THESIS

THEORY OF GRAPH TRAVERSAL EDIT DISTANCE, EXTENSIONS, AND APPLICATIONS

Submitted by

Ali Ebrahimpour Boroojeny

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2019

Master's Committee:

Advisor: Hamidreza Chitsaz

Asa Ben-Hur
Zaid Abdo

Copyright by Ali Ebrahimpour Boroojeny 2019

All Rights Reserved

ABSTRACT

THEORY OF GRAPH TRAVERSAL EDIT DISTANCE, EXTENSIONS, AND APPLICATIONS

Many problems in applied machine learning deal with graphs (also called networks), including social networks, security, web data mining, protein function prediction, and genome informatics. The kernel paradigm beautifully decouples the learning algorithm from the underlying geometric space, which renders graph kernels important for the aforementioned applications.

In this paper, we give a new graph kernel which we call graph traversal edit distance (GTED). We introduce the GTED problem and give the first polynomial time algorithm for it. Informally, the graph traversal edit distance is the minimum edit distance between two strings formed by the edge labels of respective Eulerian traversals of the two graphs. Also, GTED is motivated by and provides the first mathematical formalism for sequence co-assembly and *de novo* variation detection in bioinformatics.

We demonstrate that GTED admits a polynomial time algorithm using a linear program in the graph product space that is guaranteed to yield an integer solution. To the best of our knowledge, this is the first approach to this problem. We also give a linear programming relaxation algorithm for a lower bound on GTED. We use GTED as a graph kernel and evaluate it by computing the accuracy of an SVM classifier on a few datasets in the literature. Our results suggest that our kernel outperforms many of the common graph kernels in the tested datasets. As a second set of experiments, we successfully cluster viral genomes using GTED on their assembly graphs obtained from *de novo* assembly of next-generation sequencing reads.

In this project, we also show how to solve the problems of local and semi-global alignment between two graphs. Finally, we suggest an approach for speeding up the computations using pre-assumption on a subset of nodes that have to be paired.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Hamidreza Chitsaz, for his guidance in accomplishing this work. I also would like to thank my M.S. thesis committee member, Dr. Asa Ben-Hur and Dr. Zaid Abdo for offering their time and providing helpful advice. I wish to thank the co-authors of the paper that is published in the International Conference on Research in Computational Molecular Biology and covers a part of this project. Finally, I would like to thank the Computer Science faculty and staff for their help throughout my study at Colorado State University.

DEDICATION

I would like to dedicate this thesis to my parents, Fariba and Kohzad.

TABLE OF CONTENTS

ABSTRACT . . . ii
ACKNOWLEDGEMENTS . . . iii
DEDICATION . . . iv
LIST OF TABLES . . . vii
LIST OF FIGURES . . . viii

Chapter 1 Introduction . . . 1
1.1 Related Work . . . 3

Chapter 2 Problem Definition . . . 7

Chapter 3 Methods . . . 10
3.1 Brute Force Computation of Graph Traversal Edit Distance . . . 10
3.2 GTED as a Constrained Shortest Path Problem . . . 11
3.3 Lower Bound via Linear Programming Relaxation . . . 12
3.4 Algorithm for Graph Traversal Edit Distance . . . 13
3.5 Total Unimodularity . . . 15
3.6 Homology Theory of Alignment Graph . . . 16

Chapter 4 Generalizing GTED . . . 18
4.1 Chinese Postman Algorithm . . . 18
4.2 Relaxing the Constraints . . . 20

Chapter 5 Experiments . . . 24
5.1 Using GTED to make a kernel . . . 24
5.1.1 Data . . . 25
5.1.2 Pre-processing and post-processing . . . 25
5.1.3 Results . . . 26
5.1.4 Analysis . . . 27
5.2 Using GTED on genomic data . . . 28
5.2.1 Pre-processing and post-processing . . . 28
5.2.2 Results . . . 29

Chapter 6 Extensions of GTED . . . 30
6.1 Local Graph Traversal Edit Distance . . . 30
6.2 Fit Graph Traversal Edit Distance . . . 32

Chapter 7 More Efficiency! . . . 35

Chapter 8 Future Work . . . 44

Chapter 9 Conclusion . . . 46

Bibliography 48

Appendix A Strongly Connected Components 57

LIST OF TABLES

5.1 Comparing the accuracy of GTED to that of other common graph kernels on some benchmark datasets. 27
5.2 GTED values for pairs of corresponding assembly graphs of Hepatitis B viruses from different mammals. They belong to two different genera which are shown by gray color. 29

7.1 Running times for 4 different pairs of graphs before and after removing the edges that can be eliminated because of the master nodes. 43

LIST OF FIGURES

2.1 An edge-labeled Eulerian graph $A = (V, E, M, L, \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\})$ obtained from the $k = 4$ de Bruijn graph $G = (V, E)$ for the *circular* sequence ACAGACAT [1]. Vertices, V, correspond to $(k-1)$-mers and edges correspond to k-mers. In this case, all the edges have multiplicity one, i.e. $M \equiv 1$. Edge labels, L, show the kth nucleotide in the associated k-mers. 7

2.2 The graph of part (b) is an Eulerian traversal of the one in part (a). Therefore, the GTED value is 0 while the graphs are not the same. This example shows GTED is not a metric. 9

3.1 This grid is the alignment graph for AC versus AGC. Those edges that correspond to matches are in dashed lines (cost of a match is often 0). Solid lines show substitutions and indels which usually have a positive cost. The edit distance is the shortest distance from s to t in this graph (shown in blue). 10

4.1 A non-Eulerian graph with 4 odd nodes 19

6.1 The de Bruijn graph of R_1 for $k = 3$ 34

7.1 The nodes in a sample alignment graphs; the ones colored with red are the common nodes in the two input graphs. 37

7.2 Part (a) shows a colored de Bruijn graph in which the weight of each edge for each color is shown. Part (b) shows the two graphs extracted from the graph in (a). 38

7.3 The graph in (a) shows the alignment graph of the two graphs in Figure 7.2(b). The one in (b) is the alignment graph after all the removes that happen because of the master nodes. 39

A.1 A directed graph with 3 maximal SCCs 58

Chapter 1

Introduction

Networks, or graphs as they are called in mathematics, have become a common tool in modern biology. Biological information from DNA sequences to protein interaction to metabolic data to the shapes of important biological chemicals are often encoded in networks.

One goal of studying these networks is to compare them. We might want to know whether two DNA assembly graphs produce the same final sequences or how close the protein interaction networks of two related species are. Such comparisons are difficult owing to the fact that determining whether two graphs have an identical structure with different labels or vertex ordering is an NP-complete problem. Therefore, any comparison will need to focus on specific aspects of the graph.

Here, we present the notion of *graph traversal edit distance (GTED)*, a new method of comparing two networks. Informally, GTED gives a measure of similarity between two directed Eulerian graphs with labeled edges by looking at the smallest edit distance that can be obtained between strings from each graph via an Eulerian traversal. GTED was inspired by the problem of *differential genome assembly*, determining if two DNA assembly graphs will assemble to the same string. In the differential genome assembly problem, we have the de Bruijn graph representations of two (highly) related genome sequence data sets (e.g. one from a cancer tissue and the other from the normal tissue of the same individual), where each edge e represents a substring of size k from *reads* extracted from these genome sequences, and its multiplicity represents the number of times its associated substring is observed in the reads of the respective genome sequence. In this formulation, each vertex represents the $k-1$ length prefix of the label of its outgoing edges and the $k-1$ length suffix of the label of its incoming edges. Thus, the labels of all incoming edges of a vertex (respectively all outgoing edges) are identical with the exception of their first (last) symbol. Differential genome assembly has been introduced to bioinformatics in two flavors: (i) *reference genome free* version [2–6], and (ii) *reference genome dependent* version, which, in

its most general form, is NP-hard [7]. Both versions of the problem are attracting significant attention in biomedical applications (e.g. [8,9]) due to the reduced cost of genome sequencing (now approaching $1000 per genome sample) and the increasing needs of cancer genomics where tumor genome sequences may significantly differ from the normal genome sequence from the same individual through single symbol edits (insertions, deletions, and substitutions) as well as block edits (duplications, deletions, translocations, and reversals).

In addition to comparing assembly graphs, GTED can also be used to compare other types of networks. GTED yields a (pseudo-)metric for general graphs because it is based on the edit distance metric. Hence, it can be used as a graph kernel for a number of classification problems. GTED is the first mathematical formalism in which global traversals play a direct role in the graph metric. In this paper, we give a polynomial time algorithm using linear programming that is guaranteed to yield an integer solution. We use that as a graph kernel, and evaluate the performance of our new kernel in SVM classification over a few benchmark datasets. We also use GTED for clustering of viral genomes obtained from *de novo* assembly of next-generation sequencing reads. Note that GTED is a global alignment scheme that is not immediately scalable to full-size large genomes, like all other global alignment schemes such as Needleman-Wunsch. However, GTED can form the mathematical basis for scalable heuristic comparison of full-size large genomes in the future.

We define *graph traversal edit distance* (GTED) as follows. Let G_1 and G_2 be two directed graphs with *labeled* edges from a polynomial sized alphabet. Each edge e in each graph G_i has a non-negative *multiplicity (weight)* value, $M_i(e) \in \mathbb{Z}$ such that for any vertex of G_i, the total multiplicity of its incoming edges is equal to the total multiplicity of its outgoing edges. Equivalently, G_1 and G_2 are Eulerian graphs with their respective edge multiplicities M_1 and M_2 (chapter 4 discusses two different approaches that allows us to use GTED for non-Eulerian graphs as well). *Graph traversal edit distance* is the minimum possible (standard/Levenshtein) edit distance between the (cyclic) strings of edge labels, S_1 and S_2, implied by the (Eulerian) travelsals of G_1 and G_2 such that each edge e is traversed exactly $M_i(e)$ times in graph G_i.

1.1 Related Work

Many problems in applied machine learning deal with graphs, ranging from web data mining [10] to protein function prediction [11]. Some important application domains are biological networks such as regulatory networks, sequence assembly and variation detection, and structural biology and chemoinformatics where graphs capture structural information of macromolecules [12–14]. For instance, machine learning algorithms are often used to screen candidate drug compounds for safety and efficacy against specific diseases and also for repurposing of existing drugs [15]. Graphs are important in that in addition to the singular features or properties of a pattern, they are able to represent more information about the structure by showing the binary relationships between different parts or elements of the pattern. But, when it comes to pattern recognition, classification, and clustering, the ability to compare graphs matters. This has led to the problem of comparing graphs and graph matching. In its most basic form, we have the concept of *graph isomorphism*, which is the problem of finding a bijective function from nodes of a graph to another while preserving the structure and labels [16]. A weaker form of the problem, *subgraph isomorphism*, deals with the problem of finding a subgraph of one graph that is isomorphic to the other graph.

The binary nature of the graph isomorphism and subgraph isomorphism problems prevents us from using them directly as a distance measure; therefore, other definitions such as *maximal common subgraph* has been introduced which are using the idea of isomorphism under the hood while providing a scale for similarity. Maximal common subgraph, for two graphs A_1 and A_2 finds the common subgraph G such that there is subgraph isomorphism between G and A_1 and also between G and A_2 [17]. Comparison of the methods to solve this problem can be found in [18]. Following this definition and algorithms to deal with computing this similarity measure, different methods have been proposed to change that to a dissimilarity or distance measure [19–21].

There are two major problems in working with subgraph isomorphism, being NP-complete and lack of flexibility in capturing similarity; isomorphism requires exact similarity of the subgraph to consider it as a common part; however, when looking for a measure of similarity, usually an error-

tolerant approach would be of more use. Therefore, many error-tolerant matching approaches have been proposed [22, 23]. In error-tolerant matching not only the mapping of nodes and edges with different labels are allowed but also the edge-conserving constraint is relaxed; this allows us to even map nodes with different degrees to one another. Moreover, we are allowed to delete or insert some nodes and edges in each of the graphs. However, for any of these edit operations, a score or penalty should be assigned which depending on the problem and domain knowledge could be done differently. This freedom in assigning the weights to any of these operations might lead to a difference in the similarity measures; this elicits the necessity of model-learning or optimization in advance in many cases to be able to achieve acceptable results. Hence, the notion of the total cost for changing one graph to another based on a specific cost function can be used as a *graph edit distance*.

The major problem in using error-tolerant matching to compute the distance of two graphs is that optimizing the edit operations to minimizes the total cost for a pair of graphs is an NP-complete problem. In the literature, two general approaches have been used in order to handle this problem. The first group consists of the methods that try to make extra assumptions on the properties of the graphs and solve the problem for that specific class [24–26]. Another approach to handle the this problem, is to develop polynomial approximation methods that provide suboptimal answers to the optimization problem [27–36].

One important family of error-tolerant graph matching algorithms is *graph kernels*. Kernel methods elegantly decouple data representation from the learning part; hence, graph learning problems have been studied in the kernel paradigm [37]. Besides providing a distance measure, they make an implicit embedding of the graphs in the *Hilbert* space where a Hilbert space is an inner product space with the additional properties that it is separable and complete [38]. Following [37], other graph kernels have been proposed in the literature [39].

A graph kernel $k(G_1, G_2)$ is a (pseudo-)metric in the space of graphs. A kernel captures a notion of similarity between G_1 and G_2. For instance for social networks, k may capture similarity between their clustering structures, degree distribution, etc. For molecules, the similarity

between their sequential/functional domains and their relative arrangements is important. A kernel is usually computed from the adjacency matrices of the two graphs, but it must be invariant to the ordering (permutation) of the vertices. That property has been central in the graph kernels literature.

