threads takes on average 601 seconds for each dBG4, 1 hour for each MSA graph of TCR sequence sets and 4 hours for each MSA graph on HBV genomes (Figure S1).

We show that the deviation of FGTED from EMED is higher on genome graphs that are more expressive. We compare the FGTED computed on dBG4s and MSA graphs constructed with TCR sequences and the diameters of two types of graphs. DBG4 represents all sequences with the same 5-mer distributions as the ground truth sequences. Therefore, as expected, we observe larger sampled SUDs from dBG4 than the MSA graphs (Figure 4.4(a)). The deviation of FGTED from EMED is also larger with dBG4s than the MSA graphs (Figure 4.4(b)). This further illustrates the effect of graph construction approaches on the resulting expressiveness.

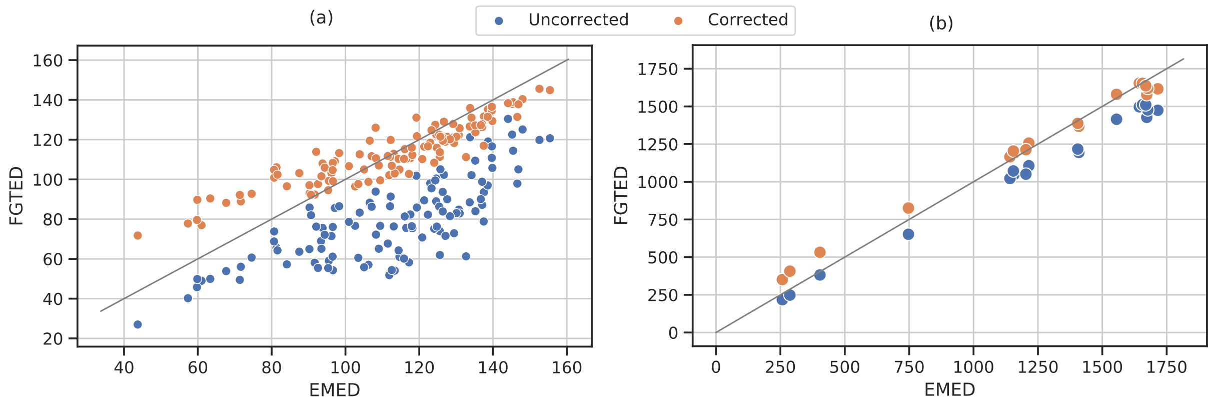


Figure 4.4: Comparison between de Bruijn graphs and MSA graphs constructed with TCR sequence sets. (a) The distribution of diameters sampled in both types of graphs. Each box shows the quartiles of the distribution, and the whiskers show the rest of the distribution. Each black dot represent the diameter of one graph. (b) The correlation between FGTED and EMED with different types of graphs. The red line denotes equality between FGTED and EMED.

4.5.7 Corrected FGTED More Accurately Estimates Distance Between Unseen String Sets Encoded With Genome Graphs

For each pair of string sets, we obtain the deviation of FGTED from EMED and sum of estimated SUDs. We fit three linear regression models, LR_{dBG4}, LR_{TCR} and LR_{HBV}, to predict deviation from sum of SUDs on simulated TCR sequences and HBV genomes of different types of graphs separately.

We evaluate the corrected and uncorrected FGTED by performing Pearson correlation experiments. We fit LR models on half of the data and compute the corrected FGTED on the other half as the test set. We evaluate the correlation between corrected and uncorrected FGTED and EMED on the test set. Two-tail P-values are calculated for each correlation experiment to test for non-correlation.

The LR models are evaluated with 10-fold cross validation. We randomly permute and split data into 10 equal parts. In each of the 10 iterations, we use one part as the test set and the rest as the training set. An average deviation is calculated across all iterations.

FGTED	Pearson Correlation		
	TCR (dBG4)	TCR (MSA Graph)	HBV (MSA Graph)
Uncorrected	**0.75**	0.74	**0.99**
Corrected	0.68	**0.90**	**0.99**

Table 4.2: Pearson correlation between EMED and corrected and uncorrected FGTED on simulated TCR and HBV sequences. Pearson correlation is calculated on a held-out set of data for both simulated TCR and HBV that consist of 50% of data, and LR model is fit on the other half.

In Table 4.2 and Table 4.3, we show that using string set universe diameters, we are able to improve the correlation between FGTED and EMED on MSA graphs of both the simulated TCR sequences and HBV genomes. On dBG4s, the correlation is reduced slightly by the correction. All Pearson correlation experiments are statistically significant with P-values < 0.01. On HBV genomes, since the correlation between uncorrected FGTED and EMED is approaching 1, no significant improvement is observed. On the other hand, significant reduction in average deviation is observed on both types of data. We are able to reduce the average deviation from 77.29 to 19.08 on de Bruijn Graphs with TCR sequences, from 32.74 to 9.13 on MSA graphs containing simulated TCR sequences and from 140.12 to 54.87 on HBV genomes.

FGTED	Average Deviation		
	TCR (dBG4)	TCR (MSA Graph)	HBV (MSA Graph)
Uncorrected	77.29	32.74	140.12
Corrected	**19.08**	**9.13**	**54.87**

Table 4.3: Average deviation of corrected and uncorrected FGTED from EMED on simulated TCR and HBV sequences. The average deviation is calculated over a 10-fold cross-validation of the LR model.

One caveat of using SUDs for correcting distances between genome graphs is that this correction is not guaranteed to always improve the distance. Given two string sets, there is usually an adversarial worst case where adjusting the distance using this approach reduces the accuracy in estimating string sets distances. When EMED between true string sets are small, the corrected FGTED may overestimate the EMED and result in a larger deviation. Nevertheless, we show that corrected FGTED reduces the anticipated deviation from EMED.

4.5.8 The scalability of FGTED

We compare the running time in real time of computing FGTED between graphs constructed with TCR sequences and HBV genomes (Figure 4.5). We ran all our experiments on a server with 24 cores (48 threads) of two Intel Xeon E5 2690 471 v3 @ 2.60GHz and 377 GB of memory. The system was running Ubuntu 18.04 with Linux kernel 472 4.15.0.

4.6. Discussion

A genome graph's string set universe diameter (SUD) provides information on the size and diversity of the represented string sets. We show that we can use SUDs to practically characterize the discrepancy between FGTED and EMED and to obtain a more accurate distance between unseen string sets encoded in genome graphs on average. While the results are obtained on short genomic sequences due to the high computational cost of FGTED and GTED, this result is encouraging.

The corrected FGTED can be used to compute a more accurate distance between heterogeneous samples represented by genome graphs in applications such as immune repertoire analysis and cancer subtyping. This opens up avenues for more comprehensive heterogeneous sam-

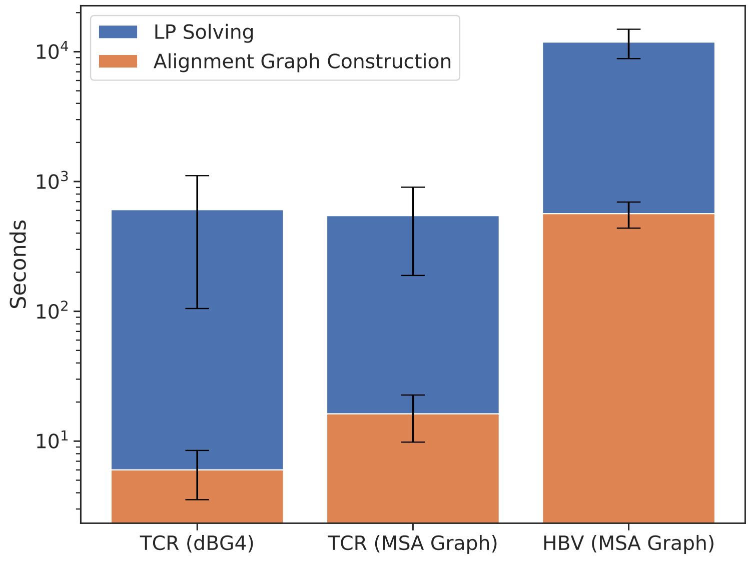


Figure 4.5: The time taken in the two steps to compute FGTED with dBG4s and MSA graphs. The orange part shows the running time to construct the alignment graph between input genome graphs. The blue part shows the running time to construct and solve the LP formulations. The length of each bar is the average running time for each graph. The error bars represent the standard deviations. Y-axis is in log-scale.

ple comparison methods. However, FGTED, as well as GTED, is not scalable to mammalian genomes due to the quadratic size of the alignment graph and time it takes to solve the LP formulations. Algorithms that compute FGTED faster or efficient approximation genome graph comparison methods [101, 102, 124] are needed for comparing large heterogeneous string sets.

