TÉCNICO LISBOA

Large Scale and Dynamic Phylogenetic Inference from Epidemic Data

Marta Alexandra Fragoso Nascimento

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Alexandre Paulo Lourenço Francisco
Prof. Cátia Raquel Jesus Vaz

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Alexandre Paulo Lourenço Francisco
Members of the Committee: Prof. Luís Manuel Silveira Russo

November 2017

Acknowledgments

I am using this opportunity to express my gratitude to everyone who supported me throughout the course of this Master Thesis.

I would like to express my special thanks to my supervisors Prof. Alexandre Francisco and Prof. Cátia Vaz for their insight, support and sharing of knowledge that has made this Thesis possible. The opportunity and knowledge acquired express my gratitude.

To João Carriço and fellow researchers at INESC-ID for introducing me to interesting problems in the bioinformatics' field. I learned a lot and working with them has been a great experience.

I would also like to thank to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life.

Last but not least, to my family for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. Thank you.

Abstract

Typing methods are widely used in the surveillance of infectious diseases, outbreak investigation and studies of the natural history of an infection. Their use is becoming standard, in particular with the introduction of High Throughput Sequencing (HTS). On the other hand, the data being generated is massive and many algorithms have been proposed for phylogenetic analysis of typing data, addressing both correctness and scalability issues, such as the goeBURST algorithm. Most of the distance based algorithms for inferring phylogenetic trees follow the closest-pair joining scheme. This is one of the approaches used in hierarchical clustering. Although phylogenetic inference algorithms may seem rather different, the main difference among them resides on how one defines cluster proximity and on which optimization criterion is used. The main goal of this thesis is the study of the most well known phylogenetic inference methods suitable for processing typing data, focusing mostly on the goeBURST algorithm and its computational problems that appear when dealing with large datasets. Moreover, this algorithm must however be run whenever new data becomes available starting from scratch. We address this issue by proposing two dynamic algorithms allowing data to be continuously integrated and updated. Experimental results show that these algorithms are efficient on integrating new data and updating inferred evolutionary patterns, improving the update running time by at least one order of magnitude.

Keywords

Phylogenetic inference; Phylogenetic trees; Dynamic algorithms; Sequence-based typing data.

Resumo

Os métodos de tipagem são vastamente utilizados uma vez que fornecem conhecimento importante na vigilância de doenças infecciosas, investigação de surtos e na história natural de uma infecção. E o seu uso está-se a tornar bastante comum, em particular com a introdução de High Throughput Sequencing (HTS). Por outro lado, a quantidade de dados que é gerada é enorme e muitos algoritmos têm sido propostos para realizarem a análise filogenética desses dados, abordando problemas de correção e escalabilidade, como é o caso do algoritmo goeBURST. Grande parte dos algoritmos baseados em distâncias que inferem árvores filogenéticas seguem um esquema de junção do par mais próximo. Esta é uma das abordagens utilizadas em *clustering* hierárquico. Apesar dos algoritmos de inferência filogenética parecerem bastante diferentes, a principal diferença entre eles reside apenas no facto de como cada um define a proximidade a um *cluster* e que critério é que usam. O objetivo principal desta tese é o estudo dos métodos mais utilizados que permitem realizar inferência filogenética, com foco maior no algoritmo goeBURST e os problemas computacionais que advém da sua utilização a grande escala. Além disso, este algoritmo deve ser executado sempre que novos dados fiquem disponíveis e a partir do zero. Este problema é abordado através da proposta de dois algoritmos dinâmicos que permitem que os dados sejam continuamente integrados e atualizados. Os resultados experimentais mostram que esses algoritmos são eficientes na integração de novos dados e na atualização dos padrões evolutivos inferidos, melhorando o tempo de execução das atualizações em pelo menos uma ordem de grandeza.

Palavras Chave

Inferência filogenética; Árvores filogenéticas; Algoritmos dinâmicos; Dados de tipagem baseados em sequências.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

BFS	Breadth-First Search
BME	Balanced Minimum Evolution
CL	Complete-Linkage
DFS	Depth-First Search
DLV	Double Locus Variant
DNA	Deoxyribonucleic Acid
FNJ	Fast Neighbor-Joining
GCP	Globally Closest Pairs
GLS	Generalized Least-Squares
GME	Greedy Minimum Evolution
HTS	High Throughput Sequencing
JC	Jukes–Cantor
LCP	Locally Closest Pair
MLST	Multilocus Sequence Typing
MLVA	Multilocus Variable Number of Tandem Repeats Analysis
MSF	Maximum Weight Forest
MST	Minimum Spanning Tree
NJ	Neighbor-Joining
OLS	Ordinary Least-Squares

OTU Operational Taxonomic Units

RNA Ribonucleic Acid

SLV Single Locus Variant

SL Single-Linkage

SNP Single Nucleotide Polymorphism

ST Sequence Type

TLV Triple Locus Variant

UNJ Unweighted Neighbor-Joining

UPGMA Unweighted Pair Group Method with Arithmetic-mean

WLS Weighted Least-Squares

WPGMA Weighted Pair Group Method with Arithmetic-mean

1

Introduction

Contents

Biological sequences, namely Deoxyribonucleic Acid (DNA), Ribonucleic Acid (RNA) and proteins, have a central role in molecular biology because they define almost every cellular activity that occurs in each organism. The key to uncover these processes relies in understanding how these sequences interact with each other in their own environment. In order to do that, it is necessary to sequence the DNA and identify the genes that are part of the genome. For this, a database with known genes for that specific organism is used and a matching is made based on some specific allele according to the chosen typing method. This way, and by the comparison between different genes, it is now possible to identify its lineage, and infer evolutionary relationships.

The choice of which typing method to use depends on the epidemiological context. These methods provide further knowledge on surveillance of infectious diseases, outbreak investigation and the natural history of an infection. Furthermore, with the introduction of "High Throughput Sequencing (HTS)" [1] technology, there has been developed new typing methods like ribossomal-Multilocus Sequence Typing (MLST) [2], Multilocus Variable Number of Tandem Repeats Analysis (MLVA) [3] or analysis of Single Nucleotide Polymorphism (SNP) [4] through comparison against a reference genome. Moreover, these recent advances and the resulting decrease in costs that allow the analysis of thousands of data, created the need for developing new efficient methods that are able to do phylogenetic analysis and that also allow the dynamic updating of data into an ongoing study.