Existing graph kernels that are vertex permutation invariant use either local invariants, such as counting the number of triangles, squares, etc. that appear in G as subgraphs, or spectral invariants such as functions of the eigenvalues of the adjacency matrix or the graph Laplacian. Essentially, different graph kernels ranging from random walks [37, 40] to shortest paths [41, 42] to Fourier transforms on the symmetric group [43] to multiscale Laplacian [44] compute local, spectral, or multiscale distances. The idea behind Random Walks kernel is to count the number of common walks in the two input graphs. In its most basic form, we have to construct the product graph of the two; then, simply by raising the adjacency matrix of the product graph to the power k, we get all the walks of length k. The runtime of the original method, which is $O(n^6)$ is a bottleneck. A follow-up work improves the time complexity to $O(n^3)$ by casting the kernel computation to Sylvester equation [45]. Also, to alleviate the tottering issue other works have been done [46]. Tottering refers to the issue caused by random walks being able to visit the same cycle of nodes repeatedly and generate unreasonably large values of similarity even for a relatively small similar substructure in some cases. One approach to resolve the tottering problem is to use paths rather than walks. This is the general idea behind the shortest path graph kernel. As expected, this kernel addresses the problem of tottering but has the disadvantage of having a time complexity of $O(n^4)$ which might not be tolerable for large graphs. Cyclic pattern kernel replaces random walks with simple cycles [47]. While being able to capture interesting properties of the structures, they are NP-hard to compute. Subtree-like graph kernels are another one similar to the random walk kernel, where subtree-like structures recursively explore the neighbors of a node and compare them with those of the other graph. The reason they are not simply called subtree is that they may contain repetitive nodes and edges [48]. Weisfeiler-Lehman kernel uses the same idea together with a hashing scheme for combining the node labels of the neighboring nodes while constructing the subtree-like structures [49].

While most subgraph counting kernels are local [50], most random walk kernels are spectral [39]. Multiscale Laplacian [44], Weisfeiler-Lehman kernel, and propagation kernel [51] are among the multiscale kernels.

In this thesis, we introduce a graph kernel based on a comparison of global Eulerian traversals of the two graphs. To the best of our knowledge, our formalism is the first to capture global architectures of the two graphs as well as their local structures (Multiscale Laplacian kernels try to capture both global and local characteristics by building a hierarchy of nested graphs and applying a feature space Laplacian graph kernel to these subgraphs recursively; however, they do not explicitly make direct use of a global property of the graph). Our kernel is based on the graph traversal edit distance introduced in this paper. We show that a lower bound for GTED can be computed in polynomial time using the linear programming relaxation of the problem. In practice, the linear program often yields an integer solution, in which case the computed lower bound is actually equal to GTED [52].

Chapter 2

Problem Definition

Due to the diversity of applications, input graphs can be obtained as molecular structure graphs, social network graphs, systems biology networks, or sequence assembly graphs such as de Bruijn graphs [53], A-Bruijn graphs [54], positional de Bruijn graphs [55], string graphs [56], or implicit string graphs [57] among numerous alternatives. Our graph traversal edit distance is inspired by those applications and can potentially be adapted to any of those frameworks. However, we choose below a general, convenient representative definition for the problem. For the sake of brevity, we assume throughout this paper that the input graph has one strongly connected component.

Definition 1 (Edge-labeled Eulerian Graph)**.** *Let* Σ *be a finite alphabet. We call a tuple* $A = (V, E, M, L, \Sigma)$ *an edge-labeled Eulerian graph, in which*

- $G = (V, E)$ *is a strongly connected directed graph,*
- $M : E \to \mathbb{N}$ *specifies the edge multiplicities,*
- $L : E \to \Sigma$ *specifies the edge labels,*

iff G *with the corresponding edge multiplicities,* M*, is Eulerian. That is,* G *contains a cycle (or path from a specified source to a sink) that traverses every edge* $e \in E$ *exactly* $M(e)$ *times. Throughout this paper, we mean* M*-compliant Eulerian by an Eulerian cycle (path) in* A*.*

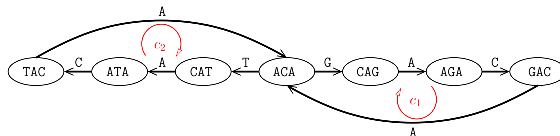

Figure 2.1: An edge-labeled Eulerian graph $A = (V, E, M, L, \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\})$ obtained from the $k = 4$ de Bruijn graph $G = (V, E)$ for the *circular* sequence ACAGACAT [1]. Vertices, V, correspond to $(k-1)$-mers and edges correspond to k-mers. In this case, all the edges have multiplicity one, i.e. $M \equiv 1$. Edge labels, L, show the kth nucleotide in the associated k-mers.

Figure 2.1 demonstrates an example edge-labeled Eulerian graph for the circular sequence ACAGACAT in the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. The sequence of edge labels over the Eulerian cycle formed by c_1 followed by c_2 yields the original sequence. The following definition makes a connection between Eulerian cycles and different sequences they spell.

Definition 2 (Eulerian Language)**.** *Let $A = (V, E, M, L, \Sigma)$ be an edge-labeled Eulerian graph. Define the* word *ω associated with an Eulerian cycle (path) $c = (e_0, \ldots, e_n)$ in A to be the word*

$$\omega(c) = L(e_0) \ldots L(e_n) \in \Sigma^*. \tag{2.1}$$

The language *of A is then defined to be*

$$\mathcal{L}(A) = \{\omega(c) \mid c \textit{ is an Eulerian cycle (path) in } A\} \subset \Sigma^*. \tag{2.2}$$

We now define graph traversal edit distance (GTED).

Problem 1 (Graph Traversal Edit Distance)**.** *Let A_1 and A_2 be two edge-labeled Eulerian graphs. We define the edit distance between A_1 and A_2 by*

$$d(A_1, A_2) = \min_{\substack{\omega_1 \in \mathcal{L}(A_1) \\ \omega_2 \in \mathcal{L}(A_2)}} d(\omega_1, \omega_2), \tag{2.3}$$

in which $d(\omega_1, \omega_2)$ is the Levenshtein edit distance between two strings ω_1 and ω_2. Throughout this paper, edit operations are single alphabet symbol insertion, deletion, and substitution, and the Levenshtein edit distance is the minimum number of such operations to transform ω_1 to ω_2 [58] (we make it symmetric by considering an equal penalty for insertion and deletion operations).

Note that $d(A_1, A_2)$ is the minimum of such edit distances over the words of possible Eulerian cycles (paths) in A_1 and A_2. Also, note that GTED is almost a metric but not a metric since there are A_1, A_2 such that $d(A_1, A_2) = 0$ even though $A_1 \neq A_2$. For instance, consider the two graphs in Figure 2.2. The one in *b* is an Eulerian cycle of the other. As a result, the graph traversal edit

distance is different from the graph edit distance because the latter is a metric whereas the former is not.

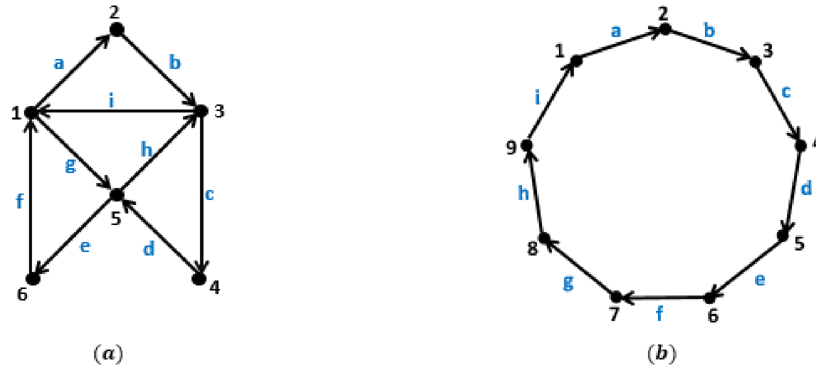

Figure 2.2: The graph of part (b) is an Eulerian traversal of the one in part (a). Therefore, the GTED value is 0 while the graphs are not the same. This example shows GTED is not a metric.

Chapter 3

Methods

3.1 Brute Force Computation of Graph Traversal Edit Distance

It is clear that there are algorithms, albeit with exponential running time, that enumerate all Eulerian cycles in a graph. Through brute force Needleman-Wunsch alignment of the words of every pair of Eulerian cycles in A_1 and A_2, we can compute the edit distance right from the definition. De Bruijn, van Aardenne-Ehrenfest, Smith, and Tutte proved the de Bruijn-van Aardenne-Ehrenfest-Smith-Tutte (BEST) theorem [59–61], which counts the number of different Eulerian cycles in A as

$$ec(A) = t_w(A) \prod_{v \in V} (\deg(v) - 1)!, \tag{3.1}$$

in which $t_w(A)$ is the number of arborescences directed towards the root at a fixed vertex w, and $\deg$ is the indegree (or equally outdegree) considering multiplicities. The number of Eulerian cycles $ec(A)$ is exponentially large in general. Therefore, the naïve brute force algorithm is intractable.

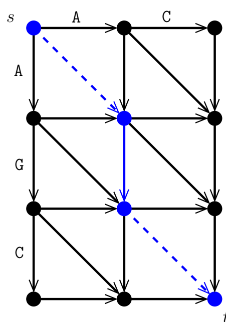

Figure 3.1: This grid is the alignment graph for `AC` versus `AGC`. Those edges that correspond to matches are in dashed lines (cost of a match is often 0). Solid lines show substitutions and indels which usually have a positive cost. The edit distance is the shortest distance from s to t in this graph (shown in blue).

3.2 GTED as a Constrained Shortest Path Problem

The conventional string alignment problem can be transformed into a shortest path problem in an alignment graph which is obtained by adding appropriate edges to the Cartesian product of the two string graphs. In this graph, the vertical edges represent a deletion operation on the characters of the first string (equivalently, an insertion operation in the second string); similarly, the horizontal edges represent an insertion in the first string (a deletion in the second string). Diagonal edges in this graph can be representative of either a match or a mismatch. Finally, for each of these edges, a weight equal to the cost of the corresponding operation is assigned. Figure 3.1 illustrates an example. To compute the optimum cost up to each node in this product graph, a dynamic programming approach called Needleman-Wunch algorithm is used. The main idea behind this algorithm is that starting from the top-left corner of the product graph, all the way down to the bottom-right corner, to compute the cost of the shortest path to a node v, we have to compute the cost of the three nodes that has an edge to v and find the one that has the minimum amount of cost to reach plus the cost of its edge to v [1]. Analogously, the graph traversal edit distance $d(A_1, A_2)$ can be written as the length of the shortest cycle (or path from a designated source to a designated sink) in the alignment graph defined below, whose projection onto A_1 and A_2 is Eulerian. To state that fact in Lemma 1, we need

Definition 3 (Alignment Graph)**.** *Let* $A_1 = (V_1, E_1, M_1, L_1, \Sigma_1)$ *and* $A_2 = (V_2, E_2, M_2, L_2, \Sigma_2)$ *be two edge-labeled Eulerian graphs. Define the alignment graph between* A_1 *and* A_2 *to be* $\mathcal{AG}(A_1, A_2) = (V_1 \times V_2, E)$*, in which* E *is a collection of horizontal, vertical, and diagonal edges as follows:*

- *Vertical:* $\forall\, e_1 = (u_1, v_1) \in E_1$ *and* $u_2 \in V_2 :\ e_1 \times u_2 = [(u_1, u_2), (v_1, u_2)] \in E,$
- *Horizontal:* $\forall\, u_1 \in V_1$ *and* $e_2 = (u_2, v_2) \in E_2 :\ u_1 \times e_2 = [(u_1, u_2), (u_1, v_2)] \in E,$
- *Diagonal:* $\forall\, e_1 = (u_1, v_1) \in E_1$ *and* $e_2 = (u_2, v_2) \in E_2 :\ [(u_1, u_2), (v_1, v_2)] \in E.$

There is a cost $\delta : E \to \mathbb{R}$ *associated with each edge of* $\mathcal{AG}$ *based on edit operation costs. Horizontal and vertical edges correspond to insertion or deletion and diagonal edges correspond*

to match or mismatch (substitution). A diagonal edge $[(u_1, u_2), (v_1, v_2)]$ *is a match iff* $L(u_1, v_1) = L(u_2, v_2)$ *and a mismatch otherwise.*

The following Lemma states the fact that GTED is equivalent to a constrained shortest path problem in the alignment graph.

Lemma 1. *For any two edge-labeled Eulerian graphs* $A_1 = (V_1, E_1, M_1, L_1, \Sigma_1)$ *and* $A_2 = (V_2, E_2, M_2, L_2, \Sigma_2)$*,*

$$\begin{aligned} d(A_1, A_2) = \underset{c}{\textit{minimize}} \quad & \delta(c) \\ \textit{subject to} \quad & c \textit{ is a cycle (path) in } \mathcal{AG}(A_1, A_2), \\ & \pi_i(c) \textit{ is an Eulerian cycle (path) in } A_i \textit{ for } i = 1, 2, \end{aligned} \tag{3.2}$$

in which $\delta(c)$ *is the total edge-cost (edit cost) of c, and* π_i *is the projection onto the* i^{th} *component graph.*

Proof. For every pair (c_1, c_2), in which c_i is an Eulerian cycle (path) in A_i, there are possibly multiple c's with $\pi_i(c) = c_i$, whose minimum total edge-cost is $d(\omega(c_1), \omega(c_2))$. Therefore, the result of the minimization in (3.2) is not more than $d(A_1, A_2)$, i.e. the right hand side is less than or equal to $d(A_1, A_2)$. Conversely, every c that satisfies the constraints in (3.2) gives rise to an Eulerian pair $(c_1, c_2) = (\pi_1(c), \pi_2(c))$ and $\delta(c) \geq d(\omega(c_1), \omega(c_2)) \geq d(A_1, A_2)$, i.e. the right hand side is greater than or equal to $d(A_1, A_2)$. □

3.3 Lower Bound via Linear Programming Relaxation

Lemma 1 easily transforms our problem into an integer linear program (ILP) as the projection operator π_i is linear and imposing path connectivity/cycle is also linear. More precisely, consider two edge-labeled Eulerian graphs A_1 and A_2 with the alignment graph $\mathcal{AG}(A_1, A_2) = (V_1 \times V_2, E)$, and let ∂ be the boundary operator, $\partial(e) = v - u$ for an edge $e = (u, v)$, which is defined in detail below. Our algorithm consists in solving the linear programming (LP) relaxation of that ILP,

$$
\begin{aligned}
\underset{x \in \mathbb{R}^{|E|}}{\text{minimize}} \quad & \sum_{e \in E} x_e \delta(e) \\
\text{subject to} \quad & \sum_{e \in E} x_e \partial(e) = 0 \quad (\text{or sink } - \text{ source}), \\
& \forall\, e \in E, \quad x_e \geq 0, \\
& \text{for } i = 1, 2, \forall\, f \in E_i, \quad \sum_{e \in E} x_e I_i(e, f) = M_i(f),
\end{aligned}
\tag{3.3}
$$

in which indicator function $I_1(e, f) = 1$ iff $e = f \times v_2$ or $e = [(u_1, u_2), (v_1, v_2)]$ with $f = (u_1, v_1)$; otherwise, $I_1(e, f) = 0$. Similarly, $I_2(e, f) = 1$ iff $e = v_1 \times f$ or $e = [(u_1, u_2), (v_1, v_2)]$ with $f = (u_2, v_2)$; otherwise, $I_2(e, f) = 0$. The linear program above is not guaranteed to give an integer solution; however, we have observed integer solutions in many scenarios. Nevertheless, the solution of (3.3) is a lower bound for GTED.