SUDs may also be used to characterize the diversity of strings represented by reference genome graphs that are used in sequence-to-graph alignment [132, 147]. In sequence-to-graph alignment, it is often desired that a more diverse set of strings than the original reference string set is represented by the graph. Here, SUDs could be used as a measure to control the right amount of variation in the string set universe of created genome graphs.

Another future direction is to use expressiveness as a regularization term in the objective function to construct better genome graphs. To ensure efficiency of genome graphs in storing sequences, we can construct genome graphs that minimize their sizes [117, 126]. However, reducing the size of a genome graph may result in graphs that are highly expressive, and the distance between these genome graphs will deviate further from distances between true string sets. Adding a SUD term to the objective may address this problem.

Chapter 5

Discussion and Conclusion

5.1. Summary of contributions

In this dissertation, we establish algorithmic foundations for constructing and comparing genome graphs by addressing the challenges posed by the discrepancies between the strings in their plain, linear representation and the graph structures. We contribute to both the theoretical and practical aspects.

In Chapter 2, we develop an algorithmic framework that leverages decades of work in string compression to the graph representation of pangenomes. This framework directly addresses the problem of minimizing the size of the genome graph which eventually leads to a type of genome graphs that are space efficient. On the theoretical side, draw the connection between the elements in an EPM-compressed form and a genome graph and show that the proposed framework constructs a small genome graph as long as it is given a small compressed string. We also identify that, while the choice of the sources in string compression problems does not affect the size of the compressed form, it affects the size of the constructed genome graph. We address this discrepancy by introducing and solving the source assignment problem. On the practical side, we implement a proof-of-concept framework that constructs a genome graph using the relative Lempel-Ziv algorithm, and we achieve a significant improvement in both the speed and space efficiency in the constructed genome graphs.

In Chapter 3, we address the property of time efficiency of genome graph comparison by revisiting the problem of genome graph traversal edit distance and the previously proposed algorithms for it. We first point out the conflict between the previous result on the complexity of GTED and other sequence-to-graph comparison problems and prove that GTED is in fact NP-complete. We then show that the previously proposed algorithm for GTED does not solve GTED but solves for a lower bound of GTED and a variant of GTED, the Closed-trail Cover Traversal Edit Distance problem. Further, we characterize the CCTED problem and show that CCTED between two genome graphs is equal to GTED when the solution to the ILP of CCTED does not find one strongly connected component. This result set allows us to check if the estimated GTED is accurate when we use CCTED to estimate GTED. We also characterize the ILP for CCTED and point out that it cannot be solved in polynomial time as opposed to the claim in Boroojeny et al. [16]. The reason that the previous GTED ILP formulations fail to model GTED correctly is that they allow strictly disjoint strongly connected components in the alignment graph. Therefore, we propose two corrected ILP formulations, that eliminate disjoint SCCs in two ways. The first ILP has an exponential number of constraints for each strongly connected subgraph, which can be added iteratively. The second ILP, compact ILP, has a polynomial number of constraints that enforces a partial ordering on all selected edges. We evaluate the efficiency of both proposed ILPs and show that the exponential ILP is faster to solve than the compact ILP in when CCTED equal to GTED. When GTED is greater than CCTED, the compact ILP is faster than the exponential ILP. This is because the iterative process of adding constraints would align pairs of all possible strongly connected subgraphs that result in the current CCTED value before moving to the next value in the alignment graph, which could easily be exponential. The ILP formulation for CCTED is the fastest to solve. However, our empirical results, combined with the previous result in Boroojeny et al. [16] suggest that none of the ILPs for solving the traversal edit distances are practical to be applied to pangenomic studies due to their slow speed.

In Chapter 4, we address the property of expressiveness of the genome graphs and study its effect on genome graph comparisons. We propose a distance metric between two collections of genomes, the Earth Mover's Edit Distance, that measures the global similarities between strings that takes into account both the composition and the string content. Based on EMED, we propose a distance metric for pangenome comparison by extending the GTED metric to flow-GTED, which is the minimum Earth Mover's Edit Distance between sets of strings reconstructed from flow decompositions of the input genome graphs. We point out that genome graphs constructed using one set of strings are more expressive than the string representation and can usually represent a larger collection of sets of strings. When we do not know the complete true strings due to limitations in sequencing technology and compare genome graphs constructed from sequencing reads (e.g. assembly graphs), we may get a metric that is far from the true distance between the genomes we compare. We show that GTED and FGTED always underestimate the distance between sets of strings encoded in the genome graph. To address the negative side effect of the expressiveness of the genome graph, we formally characterize the expressiveness and use it as a correction factor to offset expressiveness by sampling Eulerian tours or flow decompositions in the input genome graphs. To evaluate the improvement of graph comparison corrected by expressiveness, we first compute an estimate of FGTED using the adapted CCTED and correct the estimated distance using expressiveness on simulated TCR repertoires and Hepatitis B Virus genomes. We show that by using expressiveness as a correction factor, we can obtain a better estimation of the edit distance between input pan-genomes.

5.2. Limitations

In Chapter 2, we explored the connection between EPM-compression algorithms and genome graphs, but our analysis mainly focuses on relative Lempel-Ziv algorithm in empirical experiments. While the Lempel-Ziv algorithms [169] do not belong to EPM-compression scheme, they are used widely in genomics to compress indexing structures. The connection between genome graphs and other compression schemes is to be explored. Additionally, the complexity of the source assignment problem and whether this problem can be solved by an empirically efficient algorithm are to be determined.

In Chapter 3, the proposed ILPs to solve GTED are not scalable even to small, prokaryotic genomes. The proposed ILPs are only evaluated on very small genome graphs and on limited cases. For example, differences in the running time of different ILPs are not evaluated on larger genomes graphs (with > 150 edge).

In Chapter 4, we estimate the expressiveness by sampling from the genome graph using random walks. However, it is unknown if the sampling is biased toward a set of strings that may be selected with higher probability due to the graph structure. Additionally, GTED is only one formulation of comparing graphs. The robustness of string universe diameter as a correction factor could be further confirmed by using it to correct other graph comparison or pangenome comparison methods. Further, the corrected FGTED is only evaluated on a few sets of HBV genomes.

5.3. Future directions

We propose future directions that build upon the research presented in this dissertation and aim to improve the practicality of implemented algorithms and developed theories on the essential properties of the genome graphs.

5.3.1 Practical auxiliary indexing structures of RLZ-graph

In Chapter 2, the RLZ-graph data structure could be more practical if it had accompanied sequence-to-graph alignment algorithms and a coordinate system. Both of these functions could be adapted from literature on RLZ-compressed strings to RLZ-graph.

A genome graph defines the space of the pan-genomic analysis and thus should support the coordinate query that returns the location and the identifier of the sample genome given a path or a node in the genome graph [27]. The coordinate query is specified by the following problem.