The number of practical applications of phylogenetic analysis continues to grow and are by no means limited to the functional and structural study of genomes. Phylogenetic applications span much of biology, from human health [5] and forensics [6, 7] to conservation biology [8] and studies of behavior [9]. Many of these applications require accurate phylogenetic estimates, not only in terms of tree topologies, but also in branch lengths (for estimation of time and/or the amount of change), ancestral character states (for estimation of evolutionary transitions), and parameters of evolutionary models (for study of evolutionary processes). Therefore, phylogenetic analysis has become central to our understanding of biodiversity, evolution, ecology, and genomes.

1.1 Objectives

The aim of this thesis is to study some well-known algorithms for the phylogenetic inference processing large scale epidemic data. These methods have been proposed before the actual massive growth of epidemic data and, hence, most of existing implementations are facing great challenges with existing datasets. On the other hand, these methods rely often on graph theory and network mining ideas, which have been evolving in recent years, namely in what concerns dynamic graphs. Also, if we are dealing with missing data in large-scale phylogenomic datasets we may want to try different predictions since making the wrong ones will lead to negative effects on the phylogenetic inference process [10]. Hence, the challenge is to apply some techniques on two of those algorithms for phylogenetic inference, namely

the distance-based-Minimum Spanning Tree (MST) and goeBURST Full MST algorithms [11], allowing data to be continuously integrated and updated without requiring us to compute everything from start (*i.e.* dynamically update metrics and phylogenetic trees). Hence, it is expected that they construct the same phylogentic trees.

This work and related implementations will rely on, and extend the PHYLOViZ framework [12, 13].

1.2 Document Structure

This document starts by providing an overview of phylogenetic analyses regarding the main methods used to infer evolutionary relationships among different organisms and the clustering techniques they use to build the resulting phylogenetic tree. We also provide a common framework as open source software and documented at `https://gitlab.com/martanascimento/phyloviz/wikis/home`.

In the following chapters 3 and 4 we apply dynamic clustering techniques to two algorithms, distance-based-MST and goeBURST Full MST algorithms, that are available through PHYLOViZ. It also is explained which are the approaches that were used to address and achieve all that, along with their implementation details. We start in chapter 3 by providing a full description of both PHYLOViZ algorithms and then we provide a solution to allow dynamic integration of data on each. In chapter 4, we have the implementation details and we also identify possible computational bottlenecks and improvements that can be made to the implementation. Part of this work is already published [14] and the dynamic algorithms are available at `https://gitlab.com/martanascimento/dynamic-phyloviz/wikis/home`.

In chapter 5 we apply the methods developed to several public available datasets and we discuss the results regarding time and memory computational costs. And finally, chapter 6 ends this thesis with some final remarks and suggestions for future work.

2

Background

Contents

Phylogenetic analysis aims at uncovering the evolutionary relationships between different species (Operational Taxonomic Units (OTU)), to obtain an understanding of their evolution. Phylogenetic trees are widely used to address this task and are reconstructed by several different algorithms [15]. However, a phylogenetic tree will not always suffice to correctly represent the evolutionary history of a population and sometimes a network representation will be more appropriate [16]. Phylogenetic networks provide an alternative to phylogenetic trees and may be more suitable for datasets whose evolution involve significant amounts of reticulate events caused by hybridization, horizontal gene transfer, recombination, gene conversion or gene duplication and loss [17]. However, they are hard to analyze and thus phylogenetic trees are more common. Therefore, we are focusing our work on phylogenetic trees and on the algorithms that are commonly used to reconstruct them.

This chapter provides an overview of phylogenetic analyses. Beginning with a concise introduction to clustering, phylogenetic inference and some of the main methods used to infer phylogenetic trees. Different classifications and approaches are then presented for those methods.

The focus here is on distance-based analysis of DNA sequences, where the Hamming distances between pairs of sequences are computed [18]. These distances are then subject to a distance correction that is based on some appropriate model of evolution (also known as substitution model). The resulting distance matrix is then provided to a method to reconstruct a phylogenetic tree.

2.1 Clustering

Clustering is an unsupervised learning problem [19]. Given a set of elements the goal is to group them in such a way that elements in the same group (called a cluster) are more related (similar) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, bioinformatics, data compression, and computer graphics.

The main problems in clustering are how to define the similarity between elements and then find the clusters efficiently. Similarities are described according to several models like connectivity models, centroid models, distribution models, graph-based models, and others [20]. Here our focus is only on connectivity models. Connectivity models define the similarity between elements as their distance (*i.e.* elements are more similar with nearby elements than with elements farther away) and that is why this type of clustering is also known as distance-based clustering. Algorithms that implement this model differ from one another by the way distances are computed.

Clustering can be divided into two types: hierarchical clustering and flat (or partitioning) clustering.

Hierarchical clustering seeks to build a hierarchy of clusters. There are two techniques used to build the hierarchy: agglomerative and divisive. Agglomerative is a “bottom up” approach where each

element is in its own cluster and, as the hierarchy moves up, a pair of clusters is merged into one. Divisive clustering is a "top-down" approach where all elements are together in one cluster and, as the hierarchy moves down, a cluster is split in two.

Flat clustering tries to build a group of clusters all independent from each other (*i.e.* there is no relation among them) and can also be divided in two categories: hard and soft clustering. Hard clustering computes a hard assignment, where each element is a member of exactly one cluster, while the latter computes a soft assignment, where each element's assignment is a probability distribution over all clusters (*i.e.* an element can belong to several clusters).

We will concentrate our study on hierarchical clustering that use a "bottom up" approach to build the hierarchy. This type of clustering has been extensively used in bioinformatics and computational biology, namely in phylogenetic inference within most phylogenetic tree reconstruction algorithms.

2.2 Phylogenetic Inference

Phylogenetics is the study of the evolutionary history and relationships among individuals or groups of organisms (*e.g.* species, or populations) where the result is a phylogenetic network or tree. As we mentioned before we are only going to discuss this regarding phylogenetic trees.

These relationships are discovered through phylogenetic inference methods that evaluate observed heritable traits, such as DNA sequences or morphology under a model of evolution. These models try to describe the evolution process of the species from which a sequence of symbols changes into another set of traits and differ in terms of the parameters used to describe the rates at which one nucleotide replaces another during evolution. For instance, they are used during the calculation of likelihood of a tree (in Bayesian and Maximum Likelihood approaches to tree estimation) or to estimate the evolutionary distance between sequences from the observed differences. This enables us to infer evolutionary events that happened in the past, and also provides more information about the evolutionary processes operating on sequences.