3.4 Algorithm for Graph Traversal Edit Distance

The following theorem is the main result of this paper which bridges the gap between GTED and another linear programming formulation which we will show is guaranteed to have an exact integer solution. Hence, GTED has a polynomial time algorithm explained as a linear program; Corollary 1 states that fact below.

Theorem 1 (GTED)**.** *Consider two edge-labeled Eulerian graphs $A_i = (V_i, E_i, M_i, L_i, \Sigma_i)$ with $G_i = (V_i, E_i)$ for $i = 1, 2$. Let T be the collection of two-simplices in the triangulated $G_1 \times G_2$ with one-faces in $\mathcal{AG}(A_1, A_2)$. In that case,*

$$
\begin{aligned}
d(A_1, A_2) = \underset{x \in \mathbb{R}^{|E|}, y \in \mathbb{R}^{|T|}}{\textit{minimize}} \quad & \sum_{e \in E} x_e \delta(e) \\
\textit{subject to} \quad & x = x^{\textit{init}} + [\partial]\, y, \\
& \forall\, e \in E, \quad x_e \geq 0,
\end{aligned}
\tag{3.4}
$$

in which $[\partial]_{|E| \times |T|}$ is the matrix of the two-dimensional boundary operator in the corresponding homology and

$$x_e^{\textit{init}} = \begin{cases} M_1(f) & \textit{if } e = f \times s_2 \\ M_2(f) & \textit{if } e = s_1 \times f \\ 0 & \textit{otherwise} \end{cases} \tag{3.5}$$

for arbitrary fixed $s_i \in V_i$ (source/sink in the case of path).

Proof. It is sufficient to show two things:

1. GTED is equal to the solution of the integer linear program (ILP) version of the linear program in (3.4),

2. the linear program in (3.4) always yields an integer solution.

Using Lemma 1, we need to show that (3.2) and (3.4) are equivalent for the first one. By construction, x^{init} corresponds to an Eulerian cycle (path) in A_1 followed by one in A_2, which specifies a cycle (path) in $\mathcal{AG}(A_1, A_2)$ whose projection onto A_i is Eulerian. It is sufficient to note that every cycle (path) whose projection onto A_i is Eulerian is homologous to x^{init}. To see that, let c be a cycle (path) whose projection onto A_i is Eulerian. First, note that diagonal edges in c are homologous to the horizontal edge followed by the vertical edge in the corresponding cell. Hence, diagonal edges can be replaced by the horizontal followed by the vertical edge using the boundary operator $[\partial]$. Hence without loss of generality, we assume c contains only horizontal and vertical edges.

If edges in c are $h_1, h_2, \ldots, h_m, k_1, k_2, \ldots, k_n$ such that $h_i = e_i \times s_2$ and $k_i = s_1 \times f_i$ for $e_i \in E_1$ and $f_i \in E_2$, then we are done. We know that such c has exactly the same representation as x^{init}. If edges in c are $h_1, h_2, \ldots, h_m, k_1, k_2, \ldots, k_n$ such that $h_i = e_i \times v_2$ and $k_i = v_1 \times f_i$ for $e_i \in E_1$ and $f_i \in E_2$ and possibly $v_1 \neq s_1$ or $v_2 \neq s_2$, then we can rotate the cycle through adding and subtracting a perpendicular translation edge and apply the boundary replacement operation to obtain a homologous cycle (path) of the form $h_1, h_2, \ldots, h_m, k_1, k_2, \ldots, k_n$ such that $h_i = e_i \times s_2$ and $k_i = s_1 \times f_i$ for $e_i \in E_1$ and $f_i \in E_2$. Starting with an arbitrary c, we show how to obtain a homologous cycle (path) in the form of $h_1, h_2, \ldots, h_m, k_1, k_2, \ldots, k_n$ such that $h_i = e_i \times v_2$ and $k_i = v_1 \times f_i$ for $e_i \in E_1$ and $f_i \in E_2$ through basic boundary replacement operations. Essentially,

we show that we can swap vertical and horizontal edges along c until we end up with all horizontal edges grouped right up front followed by all vertical edges grouped at the end. Suppose c contains k, h as a subpath for $h = e \times v_2$ and $k = u_1 \times f$ and $e = (u_1, v_1) \in E_1$ and $f = (u_2, v_2) \in E_2$. The subpath k, h is homologous to h', k' in which $h' = e \times u_2$ and $k' = v_1 \times f$ since the four edges $k, h, -k', -h'$ form the boundary of a square. Hence, we can replace k, h with h', k' in c to obtain a homologous cycle (path) c'. Performing a number of such vertical-horizontal swaps will yield the result. The second is going to be shown in the following sections. □

Corollary 1 (GTED complexity)**.** *The graph traversal edit distance is in P and can be solved in polynomial time from the linear program in (3.4) that is guaranteed to give the integer solution.*

3.5 Total Unimodularity

Using a recent result of Dey, Hirani, and Krishnamoorthy [62], we show that (3.4) is guaranteed to yield an integer solution. The main reason is that the boundary operator matrix $[\partial]$ is totally unimodular, i.e. all its square submatrices have a determinant in $\{0, \pm 1\}$. Therefore, all vertices of the constraint polytope in (3.4) have integer coordinates; hence, the solution is integer.

Why is $[\partial]$ totally unimodular? According to [62, Theorem 5.13], $[\partial]$ is totally unimodular iff the simplicial complex $G_1 \times G_2$ has no Möbius subcomplex of dimension 2. For the sake of completeness, we include the definition of a Möbius complex below.

Definition 4 ([62, Definition 5.9])**.** *A two-dimensional cycle complex is a sequence $\sigma_0 \cdots \sigma_{k-1}$ of two-simplices such that σ_i and σ_j have a common face iff $j = (i + 1) \mod k$ and that the common face is a one-simplex. It is called a two-dimensional cylinder complex if orientable and a two-dimensional Möbius complex if nonorientable.*

Lemma 2. *A triangulated graph product space $G_1 \times G_2$ does not contain a Möbius subcomplex, for directed graphs G_i with unidirectional edges.*

Proof. It is enough to observe that in $G_1 \times G_2$, the orientation in one coordinate cannot flip. For brevity of presentation, we ignore triangulation for a moment and consider the rectangular cells.

To the contrary, assume $G_1 \times G_2$ contains a Möbius subcomplex $\sigma_0 \cdots \sigma_{k-1}$ in which every σ_i is a rectangle $e_i \times f_i$, for $e_i \in E_1$ and $f_i \in E_2$. Since every σ_i and σ_{i+1} have a common edge and G_1, G_2 are directed graphs with unidirectional edges, either $e_{i+1} = e_i$ or $f_{i+1} = f_i$ but not both. In particular, $e_0 = e_{k-1}$ or $f_0 = f_{k-1}$. That is a contradiction because $\sigma_0 \cdots \sigma_{k-1}$ is then a cylinder subcomplex (orientable) and not a Möbius subcomplex. □

Lemma 2 together with [62, Theorem 5.13] assert that $[\partial]$ is totally unimodular. Therefore, (3.4) always has an integer solution, hence the main result in Theorem 1.

Lack of Möbius subcomplexes in the product space of graphs, which are Möbius-free spaces, can also be seen from the fact that the homology groups of graph product spaces are torsion-free. The following section summarizes that characterization.

3.6 Homology Theory of Alignment Graph

An alignment graph $\mathcal{AG}(A_1, A_2)$ is essentially a topological product space with additional triangulating diagonal edges corresponding to matches and mismatches. In other words, $\mathcal{AG}(A_1, A_2)$ can be regarded as a triangulation of the two-dimensional CW complex $G_1 \times G_2$ (by horizontal, vertical, and diagonal edges). Note that $G_1 \times G_2$ has zero-dimensional vertices (v_1, v_2), one-dimensional edges $e_1 \times v_2$ and $v_1 \times e_2$, and two-dimensional squares $e_1 \times e_2$ for $v_i \in V_i$ and $e_i \in E_i$. We characterize below the homology groups of $G_1 \times G_2$ using the Künneth's theorem. Note that G_i are obtained from edge-labeled graphs $A_i = (V_i, E_i, M_i, L_i, \Sigma_i)$.

Theorem 2 (Künneth [63])**.** *For graphs $G_i = (V_i, E_i)$, $i = 1, 2$,*

$$H_m(G_1 \times G_2, \mathbb{Z}) \cong \bigoplus_{p+q=m} H_p(G_1, \mathbb{Z}) \otimes H_q(G_2, \mathbb{Z}) \;\oplus \bigoplus_{r+s=m-1} \mathit{Tor}(H_r(G_1, \mathbb{Z}), H_s(G_2, \mathbb{Z})), \tag{3.6}$$

in which H_m is the mth homology group and Tor is the torsion functor [63].

Since $G_1 \times G_2$ is a two-dimensional CW complex, $H_m(G_1 \times G_2, \mathbb{Z}) \cong 0$ for $m > 2$. Clearly, $H_0(G_1 \times G_2, \mathbb{Z}) \cong \mathbb{Z}$ since $G_1 \times G_2$ is connected. According to the Künneth's theorem above and

the fact that $\text{Tor}(\mathbb{Z},\mathbb{Z}) \cong 0$ and $\mathbb{Z}^k \otimes \mathbb{Z} \cong \mathbb{Z} \otimes \mathbb{Z}^k \cong \mathbb{Z}^k$ [64],

$$\begin{aligned} H_1(G_1 \times G_2) &\cong [H_1(G_1) \otimes H_0(G_2)] \oplus [H_0(G_1) \otimes H_1(G_2)] \oplus \text{Tor}(H_0(G_1), H_0(G_2)) \\ &\cong [\mathbb{Z}^{n_1} \otimes \mathbb{Z}] \oplus [\mathbb{Z} \otimes \mathbb{Z}^{n_2}] \oplus \text{Tor}(\mathbb{Z},\mathbb{Z}) \cong \ \mathbb{Z}^{n_1} \oplus \mathbb{Z}^{n_2} \cong \ \mathbb{Z}^{n_1+n_2}, \end{aligned} \tag{3.7}$$

in which $n_i = 1 + |E_i| - |V_i|$. Note that $H_1(G_1, G_2)$ is torsion-free.

Using the Künneth's theorem above and the fact that the tensor product of groups $\otimes$ distributes over the direct sum $\oplus$, $H_2(G_i) \cong 0$, Tor of torsion-free groups is trivial, and $\mathbb{Z} \otimes \mathbb{Z} \cong \mathbb{Z}$, we obtain

$$\begin{aligned} H_2(G_1 \times G_2) &\cong [H_1(G_1) \otimes H_1(G_2)] \oplus \text{Tor}(H_1(G_1), H_0(G_2)) \oplus \text{Tor}(H_0(G_1), H_1(G_2)) \\ &\cong [\mathbb{Z}^{n_1} \otimes \mathbb{Z}^{n_2}] \oplus \text{Tor}(\mathbb{Z}^{n_1}, \mathbb{Z}) \oplus \text{Tor}(\mathbb{Z}, \mathbb{Z}^{n_2}) \cong \ \bigoplus_{i=1}^{n_1} \bigoplus_{j=1}^{n_2} \mathbb{Z} \otimes \mathbb{Z} \cong \ \mathbb{Z}^{n_1 n_2}. \end{aligned} \tag{3.8}$$

Note that $H_2(G_1, G_2)$ is torsion-free.

Chapter 4

Generalizing GTED

As you may have noticed, based on the definition of GTED and its solution, the general assumption is that the input graphs are Eulerian; otherwise, the constraints of the LP for finding an Eulerian traversal while keeping the projections of all the edges of the input graphs equal to their weights cannot be satisfied and will lead to an LP for which no solution exists. But, being Eulerian is a very strong constraint, and even on the de Bruijn graphs, this is not usually the case. In this chapter, we are going to explain two different approaches that make us able to generalize GTED to all (strongly) connected graphs (for disconnected graphs, GTED has to be run on pair of components separately). We study each of these methods in a separate section.

4.1 Chinese Postman Algorithm

We start with the definition of the original problem and continue with explaining the solution in general. Then, we are going to explain how this method can be utilized to extend the application of GTED to non-Eulerian graphs. Chinese Postman Problem (CPP) or Route Inspection Problem is the problem of finding the minimum number of edges that have to be duplicated in a graph such that the resulting graph contains a closed path that visits every edge of the graph exactly once, i.e, the graph becomes Eulerian. This can similarly be applied to weighted graphs (the ones with discrete weights so that they can be scaled to the integer ones) by assuming that each integer weight is the number of copies of a single edge. Although the original problem considers the case of undirected graphs, the same definition can be used for the directed ones; we will mention the slight changes that have to be made to the solution of the former case in order to make one for the latter case. To illustrate the problem better, consider the graph in Figure 4.1. This graph is not Eulerian since for a graph to be Eulerian, its necessary and sufficient condition is to have even degree for each of its nodes, and it is clear that this not the case for this graph (nodes 1, 2, 3, and 4 have odd degrees).