Problem 7 (Coordinate system query)**.** *Given a genome graph $G = (V, E, \ell)$ that encodes a set of genomic strings S and a query node identifier $n \in V$, find the set of identifier-location pairs $L = \{(i, j)\}$, where for each $(i, j) \in L$, $\ell(n) = S[i][j : j + len(\ell(n))]$. Here, $S[i]$ is string i in string set S and $S[i][j : j + len(\ell(n))]$ is the substring starting at location j with length equal to the length of $\ell(n)$ in $S[i]$.*

Problem 7 has a counterpart in relative Lempel-Ziv compressed strings. Ferrada et al. [43] develop an indexing structure that enables random access in RLZ-compressed strings, which, given the location in the input string S, returns the character that corresponds to $S[i]$ without decompression through a bit-vector $\mathcal{B}_p$ that stores the start location of each phrase in the input string. The bit vector, $\mathcal{B}$, implemented in RLZ-graph stores the starting location of each node in the reference string. By combining $\mathcal{B}_p$, $\mathcal{B}$ and the sequence of pointers in the compressed form, we have a set of operations that can transform a location in the input string to a node in the graph (Figure 5.1). This correspondence is the reverse operation to the one described in Problem 7. To answer the coordinate system query, we can build a dictionary that stores the correspondence between pointers and the reference string. Given a location i in the reference string R, the dictionary returns a set of phrases that represents substring at $R[i]$. By using the select operations in $\mathcal{B}$ and $\mathcal{B}_p$, we can get from a node to the phrases and the location in the input

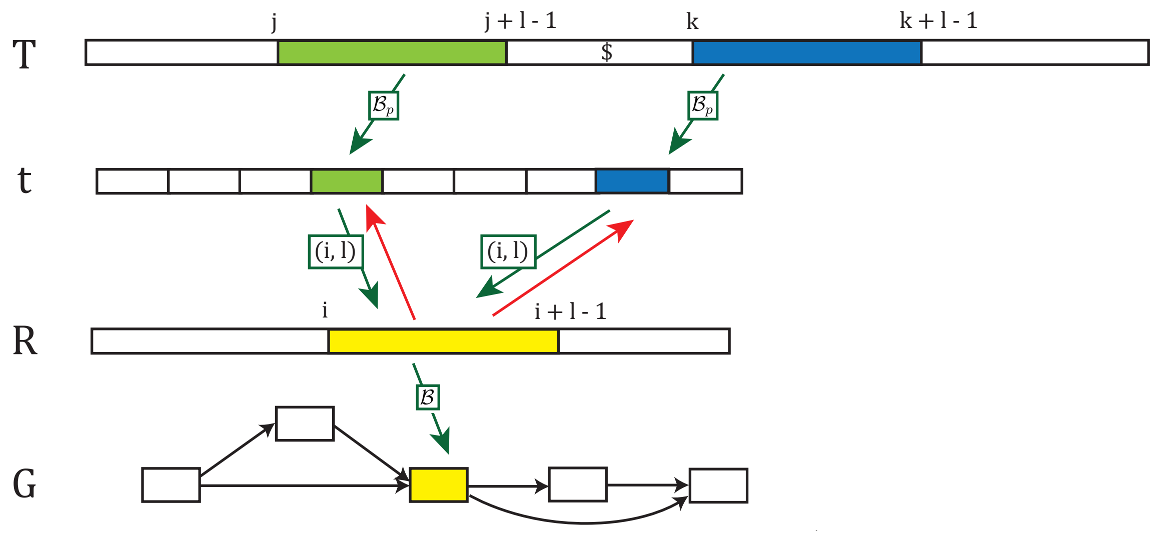


Figure 5.1: Auxilary structure for building a coordinate system on RLZ-graphs. T is the input string, t is the sequence of pointers/phrases from RLZ factorization, R is the reference string, and G is the RLZ-graph. Green arrows represents constant-time queries to obtain pointers, substrings from the reference string, and nodes using bit vectors $\mathcal{B}_p$, pointers, and $\mathcal{B}$, respectively. Red arrows represent the operation that returns a set of pointers given a substring on the reference string.

strings (Figure 5.1). Since in RLZ-graph, the input strings are concatenated into one string with the special character “$” as the boundary between strings, we can use another bit-vector to store the location of the “$” characters to get the sample identifier.

The auxiliary data structure developed for the coordinate system query can also be used to answer the read mapping query, which asks where in the genome graph and corresponding sample strings a query string is mapped. The read mapping query is defined in the following problem.

Problem 8 (Read mapping query)**.** *Given a genome graph $G = (V, E, \ell)$ that encodes a set of strings S, a query string q and a threshold t, find the set of identifier-location-length pairs $L = \{(i, j, k)\}$ such that for each $(i, j) \in L$,*

$$\mathit{edit}(\mathit{str}(q), \mathit{str}(S[i][j : j + k - 1])) \leq t.$$

Problem 8 is a generalization of sequence-to-pangenome mapping that could be answered by first aligning a query string to a genome graph and using the coordinate system of the genome graph to convert the alignment back to the coordinate in the set of genomes. While existing read mapping indexing structures [71, 87, 132, 145, 146, 166] could be applied to RLZ-graph directly, it might be more space- and time-efficient to adapt existing indexing structures designed for sequence query [30, 40, 46] in RLZ-compressed strings to the RLZ-graph.

By adapting existing algorithmic innovations for RLZ-compressed strings, the RLZ-graph could be more practical and scalable to pan-genomic analysis on thousands of eukaryotic genomes.

The compression-to-graph framework can be further improved to reduce the graph size by hierarchical compression. Hierarchical compression was applied to string compression and shows significant improvement in compression ratio without compromising the speed of alignment significantly [13, 114]. Additionally, a hierarchical genome graph representation of the genome graph can be used to encode the phylogeny relationship between the genome graphs [27].

5.3.2 Faster pan-genome comparisons

In chapter 3, unfortunately, we show that the family of traversal edit distance problems are either NP-hard (GTED) or at least very difficult to solve efficiently on larger genomes via integer linear programming (both GTED and CCTED). This result, however, is in concordance with complexity results on many sequence-to-graph matching problems. Sequence-to-graph matching has been sped up by the seed-and-extend techniques. A future direction of work is to speed up the traversal distance computation by reducing it to sequence-to-graph traversal distance computation and applying the seed-and-extend techniques.

In traversal edit distance computation, by solving an ILP based on an alignment graph, two sets of strings are found directly from the string universes of two input graphs, which requires exploring a huge feasible space. One way to reduce the feasible space is to shift to an iterative scheme and find sets of strings that are closer in each iteration.

An iterative framework. Given a pair of genome graphs, G_1 and G_2, we first sample a set of strings S_1 from $SU(G_1)$. We then find $S_2 \in SU(G_2)$ such that emedit(S_1, S_2) is minimized. Then, find $S_1' \in SU(G_1)$ such that emedit(S_1', S_2) is minimized. Repeat the process of finding the closest string set to the previously found string set until the Earth Mover's Edit Distance has converged. This process is described by a pseudo-code in Algorithm 2. It is open whether the proposed framework yields a solution that approximates FGTED within a factor for general genome graphs or for certain types of genome graphs. Although the sequence-to-graph matching under traversal edit distance is also NP-complete as shown in Chapter 3, reducing graph-to-graph matching to sequence-to-graph matching reduces the size of the problem and therefore may speed up traversal edit distance computation.

Algorithm 2: An iterative framework to estimate FGTED

```
1: Input Two genome graphs $G_1$ and $G_2$.
2: $S_1 \leftarrow \text{sample}(G_1)$
3: $\text{FGTED}' \leftarrow \text{inf}$
4: while true do
5:     $S_2 = \text{argmin}_{S \in SU(G_2)} \text{emedit}(S_1, S)$
6:     if $\text{edit}(S_1, S_2) \geq \text{FGTED}'$ then return the converged estimate $\text{FGTED}'$
7:     else
8:         $\text{FGTED}' = \text{emedit}(S_1, S_2)$
9:         $S_1 = \text{argmin}_{S \in SU(G_1)} \text{emedit}(S, S_2)$
10:    end if
11: end while
```

Heuristics to estimate traversal edit distance between a string set and a genome graph. Since sequence-to-graph matching is NP-complete with the Eulerian constraint, we propose a heuristic to estimate the traversal edit distance between a string and a genome graph, which is defined by $\min_{S \in SU(G)}$ emedit(S_1, S) given a string set S_1 and a genome graph G.

An Eulerian tour in a graph can be defined by a set of pairings between edges at each node with more than one unique out-neighbors [153]. A node, v, is an out-neighbor of u if edge (u, v) is in the graph. Similarly, a node w is an in-neighbor of u if edge (w, u) is in the graph. For a node with in-neighbors set I and out-neighbors set O, an edge matching is $M \in I \times O$ such that each edge in I is matched once with one edge in O. A pair of matched edges, (e_1, e_2), defines a traversal ordering where e_2 is traversed immediately after e_1 in an Eulerian trail. Such a matching always exists if the graph is an Eulerian graph. A set of edge matchings for all nodes is called valid if the traversal orders defined by the matching results in an Eulerian tour.