Evolution models can be classified as neutral, independent and finite-sites [21]. A model is considered *neutral* if the majority of evolutionary changes are caused by genetic drift (dominated by random processes) rather than natural selection. This means that the change in the frequency of a site in a population is due to random sampling of organisms instead of being related with its beneficial or deleterious effect. The most commonly used neutral models are infinite alleles model [22] and the stepwise mutation model [23]. They can also be considered *independent* if changes in one site do not affect the probability of changes in another site and *finite-site* if there are finitely many sites, and so over evolution, a single site can be changed multiple times independently of each other. The simplest and most well-known finite sites model is the Jukes–Cantor (JC) model [24] where all positions are equally likely to mutate and the mutant is chosen with equal probability among the three possible nucleotides. This model was

later modified by Kimura [25] to accommodate the fact that transition events ($A \leftrightarrow T$ and $C \leftrightarrow G$) occur at a faster rate than all other events. Both models are unrealistic in the sense that all nucleotides are expected to occur with the same frequency in a random sequence, which is not likely to be the case for any sequence. For that reason, more sophisticated models have been introduced to account for subtle differences in substitution rates(*e.g.* Felsenstein [26], Hasagawa [27], etc.).

We can also impose a process of mutation on top of the these models, and independently of the model, by assigning a mutation event with probability μ on each gene. For instance, under the finite sites model a site is chosen randomly and a mutation occurs in that position according to a defined mutation process [21].

These models can assume a *molecular clock* if the expected number of substitutions is constant regardless of which species' evolution is being examined and can be *time-reversible* if it does not care which sequence is the ancestor and which is the descendant so long as all other parameters (*e.g.* number of substitutions) are held constant.

There are several methods that construct phylogenetic trees and they can all be seen as clustering methods because they apply several clustering techniques in their approach. Note also that the goal of phylogenetic analyses is to discover relationships between species or populations by grouping them based on some similarity criterion that underlies some evolution model.

These methods can be classified by the type of data they use to build the tree or by the type of tree search algorithm.

2.2.1 Classification by Type of Data

Phylogenetic trees can be built using either distance matrix methods or character-state methods. *Distance matrix* methods infer the relationship between individuals as the number of genetic differences between pairs of sequences, whereas in *character-state* methods is used an array of character states. Nucleotide sequences and amino acid sequences are the two representative character states used for phylogeny construction (*e.g.* adenine (A), cytosine (C), guanine (G), thymine (T), or gap in the case of multiple alignment of nucleotide sequences) [28]. In table 2.1 we can observe some examples of methods according to this classification however the focus of this work is on distance matrix methods.

Distance matrix methods compute a distance matrix D and then a phylogenetic tree T. This matrix D can be built directly from the pairwise distance between two sequences or from exposing them to a distance correction according to some model of evolution.

The simplest approach to compute D is to use the *normalized* Hamming distance $H(A, B)$, defined as the proportion of positions at which two aligned sequences A and B differ. This distance does not take into consideration the number of back mutations[1] and multiple mutations that occurred at the same

[1] back mutation is a mutation that restores the original sequence and hence the original phenotype.

Table 2.1: Classification by type of data.

Classification	Data	Method
Distance matrix methods	Set of $n(n-1)/2$ distance values for n OTUs	UPGMA WPGMA Neighbor-Joining Minimum Deviation[1] Minimum Evolution Distance Parsimony ...
Character-state methods	Array of character states (e.g. nucleotide sequences, amino acid sequences, ...)	Maximum Parsimony Maximum Likelihood Bayesian ...

[1] Also known as Fitch and Margoliash's method.

position and therefore it underestimates the true evolutionary distance. To rectify this, a correction formula based on some model of evolution is often used [17]. For example, in the case of the Jukes-Cantor model, it assumes all substitutions are independent, equal base frequencies (*i.e.* all sequence positions are equally subject to change), equal mutation rates and no insertions or deletions have occurred. Given the proportion of positions that differ between two sequences A and B, $H(A,B)$, the Jukes-Cantor estimate of the evolutionary distance (in terms of the expected number of changes) between two sequences is given by eq. (2.1).

$$JC(A,B) = -\frac{3}{4}\ln(1-\frac{4}{3}H(A,B)) \tag{2.1}$$

So, if we assume that those sequences evolved according to the Jukes-Cantor model of evolution, to compute a distance matrix that approximates the true evolutionary distances, we first determine the *normalized* Hamming distance between any two sequences A and B and then apply this transformation to get a corrected value.

An artificial example of a matrix D, built directly from the pairwise distance (*unnormalized* Hamming distance), is presented in matrix (2.1) where each entry represents the pairwise distance, D_{ij}, between elements i and j where each element is an OTU (*i.e.* belong to a specie or population). It is easy to understand that this matrix is symmetric because $D_{ij} = D_{ji}$.

The phylogenetic tree that represents a distance matrix depends on the chosen method and evolution model. Figure 2.1 shows two artificial examples of different types of trees built by two different methods in which the first uses a correction formula based on the minimum evolution criterion (explained later on section 2.3.2) and the latter assumes a molecular clock.

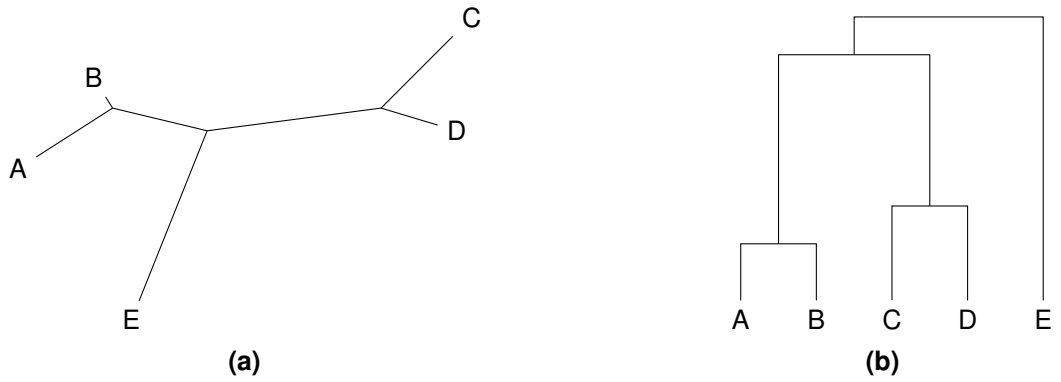

Figure 2.1: Examples of phylogenetic trees: (a) an unrooted tree built by the Neighbor-Joining method and (b) a rooted tree, or dendrogram, built by the UPGMA method.

$$\mathbf{D} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 2 & 7 & 7 & 6 \\ 2 & 0 & 7 & 7 & 6 \\ 7 & 7 & 0 & 5 & 5 \\ 7 & 7 & 5 & 0 & 3 \\ 6 & 6 & 5 & 3 & 0 \end{bmatrix} \end{array} \tag{2.1}$$

Phylogenetic trees can be additive or ultrametric. A tree $\mathcal{T}$ is considered to be *additive* if every distance between two different elements in the tree, $D_{ij}^{\mathcal{T}}$, reflects the exact distance in the matrix (*i.e.* $D_{ij}^{\mathcal{T}} = D_{ij}$). This means that the result of summing all branch lengths in the tree $\mathcal{T}$ between elements i and j needs to be the exact value present in the matrix. An *ultrametric* tree, fig. 2.1 (b), is an additive tree in which all elements are equidistant from the root. For this reason, methods that produce this kind of tree require a constant mutation rate over all sequences (*i.e.* the rate at which various types of mutations occur over time must be constant).