To solve the problem, we should notice that a graph has even number of odd nodes. This is due to the fact that the sum of the degrees of all the nodes of the graph is even because each edge appears two times in computing this sum, once for each of its endpoints. Another thing to consider when solving the CPP is that once we duplicate a path that connects two nodes of the graph, the parity of the degree of each of its nodes except for the two end nodes remains the same because two edges have to be added for each pass through a node; however, the parity of the two endpoints change. Therefore, if we duplicate a path that connects two odd nodes of the graph, the odd nodes become even while keeping the parity of all the other nodes intact. Based on aforementioned properties, one brute force method is to consider all pairings of odd nodes, and compute the shortest path between each of the two in a pair, and choose the pairing that provides us with the minimum sum of distances over all its pairs. For example, for the graph of Figure 4.1, there are 3 possible pairings, $\{(1,2),(3,4)\}$, $\{(1,3),(2,4)\}$, and $\{(1,4),(2,3)\}$ with sum of minimum distances 2, 2, and 4 respectively. So, any of the first two pairings would provide us with the minimum number of edges that have to be duplicated to make the graph Eulerian, which is 2 in this case (e.g. duplicating edges $(1,2)$ and $(3,4)$).

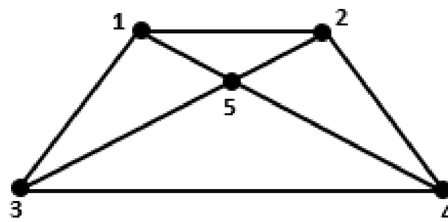

Figure 4.1: A non-Eulerian graph with 4 odd nodes

Since the total number of pairings is exponential in the number of odd nodes, this brute force solution is not practical when the number of such nodes increases. However, this problem can be solved in polynomial time [65]. This method makes a complete weighted graph on the set of all odd nodes where the weight of each edge is the weight of the shortest path between its

two end nodes in the original graph. This graph can be built in $O(V^3)$ using Floyd-Warshall algorithm ([?,66]) which is capable of finding all pair-wise shortest distances in a graph in $O(V^3)$. Afterward, a minimum perfect matching has to be found on the complete graph, which can be done in $O(V^{2.376})$ [67]. The solution to the matching problem provides us with a pairing on the set of odd nodes such that the amount of weights that has to be added to the graph to make it Eulerian is minimum; finally, we have to increase the weights of the edges of the original graph that are present on a path that corresponds to an edge of the chosen pairing on the complete graph of odd nodes by one. According to the reasoning made earlier about how this change affects the parity of the nodes, the final graph will be Eulerian.

Hence, the Chinese Postman algorithm provides us with a method to change each graph such that it becomes Eulerian and can be used as an input to GTED; however, we have to notice the fact that by using this method, we are actually changing the original graphs which might not be desirable in some cases. In the following section, we will introduce another approach to generalize GTED that does not need any changes to the input graphs.

4.2 Relaxing the Constraints

In this section, instead of changing the input graphs, we are going to make some changes to the LP that is generated to derive the GTED value. This becomes specifically useful for the de Bruijn graphs which are often large graphs where the time complexity of the algorithm to CPP is not tolerable or at least desirable. Another important reason to look for such method is avoiding to change the input graphs, and preserve the original information they contain. A standard Linear Program (LP) is an optimization problem of the form:

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^{\intercal} x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{4.1}$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, and $A \in \mathbb{R}^{m \times n}$. In case of having inequality in the constraints, we can use *surplus* and *slack* variables to change the LP to the standard form. Equation 4.2 shows how to handle the case in which we have *greater than* ($>$) inequalities in the constraints. The trick is to subtract a single variable $y_i \geq 0$ from the left side of the inequality constraints to let the value of the rest of that be greater than the value on the right side. These y_i variables are called surplus variables [68].

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^\intercal x \\ \text{subject to} \quad & Ax - I_m y = b \\ & x \geq 0 \end{aligned} \tag{4.2}$$

When we have *less than* inequalities instead of the equations in the constraints, we add a single variable $y_i \geq 0$ to the left side of each such constraint to let the value of the rest of that be less than the value on the right side. Here, we call the y_is, slack variables. Equation 4.3 shows the resulting LP, which is now in the standard form [68].

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^\intercal x \\ \text{subject to} \quad & Ax + I_m y = b \\ & x \geq 0 \end{aligned} \tag{4.3}$$

As mentioned before, in the original LP for computing GTED we have two sets of constraints, the Eulerian constraints and the projection constraints. One important thing to notice is that if we do not require the projection constraints to be satisfied, then the input graphs do not have to be Eulerian anymore; however, the projection of the tour on the alignment graph onto any of the input graphs has to be an Eulerian tour, and this is due to the first set of constraints. Henceforth, we do not need the graphs to be Eulerian; rather, to be able to generate a solvable LP, we have to make sure that each of the input graphs has at least one Eulerian subgraph.

When the input graphs are directed, which also is the case for de Bruijn graphs, the only requirement in order to have Eulerian tours in the graphs is that the graph is a single Strongly

Connected Component (SCC). The definition of SCC and the algorithm we have used to find the SCCs in the de Bruijn graphs are presented in Appendix A. From the definition, it is easy to observe that in any SCC, we have directed cycles which are Eulerian tours on subgraphs of the original graph; for any two nodes u and v in the SCC, based on the definition, we know that there is at least one directed path from u to v, and there is at least one path from v to u. By concatenating the latter path to the former one, a directed cycle will be derived.

Therefore, by changing the equality constraints for the projections to less than or equal constraints we derive an LP like what we have in equation 4.3 where y_is are the slack variables added to the projection constraints. This new LP finds Eulerian subgraphs of the input graphs that minimize the objective function of the LP. However, we still want these subgraphs to be reminiscent of the original graphs as much as possible by making their size as close to that of the original ones as possible. To achieve this goal, we add the slack variables (y_is) to the objective function. This will help us to minimize the distance of the Eulerian traversals while keeping the number of times each is used in the traversal as close as to their original weights.

The method we just explained always provides us with an answer in which the number of times each edge is used in the traversal is less than its weight. However, in many cases, we might get a better answer by letting the traversal to use some edges more than their weights. In other words, we are looking for a traversal that is made by making the minimum amount of changes to the original weights. To form an LP with this property, rather than projection constraints, we need to minimize the absolute value of the difference between the projections and the original weights. To make such LP, we have to remove the projection constraints, and instead of each one we have to add the absolute value of the difference of the left and right side of the equation to the objective function. Finally, we have to change the LP which has absolute values in the objective function, like the one in equation 4.4, to one that is in the standard form of equation 4.1.

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^{\mathsf{T}}|x| \\ \text{subject to} \quad & Ax = b \end{aligned} \tag{4.4}$$

In order to make this change, we use a new vector z with the same size as x and replace the $|x|$ term in the objective function by that. Also, we add two new sets of constraints to the LP, $x \leq z$ and $-x \leq z$. These two sets of constraints are equivalent to $|x| \leq z$, and because we added z to the objective function, we are actually minimizing $|x|$ which is bounded by z, and that is what the original LP in 4.4 was meant to do. After adding two vectors of slack variables $y^{(1)}$ and $y^{(2)}$, the final formulation 4.5 is derived. Using the same technique, we can change our original LP 3.3 to one that minimizes the absolute difference of the projections of an edge in the output traversal and its original weight in the input graph. Notice that this absolute difference will play the role of x_is in the representation of 4.5. Also, $Ax = b$ could be considered as the Eulerian constraints if we consider $b = 0$.

$$
\begin{aligned}
\underset{z \in \mathbb{R}^n}{\text{minimize}} \quad & c^{\mathsf{T}} z \\
\text{subject to} \quad & Ax = b \\
& x + y^{(1)} = z \\
& -x + y^{(2)} = z \\
& y \geq 0, z \geq 0
\end{aligned}
\tag{4.5}
$$

The latter solution makes more sense for de Bruijn graphs compared to the one that bounds the number of projections to the weight for each edge because usually, the weights are not very accurate due to the possibility of error in the sequencing reads and other problems such as non-uniform coverage and some missing regions due to low coverage. Hence, we used this method in our experiments involving de Bruijn graphs.

There are other tricks that can be utilized to fulfill other constraints. For example, if we want to have at least one projection for each edge of the input graphs (i.e. we want to use each edge of the input graphs at least once in the alignment) we can simply add a new set of constraints $x \geq 1$ to the LP.

Chapter 5

Experiments

5.1 Using GTED to make a kernel

As mentioned earlier, since GTED is a measure of distance or dissimilarity between two graphs, we can use it to make a kernel of the distance of pair of graphs in a dataset, and this can be used for classification problems. We implemented a C++ program that generates the linear program for the problem. First, it builds the alignment graph $\mathcal{AG}$ for two given graphs $A_1 = (V_1, E_1)$ and $A_2 = (V_2, E_2)$ where V_i and E_i are vertices and edges of the ith graph. It begins with $|V_1| \times |V_2|$ vertices that are labeled as (v_1, v_2) for each $v_1 \in V_1$ and $v_2 \in V_2$. For each edge $(u_1, v_1) \in E_1$ and vertex $u_2 \in V_2$ we add the vertical edge $[(u_1, u_2), (v_1, u_2)]$ with a gap penalty δ_1 to our grid, $\mathcal{AG}$. We also add a horizontal edge $[(u_1, u_2), (u_1, v_2)]$ for each vertex $u_1 \in V_1$ and edge $(u_2, v_2) \in E_2$ with the same cost δ_1. Then, for each pair of edges $(u_1, v_1) \in E_1$ and $(u_2, v_2) \in E_2$ we add a diagonal edge $[(u_1, u_2), (v_1, v_2)]$, with a mismatch penalty δ_2 if (u_1, v_1) has a different label from (u_2, v_2), or a match bonus δ_3 if the labels are the same. The cost values are taken as arguments, with default values of $\delta_1 = \delta_2 = 1$ and $\delta_3 = 0$. This can be further extended to different penalties for insertion and deletion (i.e. different cost for horizontal and vertical edges).

The C++ program also creates a projection set for each edge in either of the input graphs. Each vertical edge $[(u_1, u_2), (v_1, u_2)]$ is added to the projection set of the edge $(u_1, v_1) \in E_1$, each horizontal edge $[(u_1, u_2), (u_1, v_2)]$ to the set of $(u_2, v_2) \in E_2$, and each diagonal edge $[(u_1, u_2), (v_1, v_2)]$ to projection sets of both $(u_1, v_1) \in E_1$ and $(u_2, v_2) \in E_2$.

Our program then extracts a linear programming problem from the alignment graph by assigning a variable x_i to the ith edge of $\mathcal{AG}$. The objective function minimizes weighted sum $\sum_{e \in E, \delta(e) > 0} x_e \delta(e)$. Then, the constraints will be generated. There are two different groups of constraints. The first group forces the vertices of the grid to have the same number of incoming edges and outgoing edges, forcing the output to be a cycle in the alignment graph. The second group

forces the size of the projection set for each edge of the input graphs to be equal to its weight in that input graph, forcing the projection of the output to be Eulerian in both input graphs.

We used an academic license of Gurobi optimizer to solve the linear program. Since the variables are already supposed to be non-negative, it was not necessary to add inequalities to the LP for this purpose.

5.1.1 Data

We tested our graph kernel on four datasets. The Mutag dataset consists of aromatic and heteroaromatic nitro compounds tested for mutagenicity. Nodes in the graphs represent the names of the atoms. The Enzymes dataset is a protein graph model of 600 enzymes from BRENDA database which contains 100 proteins each from 6 Enzyme Commission top-level classes (Oxidoreductases, Transferases, Hydrolases, Lyases, Isomerases, and Ligases). Protein structures are represented as nodes, and each node is connected to three closest proteins on the enzyme. The NCI1 dataset is derived from the PubChem website [pubchem.ncbi.nlm.nih.gov] which is related to screening of human tumor (Non-Small Cell Lung) cell line growth inhibition. Each chemical compound is represented by their corresponding molecular graph where nodes are various atoms (Carbon, Nitrogen, Oxygen etc) and edges are the bonds between atoms (single, double etc). The class labels on this dataset are either active or inactive based on the cancer assay. The PTC dataset is part of Predictive Toxicology Evaluation Challenge. This dataset is composed of graphs representing the chemical structure and their outcomes of biological tests for the carcinogenicity in Male Rats (MR), Female Rats (FR), Male Mice (MM) and Female Mice (FM). The task is to classify whether a chemical is POS or NEG in MR, FR, MM, and FM in terms of carcinogenicity.

5.1.2 Pre-processing and post-processing

We use the Chinese Postman algorithm (the description of the algorithm can be found in section 4.1) to make the input graphs Eulerian by adding the minimum amount of weights to the existing edges of the graphs. For directed graphs, we can use them directly in our algorithm, but for

undirected graphs, we consider two edges in opposite directions for each undirected edge and treat the two created opposite edges as separate variables in our linear programming problem.

Because our method requires edge labels, for those datasets such as Enzymes that have no edge labels, we use the concatenation of the source node label and the destination node label to make a label for every edge. To make the direction of the edge irrelevant, when we are comparing the two edge labels to see whether they match, we check both the equality of label of one to the label of the other or to the reverse label of the other edge which is obtained by reversing ordering of the source and destination nodes.