The set of valid edge matchings defines an Eulerian tour. Given a string s, we can estimate the traversal edit distance between s and G by finding the minimum-cost valid edge matching with a cost function based on the local similarity between s and each in-out edge pairs in G.

Problem 9 (Min-cost edge matching problem)**.** *Given an edge-labeled Eulerian graph $G = (V, E, \ell)$ and a string s, find a set of valid edge matching $\mathcal{M}$ such that the total cost:*

$$cost(\mathcal{M}, s) = \sum_{M_v \in \mathcal{M}} \sum_{(e_i, e_j) \in M_v} edit_l(\ell(e_i) \cdot \ell(e_j), s),$$

where M_v is the min-cost matching for edges adjacent to v, is minimized.

Here, $\ell(e_i) \cdot \ell(e_j)$ is the concatenated labels on e_i and e_j, edit$_l$ is the minimum cost to align the string $\ell(e_i) \cdot \ell(e_j)$ to a substring of s.

The min-cost edge matching problem is a variant of the minimum reload cost Eulerian tour problem [8] where the input graph is an Eulerian graph with no edge labels, and the matching cost is predetermined. The minimum reload cost Eulerian tour problem is proven to be NP-complete.

An approximation heuristics to the min-cost edge matching problem is by finding the min-cost s-t trails iteratively in the line graph of the input graph. The line graph $L(G)$ of a graph G is constructed by creating a node for each edge in G and connecting two nodes if the corresponding edges are adjacent. Starting with an input Eulerian graph G and a string s, we first compute the edit$_l$ for all pairs of adjacent edges. Obtain $L(G)$ where each node v_e corresponds to edge e in G. The cost of each edge (v_{e_1}, v_{e_2}) in $L(G)$ is equal to the cost of matching e_1 and e_2. Repeat the following procedure iteratively until the remaining graph is empty. We choose a pair of source and sink nodes, s and t. We find a s-t trail with minimum cost and remove it from $L(G)$. If t is still reachable from s, repeat the process of finding a min-cost s-t trail and removing it from $L(G)$. If t is no longer reachable from s, choose a new pair of s and t and repeat the process of finding and removing a min-cost s-t trail. When $L(G)$ is empty, collapse all selected shortest paths into edges and obtain a new graph $L(G)'$. Repeat the process described above until the final graph contains only one edge, which has the edge label that spells a string constructed from an Eulerian trail in the original graph G.

The running time of finding a min-cost trail between two nodes under reload cost, or pair-edge cost is polynomial [8], and therefore the approximation heuristics described above is polynomial.

It is open whether such a heuristics could find a lower bound on the solution to Problem 9 within a constant or a polynomial factor of the optimal distance between a string and a graph.

5.3.3 Integrating expressiveness evaluation and pan-genome construction

As mentioned in Chapter 1, expressiveness is a desired feature of a pan-genome representation as it allows the representation to encode a larger population of genomes that enables flexibility and sensitivity. However, too much expressiveness hinders the accuracy of pan-genomic operations. Therefore, a pan-genome construction method that controls not only the size but also expressiveness is beneficial for more accurate representations of a pangenome. We describe the Genome graph size and expressiveness optimization problem as follows:

Problem 10 (Genome graph size and expressiveness optimization problem)**.** *Given a set of strings S and a threshold t, construct a genome graph G such that*

$$\begin{aligned} G = \arg\min_{G} \quad & \textit{size}(G) + \alpha \cdot |\textit{express}(G) - t| \\ \textit{subject to} \quad & S \in SU(G). \end{aligned}$$

Here, size(G) is the combined space taken by storing nodes, edges and node labels. express(G) is a quantity representing the level of expressiveness of G. α is a scaling factor. $SU(G)$ is the string universe of G, which is equal to the collection of all sets of strings G encodes, which is defined in Chapter 4 Section 4.4.3. In Chapter 4, express(G) is defined as the maximum distance between string sets encoded by a genome graph, which is computed by sampling.

Alternative definitions of expressiveness could be used to reduce the false positive rate of aligning a read to a panel of reference genomes in a population. A read mapped to the genome graph is a false positive mapping if this read comes from a genome that does not belong to the genome population encoded by the genome graph. For example, aligning a bacterial genomic segment to a human genome is a false positive alignment. This type of false positive alignment error could be reduced by restricting the deviation of strings encoded in the genome graph from the genomes that are used to construct the genome graph.

Given a set of founding strings S and a genome graph G constructed based on S, the expressiveness of G can be defined as $\max_{S' \in SU(G)}$ emedit(S', S). The use of the founding strings and comparing them with the "peripheral" strings in the genome graph is analogous to the concept of core and accessory genomes in metagenomics, where the core genome is the set of gene sequences shared among the community of microbes and the accessory genome is the set of genes unique to some subsets of microbes. By restricting the expressiveness based on the founding strings, we restrict the variability of strings encoded by the genome graph, and only store strings that are closely related to the founding strings in the genome graph.

The challenges of solving the genome graph size and expressiveness optimization problem is the complexity to quantify the expressiveness of a genome graph and the complexity to solve the problem. The definition of expressiveness could change based on the application of the genome graph and it is interesting to properly define and compute the expressiveness of genome graphs. Similar to the complexity of Problem 1 in Chapter 2, the NP-completeness of Problem 10 is open.

5.4. Conclusion

In this dissertation, we draw connections between pangenome representations and well-studied computational problems in strings and graph theory, such as string compression and flow decomposition. We propose algorithmic frameworks using these connections and address the foundational properties of genome graphs that are essential to efficient and accurate pangenomic analyses but are not well characterized by existing research. The compression-to-graph framework makes pangenome representation more scalable and allows more reference genomes to be incorporated in a pangenome reference, which increases the sensitivity of read mapping. We explore the connections between genome graph comparison problems and optimal traversal finding problems, which enable us to provide corrected complexity analysis and new algorithms for comparing genome graphs. We use the connection between flow decomposition and genome graphs to adapt genome graph comparisons from graphs that contain single genomes to graphs that contain multiple genomes and enable comparison between mixtures of genomes that are more commonly seen in biomedical applications. By characterizing genome graph expressiveness, we contribute to a genome graph comparison framework that is more robust to unwanted variations among genomes stored in the graph. This work will enable faster, more accurate, and more interpretable pan-genomic analyses in an era where joint use of large collections of sequencing data is becoming essential.

Bibliography

[1] Human Pangenome Reference Consortium, 2023. URL https://humanpangenome.org/. 1.1

[2] 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015. 1.1, 2.1, 2.8.1

[3] Ravindra K Ahujia, Thomas L Magnanti, and James B Orlin. Network flows: Theory, algorithms and applications. *New Jersey: Prentice-Hall*, 1993. 3.3.4, 4.2.2

[4] Jarno Alanko and Tuukka Norri. Greedy shortest common superstring approximation in compact space. In *International Symposium on String Processing and Information Retrieval*, pages 1–13. Springer, 2017. 2.4.2

[5] Fatemeh Almodaresi, Prashant Pandey, and Rob Patro. Rainbowfish: a succinct colored de Bruijn graph representation. In *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. 1.2.2, 2.1, 2.1

[6] Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, and Rob Patro. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*, 34(13):i169–i177, 2018. 4.1

[7] Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. An efficient, scalable, and exact representation of high-dimensional color information enabled using de Bruijn graph search. *Journal of Computational Biology*, 27(4):485–499, 2020. 2.1, 2.1

[8] Edoardo Amaldi, Giulia Galbiati, and Francesco Maffioli. On minimum reload cost paths, tours, and flows. *Networks*, 57(3):254–260, 2011. 5.3.2

[9] Joel Armstrong, Ian T Fiddes, Mark Diekhans, and Benedict Paten. Whole-genome alignment and comparative annotation. *Annual Review of Animal Biosciences*, 7:41–64, 2019. 1.3, 1.3.1

[10] S. Ballouz, A. Dobin, and J. A. Gillis. Is it time to change the reference genome? *Genome Biology*, 20(1):159, 2019. 1.2, 2.1

[11] Philipp E Bayer, Agnieszka A Golicz, Armin Scheben, Jacqueline Batley, and David Edwards. Plant pan-genomes are the new reference. *Nature Plants*, 6(8):914–920, 2020. 1.1