2.2.2 Classification by Type of Tree Search Algorithm

As mentioned before, different methods can build different trees. Therefore, we can classify those trees by the strategy that each method uses to find the best tree topology: exhaustive search methods and stepwise clustering methods [29]. Some examples of these methods are presented in table 2.2.

The strategy used by *exhaustive search* methods is to examine all or a large number of possible tree topologies and choose the best one according to a certain criterion. However, when having hundreds of sequences, these methods will not be able to compute and compare all possible topologies within a reasonable amount of time. Taking that into consideration, a new strategy called *completely bifurcating*

Table 2.2: Classification by Type of Tree Search Algorithm.

Classification		Method
Exhaustive Search method	Standard	Minimum Deviation Minimum Evolution Maximum Parsimony Maximum Likelihood ...
	Completely Bifurcating Tree Search method	
Stepwise Clustering method		Neighbor-Joining UPGMA WPGMA Single-Linkage Complete-Linkage “distance Wagner” ...

tree search method was developed where the number of tree topologies is limited [30] . Various measures are computed for a limited number of tree topologies to find the best tree in methods belonging to this category.

The minimum deviation method uses the number of differences between observed and estimated distances (“percent standard deviation”) as criterion of choosing the best tree topology [31]. The estimated distance between a pair of OTUs for a given tree topology is obtained by summing all branches connecting these two OTUs.

Another criterion is the minimization of the sum of branch lengths (principle of minimum evolution). This principle is used in the minimum evolution method where the tree topology which has the smallest sum of branch lengths is searched [32]. Neighbor-Joining (NJ) method also uses this principle of minimum evolution (by using least-squares approach for estimating the sum of branch lengths), although it is a stepwise clustering method while the minimum evolution method is a completely bifurcating tree search method. The basic procedure of this method is first to obtain the neighbor-joining tree (Saitou and Nei [33]) and then to search for a tree with the minimum value of the sum of branch lengths by examining all trees that are closely related to the NJ tree [34].

The maximum parsimony method is related to the principle used in minimum evolution methods since it tries to minimize (or maximize parsimony) the number of changes on character states, such as nucleotide or amino acid sequences, on the given tree topology [35].

The maximum likelihood method uses standard statistical techniques for inferring probability distributions to assign probabilities to particular possible phylogenetic trees [36].

Finally, *stepwise clustering* methods recursively examine a local relationship between different OTUs and find the best one. Most of distance matrix methods use this strategy as we can see in table 2.2.

Algorithm 2.1: General scheme for hierarchical agglomerative clustering methods based on distance matrices.

Input: A matrix D over a set of elements S (OTUs).
Output: A cluster-hierarchy $\mathcal{H}$ over S.
Initialization: Initialize the cluster-set C by defining a singleton cluster $C_i = \{i\}$ for every element $i \in S$. Initialize output hierarchy $\mathcal{H} \leftarrow C$.
Loop: While $|C| > 1$ do:

1. **Cluster-pair selection:** Select a pair of distinct clusters $\{C_i, C_j\} \subseteq C$ from D, according to some criterion.
2. **Cluster-pair joining:** Remove C_i, C_j from the cluster set C and replace them with $C_u = \{C_i \cup C_j\}$. Add C_u to the hierarchy $\mathcal{H}$ and calculate the branch length for each element (D_{iu} and D_{ju}).
3. **Reduction:** Update matrix D by calculating the new values (C_{uk}) for every $C_k \in C' \setminus \{C_i \cup C_j\}$.

Finalize: Return the hierarchy $\mathcal{H}$.

2.3 Distance Matrix Methods

Distance matrix methods take as input the pairwise distances for a set of OTUs, represented by a matrix D, possibly corrected according to some evolution model. The actual tree is then computed from the distance matrix by running a clustering algorithm that starts with the most similar sequences (Globally Closest Pair) or by trying to minimize the total branch length of the tree (Minimum Evolution). Table 2.3 provides a categorization of most well known phylogenetic inference methods. All these methods are variations of algorithm 2.1.

2.3.1 Globally Closest Pairs methods

Globally Closest Pairs (GCP) based algorithms are widely used in phylogeny. They receive as input a dissimilarity matrix containing all pairwise differences between elements and return a hierarchy of clusters.

First, a pair of clusters is chosen based on the minimum dissimilarity criterion over the matrix D. If a tie occurs, the selection between those cluster-pairs is arbitrary. Then, the selected pair (C_i, C_j) is removed from the set, joined together to form one single cluster ($C_u = \{C_i \cup C_j\}$) and added to the hierarchy $\mathcal{H}$. The distance between each element of the selected pair to the new cluster (D_{iu} and D_{ju}) is $D_{ij}/2$. Finally, in the reduction step 3, all dissimilarities from C_i and C_j to any other element need to be recalculated taking into account the new cluster C_u in order to update the dissimilarity matrix. The algorithm ends when there are no more clusters to join.

Unweighted Pair Group Method with Arithmetic-mean (UPGMA), Weighted Pair Group Method with Arithmetic-mean (WPGMA), Single-Linkage (SL) andComplete-Linkage (CL) are different variants of GCP, and they all differ on the reduction formula used in step 3:

Table 2.3: Clustering Algorithms

		Distance-matrix	Tree	Complexity	Proposed by
Globally Closest Pairs (Molecular-Clock)	UPGMA	Ultrametric	Rooted (*e.g.* dendrogram)	$O(n^2)$	Sokal and Michener[37]
	WPGMA			$O(n^2)$	Sokal and Michener[37]
	SL			$O(n^2)$	Sibson[38]
	CL			$O(n^2)$	Defays[39]
	UPGMC			$O(n^2)$	Sneath and Sokal[40]
	WPGMC			$O(n^2)$	Sneath and Sokal[40]
Minimum Evolution	NJ	Additive	Unrooted	$O(n^5)$	Saitou and Nei[33]
	NJ			$O(n^3)$	Studier and Kepler[41]
	BIONJ			$O(n^3)$	Gascuel[42]
	UNJ			$O(n^3)$	Gascuel[43]
	RapidNJ			$O(n^3)$	Simonsen, Mailund, and Pedersen[44]
	ERapidNJ			$O(n^3)$	Simonsen, Mailund, and Pedersen[45]
	QuickTree			$O(n^3)$	Howe, Bateman, and Durbin[46]
	QuickJoin			$O(n^3)$	Mailund and Pedersen[47]
	NINJA			$O(n^3)$	Wheeler[48]
	FastJoin			$O(n^3)$	Wang, Guo, and Xing[49]
	ClearCut (RNJ)			$O(n^3)$	Sheneman, Evans, and Foster[50]
	FastME			$O(n^3)$	Lefort, Desper, and Gascuel[51]
	FNJ			$O(n^2)$	Elias and Lagergren[52]
	GME			$O(n^2)$	Desper and Gascuel[53]
	BME			$O(n^3)$	Desper and Gascuel[53]
	goeBURST			$O(n^2)$	Francisco, Bugalho, Ramirez, and Carriço[54]
	FHP			$O(n^2)$	Foulds, Hendy, and Penny[55]

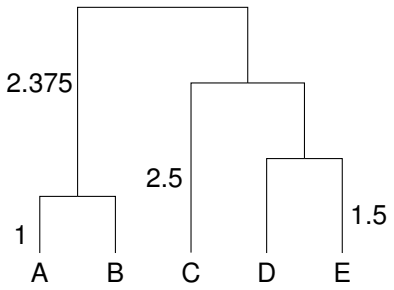

Figure 2.2: UPGMA phylogenetic tree regarding matrix 2.1.

UPGMA:

$$D((C_i \cup C_j), C_k) \leftarrow \frac{|C_i|}{|C_i| + |C_j|} D(C_i, C_k) + \frac{|C_j|}{|C_i| + |C_j|} D(C_j, C_k) \tag{2.2}$$

WPGMA:

$$D((C_i \cup C_j), C_k) \leftarrow \frac{1}{2}(D(C_i, C_k) + D(C_j, C_k)) \tag{2.3}$$

Single-Linkage:

$$D((C_i \cup C_j), C_k) \leftarrow min\{D(C_i, C_k), D(C_j, C_k)\} \tag{2.4}$$

Complete-Linkage:

$$D((C_i \cup C_j), C_k) \leftarrow max\{D(C_i, C_k), D(C_j, C_k)\} \tag{2.5}$$

UPGMA defines the similarity between two clusters as their average dissimilarity (eq. (2.2)). WPGMA defines it by weighting the two clusters by $1/2$ (eq. (2.3)). Single-linkage chooses the minimum value and complete-linkage the maximum (eqs. (2.4) and (2.5)).

As mentioned before, despite their major advantage ($O(n^2)$ running time) they can construct an erroneous tree because they assume a constant evolutionary rate (*i.e.* constant rate of mutations over time and for all lineages in the tree) [56]. Therefore, these methods often generate a wrong tree topology since in reality the OTUs branches are very unlikely to have the same mutation rate.

An example of the phylogenetic tree built by UPGMA regarding the distance matrix (matrix 2.1) is presented in fig. 2.2.

2.3.2 Minimum Evolution principle

Clustering methods try to find the optimal tree using different evolution models. The resulting tree may be less accurate or inconsistent when the assumed evolutionary model is wrong. The importance of these models are not just because of their consequences in phylogenetic analysis, but also because the characterization of the evolutionary process. Some models use the minimum evolution principle, where a tree is considered to be optimal if it has the shortest total branch lengths.

In order to get the minimum evolution tree it must be decided how the branch lengths are estimated and then how the tree length is calculated from these branch lengths. As it will be demonstrated next, NJ simply defines the tree length as the sum of all branch lengths, regardless of whether they are positive or negative. In practice, branch lengths are usually estimated within the least-squares framework and can be classified as Ordinary Least-Squares (OLS), Weighted Least-Squares (WLS) and Generalized Least-Squares (GLS). If all distance estimates can be assumed to be independent and to have the same variance, *OLS* should be used. If distance estimates are independent but could have different variances, then *WLS* or *GLS* should be considered, with *GLS* not imposing any restriction and being able to benefit from the covariances of the distance estimates [53].

The neighbor-joining is the most common method and has been widely used in phylogeny inference. It is an agglomerative clustering algorithm, which produce a tree in a bottom-up approach by iteratively combining OTUs. All variants of this method try to improve on its accuracy in reconstructing the true phylogenetic tree and also by trying to decrease its computational cost. Here we describe and present the most widely used variants of this method.

2.3.2.1 Neighbor-Joining and variants

Neighbor-Joining algorithm is the most commonly used method in phylogenetics and several variants of this algorithm have been introduced over the years. While some try to optimize the formulas used by NJ to better estimate the true and optimal tree others try to improve NJ's efficiency, both in terms of running time and memory usage. We will start by explaining NJ along with the variants that differ on the formulas they use and then a brief description of the ones that seek to reduce the overall complexity.

NJ (by Studier and Kepler [41]), Unweighted Neighbor-Joining (UNJ), BIONJ, Fast Neighbor-Joining (FNJ) and ClearCut methods are variants of NJ (by Saitou and Nei [33]) that differ from the latter by applying different criteria over the three steps of the generalized algorithm 2.1.

These algorithms take as input a dissimilarity matrix D and then create a new matrix Q based on the application of some criterion over D; Q is then used in the step 1 of algorithm 2.1 as criterion, being updated along D. This criterion relies on the minimum evolution principle. Original NJ (by Saitou and Nei) defines Q as minimizing the least-squares length estimate of the tree (eq. (2.6)).

$$Q_{ij} = \frac{1}{2}D_{ij} + \frac{1}{2(r-2)} \sum_{k=1,k\neq i,j}^{r} (D_{ik} + D_{jk}) + \frac{1}{r-2} \sum_{k=1,k\neq i,j}^{r} \sum_{l=k,l\neq i,j}^{r} D_{kl}, \tag{2.6}$$

where r represents the number of OTUs in D. Studier and Kepler [41] propose replacing the Saitou and Nei criterion [33] by eq. (2.7).

$$Q_{ij} = (r-2)D_{ij} - \sum_{k=1,k\neq i,j}^{r} D_{ik} - \sum_{k=1,k\neq i,j}^{r} D_{jk} \tag{2.7}$$

This new criterion has the advantage of leading to a complexity of $O(n^3)$ while the first leads to $O(n^5)$. These two criteria were proven to be equivalent by Vach and Degens [57], with several implementations available [13].