After computing the distance value between each pair of graphs, we have higher values for more distant (less similar) graphs. To prepare a normalized kernel to be used in other implemented classifiers like Support Vector Machines (SVM), we have to map initial values such that for more similar graphs we obtain higher values (1 for identical pairs). To make this transformation, we have used two simple methods, and for each dataset, we have used both of them and chose the one that gives us the best results during the cross-validation on the training set. Then, this chosen method is used on the test set to get the final accuracy. The first method is to use $f(x) = \frac{1}{x+1}$ as the map function. The second method is to use the function $f(x) = 1 - \frac{x-min}{max-min}$ to map the distance values. Here, the max and min show the maximum and minimum distance values that we have among all possible pairs of graphs. Since we get 0 for identical graphs, the min is always 0. Hence, the map function can be simplified to $f(x) = 1 - \frac{x}{max}$. Both methods will give us 1 for similar graphs that have GTED values of 0, and numbers between 0 and 1 for more distant graphs. The more distant the pair of graphs are, the less the corresponding value in the kernel will be.

5.1.3 Results

To evaluate whether this method works well at capturing the similarity and classifying the graphs, we used some benchmark datasets that are used to compare the graph kernels. We compare the kernels by evaluating the accuracy of an SVM classifier that uses them for classification. We used the same settings as in [44] so we can compare our results with previously computed results

for other kernels. In this setting, we split the data randomly into two parts, 80% for training and 20% for testing. Then, we computed results for 20 different splitting using different random seeds. It can be seen in Table 5.1 that for the Mutag [69] and Enzymes [11] datasets, our kernel outperforms the other kernels. In the results table, we copied the values in [44] for other kernels.

Table 5.1: Comparing the accuracy of GTED to that of other common graph kernels on some benchmark datasets.

Kernel/Dataset	Mutag [69]	Enzymes [11]	NCI1 [70]	PTC [71]
WL [49]	84.50($\pm$2.16)	53.75($\pm$1.37)	**84.76**($\pm$0.32)	59.97($\pm$1.60)
WL-Edge [50]	82.94($\pm$2.33)	52.00($\pm$0.72)	84.65($\pm$0.25)	60.18($\pm$2.19)
SP [41]	85.50($\pm$2.50)	42.31($\pm$1.37)	73.61($\pm$0.36)	59.53($\pm$1.71)
Graphlet [50]	82.44($\pm$1.29)	30.95($\pm$0.73)	62.40($\pm$0.27)	55.88($\pm$0.31)
p-RW [37]	80.33($\pm$1.35)	28.17($\pm$0.76)	TIMED OUT	59.85($\pm$0.95)
MLG [44]	84.21($\pm$2.61)	57.92($\pm$5.39)	80.83($\pm$1.29)	**63.62**($\pm$4.69)
GTED	**90.12**($\pm$4.48)	**59.66**($\pm$1.84)	65.83($\pm$1.14)	59.08($\pm$2.11)

5.1.4 Analysis

As shown in the table, our kernel achieves the highest accuracy on the Mutag and Enzymes datasets but gets an average result on PTC and relatively weaker result on NCI1, as compared to other methods. Actually, none of the existing kernels can get the best results on all different kinds of data because each kernel captures only some features of the graphs. The Eulerian traversals of the graphs can be very informative for some specific applications, like Mutag. The aromatic and heteroaromatic chemical compounds in Mutag mostly consist of connected rings of atoms. These constituent rings can give us a good measure of the proximity of two compounds. Since the language of Eulerian traversals includes the traversal of these rings in each compound, finding the minimum distance between the strings of the languages (which are built by the labels of the nodes that represent the name of atoms) for two different compounds can provide a measure of the

similar structures that they contain. That is why we get the best result for this dataset using our kernel.

Similarly, GTED outperforms the other kernels in the enzymes dataset. The enzymes in this dataset have certain shapes consisting of various protein structures (the nodes), and the combination of the individual structures and the nearby proteins gives us a good sense of the structure of the enzyme. In this case, Eulerian cycles usually give us a good approximation for the general spatial structure of the enzyme which leads to a good score.

The algorithm performed less well on the NCI1 and PTC data sets. We are uncertain of why this is, but it seems likely that the critical properties of the relevant chemicals are not captured by the Eulerian traversal.

5.2 Using GTED on genomic data

As mentioned earlier, the original goal of GTED was to find the best alignment of two genomes using only the assembly graphs, without having to create an assembled sequence first. The common alignment methods that compute the Levenshtein edit distance cannot take many factors into account, like having trans-locations in the genome, or the fact that assembly graphs could have multiple Eulerian cycles. Our method finds the best alignment among all possible alignments for all possible pairs of reference genomes that can be derived from the assembly graphs. As a result, it gives us a good measure to compute the distance (or similarity) between genomic sequences, and hence a way to cluster a group of samples. Therefore, to evaluate our method on genomic data, we chose genomes of Hepatitis B viruses in five different vertebrates; the virus in two of them (Heron and Tinamou) belong to Avihepadnavirus genus, and the ones in three of them (Horseshoe bat, Tent-making bat, and Woolly monkey) belong to Orthohepadnavirus genus.

5.2.1 Pre-processing and post-processing

First, for each pair of sequences we wished to compare, we generated a *colored de Bruijn graph*, a de Bruijn graph (assembly graph) that combines multiple samples in a single assem-

bly graph with k-mers from different samples identified using different colored edges. We then extracted the graph for each specific color (genome). The linear programming problem for this experiment is produced almost like before; the difference here is that instead of using the second set of constraints to enforce that all edges of the input graphs are used exactly as many times as their multiplicities (an Eulerian cycle), we add the absolute value of the difference of the number of times that an edge is used in the alignment graph and its original weight in the corresponding input graph to the objective function of the LP. This way, we try to minimize this difference but allow some discrepancies. The extra flexibility seems necessary in this case, because the input graphs are large and contain numerous sources of error: sequencing errors, using cutoffs for edges, and crude estimates of the weights of the edges based on the coverage of sequences in the colored de Bruijn graph mean that the edge multiplicities are not completely accurate.

5.2.2 Results

Numbers in Table 7.1 are the computed distance of each of these pairs of graphs. As represented in this table, it can be seen that the intra-genus distances are lower than inter-genus distances. We believe, based on these numbers, a good estimate of the similarity of the genomes can be made, both for genomes in the same genus and the ones with different genera.

Table 5.2: GTED values for pairs of corresponding assembly graphs of Hepatitis B viruses from different mammals. They belong to two different genera which are shown by gray color.

	Heron	Tinamou	Horseshoe bat	Tent-making bat	Woolly monkey
Heron	-	1016	1691	1639	1659
Tinamou	1016	-	1699	1638	1640
Horseshoe bat	1691	1699	-	1347	1296
Tent-making bat	1639	1638	1347	-	1429
Woolly monkey	1659	1640	1296	1429	-

Chapter 6

Extensions of GTED

As in the problem of finding optimal alignment between two strings where the notion of local alignment and fit alignment are essential to deal with different problems, for the alignment of two graphs, these notions can become useful to solve problems that differ from what the original GTED is designed for. In this chapter, we mention variants of the Needleman-Wunch algorithm that has been developed to solve the two new problems for pair of strings, and then, we demonstrate the modifications that have to be made to the original GTED in order to derive the analogous algorithms for graphs.

6.1 Local Graph Traversal Edit Distance

When analyzing genomic sequences, we might be interested in finding a conserved section in two different genes from the same or different species. The original Needleman-Wunch algorithm optimizes the score of the alignment between the whole sequences which might be low between two different genes even if they both contain a highly conserved subsequence. So, for solving this new problem, a new solution is required. First, let provide a formal definition for the problem of local alignment between a pair of strings.

Problem 2 (Local Alignment for Strings)**.** *If we represent the set of all subsequences of a string S by $\gamma(S)$, Given 2 strings S_1 and S_2, we define this problem as:*

$$d_l(S_1, S_2) = \min_{\substack{s_1 \in \gamma(S_1) \\ s_2 \in \gamma(S_2)}} d(s_1, s_2), \tag{6.1}$$

in which $d(s_1, s_2)$ is the Levenshtein edit distance between two strings s_1 and s_2. As mentioned earlier, edit operations are single alphabet symbol insertion, deletion, and substitution, and the Levenshtein edit distance is the minimum number of such operations to transform s_1 to s_2 (it is usually symmetric by considering an equal penalty for insertion and deletion operations).

To solve Problem 2, the Smith-Waterman algorithm has been introduced which has subtle differences with the Needleman-Wunch algorithm. Here, we have to change the initial values of the first row and first column of the table we discussed for computing the global alignment in section 3.2 to 0. Making this change will allow us to start at any position of the input strings without being penalized for the indels up to that point. Another change that that has to be made while filling in the values of the table is changing the negative values to 0; because instead of accumulating the negative scores (penalty) up to that point, we have the opportunity of starting at that position with a score of 0. To find a pair of substrings that are a solution to Problem 2, we have to find the maximum value in the table (which shows the end point of substrings on the corresponding input strings) and backtrace until a value 0 is found. This algorithm has the same time/space complexity as the Needleman-Wunch algorithm.

Now, we are going to define a variant of the original GTED, the problem of Local Graph Traversal Edit Distance, which is analogous to the problem of alignment for strings (Problem 2).

Problem 3 (Local Graph Traversal Edit Distance)**.** *Let A_1 and A_2 be two edge-labeled Eulerian graphs. We define the local edit distance between A_1 and A_2 by*

$$d_l(A_1, A_2) = \min_{\substack{s_1 \in \mathcal{C}(A_1) \\ s_2 \in \mathcal{C}(A_2)}} d(s_1, s_2), \tag{6.2}$$

where $\mathcal{C}(A_i)$ is the set of corresponding strings for all simple cycles in A_i for $i = 1, 2$, and $d(s_1, s_2)$ is the Levenshtein edit distance between two strings s_1 and s_2. [58].

To solve this problem using the solution to the original GTED, we should notice that the second set of constraints does not have to be fulfilled anymore. This means that the projection of the cyclic traversal of the alignment graph onto the constituent graphs no more needs to exhaust their edges. Therefore, removing the second set of constraints provides us with an optimum traversal in $\mathcal{AG}$ whose projections to A_1 and A_2 are cyclic traversals; this will be the best local alignment that we have between A_1 and A_2.

For local GTED, something that has to be noted is the necessity of assigning a negative value for the score of a match; otherwise, the solution to the LP will be an empty traversal, which has the optimum value of 0 (minimum possible distance). By assigning a negative value to a match, we encourage the local alignment to find larger cycles in the two graphs which represent more similar strings.

6.2 Fit Graph Traversal Edit Distance

There are other problems where we have to align a short sequence to a much longer sequence in order to find out if the longer sequence has a similar subsequence to the shorter one. This problem might arise in various applications, such as mapping RNA-seq data to a genome, sequencing using a reference genome, checking if a genome contains a subsequence similar to a highly persevered sequence in a family or genera, etc. We start by defining the problem of fit alignment on strings. Problem 4 formalizes this concept.

Problem 4 (Fit Alignment for Strings)**.** *If we represent the set of all subsequences of a string S by $\gamma(S)$, Given 2 strings S_1 and S_2, we define the problem of fitting a subsequence of S_2 to S_1 as:*

$$d_l(S_1, S_2) = \min_{s \in \gamma(S_2)} d(S_1, s), \tag{6.3}$$

in which $d(s_1, s_2)$ is the Levenshtein edit distance between two strings s_1 and s_2.

To solve Prob 4, similar to the solution to Problem 2, we have to change the initial values of the table we had in for global alignment in section 3.2; but, this time, we only change the values of the first row to 0; this allows us to start the alignment anywhere on the second string while considering the insertion errors from traversing a path along the first column because we have to completely use the first string in the alignment. Finally, to find the answer to Problem 4, we have to find the maximum value of the last row and trace back to an element of the table with value 0 (which happens only at the first row in this case). By finding the maximum value on the last row,

we allow the alignment to finish anywhere on the second string while forcing it to exhaust the first one.

In the following, we define a variant of the original GTED, the problem of Fit Graph Traversal Edit Distance, which is analogous to the problem of fit alignment on strings.

Problem 5 (Fit Graph Traversal Edit Distance)**.** *Let A_1 and A_2 be two edge-labeled Eulerian graphs. We define the local edit distance between A_1 and A_2 by*

$$d_f(A_1, A_2) = \min_{\substack{s \in \mathcal{C}(A_1) \\ \omega \in \mathcal{L}(A_2)}} d(s, \omega), \tag{6.4}$$

in which $\mathcal{C}(A_1)$ is the set of corresponding strings for all cycles in A_1 and $d(s, \omega)$ is the Levenshtein edit distance between two strings s and ω [58].

For solving this problem, again we are going to use the solution to the original GTED (Problem 1). Actually, the solution to this problem is very similar to the one for problem 4; here, we have to exhaust all the edges of A_2, but we can use only a subset of edges of A_1 which forms a cycle. So, by ignoring the projection constraints only for A_1, the optimum alignment that will be found by the LP will be the solution to problem5. However, this solution requires A_2 to be an Eulerian graph, and for general applications, this might not be the case. To solve this problem, we have to use the techniques discussed in section 4.2 to relax the projection constraints for A_2, and try to minimize the distance while trying to use as many edges of A_2 as possible in the alignment.

Now, we will use a small example to see how our solutions work in practice to find the optimum local and fit graph traversal edit distance. Consider the set of reads (strings) $R_1 = \{ACC$, $CCGTC$, $TCGTACC$, $CCTGTCG\}$. The de Bruijn graph for R_1 using $k = 3$ is depicted in Figure 6.1. Let also consider the string $s = CCGTACCG$. Obviously, s is not a member of R_1 or substring of any of its members; also, it cannot be generated by attaching any ordering of any subset of R_1. However, if we make a de Bruijn graph for this string, the subgraph of the original graph containing nodes 2, 4, 9, 8, and 1, along with the edges between them, will be formed. By using fit graph traversal edit distance, a distance of 0 will be derived (assuming values 0, 1, and 1

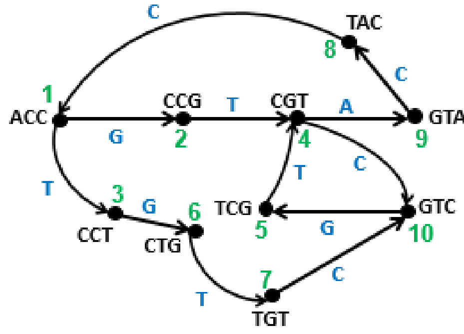

Figure 6.1: The de Bruijn graph of R_1 for $k = 3$

are used for match, mismatch, and indel respectively), which implies the presence of an Eulerian traversal of the corresponding graph of s in the de Bruijn graph of R_1. The same holds for the string $s = ACCGTACC$ which is another Eulerian traversal of the same subgraph the previous string formed. For this string, we get the same distance of 0 when fit GTED is used.