[12] Giulia Bernardini, Nadia Pisanti, Solon P Pissis, and Giovanna Rosone. Approximate

pattern matching on elastic-degenerate text. *Theoretical Computer Science*, 812:109–122, 2020. 1.2.1

[13] Philip Bille, Inge Li Gørtz, Simon J Puglisi, and Simon R Tarnow. Hierarchical relative lempel-ziv compression. *arXiv preprint arXiv:2208.11371*, 2022. 1.2.1, 5.3.1

[14] Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. *Journal of the ACM (JACM)*, 41(4):630–647, 1994. 2.4.2

[15] Christopher R Bolen, Florian Rubelt, Jason A Vander Heiden, and Mark M Davis. The repertoire dissimilarity index as a method to compare lymphocyte receptor repertoires. *BMC Bioinformatics*, 18(1):1–8, 2017. 4.1

[16] Ali Ebrahimpour Boroojeny, Akash Shrestha, Ali Sharifi-Zarchi, Suzanne Renick Gallagher, S Cenk Sahinalp, and Hamidreza Chitsaz. Gted: Graph traversal edit distance. pages 37–53, 2018. (document), 1.3.3, 1.5, 3.1, 3, 3.1, 3.2.1, 3.3, 3.3.2, 5, 3.3.2, 3.3.3, 3.3.3, 3.3.4, 3.5.2, 3.6, 3.6.1.1, 3.8, 4.1, 4.2.5, 6, 5.1

[17] Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big bwts. *Algorithms for Molecular Biology*, 14(1):1–15, 2019. 1.2.1

[18] Guillaume Bourque, Pavel A Pevzner, and Glenn Tesler. Reconstructing the genomic architecture of ancestral mammals: lessons from human, mouse, and rat genomes. *Genome Research*, 14(4):507–516, 2004. 3.5.1

[19] Stephen P Bradley, Arnoldo C Hax, and Thomas L Magnanti. *Applied mathematical programming*. Addison-Wesley, 1977. 3.4.2

[20] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997. 1.3.1

[21] Michael Burrows. A block-sorting lossless data compression algorithm. *SRC Research Report, 124*, 1994. 1.2.1

[22] Nae-Chyun Chen, Brad Solomon, Taher Mun, Sheila Iyer, and Ben Langmead. Reducing reference bias using multiple population reference genomes. *BioRxiv*, 2020. 2.1

[23] Nae-Chyun Chen, Brad Solomon, Taher Mun, Sheila Iyer, and Ben Langmead. Reference flow: reducing reference bias using multiple population genomes. *Genome Biology*, 22 (1):1–17, 2021. 1.2.1

[24] Ningbo Chen, Weiwei Fu, Jianbang Zhao, Jiafei Shen, Qiuming Chen, Zhuqing Zheng, Hong Chen, Tad S Sonstegard, Chuzhao Lei, and Yu Jiang. Bgvd: an integrated database for bovine sequencing variations and selective signatures. *Genomics, Proteomics & Bioinformatics*, 18(2):186–193, 2020. 1.1

[25] Aleksander Cisłak, Szymon Grabowski, and Jan Holub. Sopang: online text searching over a pan-genome. *Bioinformatics*, 34(24):4290–4292, 2018. 1.2.1

[26] Karen Clark, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. GenBank. *Nucleic Acids Research*, 44(D1):D67–D72, 2016. 2.1, 2.9

[27] Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2018. 1.3, 2.1, 5.3.1, 5.3.1

[28] Human Microbiome Jumpstart Reference Strains Consortium, Karen E Nelson, George M Weinstock, Sarah K Highlander, Kim C Worley, Heather Huot Creasy, Jennifer Russo Wortman, Douglas B Rusch, Makedonka Mitreva, Erica Sodergren, et al. A catalog of reference genomes from the human microbiome. *Science*, 328(5981):994–999, 2010. 1.1, 1.3

[29] Isidro Cortés-Ciriano, Jake June-Koo Lee, Ruibin Xi, Dhawal Jain, Youngsook L Jung, Lixing Yang, Dmitry Gordenin, Leszek J Klimczak, Cheng-Zhong Zhang, David S Pellman, et al. Comprehensive analysis of chromothripsis in 2,658 human cancers using whole-genome sequencing. *Nature Genetics*, 52(3):331–341, 2020. 1.2.2

[30] Anthony J Cox, Andrea Farruggia, Travis Gagie, Simon J Puglisi, and Jouni Sirén. RLZAP: relative Lempel-Ziv with adaptive pointers. In *International Symposium on String Processing and Information Retrieval*, pages 1–14. Springer, 2016. 1.2.1, 5.3.1

[31] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954. 3.1

[32] Eva Darai-Ramqvist, Agneta Sandlund, Stefan Müller, George Klein, Stefan Imreh, and Maria Kost-Alimova. Segmental duplications and evolutionary plasticity at tumor chromosome break-prone regions. *Genome Research*, 18(3):370–379, 2008. 3.7.3

[33] Aaron E Darling, Bob Mau, and Nicole T Perna. progressivemauve: multiple genome alignment with gene gain, loss and rearrangement. *PloS one*, 5(6):e11147, 2010. 1.3.1

[34] Arthur L Delcher, Simon Kasif, Robert D Fleischmann, Jeremy Peterson, Owen White, and Steven L Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11): 2369–2376, 1999. 1.3.1

[35] Sebastian Deorowicz and Szymon Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21):2979–2986, 2011. 1.2.1, 2.1

[36] Sebastian Deorowicz, Agnieszka Danek, and Marcin Niemiec. GDC2: Compression of large collections of genomes. *Scientific Reports*, 5:11565, 2015. 1.2.1, 2.1

[37] Tamal K Dey, Anil N Hirani, and Bala Krishnamoorthy. Optimal homologous cycles, total unimodularity, and linear programming. *SIAM Journal on Computing*, 40(4):1026–1044, 2011. 3.6.1.1, 3.6.1.1

[38] Fernando HC Dias, Lucia Williams, Brendan Mumey, and Alexandru I Tomescu. Minimum flow decomposition in graphs with cycles using integer linear programming. *arXiv preprint arXiv:2209.00042*, 2022. 3.1, 3.4, 3.4.1, 3.4.1, 3.4.1, 3.4.2

[39] Alexander Dilthey, Charles Cox, Zamin Iqbal, Matthew R Nelson, and Gil McVean. Improved genome inference in the MHC using a population reference graph. *Nature Genetics*, 47(6):682–688, 2015. 1.2.2, 2.1, 4.1

[40] Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative

Lempel–Ziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30, 2014. 2.10, 5.3.1

[41] Peter Ebert, Peter A Audano, Qihui Zhu, Bernardo Rodriguez-Martin, David Porubsky, Marc Jan Bonder, Arvis Sulovari, Jana Ebler, Weichen Zhou, Rebecca Serra Mari, et al. Haplotype-resolved diverse human genomes and integrated analysis of structural variation. *Science*, 372(6537), 2021. 2.1, 2.8.2

[42] Jana Ebler, Peter Ebert, Wayne E Clarke, Tobias Rausch, Peter A Audano, Torsten Houwaart, Yafei Mao, Jan O Korbel, Evan E Eichler, Michael C Zody, et al. Pangenome-based genome inference allows efficient and accurate genotyping across a wide spectrum of variant classes. *Nature Genetics*, 54(4):518–525, 2022. 1.2.2

[43] Héctor Ferrada, Travis Gagie, Simon Gog, and Simon J Puglisi. Relative Lempel-Ziv with constant-time random access. In *International Symposium on String Processing and Information Retrieval*, pages 13–17. Springer, 2014. 1.2.1, 2.1, 5.3.1

[44] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000. 1.2.1

[45] Lars Feuk, Andrew R Carson, and Stephen W Scherer. Structural variation in the human genome. *Nature Reviews Genetics*, 7(2):85–97, 2006. 1.2.2

[46] Travis Gagie, Paweł Gawrychowski, and Simon J Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *Algorithms and Computation: 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings 22*, pages 653–662. Springer, 2011. 5.3.1

[47] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J Puglisi. A faster grammar-based self-index. In *International Conference on Language and Automata Theory and Applications*, pages 240–251. Springer, 2012. 2.10

[48] Travis Gagie, Simon J Puglisi, and Daniel Valenzuela. Analyzing relative Lempel-Ziv reference construction. In *International Symposium on String Processing and Information Retrieval*, pages 160–165. Springer, 2016. 2.7.1

[49] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477. SIAM, 2018. 1.2.1

[50] Erik Garrison and Andrea Guarracino. Unbiased pangenome graphs. *Bioinformatics*, 39 (1):btac743, 2023. 1.2.2

[51] Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, Benedict Paten, and Richard Durbin. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36(9):875–879, 2018. 1.2.2, 2.1, 2.1, 2.8, 2.8.2, 4.1

[52] Erik Garrison, Andrea Guarracino, Simon Heumos, Flavia Villani, Zhigui Bao, Lorenzo Tattini, Jörg Hagmann, Sebastian Vorbrugg, Santiago Marco-Sola, Christian Kubica, et al.