FNJ method uses a similar criterion as the latter NJ (eq. (2.7)), but instead of choosing the minimum in Q chooses from a different set (called *visible set*), of size $O(n)$. Given i, this set contains all *visible pairs* (C_i, C_j) such that $C_j = \min\{\sum_{k=1,k\neq i}^{r} Q_{ik}\}$. BIONJ method uses a simple first-order model of the variances and covariances of evolutionary distance, $Q_{ij} = D_{ij}/l_s$, where l_s represents the sequence length. At each step it allows the selection of the pair which minimizes the variance of the dissimilarity matrix. ClearCut relaxes the requirement of exhaustively searching the input matrix at each step to find the closest pair of nodes to join. Although it determines and joins the pair of nodes that are closer to each other, it does so while matrix updates due to joins are delayed. Hence candidate nodes may not be the closest pair of nodes in NJ sense. All the remaining variants use the same criterion defined by Studier and Kepler [41], minimizing the least-squares estimate of the tree length.

Branch lengths are computed in step 2. UNJ (eq. (2.9)) differs from the remaining (eq. (2.8)) because it uses an unweighted version of the equation used by NJ.

$$D_{iu} = \frac{1}{2}D_{ij} + \frac{1}{2(r-2)}\left[\sum_{k=1,k\neq i,j}^{r}(D_{ik} - D_{jk})\right], \quad D_{ju} = D_{ij} - D_{iu} \tag{2.8}$$

$$D_{iu} = \frac{1}{2}D_{ij} + \frac{1}{2(|S| - |C_u|)}\left[\sum_{k=1,k\neq i,j}^{r}|C_k|(D_{ik} - D_{jk})\right], \quad D_{ju} = D_{ij} - D_{iu} \tag{2.9}$$

It can be easily observed that this equation can be obtain from NJ by setting $|C_k| = 1$ for all $k \neq i, j$ and by replacing $(|S| - |C_u|)$ by $\sum_{k\neq i,j}|C_k|$ which is then equal to $(r-2)$.

Finally, in step 3, the dissimilarity matrix is reduced by deleting a pair of OTUs (i, j) and by estimating new distances between the new OTU u to any other OTU k (D_{uk}) using the general reduction formula:

$$D_{uk} = \lambda(D_{ik} - D_{iu}) + (1-\lambda)(D_{jk} - D_{ju}) \tag{2.10}$$

where D_{iu} and D_{ju} are given by either eq. (2.8) or (2.9), and λ is the weight that each algorithm assigns to each branch. NJ method defines λ by giving an equal weight to both original branch lengths (D_{ik} and D_{jk}), i.e., $\lambda = \frac{1}{2}$. But Saitou and Nei [33] do not consider the newly computed branches, i.e., $D_{iu} = D_{ju} = 0$, whereas Studier and Kepler [41] do. UNJ defines the weight λ as proportional to the number of OTUs contained in the clusters and hence gives the same weight to each of these OTUs,

$\lambda = \frac{|C_i|}{|C_i|+|C_j|}$. And BIONJ assigns the weight according to their variance:

$$\lambda = \frac{1}{2} + \frac{\sum_{k=1,k\neq i,j}^{r} (V_{jk} - V_{ik})}{2(r-2)V_{ij}} \tag{2.11}$$

with $\lambda \in [0, 1]$.

Regarding the reduction of Q, all methods that use NJ criterion need to update the entire matrix using the same equations that were used at the beginning (eqs. (2.6) and (2.7)), while BIONJ just needs to calculate the new values between the new OTU u to any other OTU k (D_{uk}) using the eq. (2.12), that is identical to eq. (2.10), but instead of using the distance matrix it uses the variance matrix:

$$V_{uk} = \lambda(V_{ik} - V_{iu}) + (1-\lambda)(V_{jk} - V_{ju}) \tag{2.12}$$

FNJ method also uses the same equation as NJ (eq. (2.10)) but has an additional step for updating the *visible set* by removing the previously selected *visible pair* and adding a new *visible pair* for C_u.

The complexity $O(n^3)$ of neighbor-joining method is directly related with the selection and reduction steps because it joins the two clusters that are closest to each other in $O(n^2)$ and then updates the distance matrix with the new distance values, at each step, for n OTUs.

Regarding neighbor-joining efficiency, several methods were developed to address running time and memory issues [58]. These include QuickJoin by Mailund and Pedersen [47]. NINJA by Wheeler [48]. RapidNJ by Simonsen et al. [44], using heuristics to find the next two elements to join avoiding the exploration of all possible pairs. ERapidNJ by Simonsen et al. [45], through reducing the memory requirements of RapidNJ. FastJoin by Wang et al. [49], decreasing the search time and the updating time of the distance matrix. QuickTree by Howe et al. [46], making use of low data-structure overhead and using an heuristic for handling redundant data. Moreover, regarding QuickJoin, RapidNJ and NINJA methods, they all produce the same phylogenetic trees as NJ but improve the running time by using some techniques to find the globally minimum value of sum matrix rather than by traversing the whole sum matrix in each iteration. Although all these methods take $O(n^3)$ running time in the worst case, in practice they perform faster.

Another optimization that can be made is the application of different techniques for topological rearrangements in the tree. Usually, they are used by heuristic algorithms like FastME by Lefort et al. [51], which searches for an optimal tree structure. These techniques are Nearest Neighbor Interchanges and Subtree Pruning and Regrafting. Nevertheless, and regardless of the implemented optimization technique, their ultimate goal is always trying to obtain the most accurate tree efficiently.

An example of a phylogenetic tree obtained with NJ with the criterion by Studier and Kepler [41] is presented in fig. 2.3.

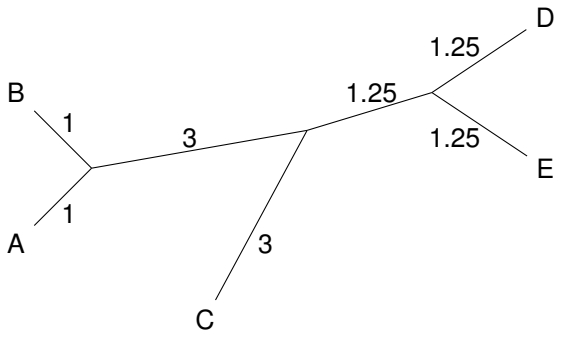

Figure 2.3: Phylogenetic tree obtained with NJ (Studier and Kepler [41]) for matrix 2.1.

2.3.2.2 GME and BME

Greedy Minimum Evolution (GME) and *Balanced Minimum Evolution (BME)* addition algorithms are stepwise algorithms that at each step insert a new element to a growing tree [53]. This approach differs from the last ones because instead of having a selection criteria for choosing the best element to be added in a fixed position in the tree, it has a selection criteria for choosing the best position in the tree to add any element. These algorithms add iteratively every element into the tree by minimizing the cost of insertion that is calculated for every edge in the tree. The scheme for both of these is presented in algorithm 2.2 and they both try to obtain the minimum evolution tree by minimizing the total tree length.