Now, we are going to introduce some indels to the string s that is supposed to be fitted to R_1. Let assume $s = TCGTCGTAC$. It is not obvious that what is the best fit alignment of this string to the de Bruijn graph of R_1. If we use the original fit GTED on the corresponding assembly graph of s and the graph of R_1, an unsolvable LP will be formed because the de Bruijn graph of s is not Eulerian. However, by using the technique introduced in section 4.2, which allows us to use GTED for non-Eulerian graphs by relaxing the projection constraints, a value of 3 (assuming values 0, 1, and 1 are used for match, mismatch, and indel respectively) is returned as the minimum distance which can be derived by traversing nodes 5, 4, 10, 5, 4, and 9. Although all the edges are present in the original graph, we traversed the edge $(5, 4)$ twice. To make the degree of the nodes 5 and 4 even so that the resulting graph becomes Eulerian, we have to ignore one of its repeats. Since this edge appears in the projection equations of both of its end nodes, a penalty of 2 will be added to the objective function. The other increment of 1 in the distance comes from the penalty for the edge $(4, 9)$ which has to be removed from the graph of s in order to make it Eulerian, and because the absolute value of the difference of its weight, which is 1, and its projection, which is now 0, is added to the objective function of the LP, the minimum distance increases by 1.

Chapter 7

More Efficiency!

In the original GTED, for two graphs A_1, A_2 with m_1, m_2 nodes and n_1, n_2 edges, the alignment graph has $m_1 \times n_2 + m_2 \times n_1 + n_1 \times n_2$ edges, and the resulting LP will have the same number of variables. Since the time complexity of an LP depends on the number of variables, for larger graphs like the de Bruijn graphs, solving the LP becomes very time-consuming. In this chapter, we introduce a method which helps us reduce the number of variables and making the final LP easier to solve, especially for the de Bruijn graphs. We start with some pre-assumption on the nodes that can be considered as a match. Fixating these corresponding nodes on the alignment graph will reduce the number of edges that are added to the alignment graph which results in less number of variables for the final linear programming problem. First, we explain the strategy for pruning the edges of the alignment graph and then, we illustrate the algorithm on a small graph. Afterward, we count the number of edges that will be removed in case of the de Bruijn graphs. Finally, we evaluate the gain in performance by running the method on some sample graphs and solving the resulting LP and comparing the time to that of the original method.

If based on some domain knowledge or heuristics we assume two nodes v_1 and v_2 of the input graphs $A_1 = (V_1, E_1, M_1, L_1, \Sigma_1)$ and $A_2 = (V_2, E_2, M_2, L_2, \Sigma_2)$ are corresponding nodes, we can fix them on the alignment graph ($\mathcal{AG}$). This means that among all the possible traversals of the alignment graph we only have to consider those that align these pair of nodes together. Based on the definition of the alignment graph, we know that for each pair of nodes of A_1 and A_2, there is one node on the alignment graph which we represent by $\mathcal{AG}_{v_1,v_2}$. Also, any traversal of $\mathcal{AG}$ provides us with one traversal on each of the input graphs; Any horizontal edge on this graph can be considered as a *pause* (staying on the same node) on the traversal of A_1 and a *move* (traversing an edge) in the traversal of A_2. Similarly, a vertical edge represents a pause on the traversal of A_2 and a move in the traversal of A_1. A diagonal edge corresponds to a move in the traversal of both graphs. So, for fixing the alignment of v_1 and v_2, the traversal on $\mathcal{AG}$ has to take the traversals of

A_1 and A_2 to the nodes v_1 and v_2 at the same time. Corresponding nodes of the nodes v_1 and v_2 in the alignment graph consist of the nodes of a row and column in $\mathcal{AG}$ respectively. If a traversal on $\mathcal{AG}$ enters this row or column, it has to happen at the intersection of them, which is $\mathcal{AG}_{v_1,v_2}$ (we call such node a *master node* from now on). Therefore, we can immediately eliminate all the edges of $\mathcal{AG}$ which are pointing to a node in this row and column except for the ones that are entering $\mathcal{AG}_{v_1,v_2}$.

As you can see in Figure 7.1, this technique could be applied with slight changes in case of having multiple options in one graph to be matched to a node in the other graph. In Figure $7.1(b)$, 2 different options u_1 and v_1 in graph A_1 are the only possible matches for node v_2 in A_2. In this case, we can again eliminate all the edges that enter a node in the corresponding column of v_2 in a point other than or $\mathcal{AG}_{v_1,v_2}$ and $\mathcal{AG}_{u_1,v_2}$; however, we cannot do the same for the two rows surrounded by dashed lines. The reason is that we do not know which one of the matches will provide us with the best score, and for any of the 2 possible matching the other row has to be treated as a regular row.

Figure $7.1(c)$ shows the a case where we have 2 nodes u_1, v_1 in A_1 and 2 nodes u_2, v_2 in A_2 such that either (v_1, v_2) and (u_1, u_2) are matching pairs or (v_1, u_2) and (u_1, v_2). Here, we can eliminate all the edges that enter one of the nodes in corresponding rows or columns except the ones entering one of the 4 intersection points (shown with red color in the figure). To see an application of this case you may consider comparing the structural graph of two molecules each with two Carbon atoms. To align these two molecules there are two separate ways of matching these Carbon atoms (assuming that we are certain that the best alignment makes a correspondence between the Carbon atoms of one molecule and those of the other one).

In general, it is easy to observe that if each node in a set $k_1 \subset V_1$ can be paired with any of the nodes in the set $k_2 \subset V_2$ and $|k_1| < |k_2|$ ($|k_1|$ and $|k_2|$ represent the cardinality of the sets k_1 and k_2), then all the edges that enter to a corresponding row to a node in k_1 at a node other than the intersections with the corresponding columns of the nodes in k_2 can be deleted from $\mathcal{AG}$. However, this procedure cannot be applied to any of the corresponding columns to the nodes in k_2 because

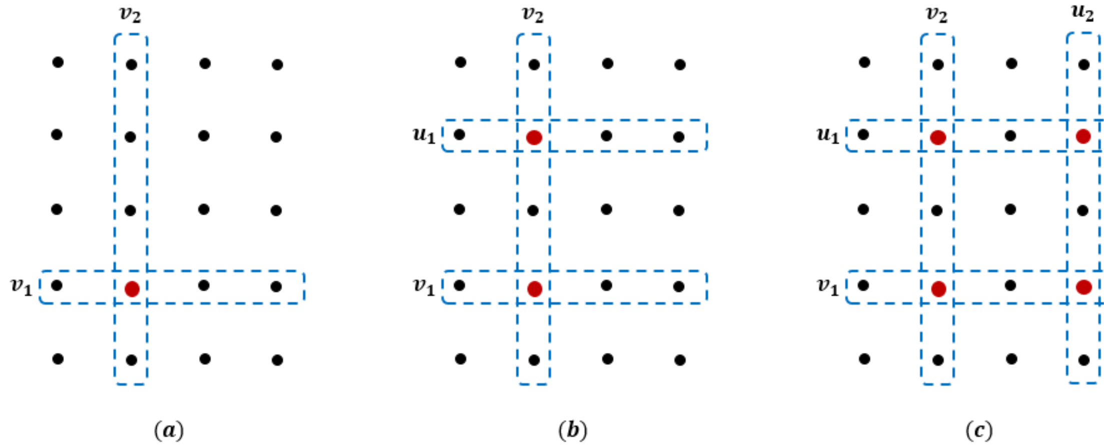

Figure 7.1: The nodes in a sample alignment graphs; the ones colored with red are the common nodes in the two input graphs.

no information is available about the nodes that will remain unmatched to one of the nodes in k_1. A similar reasoning can be made when $|k_1| > |k_2|$.

For de Bruijn graphs, however, we will only deal with case (a) of Figure 7.1. This applies to the case where it is highly likely that two nodes, which represent two k-mers or two contigs with length more than k in the graph, are a certain match. What makes this possibility available is having an exact match of the corresponding strings of these nodes. Of course, when two matching strings are longer, the probability of them being a correct match increases. If we want to find all the nodes in two de Bruijn graphs that have the same string, the procedure becomes very time-consuming. This difficulty comes from the fact that the string of a node in one graph can be a substring of the string of a node in the other graph. This case might happen because of the merging step in making de Bruijn graphs where we merge a node with only one outgoing edge to the succeeding node and make contigs. To overcome this difficulty, we make a *colored* de Bruijn graph (co-assembly graph) using HyDA [72]. It gets as input 2 different sets of sequencing reads and provides us with a single graph where nodes and edges contain substrings of one or both of the constituent graphs. In this graph, each edge has 2 weights, each for one of the contained graphs. In colored de Bruijn graphs the common contigs are formed when there is a sequence of nodes with only one outgoing edge such that each of those edges has a nonzero weight for each color. Finally, we prepared a script

to extract those two graphs from the single graph generated by HyDA and used as input graphs to make the alignment graph; in this procedure, we keep track of the common nodes which are the nodes that are present in at least one of the edges in each graph.

Here, we make the whole procedure clearer by showing how it works on a small toy example. Consider the two sets of reads $S_1 = \{ACC, CA\}$ and $S_2 = \{AC, CA\}$. By making a colored de Bruijn graph for these sets the graph in Figure 7.2(a) will be generated. In this graph, the blue and red numbers show the weights of the edges corresponding to S_1 and S_2 respectively. After extracting the 2 constituent graphs, we get the two graphs of Figure 7.2(b). Notice that we do not merge any two nodes after extracting the graphs because this will destroy what we were looking for in the first place by making the colored de Bruijn graph.

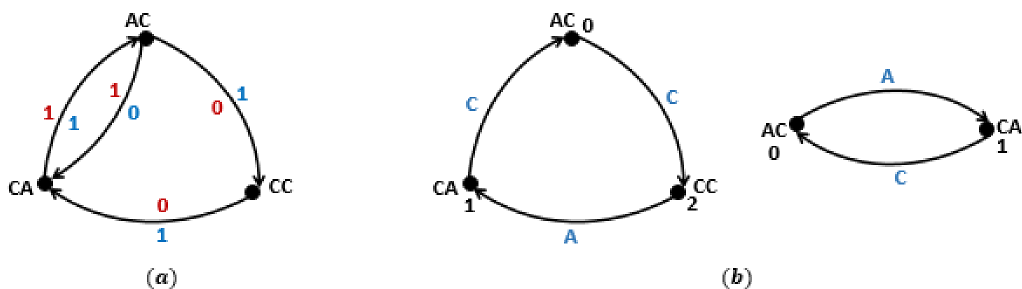

Figure 7.2: Part (a) shows a colored de Bruijn graph in which the weight of each edge for each color is shown. Part (b) shows the two graphs extracted from the graph in (a).

While extracting the two graphs, we mark the nodes that correspond to each other. So, we know that the nodes for AC are a match, as well as the nodes for CA. We can show this correspondence by using the same node numbers. Figure 7.3(a) shows the alignment graph for these 2 graphs. For each row and column, we have written the corresponding node numbers of the constituent graphs (the columns and rows correspond to the nodes of the graphs with 2 and 3 nodes respectively. In this graph, the edges that correspond to an insertion or deletion in the alignment are represented with the black color, the nodes that correspond to a mismatch are shown with red, and the ones that represent a match are shown with green.

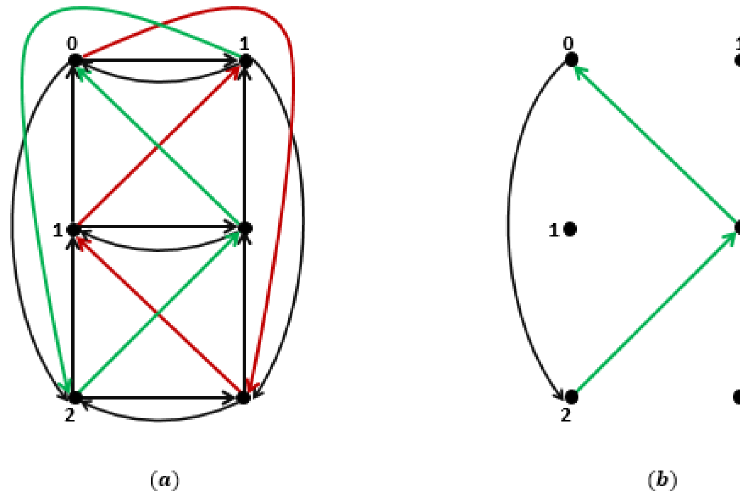

Figure 7.3: The graph in (a) shows the alignment graph of the two graphs in Figure 7.2(b). The one in (b) is the alignment graph after all the removes that happen because of the master nodes.

Figure 7.3(a), the nodes $\mathcal{AG}_{0,0}$ and $\mathcal{AG}_{1,1}$ are master nodes. As explained earlier, we have to force the entrance to the nodes that are corresponding to a common master node to happen at the same time. Therefore, we can eliminate all the edges pointing to a node in one of the corresponding rows or columns (other than the ones entering the master nodes). By doing so, only 3 edges remain in the graph (more details about the extra edges that are removed will be discussed in the following), which itself is the final answer to the LP; it is an Eulerian traversal of the alignment graph, which we have proved to be corresponding to an Elerian traversal in each of the constituent graphs. It also satisfies the second set of constraints which are assuring that all the edges in the constituent graphs are being used. Moreover, since this is the only traversal satisfying the constraints of the LP, it will be chosen as the answer to the optimization problem. Starting from node $(0, 0)$ on the alignment graph, the traversal in this graph is $(0, 0), (2, 0), (1, 1), (0, 0)$. This corresponds to the alignment CAC, $-AC$ which has two matches and one deletion; this can be also seen in Figure 7.3 from the colors assigned to the edges (in the code scores will replace the colors to compute the score of the alignment).