Building pangenome graphs. *bioRxiv*, pages 2023–04, 2023. 1.2.2

[53] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. 2.7.2

[54] Catherine Grasso and Christopher Lee. Combining partial order alignment and progressive multiple sequence alignment increases alignment speed and scalability to very large alignment problems. *Bioinformatics*, 20(10):1546–1556, 2004. 1.3.3

[55] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016. 1.3.2

[56] Cristian Groza, Tony Kwan, Nicole Soranzo, Tomi Pastinen, and Guillaume Bourque. Personalized and graph genomes reveal missing signal in epigenomic data. *Genome biology*, 21(1):1–22, 2020. 1.2.2

[57] Andrea Guarracino, Moses Njagi Mwaniki, Santiago Marco-Sola, and Erik Garrison. Whole-chromosome pair-wise alignment using the hierarchical wavefront algorithm, 2021. URL https://github.com/ekg/wfmash. 1.3.1

[58] Andrea Guarracino, Silvia Buonaiuto, Leonardo Gomes de Lima, Tamara Potapova, Arang Rhie, Sergey Koren, Boris Rubinstein, Christian Fischer, Jennifer L Gerton, et al. Recombination between heterologous human acrocentric chromosomes. *Nature*, 617 (7960):335–343, 2023. 1.1

[59] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021. URL https://www.gurobi.com. 3.7.1

[60] Kevin Hadi, Xiaotong Yao, Julie M Behr, Aditya Deshpande, Charalampos Xanthopoulakis, Huasong Tian, Sarah Kudman, Joel Rosiene, Madison Darmofal, Joseph DeRose, et al. Distinct classes of complex structural variation uncovered across thousands of cancer genome graphs. *Cell*, 183(1):197–210, 2020. 1.2.2

[61] Eneida L Hatcher, Sergey A Zhdanov, Yiming Bao, Olga Blinkova, Eric P Nawrocki, Yuri Ostapchuck, Alejandro A Schäffer, and J Rodney Brister. Virus variation resource–improved response to emergent viral outbreaks. *Nucleic Acids Research*, 45(D1):D482–D490, 2017. 2, 4.5.4.2

[62] Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biology*, 21(1):249–269, 2020. 1.2.2, 2.1, 2.1, 2.8.1, 2.8.3, 4.1

[63] Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):i361–i370, 2013. 1.2.1

[64] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44 (2):226–232, 2012. 1.2.2, 2.1, 2.1, 2.8, 4.1

[65] Chirag Jain, Sergey Koren, Alexander Dilthey, Adam M Phillippy, and Srinivas Aluru. A fast adaptive algorithm for computing whole-genome homology maps. *Bioinformatics*, 34

(17):i748–i756, 2018. 1.3.1

[66] Chirag Jain, Haowen Zhang, Yu Gao, and Srinivas Aluru. On the complexity of sequence-to-graph alignment. *Journal of Computational Biology*, 27(4):640–654, 2020. 1.3.3, 2.1, 3.1, 3.3.1

[67] David S Johnson and Michael R Garey. *Computers and intractability: A guide to the theory of NP-completeness*. WH Freeman, 1979. 1.3

[68] Juha Kärkkäinen, Dominik Kempa, and Simon J Puglisi. Hybrid compression of bitvectors for the FM-index. In *2014 Data Compression Conference*, pages 302–311. IEEE, 2014. 2.4.1

[69] Jamshed Khan and Rob Patro. Cuttlefish: fast, parallel and low-memory compaction of de bruijn graphs from large-scale genome collections. *Bioinformatics*, 37(Supplement_1): i177–i186, 2021. 1.2.2

[70] Bryce Kille, Advait Balaji, Fritz J Sedlazeck, Michael Nute, and Todd J Treangen. Multiple genome alignment in the telomere-to-telomere assembly era. *Genome Biology*, 23(1): 182, 2022. 1.3.1

[71] Daehwan Kim, Joseph M Paggi, Chanhee Park, Christopher Bennett, and Steven L Salzberg. Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype. *Nature Biotechnology*, 37(8):907–915, 2019. 5.3.1

[72] Carl Kingsford, Michael C Schatz, and Mihai Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics*, 11(1):21, 2010. 4.1

[73] Orna Kupferman and Gal Vardi. Eulerian paths with regular constraints. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 62:1–62:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-016-3. 1.3.3, 1.4.1, 3.1, 3.2.1, 4, 3.2.1, 3.2.2, 3.2.2

[74] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In Edgar Chavez and Stefano Lonardi, editors, *String Processing and Information Retrieval*, pages 201–206, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 1.2.1, 2.1, 2.7.1, 2.7.1, 2.10

[75] Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Optimized relative Lempel-Ziv compression of genomes. In *Proceedings of the Thirty-Fourth Australasian Computer Science Conference-Volume 113*, pages 91–98. Australian Computer Society, Inc., 2011. 2.7.3, 2.9

[76] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In *International Conference on Machine Learning*, pages 957–966. PMLR, 2015. 4.1

[77] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3): 1–10, 2009. 1.2.1

[78] Christopher Lee, Catherine Grasso, and Mark F Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002. 4.5.4.2

[79] Heewook Lee and Carl Kingsford. Kourami: graph-guided assembly for novel human leukocyte antigen allele discovery. *Genome Biology*, 19(1):1–16, 2018. 4.1

[80] Marie-Paule Lefranc. IMGT, the international ImMunoGeneTics information system. *Cold Spring Harbor Protocols*, 2011(6):595–603, 2011. 3.7.2

[81] Marie-Paule Lefranc and Gérard Lefranc. *The immunoglobulin factsbook*. Academic Press, 2001. 1, 4.5.4.1

[82] Li Lei, Eugene Goltsman, David Goodstein, Guohong Albert Wu, Daniel S Rokhsar, and John P Vogel. Plant pan-genomics comes of age. *Annual Review of Plant Biology*, 72: 411–435, 2021. 1.1

[83] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710. Soviet Union, 1966. 4.2.1, 4.4.3.1

[84] Elizaveta Levina and Peter Bickel. The Earth Mover's distance is the mallows distance: Some insights from statistics. In *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, volume 2, pages 251–256. IEEE, 2001. 4.1

[85] Heng Li. A statistical framework for SNP calling, mutation discovery, association mapping and population genetical parameter estimation from sequencing data. *Bioinformatics*, 27(21):2987–2993, 2011. 2.8.1

[86] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 2016. 2.8.1

[87] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34 (18):3094–3100, 2018. 1.3.1, 5.3.1

[88] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009. 1.2.1

[89] Heng Li, Xiaowen Feng, and Chong Chu. The design and construction of reference pangenome graphs with minigraph. *Genome Biology*, 21(1):265–283, 2020. 1.2.2, 1.4.4, 2.1, 4.1

[90] Yilong Li, Nicola D Roberts, Jeremiah A Wala, Ofer Shapira, Steven E Schumacher, Kiran Kumar, Ekta Khurana, Sebastian Waszak, Jan O Korbel, James E Haber, et al. Patterns of somatic structural variation in human cancer genomes. *Nature*, 578(7793):112–121, 2020. 1.1, 3.7.3

[91] Wen-Wei Liao, Mobin Asri, Jana Ebler, Daniel Doerr, Marina Haukness, et al. A draft human pangenome reference. *Nature*, 617(7960):312–324, May 2023. ISSN 1476-4687. doi: 10.1038/s41586-023-05896-x. URL https://doi.org/10.1038/s41586-023-05896-x. 1.2.2