While GME algorithm uses OLS framework enabling it to be faster than the most common minimum evolution algorithms, BME uses a new version of OLS that tries to simplify the tree length computation that ends up being more appropriate but slower than the OLS version. In this new version, sibling subtrees have equal weight, as opposed to the OLS, where the weight of a subtree is equal to the number of OTUs it contains. This balanced version can also be seen as a weighted version of OLS, just like WPGMA is the weighted version of UPGMA.

Given an ordering on the OTUs, denoted as $(1, 2, 3, \ldots, n)$, for $k = 4$ to n, a tree $\mathcal{T}_k$ is created. This is done by testing each edge of $\mathcal{T}_{k-1}$ as a possible insertion point for k and comparing them using the OLS (Ordinary Least Squares) minimum evolution criterion in GME and the balanced minimum evolution criterion in BME. Then, k is inserted on any edge e of $\mathcal{T}_{k-1}$, that edge e is removed and the length of every other already existing edge is changed. After this, it is required the computation of the length of the three newly created edges.

The *tree length* $l(\mathcal{T})$ is defined as the sum of all edge lengths $l(e)$ of $\mathcal{T}$. In GME, the minimum evolution tree is an OLS minimum evolution tree that minimizes the length of the tree $l(\mathcal{T})$ and where $\mathcal{T}$ has the OLS edge length estimates. In BME, the definition for tree length holds but instead of using OLS distance it uses a balanced distance between subtrees. Given two subtrees S_i and S_j, if they are subtrees with only one OTU ($S_i = \{i\}$ and $S_j = \{j\}$) then the dissimilarity comes directly from

Algorithm 2.2: Minimum Evolution scheme.

Input: A dissimilarity matrix D between distant-2 subtrees S (*i.e.* if A and B are two disjoint subtrees, with roots a and b respectively, we'll say that A and B are distant-k subtrees if there are k edges in the path from a to b).
A tree $\mathcal{T}_k$ containing three subtrees.
k, number of already processed OTUs ($k = 3$).
An array P with the number of OTUs in each subtree.
n, number of initial OTUs.

Output: A tree $\mathcal{T}_n$ over all n OTUs.

Initialization: Initialize $k = 4$.

Loop: While $k < n$ do:

1. Calculate the dissimilarity between OTU k and any subtree S from $\mathcal{T}_{k-1}$.
2. Calculate the cost of inserting k along edge e as $f(e)$ for all edges e in $\mathcal{T}_{k-1}$.
3. Select the best edge by minimizing f and insert k on that edge to form $\mathcal{T}_k$.
4. Update the dissimilarity matrix D between every pair of distant-2 subtrees as well as the array P with the number of OTUs per subtree.

Finalize: Return the tree $\mathcal{T}_n$.

the distance matrix. Otherwise, if $S_i = \{i\}$ and $S_j = \{S_{j_1}, S_{j_2}\}$ the dissimilarity can be computed by eq. (2.13) for GME or eq. (2.14) for BME because under a balanced weighting scheme, sibling subtrees are assigned equal weights, while in the OLS scheme the weight of a subtree is equal to the number of OTUs it contains.

$$D_{ij} = \frac{|S_{j_1}|}{|S_j|} D_{ij_1} + \frac{|S_{j_2}|}{|S_j|} D_{ij_2} \tag{2.13}$$

$$D_{ij} = \frac{1}{2}(D_{ij_1} + D_{ij_2}) \tag{2.14}$$

Suppose that we have a tree $\mathcal{T}$ with four subtrees A, B, C and D and that for any subtree K the distance between its root and that subtree is D_{kK}. Then the tree length can be computed recursively by applying eq. (2.15).

$$\begin{aligned} l(\mathcal{T}) &= \tfrac{1}{2}[\lambda(D_{AC} + D_{BD}) + (1-\lambda)(D_{AD} + D_{BC}) + D_{AB} + D_{CD}] \\ &+ l(A) + l(B) + l(C) + l(D) - D_{aA} - D_{bB} - D_{cC} - D_{dD} \end{aligned} \tag{2.15}$$

Now, in order to compute the length of an edge e it is necessary to distinguish when e is an internal or external edge. First, let us assume that e is an internal edge that connects subtrees A and B with C and D. Then, the length estimate of e can be defined by eq. (2.16). If we consider e an external edge that connects subtrees A, B with a single OTU k, we need to apply eq. (2.17).

$$l(e) = \frac{1}{2}[\lambda(D_{AC} + D_{BD}) + (1 - \lambda)(D_{AD} + D_{BC}) - D_{AB} - D_{CD}] \tag{2.16}$$

$$l(e) = \frac{1}{2}(D_{AC} + D_{BC} - D_{AB}) \tag{2.17}$$

For the GME version, λ is replaced in both eqs. (2.15) and (2.16) by eq. (2.18). This demonstrates an important property of OLS edge length estimation because the length estimate of any given edge does not depend on the topology of the subtrees A, B, C and D, but only on the number of OTUs contained in those subtrees.

In the balanced version of minimum evolution, λ is replaced by $1/2$ (eq. (2.19)) to assign equal weights to all subtrees, regardless of the number of OTUs they contain. The edge length estimates now depend on the topology of the subtrees, simply because the balanced distances between these subtrees depend on their topologies.

$$\lambda = \frac{|A||D| + |B||C|}{(|A| + |B|)(|C| + |D|)} \tag{2.18}$$

$$\lambda = \frac{1}{2} \tag{2.19}$$

Starting with GME algorithm, on step 1, the dissimilarity between OTU k and any other subtree can be computed by applying eq. (2.13) recursively. On step 2, suppose that a tree $\mathcal{T}'$ is formed by moving the insertion of k from edge e_1 to the edge e_2, where e_2 is a sibling edge of e_1 in $\mathcal{T}$. The length of $\mathcal{T}'$ is computed by eq. (2.15) as $l(\mathcal{T}') = L + f(e)$, where L is the length of tree $\mathcal{T}$ and $f(e)$ depends on the computations for both $\mathcal{T}$ and $\mathcal{T}'$. Then, the process of searching for every edge e of $\mathcal{T}_{k-1}$ continues by recursively moving from one edge to its neighboring edges, to obtain the cost that corresponds to the length of the tree $\mathcal{T}_{k-1}$ plus k inserted on e. Moreover, this cost can be written as $L + f(e)$ and for the first considered edge it is sufficient to minimize $f(e)$ with $f(e) = 0$. Finally, on the last step, eq. (2.20) is used to update the distance between any pair A, B of distant-2 subtrees, if k is inserted in the subtree A.

$$D_{(k \cup A)B} = \frac{1}{1 + |A|} D_{kB} + \frac{|A|}{1 + |A|} D_{AB} \tag{2.20}$$

The main difference between GME and BME is that in the latter the update can no longer be achieved using a fast method as expressed by eq. (2.20), because the balanced distance between $k \cup A$ and B now depends of the position of k within A.