As you can see, the number of edges can be reduced noticeably depending on the number of master nodes in the graph. Actually, it is straightforward to count the total number of edges that get eliminated from the graph by having only one master node. Let consider two graphs $A_1 = (V_1, E_1)$ and $A_2 = (V_2, E_2)$ such that $|V_1| = m_1$, $|V_2| = m_2$, $|E_1| = n_1$, and $|E_2| = n_2$. As mentioned earlier, the total number of edges in the alignment graph is $m_1 \times n_2 + m_2 \times n_1 + n_1 \times n_2$. Now, suppose we introduce a master node to the graph by making a correspondence between the nodes $v_1 \in V_1$ and $v_2 \in V_2$. Also, let represent the indegree and outdegree of a node v in its graph by $I(v)$ and $L(v)$ respectively. The total number of vertical edges that get eliminated from the alignment graph is equal to the number of edges that enter node v_1 in A_1 (which shows the number of vertical edges that enter the corresponding row of v_1 in a single column of the alignment graph) multiplied by the number of columns, which is m_2. Notice that we used m_2 in the multiplication rather than $m_2 - 1$, which is the number of columns that do not correspond to v_2. In fact, we also counted the column which corresponds to v_2 because, as mentioned earlier, we require that entering to node v_1 and v_2 occurs simultaneously, so it has to be via a diagonal edge. We can write this number as $I(v_1) \times m_2$. Similarly, we can compute the number of horizontal edges that get eliminated from the alignment graph, which is $I(v_2) \times m_1$.

For computing the number of diagonal edges that get removed from the graph by introducing a master node, we should notice that each diagonal edge corresponds to one edge in each of the graphs A_1 and A_2. So, we have to count all the edges in $\mathcal{AG}$ for which either the corresponding edge in E_1 enters v_1 and the corresponding edge in E_2 does not enter v_2 or vice versa. The cardinality of the former case is $I(v_1) \times (n_2 - I(v_2))$ and the latter case $I(v_1) \times (n_2 - I(v_2))$. So, if $R(v_1, v_2)$ represents the number of edges in $\mathcal{AG}$ that are removed because of the master node (v_1, v_2), the equation 7.1 holds.

$$R(v_1, v_2) = I(v_1) \times m_2 + I(v_2) \times m_1 + I(v_1) \times (n_2 - I(v_2)) + I(v_1) \times (n_2 - I(v_2)) \quad (7.1)$$

Because in a de Bruijn graph the number of outgoing edges for each node is at most 4, the total number of removed edges for a single master node represented in equation 7.1 is $O(m_1 +$

$m_2 + n_1 + n_2$). What happens when we have more than one master node? Does the above number gets multiplied by the number of master nodes or many of them will be duplicates? Let consider 2 master nodes u and v which correspond to (u_1, u_2) and (v_1, v_2) respectively (as mentioned earlier in case of having a de Bruijn graphs, we know $u_1 \neq v_1$ and $u_2 \neq v_2$. In the following, we count the number of edges that are removed because of these master nodes.

The intersection of the sets of edges that get removed from the graph because of the master nodes is not empty! So, we do have duplicate removes in case of having more than one master node. According to how we make the alignment graph, any edge from a node in A_1 to u_1 will create a diagonal edge in the alignment graph when considered a corresponding edge for any of the edges of A_2, including the edges pointing to v_2. Also, we know that any of these edges that do not point to u_2 are going to be removed, including the ones pointing to v_2. On the other hand, when we are processing the master node V and removing the diagonal edges that point to v_2 but do not point to v_1, we are removing the edges of $\mathcal{AG}$ that correspond to an edge to v_2 in A_2 and an edge to u_1 in A_1, again. We also have this duplicate removing for edges in the alignment graph that point to v_2 in A_1 and u_1 in A_2. For the case of having more than 2 master nodes, we have similar duplicates for each of them. Hence, the total number of duplicate eliminations for k master nodes $V^1, V^2, ..., V^k$ where V^i corresponds to (v_1^1, v_2^2) for i: 1 to k will be:

$$\sum_{i=1}^{k}(k-1) \times I(v_1^i) + (k-1) \times I(v_2^i) = (k-1) \times \sum_{i=1}^{k} I(v_1^i) + I(v_2^i) \tag{7.2}$$

Since in the de Bruijn graph indegree of each node is limited to 4, the summation on the right side of equation 7.2 is $O(k)$, which makes the total duplicates in counting the number of eliminations $O(k^2)$ if when we multiply equation 7.1 by k to find the total number of edges that get eliminated by introducing k master nodes.

What about the horizontal and vertical edges? It is clear that edges of this kind that get removed because of these master nodes are mutually exclusive; the reason is that the end nodes of all those edges are different. But, we are not done! In case of having more than one master node, there will be a new set of horizontal and vertical edges that are removed; Those are the edges that correspond

to leaving u_1 and staying in v_2. Due to our constraint of entering a master node at the same time on the traversals of A_1 and A_2, these edges are not allowed because either u has to happen earlier or v, and whichever one occurs first, the other one has to happen together in the traversal of the graphs which means neither one can still be in the former master node while the other enters the latter one. So, the total number of edges that will be eliminated in addition to the number we get by multiplying equation 7.1 by k for k master nodes $V^1, V^2, ..., V^k$ where V^i corresponds to (v_1^1, v_2^2) for i: 1 to k will be:

$$\sum_{i=1}^{k}(k-1) \times L(v_1^i) + (k-1) \times L(v_2^i) = (k-1) \times \sum_{i=1}^{k} L(v_1^i) + L(v_2^i) \tag{7.3}$$

Since the outdegree of each node in a de Bruijn graph is at most 4, the summation on the right side of equation 7.3 is $O(k)$; therefore, the total number of edges that are added to the previously eliminated edges is $O(k^2)$. Hence, after multiplying equation 7.1 by k, the total number of edges that will be added has the same asymptotic complexity as the number of duplicate removed edges.

Considering how we make a de Bruijn graph, we know that every node has at most 4 ingoing and 4 outgoing edges. Therefore, the number of edges of a de Bruijn graph is at most 4 times the number of its nodes; this means $n_1 = O(m_1)$ and $n_2 = O(m_2)$. In case of analysis, if we assume the number of nodes in A_1 as almost the same as the number of nodes in A_2, and show it with m, then the total number of edges in $\mathcal{AG}$ is $O(m^2)$. Based on a similar analysis, we can simplify the complexity of equation 7.1 to $O(m)$. Considering the discussion we had for the case of having k master nodes, we know the number of removed edges from the alignment graph will be $O(km)$. Therefore, after removing these edges, the total number of edges of $\mathcal{AG}$ will be $O(m(m-k))$. This means that by having k master nodes, in general, $O(\frac{k}{m})$ of edges will be removed. However, the number of remained edges is still $O(m^2)$. Hence, it does not change the number of variables in the optimization problem significantly except for very similar graphs, like the ones belonging to the same family or species. In practice, however, the effect of this method is much more than expected. This is due to simplifying many of the equations in the constraints of the LP. After using this technique, many of the equations and therefore the values of some of the variables will be

resolved before a general algorithm for solving the LP, such as Simplex, Interior-Point Method, and Primal Simplex, gets started.

To show the time differences in practice, before and after using this technique, we have used the Gurobi optimizer tool which tries to simplify the problem as much as possible before starting the general solver algorithm. Table 7.1 shows the number of nodes and edges in 2 sample graph, along with the number of common nodes and the time it takes for their LP to be solved ($|\mathcal{AG}|$ shows the number of edges of $\mathcal{AG}$, $|\mathcal{M}|$ shows the number of master nodes, and $\mathcal{T}$ stands for time).

Table 7.1: Running times for 4 different pairs of graphs before and after removing the edges that can be eliminated because of the master nodes.

n_1	m_1	n_2	m_2	$\|\mathcal{M}\|$	$\|\mathcal{AG}\|$ (before)	$\|\mathcal{AG}\|$ (after)	$\mathcal{T}$ (before)	$\mathcal{T}$ (after)
2.5 k	2.6 k	2.4 k	2.4 k	192	18 M	16 M	7.2 k s	0.7 k s
3.1 k	3.1 k	2.9 k	2.9 k	169	27 M	25 M	19.6 k s	3.9 k s
0.4 k	0.4 k	0.4 k	0.4 k	192	591 k	252 k	98 s	1 s
0.5 k	0.5 k	0.5 k	0.5 k	169	717 k	375 k	153 s	3 s

Chapter 8

Future Work

Although GTED seems to be a promising measure for comparing two graphs, especially de Bruijn graphs, it is still limited to graphs of size several thousand. For the original motivation which is differential genome assembly, this bound on the size of the graphs limits us to small viruses which are not challenging to study with the current methods in the literature. Even the method introduced in chapter 7 does not make the computations fast enough to leverage the utility of GTED to the realm of bacteria. Therefore, smarter techniques are necessary to either reduce the time complexity of the algorithm or make us able to divide the whole problem into smaller substances and solve them recursively. For the latter one, the notion of master nodes might become useful given certain conditions. This technique is currently under study.

GTED provides us with a distance measure whereas for graph kernels a similarity measure is required. Therefore, the original values have to be altered in order to be used in a graph kernel. Different ways for changing these measures to one another has been introduced. Hence, further exploration of the results of each can be conducted. This choice can even be considered as a hyper-parameter which can be tuned for each dataset during the cross-validation step.

As explained earlier, different values can be used as the score for a match, mismatch, and indel. This provides room for optimizing these scores to meet our needs. The authors plan to consider these values as additional hyper-parameters when GTED is used as a graph kernel for classification. This might further increase the classification accuracy on the benchmark datasets.

As we discussed in section 5.1, the performance of GTED varies considerably on the benchmark datasets that are used in our experiment. Although some general and intuitive analysis is provided in that section, a more thorough study of the properties that makes GTED the preferable choice as a graph kernel seems necessary. However, a run of GTED on a sample of the dataset should be enough in most cases to provide the user with a good estimate on the performance of this method on the dataset of interest.

Deep learning is becoming the most used approach of machine learning in various domains. Therefore, a more comprehensive comparison of the graph kernels used in our study to the state-of-the-art methods in deep learning, especially the ones that belong to the thriving area of Graph Convolutional Networks that are designed for performing the task of learning on graphs, seems an interesting experiment. Also, investigating the possibility of making hybrid methods that gain the benefits of the two would be beneficial.

Chapter 9

Conclusion

In this thesis, we have introduced GTED, a new method for comparing networks based on a traversal of their edge labels. Although the brute-force solution can be impractical due to the exponential number of Eulerian traversals, we have shown that GTED admits a polynomial time algorithm using a linear program. This linear program is guaranteed to have an integer solution due to the fact that the boundary operator function is totally unimodular, giving us an exact solution for the minimum possible edit distance.

The GTED problem was originally designed to be a formalization of the differential genome assembly problem, comparing DNA assembly graphs by considering all their possible assembled strings. It performs well at that task, successfully differentiating different genera of the Hepatitis B virus. We tested GTED on viral genomes since GTED is a global alignment scheme that is not immediately scalable to full-size large genomes, like all other global alignment schemes such as Needleman-Wunsch. However, GTED can form the mathematical basis for scalable heuristic comparison of full-size large genomes in the future. GTED can also be used as a general graph kernel on other types of networks, performing particularly well on graphs whose Eulerian traversals provide a good insight into their important structural features.

GTED not only has the potential to be a valuable tool in the study of biological networks, but it also provides us with a new way of measuring the similarity between networks. It has many applications in differential genome assembly, but it also performs well in domains beyond assembly graphs. As an example of these applications, we used it as a graph kernel for solving some benchmark classification problems, and a higher accuracy was achieved for some of them.

Although GTED works based on the Eulerian traversals, two different approaches to make make it applicable to non-Eulerian graphs were fully discussed. We also defined the problems of local alignment and fit alignment based on the Eulerian traversals, and explained in detail the

changes that have to be made to the original linear programming formulation in order to solve these problems.

Finally, we have provided a technique for optimizing the computation of GTED based on a pre-assumption on the nodes of the graphs that are a certain match. We counted the number of edges that are eliminated in terms of the number of such nodes, and by measuring the running time of the solving GTED we showed the improvement is even more than the portion of the eliminated edges due to simplifying the constraints and making increasing the number of ones that get pre-solved.

Bibliography

[1] Neil C Jones and Pavel Pevzner. *An introduction to bioinformatics algorithms*. MIT press, 2004.

[2] Y. Li *et al.* Structural variation in two human genomes mapped at single-nucleotide resolution by whole genome *de novo* assembly. *Nature Biotechnology*, 29:723–730, 2011.

[3] Narjes S. Movahedi, Elmirasadat Forouzmand, and Hamidreza Chitsaz. De novo co-assembly of bacterial genomes from multiple single cells. In *IEEE Conference on Bioinformatics and Biomedicine*, pages 561–565, 2012.

[4] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de bruijn graphs. *Nat Genetics*, 44:226–232, 2012.

[5] Z. Taghavi, N. S. Movahedi, S. Draghici, and H. Chitsaz. Distilled single-cell genome sequencing and de novo assembly for sparse microbial communities. *Bioinformatics*, 29(19):2395–2401, Oct 2013.

[6] Narjes S. Movahedi, Mallory Embree, Harish Nagarajan, Karsten Zengler, and Hamidreza Chitsaz. Efficient synergistic single-cell genome assembly. *Frontiers in Bioengineering and Biotechnology*, 4(42), 2016.

[7] F. Hormozdiari, I. Hajirasouliha, A. McPherson, E. Eichler, and S. Cenk Sahinalp. Simultaneous structural variation discovery among multiple paired-end sequenced genomes. *Genome Research*, 21:2203–2212, 2011.

[8] C. Mak. Multigenome analysis of variation (research highlights). *Nature Biotechnology*, 29, 2011.