[92] Oksana Lukjancenko, Trudy M Wassenaar, and David W Ussery. Comparison of 61 sequenced Escherichia coli genomes. *Microbial Ecology*, 60:708–720, 2010. 1.1, 1.3

[93] Altti Ilari Maarala, Ossi Arasalo, Daniel Valenzuela, Veli Mäkinen, and Keijo Heljanko. Distributed hybrid-indexing of compressed pan-genomes for scalable and fast sequence alignment. *Plos One*, 16(8):e0255260, 2021. 1.2.1

[94] Veli Mäkinen, Bastien Cazaux, Massimo Equi, Tuukka Norri, and Alexandru I. Tomescu. Linear time construction of indexable founder block graphs. In *20th International Workshop on Algorithms in Bioinformatics (WABI 2020)*, volume 172 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 1.2.2, 2.1

[95] Serghei Mangul and David Koslicki. Reference-free comparison of microbial communities via de Bruijn graphs. In *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 68–77, 2016. 1.3.3, 3.1, 4.1

[96] Guillaume Marçais, Arthur L Delcher, Adam M Phillippy, Rachel Coston, Steven L Salzberg, and Aleksey Zimin. Mummer4: A fast and versatile genome alignment system. *PLoS Computational Biology*, 14(1):e1005944, 2018. 1.3.1

[97] Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics*, 37(4):456–463, 2021. 1.3.1, 1.4.1

[98] Duccio Medini, Claudio Donati, Hervé Tettelin, Vega Masignani, and Rino Rappuoli. The microbial pan-genome. *Current Opinion in Genetics & Development*, 15(6):589–594, 2005. 1.1

[99] Karen H Miga and Ting Wang. The need for a human pangenome reference sequence. *Annual Review of Genomics and Human Genetics*, 22:81–102, 2021. 1.2

[100] Clair E Miller, Albert W Tucker, and Richard A Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM (JACM)*, 7(4):326–329, 1960. 3.1, 3.4.2

[101] Ilia Minkin and Paul Medvedev. Scalable pairwise whole-genome homology mapping of long genomes with BubbZ. *iScience*, 23(6):101224, 2020. 1.3.3, 3.1, 4.6

[102] Ilia Minkin and Paul Medvedev. Scalable multiple whole-genome alignments and locally collinear block construction with SibeliaZ. *Nature communications*, 11(1):6327, 2020. 1.3.3, 3.1, 4.1, 4.6

[103] Ilia Minkin, Son Pham, and Paul Medvedev. TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics*, 33(24): 4024–4032, 2017. 1.2.2, 2.1, 2.1, 4.1

[104] Luc GT Morris, Nadeem Riaz, Alexis Desrichard, Yasin Şenbabaoğlu, A Ari Hakimi, Vladimir Makarov, Jorge S Reis-Filho, and Timothy A Chan. Pan-cancer analysis of intratumor heterogeneity as a prognostic determinant of survival. *Oncotarget*, 7(9):10051, 2016. 1.1, 1.3, 4.1

[105] Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. Succinct colored de

bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017. 1.2.2

[106] Martin D Muggli, Bahar Alipanahi, and Christina Boucher. Building large updatable colored de Bruijn graphs via merging. *Bioinformatics*, 35(14):i51–i60, 2019. 1.2.2, 2.1, 2.1

[107] Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching reads to many genomes with the r-index. *Journal of Computational Biology*, 27(4):514–518, 2020. 1.2.1

[108] Taher Mun, Naga Sai Vaddadi, and Ben Langmead. Pangenomic genotyping with the marker array. *Algorithms for Molecular Biology*, 18(1), 2023. doi: 10.1186/s13015-023-00225-3. 1.2.1

[109] James R Munkres. *Elements of algebraic topology*. CRC press, 2018. 3.3.3, 3.3.3

[110] Gonzalo Navarro and Víctor Sepúlveda. Practical indexing of repetitive collections using relative Lempel-Ziv. In *2019 Data Compression Conference (DCC)*, pages 201–210. IEEE, 2019. 2.10

[111] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. 1.3.1, 1.4.1, 3.1, 3.3.1

[112] Adam M Novak, Glenn Hickey, Erik Garrison, Sean Blum, Abram Connelly, Alexander Dilthey, Jordan Eizenga, MA Saleh Elmohamed, Sally Guthrie, André Kahles, et al. Genome graphs. *BioRxiv*, page 101378, 2017. 1.1, 2.1

[113] Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022. 1.2

[114] Marco Oliva, Travis Gagie, and Christina Boucher. Recursive prefix-free parsing for building big bwts. *bioRxiv*, 2023. 1.2.1, 5.3.1

[115] Branden J Olson, Stefan A Schattgen, Paul G Thomas, Philip Bradley, and Frederick A Matsen IV. Comparing t cell receptor repertoires using optimal transport. *PLOS Computational Biology*, 18(12):e1010681, 2022. 1.3

[116] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biology*, 17(1), June 2016. doi: 10.1186/s13059-016-0997-x. URL https://doi.org/10.1186/s13059-016-0997-x. 1.3.1

[117] Prashant Pandey, Yinjie Gao, and Carl Kingsford. VariantStore: an index for large-scale genomic variant search. *Genome Biology*, 22(1), August 2021. doi: 10.1186/s13059-021-02442-8. URL https://doi.org/10.1186/s13059-021-02442-8. 1.2.2, 4.6

[118] Benedict Paten, Mark Diekhans, Dent Earl, John St John, Jian Ma, Bernard Suh, and David Haussler. Cactus graphs for genome comparisons. *Journal of Computational Biology*, 18(3):469–481, 2011. 1.2.2, 1.3.1, 2.1, 4.1

[119] Benedict Paten, Adam M Novak, Jordan M Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *Genome Research*, 27(5):665–676, 2017. 2.1, 4.1

[120] Benedict Paten, Jordan M Eizenga, Yohei M Rosen, Adam M Novak, Erik Garrison, and Glenn Hickey. Superbubbles, ultrabubbles, and cacti. *Journal of Computational Biology*, 25(7):649–663, 2018. 1.4.2, 4.1

[121] Rob Patro and Carl Kingsford. Global network alignment using multiscale spectral signatures. *Bioinformatics*, 28(23):3105–3114, 2012. 1.3.2

[122] Ofir Pele and Michael Werman. A linear time histogram metric for improved sift matching. In *Computer Vision–ECCV 2008: 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008, Proceedings, Part III 10*, pages 495–508. Springer, 2008. 3.5.1

[123] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences of USA*, 98 (17):9748–9753, 2001. 1.2.2, 3.1, 4.3, 4.5.4.2

[124] Evgeny Polevikov and Mikhail Kolmogorov. Synteny paths for assembly graphs comparison. In *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. 1.3.3, 3.1, 3.5.1, 4.1, 4.6

[125] Nataša Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007. 1.3.2

[126] Yutong Qiu and Carl Kingsford. Constructing small genome graphs via string compression. *Bioinformatics*, 37(Supplement 1):i205–i213, 2021. 2, 4.6

[127] Yutong Qiu and Carl Kingsford. The effect of genome graph expressiveness on the discrepancy between genome graph distance and string set distance. *Bioinformatics*, 38: i404–i412, 2022. 3.3.4, 3.5.1, 3.8, 4

[128] Kari-Jouko Räihä and Esko Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16(2):187–198, 1981. 2.4.2

[129] Goran Rakocevic, Vladimir Semenyuk, Wan-Ping Lee, James Spencer, John Browning, Ivan J Johnson, Vladan Arsenijevic, Jelena Nadj, Kaushik Ghose, Maria C Suciu, et al. Fast and accurate genomic analyses using genome graphs. *Nature Genetics*, 51(2):354–362, 2019. 1.2.2, 2.1

[130] Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on discrete algorithms*, pages 233–242. Society for Industrial and Applied Mathematics, 2002. 2.4.1

[131] Lucas P Ramos, Felipe A Louza, and Guilherme P Telles. Genome comparison on succinct colored de bruijn graphs. In *String Processing and Information Retrieval: 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8–10, 2022, Proceedings*, pages 165–177. Springer, 2022. 1.3.3

[132] Mikko Rautiainen and Tobias Marschall. GraphAligner: rapid and versatile sequence-to-graph alignment. *Genome Biology*, 21(1):1–28, 2020. 4.6, 5.3.1

[133] Anthony R Rees. Understanding the human antibody repertoire. In *MAbs*, volume 12,

page 1729683. Taylor & Francis, 2020. 1.3

[134] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. Moni: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology*, 29(2):169–187, 2022. 1.2.1

[135] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The Earth Mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000. 3.5.1, 4.1, 4.3.1, 4.4.3.1

[136] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998. 3.6.1.1, 3.6.1.1

[137] Russell Schwartz and Alejandro A Schäffer. The evolution of tumour phylogenetics: principles and practice. *Nature Reviews Genetics*, 18(4):213–229, 2017. 1.3

[138] Scott Schwartz, W James Kent, Arian Smit, Zheng Zhang, Robert Baertsch, Ross C Hardison, David Haussler, and Webb Miller. Human–mouse alignments with blastz. *Genome Research*, 13(1):103–107, 2003. 1.3.1

[139] Michael M Shen. Chromoplexy: a new category of complex rearrangements in the cancer genome. *Cancer Cell*, 23(5):567–569, 2013. 1.2.2

[140] Rachel M Sherman and Steven L Salzberg. Pan-genomics in the human genome era. *Nature Reviews Genetics*, 21(4):243–254, 2020. 2.1

[141] Rachel M Sherman, Juliet Forman, Valentin Antonescu, Daniela Puiu, Michelle Daya, Nicholas Rafaels, Meher Preethi Boorgula, Sameer Chavan, Candelaria Vergara, Victor E Ortega, et al. Assembly of a pan-genome from deep sequencing of 910 humans of african descent. *Nature Genetics*, 51(1):30–35, 2019. 1.2

[142] Jonas A Sibbesen, Jordan M Eizenga, Adam M Novak, Jouni Sirén, Xian Chang, Erik Garrison, and Benedict Paten. Haplotype-aware pantranscriptome analyses using spliced pangenome graphs. *Nature Methods*, pages 1–9, 2023. 1.2.2

[143] Fabian Sievers, Andreas Wilm, David Dineen, Toby J Gibson, Kevin Karplus, Weizhong Li, Rodrigo Lopez, Hamish McWilliam, Michael Remmert, Johannes Söding, et al. Fast, scalable generation of high-quality protein multiple sequence alignments using clustal omega. *Molecular Systems Biology*, 7(1):539, 2011. 4.5.4.2

[144] François Sigaux. Cancer genome or the development of molecular portraits of tumors. *Bulletin de L'academie Nationale de Medecine*, 184(7):1441–7, 2000. 1.3

[145] Jouni Sirén. Indexing variation graphs. In *2017 Proceedings of the ninteenth workshop on algorithm engineering and experiments (ALENEX)*, pages 13–27. SIAM, 2017. 2.1, 2.4.1, 2.8.2, 2.10, 5.3.1

[146] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014. 2.1, 5.3.1

[147] Jouni Sirén, Erik Garrison, Adam M Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2020. 2.1, 2.4.1, 2.8.2, 2.10, 4.6

[148] Jouni Sirén, Jean Monlong, Xian Chang, Adam M Novak, Jordan M Eizenga, Charles Markello, Jonas A Sibbesen, Glenn Hickey, Pi-Chuan Chang, Andrew Carroll, et al. Pangenomics enables genotyping of known structural variants in 5202 diverse genomes. *Science*, 374(6574):abg8871, 2021. 1.1, 1.2.2

[149] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. 1.3.1

[150] J Storer. NP-completeness results concerning data compression. *Technical Report*, 234, 1977. 1.4.1, 2.1

[151] James A Storer and Thomas G Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982. 2.1, 2.3.2, 4, 2.3.3

[152] Cathie Sudlow, John Gallacher, Naomi Allen, Valerie Beral, Paul Burton, John Danesh, Paul Downey, Paul Elliott, Jane Green, Martin Landray, et al. UK Biobank: an open access resource for identifying the causes of a wide range of complex diseases of middle and old age. *PLoS Medicine*, 12(3):e1001779, 2015. 1.1

[153] Prasad Tetali and Santosh Vempala. Random sampling of euler tours. *Algorithmica*, 30: 376–385, 2001. 5.3.2

[154] Hervé Tettelin, Vega Masignani, Michael J Cieslewicz, Claudio Donati, Duccio Medini, Naomi L Ward, Samuel V Angiuoli, Jonathan Crabtree, Amanda L Jones, A Scott Durkin, et al. Genome analysis of multiple pathogenic isolates of streptococcus agalactiae: implications for the microbial “pan-genome”. *Proceedings of the National Academy of Sciences*, 102(39):13950–13955, 2005. 1.1

[155] Jonathan S Turner. Approximation algorithms for the shortest common superstring problem. *Information and computation*, 83(1):1–20, 1989. 2.4.2

[156] Leonid Nisonovich Vaserstein. Markov processes over denumerable products of spaces, describing large systems of automata. *Problemy Peredachi Informatsii*, 5(3):64–72, 1969. 4.3.1

[157] GS Vernikos. A review of pangenome tools and recent studies. *The pangenome: diversity, dynamics and evolution of genomes*, pages 89–112, 2020. 1.1, 1.3

[158] Mitchell R Vollger, Xavi Guitart, Philip C Dishuck, Ludovica Mercuri, William T Harvey, Ariel Gershman, Mark Diekhans, Arvis Sulovari, Katherine M Munson, Alexandra P Lewis, et al. Segmental duplications and their variation in a complete human genome. *Science*, 376(6588):eabj6965, 2022. 3.5.1

[159] Mitchell R Vollger, Philip C Dishuck, William T Harvey, William S DeWitt, Xavi Guitart, Michael E Goldberg, Allison N Rozanski, Julian Lucas, Mobin Asri, et al. Increased mutation and gene conversion within human segmental duplications. *Nature*, 617(7960): 325–334, 2023. 3.5.1

[160] Leonid N Wasserstein et al. Markov processes over denumerable products of spaces describing large systems of automata. *Problems of Information Transmission*, 5(3):47–52, 1969. 4.1, 4.3.1

[161] John N Weinstein, , Eric A Collisson, Gordon B Mills, Kenna R Mills Shaw, Brad A

Ozenberger, Kyle Ellrott, Ilya Shmulevich, Chris Sander, and Joshua M Stuart. The cancer genome atlas pan-cancer analysis project. *Nature Genetics*, 45(10):1113–1120, September 2013. doi: 10.1038/ng.2764. URL https://doi.org/10.1038/ng.2764. 1.1

[162] Richard C Wilson and Ping Zhu. A study of graph spectra for comparing graphs and trees. *Pattern Recognition*, 41(9):2833–2841, 2008. 1.3.2

[163] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2020. 1.3.2

[164] Ömer Nebil Yaveroğlu, Noël Malod-Dognin, Darren Davis, Zoran Levnajic, Vuk Janjic, Rasa Karapandza, Aleksandar Stojmirovic, and Nataša Pržulj. Revealing the hidden language of complex networks. *Scientific Reports*, 4(1):4547, 2014. 1.3.2

[165] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821–829, 2008. 4.1

[166] Yun Zhang, Chanhee Park, Christopher Bennett, Micah Thornton, and Daehwan Kim. Rapid and accurate alignment of nucleotide conversion sequencing reads with hisat-3n. *Genome Research*, 31(7):1290–1295, 2021. 5.3.1

[167] Lan Zhao, Victor HF Lee, Michael K Ng, Hong Yan, and Maarten F Bijlsma. Molecular subtyping of cancer: current status and moving toward clinical applications. *Briefings in Bioinformatics*, 20(2):572–584, 2019. 4.1

[168] Yang Zhou, Lv Yang, Xiaotao Han, Jiazheng Han, Yan Hu, Fan Li, Han Xia, Lingwei Peng, Clarissa Boschiero, Benjamin D Rosen, et al. Assembly of a pangenome for global cattle reveals missing sequences and novel structural variations, providing new insights into their diversity and evolutionary history. *Genome Research*, 32(8):1585–1601, 2022. 1.1

[169] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. 2.1, 5.2