[9] S. Jones. True colors of genome variation (research highlights). *Nature Biotechnology*, 30, 2012.

[10] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354, 2003.

[11] Karsten M. Borgwardt, Cheng Soon Ong, Stefan Schönauer, S. V. N. Vishwanathan, Alex J. Smola, and Hans-Peter Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(1):47–56, January 2005.

[12] Pierre Mahé, Nobuhisa Ueda, Tatsuya Akutsu, Jean-Luc Perret, and Jean-Philippe Vert. Graph kernels for molecular structure- activity relationship analysis with support vector machines. *Journal of chemical information and modeling*, 45(4):939–951, 2005.

[13] Benoit Gaüzere, Luc Brun, and Didier Villemin. Two new graphs kernels in chemoinformatics. *Pattern Recognition Letters*, 33(15):2038–2047, 2012.

[14] Luc Brun, Donatello Conte, Pasquale Foggia, Mario Vento, and Didier Villemin. Symbolic learning vs. graph kernels: An experimental comparison in a chemical application. In *ADBIS (Local Proceedings)*, pages 31–40, 2010.

[15] H. Kubinyi. Drug research: myths, hype and reality. *Nat Rev Drug Discov*, 2(8):665–668, Aug 2003.

[16] Kaspar Riesen. Structural pattern recognition with graph edit distance. *Advances in computer vision and pattern recognition, Cham*, 2015.

[17] Giorgio Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4):341, 1973.

[18] Horst Bunke, Pasquale Foggia, Corrado Guidobaldi, Carlo Sansone, and Mario Vento. A comparison of algorithms for maximum common subgraph on randomly connected graphs. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 123–132. Springer, 2002.

[19] Horst Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(8):689–694, 1997.

[20] Horst Bunke and Kim Shearer. A graph distance metric based on the maximal common subgraph. *Pattern recognition letters*, 19(3-4):255–259, 1998.

[21] Walter D Wallis, Peter Shoubridge, M Kraetz, and D Ray. Graph distances using graph union. *Pattern Recognition Letters*, 22(6-7):701–704, 2001.

[22] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[23] Pasquale Foggia, Gennaro Percannella, and Mario Vento. Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(01):1450001, 2014.

[24] Xiaoyi Jiang and Horst Bunke. Optimal quadratic-time isomorphism of ordered graphs. *Pattern Recognition*, 32(7):1273–1283, 1999.

[25] Peter J Dickinson, Horst Bunke, Arek Dadej, and Miro Kraetzl. On graphs with unique node labels. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 13–23. Springer, 2003.

[26] Michel Neuhaus and Horst Bunke. An error-tolerant approximate matching algorithm for attributed planar graphs and its application to fingerprint classification. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 180–189. Springer, 2004.

[27] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision computing*, 27(7):950–959, 2009.

[28] MA Eshera and King-Sun Fu. A graph distance measure for image analysis. *IEEE transactions on systems, man, and cybernetics*, (3):398–408, 1984.

[29] Salim Jouili, Salvatore Tabbone, and Ernest Valveny. Comparing graph similarity measures for graphical recognition. In *International Workshop on Graphics Recognition*, pages 37–48. Springer, 2009.

[30] Maria C Boeres, Celso C Ribeiro, and Isabelle Bloch. A randomized heuristic for scene recognition by graph matching. In *International Workshop on Experimental and Efficient Algorithms*, pages 100–113. Springer, 2004.

[31] Sébastien Sorlin and Christine Solnon. Reactive tabu search for measuring graph similarity. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 172–182. Springer, 2005.

[32] Michel Neuhaus, Kaspar Riesen, and Horst Bunke. Fast suboptimal algorithms for the computation of graph edit distance. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 163–172. Springer, 2006.

[33] Derek Justice and Alfred Hero. A binary linear programming formulation of the graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(8):1200–1214, 2006.

[34] Zhiping Zeng, Anthony KH Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment*, 2(1):25–36, 2009.

[35] Benoit Gaüzère, Sébastien Bougleux, Kaspar Riesen, and Luc Brun. Approximate graph edit distance guided by bipartite matching of bags of walks. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 73–82. Springer, 2014.

[36] Andreas Fischer, Ching Y Suen, Volkmar Frinken, Kaspar Riesen, and Horst Bunke. Approximation of graph edit distance based on hausdorff matching. *Pattern Recognition*, 48(2):331–343, 2015.

[37] T. G"artner. Exponential and geometric kernels for graphs. In *NIPS 2002 Workshop on Unreal Data*, 2002. Principles of modeling nonvectorial data.

[38] John Shawe-Taylor, Nello Cristianini, et al. *Kernel methods for pattern analysis*. Cambridge university press, 2004.

[39] S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. Graph kernels. *J. Mach. Learn. Res.*, 11:1201–1242, August 2010.

[40] Hisashi Kashima, Koji Tsuda, and Akihiro Inokuchi. Marginalized kernels between labeled graphs. In *Proceedings of the 20th international conference on machine learning (ICML-03)*, pages 321–328, 2003.

[41] K. M. Borgwardt and H. P. Kriegel. Shortest-path kernels on graphs. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 8 pp.–, Nov 2005.

[42] Aasa Feragen, Niklas Kasenburg, Jens Petersen, Marleen de Bruijne, and Karsten Borgwardt. Scalable kernels for graphs with continuous attributes. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 216–224. Curran Associates, Inc., 2013.

[43] Risi Kondor and Karsten M. Borgwardt. The skew spectrum of graphs. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 496–503, New York, NY, USA, 2008. ACM.

[44] Risi Kondor and Horace Pan. The multiscale laplacian graph kernel. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2990–2998. Curran Associates, Inc., 2016.

[45] SVN Vishwanathan, Karsten M Borgwardt, Nicol N Schraudolph, et al. Fast computation of graph kernels. In *NIPS*, volume 19, pages 131–138, 2006.

[46] Pierre Mahé, Nobuhisa Ueda, Tatsuya Akutsu, Jean-Luc Perret, and Jean-Philippe Vert. Extensions of marginalized graph kernels. In *Proceedings of the twenty-first international conference on Machine learning*, page 70. ACM, 2004.

[47] Tamás Horváth, Thomas Gärtner, and Stefan Wrobel. Cyclic pattern kernels for predictive graph mining. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 158–167. ACM, 2004.

[48] Jan Ramon and Thomas Gärtner. Expressivity versus efficiency of graph kernels. In *First international workshop on mining graphs, trees and sequences*, pages 65–74. Citeseer, 2003.

[49] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, November 2011.

[50] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. Efficient graphlet kernels for large graph comparison. In David van Dyk and Max Welling, editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 488–495, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR.

[51] Marion Neumann, Roman Garnett, Christian Bauckhage, and Kristian Kersting. Propagation kernels: efficient graph kernels from propagated information. *Machine Learning*, 102(2):209–245, 2016.

[52] Ali Ebrahimpour Boroojeny, Akash Shrestha, Ali Sharifi-Zarchi, Suzanne Renick Gallagher, S Cenk Sahinalp, and Hamidreza Chitsaz. Gted: Graph traversal edit distance. In *Interna-*

tional Conference on Research in Computational Molecular Biology, pages 37–53. Springer, 2018.

[53] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. U.S.A.*, 98:9748–9753, Aug 2001.

[54] P. A. Pevzner, H. Tang, and G. Tesler. De novo repeat classification and fragment assembly. *Genome Res.*, 14(9):1786–1796, Sep 2004.

[55] Roy Ronen, Christina Boucher, Hamidreza Chitsaz, and Pavel Pevzner. SEQuel: improving the accuracy of genome assemblies. *Bioinformatics*, 28(12):i188–i196, 2012. Also ISMB proceedings.

[56] E. W. Myers. Toward simplifying and accurately formulating fragment assembly. *J. Comput. Biol.*, 2:275–290, 1995.

[57] J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26:367–373, Jun 2010.

[58] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics-Doklady*, 10(8):707–710, 1966. Original in *Doklady Akademii Nauk SSSR* 163(4): 845–848 (1965).

[59] W.T. Tutte and C.A.B. Smith. On unicursal paths in a network of degree 4. *The American Mathematical Monthly*, 48(4):233–237, 1941.

[60] T. van Aardenne-Ehrenfest and N.G. Bruijn. Circuits and trees in oriented linear graphs. In Ira Gessel and Gian-Carlo Rota, editors, *Classic Papers in Combinatorics*, Modern Birkhäuser Classics, pages 149–163. Birkhäuser Boston, 1987.

[61] T. van Aardenne-Ehrenfest and N.G. de Bruijn. Circuits and trees in oriented linear graphs. In Ira Gessel and Gian-Carlo Rota, editors, *Classic Papers in Combinatorics*, Modern Birkhäuser Classics, pages 149–163. Birkhäuser Boston, 1987.

[62] T. Dey, A. Hirani, and B. Krishnamoorthy. Optimal homologous cycles, total unimodularity, and linear programming. *SIAM Journal on Computing*, 40(4):1026–1044, 2011.

[63] James W Vick. *Homology theory: an introduction to algebraic topology*, volume 145. Springer, 1994.

[64] William Massey. *A basic course in algebraic topology*, volume 127. Springer, 1991.

[65] Jack Edmonds and Ellis L Johnson. Matching, euler tours and the chinese postman. *Mathematical programming*, 5(1):88–124, 1973.

[66] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[67] Marcin Mucha and Piotr Sankowski. Maximum matchings via gaussian elimination. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 248–255. IEEE, 2004.

[68] Edwin KP Chong and Stanislaw H Zak. *An introduction to optimization*, volume 76. John Wiley & Sons, 2013.

[69] Asim Kumar Debnath, Rosa L Lopez de Compadre, Gargi Debnath, Alan J Shusterman, and Corwin Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of medicinal chemistry*, 34(2):786–797, 1991.

[70] Nikil Wale, Ian A Watson, and George Karypis. Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowledge and Information Systems*, 14(3):347–375, 2008.

[71] H. Toivonen, A. Srinivasan, R. D. King, S. Kramer, and C. Helma. Statistical evaluation of the predictive toxicology challenge 2000-2001. *Bioinformatics*, 19(10):1183âĂŞ1193, Jan 2003.

[72] Seyed Basir Shariat Razavi, Narjes Sadat Movahedi Tabrizi, Hamidreza Chitsaz, and Christina Boucher. HyDA-Vista: Towards optimal guided selection of k-mer size for sequence assembly. *BMC Genomics*, 15(Suppl 10):S9, 2014. Also GIW/ISCB-Asia proceedings.

Appendix A

Strongly Connected Components

One of the main differences between the directed graphs and undirected graphs is having ordered pairs rather than unordered pairs when speaking of paths and distances. This roots from the fact that each edge can be traversed in only one direction. This, itself, results in the property a path from a node u to v in a. Definition 5 formalizes this notion.

Definition 5. *In a directed graph $G = (V, E)$, a Strongly Connected Component (SCC) is a subgraph of G, $G\prime = (V\prime, E\prime)$, such that for every two nodes $u, v \in V\prime$ there is at least one path from u to v and at least one path from v to u. A Maximal Strongly Connected Component is an SCC that cannot be augmented by adding any subset of nodes in $V - V\prime$ and any subset of edges in $E - E\prime$ while it remains an SCC.*

As an example, consider the directed graph in Figure A.1. In this graph we have 3 Maximal SCCs with set of nodes $A = \{0, 1, 2, 3\}$, $B = \{4\}$, and $C = \{5, 6, 7\}$. Notice that from any node in A there exists a path to any node in B, but no node in B has a path to the nodes in A. Similarly, from any node in C there exists a path to any node in the other two sets whereas neither of the nodes in A and B has a path to a node in C.

Now, the question is how one can find Maximal SCCs in a reasonable time. Let start with some Depth First Search (DFS) traversal of the graph in Figure A.1. Before we start, let make a small change to the DFS algorithm; we stop whenever the stack becomes empty. Now, if we start from the node in B, node 4 will be marked and then this run of DFS stops because node 4 has no outgoing edge. What if we start from a node in A? DFS will find every node in A (because A is an SCC) and also the node in B. If we start DFS from a node in C it will find all the nodes in the graph. Notice, if we keep track of the nodes that are visited to exclude them from the succeeding runs of DFS, we will find our 3 maximal SCCS by the 3 mentioned runs of DFS, in the same order.

Although we were able to find the SCCs of the graph by several runs of DFS, not all the choices of the start points and their ordering will lead to the desired solution. In our aforementioned solution, we started from a maximal SCC that had no path to any other SCC (if the whole graph is not an SCC such maximal SCC exists because otherwise the whole graph is a single SCC and this can be proved by contradiction). If we traverse this SCC and remove all its nodes and edges from the graph, we can recursively find all the maximal SCCs. To find an ordering of the starting nodes, one can make the transpose of the graph; for a directed graph $G = (V, E)$ its transpose is the graph $G^T = (V, E^T)$ where $(v, u) \in E^T$ if and only if $(u, v) \in E$. Then, we run DFS on the transpose of the graph and use timestamps whenever we pop a node from the stack. Finally, we start running DFS on the original graph in the decreasing order of timestamps while keeping track of the visited nodes. [**?**] The complexity of this algorithm is $\theta(V + E)$

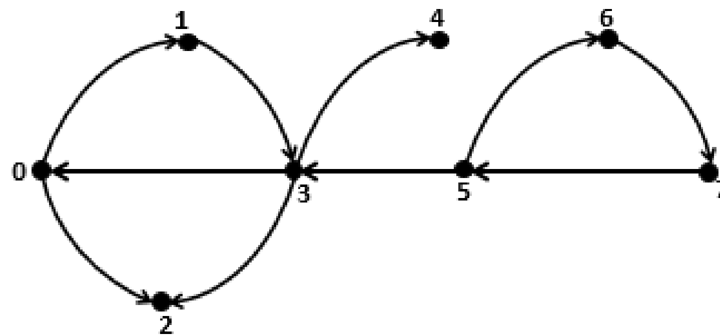

Figure A.1: A directed graph with 3 maximal SCCs