Despite of BME algorithm requiring $O(n^3)$ operations on the worst-case, in practice it needs only $O(n^2 \log n)$ [53]. Therefore, both GME and BME algorithms are faster than NJ-like algorithms which require $O(n^3)$ operations, allowing trees with thousands of elements to be constructed in few minutes.

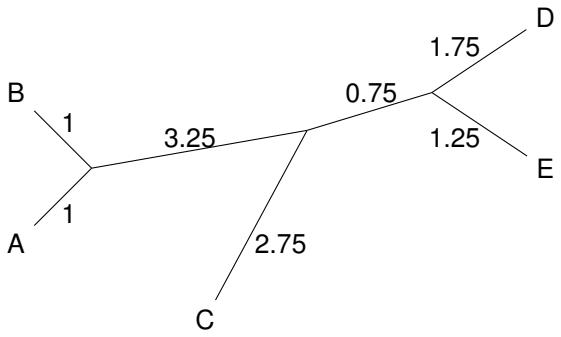

Figure 2.4: BME phylogenetic tree according to matrix 2.1.

Also, the balanced minimum evolution appears much more appropriate for phylogenetic inference than the ordinary least-squares version (GME). This is likely due to the fact that it gives less weight to the topologically long distances (*i.e.* those containing numerous edges), while the OLS method puts the same confidence on each distance, regardless of its length. Another disadvantage of GME is that it does not guarantee that the global optimum will be reached but only a local optimum. Furthermore, the balanced algorithms produced output trees with better topological accuracy than those from NJ. Still, NJ continues to be the most used and preferred algorithm by the community because it guarantees that the reconstructed tree will be the correct tree if the distance matrix is "nearly additive" [59]. In practice the distance matrix rarely satisfies this condition, but neighbor joining often constructs the most correct tree topology anyway [60].

An example of BME phylogenetic tree according to distance matrix 2.1 is presented in fig. 2.4.

2.3.2.3 Distance-based-MST-like methods

As we have seen earlier, the theory of evolution predicts that existing biological species have been linked in the past by common ancestors and their relationships can be depicted as a branched diagram like phylogenetic trees. The methods that we will present here use the same evolutionary model of NJ (minimum evolution) to better estimate the true phylogenies however they were developed following a graph theoretic approach. Thus, the problem of determining the minimal phylogenetic tree will be discussed in relation to graph theory.

Graph theory is the branch of mathematics which is concerned with the study of discrete arrangements of, and relationships between objects [61]. A graph is a mathematical structure which can be represented by a set of points drawn in a plane, pairs of which may be connected by lines called links. A path is a sequence of distinct links in a graph with the property that each link in the sequence, other than the first, begins at the point where its predecessor ends. A cycle is a sequence of distinct links in a graph which is identical to a path except that the first point of the first link is coincident with the last point of the last link in the sequence. A graph is said to be connected if there exists at least one path between

Algorithm 2.3: Generic-MST pseudocode.

```
1  Input: A graph G = (V, E) where V and E are the set of vertices and edges, respectively.
2         cmp, edge weight comparator.
3  Output: A minimum spanning tree T.
4
5  begin
6      T ← ∅
7      while T does not form a spanning tree do
8          find an edge (u, v) ∈ E that is safe for T
9          T ← T ∪ {(u, v)}
10     return T
11 Finalize: Return T.
```

every pair of distinct points. A tree is a graph that is connected but does not contain any cycles.

The problem of determining the minimal phylogenetic tree is discussed in relation to graph theory where each OTU is represented by a point in the tree to be constructed and each link in the tree is associated with the changes between the species it connects. This problem is similar to one known in graph theory as the minimal spanning tree problem on trying to identify a subset of links in a graph that connects all points without any cycles and with the minimum possible total link length.

Next, we will describe four methods, generic-MST, Foulds-Hendy-Penny, goeBURST and goeBURST Full MST that have been developed based on algorithms that solve the MST problem (*e.g.* Borůvka [62], Kruskal [63], Prim [64], etc.) to determine the minimal phylogenetic tree.

2.3.2.3.1 Generic-MST

The problem of finding a minimum spanning tree can be captured by the generic algorithm algorithm 2.3, which grows the minimum spanning tree one edge at a time. At each step, we determine a safe edge (u, v) that we can add to $\mathcal{T}$. An edge is considered a *safe* edge if and only if we can add it to $\mathcal{T}$ without creating a cycle in it. Therefore, all edges added to $\mathcal{T}$ guarantee that $\mathcal{T}$, in line 10, must be a minimum spanning tree. In the case of a tie, *i.e.*, if a new vertex u has two safe edges, (u, v) and (u, w), connecting to the tree $\mathcal{T}$ with equal weights, we must define a comparator, cmp, to decide which one to choose.

There exist several greedy algorithms that elaborate on this generic method and they use a specific criterion to determine a safe edge, in line 8. Kruskal chooses always a least-weight edge in the graph that connects two distinct components. Prim chooses always a least-weight edge connecting the tree to a vertex not in the tree. And Borůvka chooses the minimum-weight edge incident to each vertex of the graph.

Distance-based-MST algorithm also elaborate on this generic method but the criterion is based on the computation of distances between vertices. This algorithm differs from the remaining on the com-

parator used to break the ties on the edges, while the others choose them arbitrarily.

The reason underlying why greedy algorithms are effective at finding minimum spanning trees is that the set of forests of a graph forms a graphic matroid [65, 66] (see section 3.1).

2.3.2.3.2 L.R. Foulds, M.D. Hendy and David Penny

The proposed method of Foulds, Hendy, and Penny [55] is an heuristic method of approaching the phylogenetic problem by transforming it into a Steiner problem [67] combined with Kruskal's method. It requires not only a distance matrix to represent the number of changes between the species' sequences but also the position in that sequence at which they differ, and which characters were changed from one sequence to the other.

At each iteration, the method identifies the shortest link joining two unconnected points, which when added individually to the existing graph do not create a cycle. Then, it tries to reduce the graph by "coalescing" (*i.e.* introducing Steiner points that do not represent any of the original species but represent instead a sequence of a possible intermediary specie), where necessary. This means that after a link is selected and added to the tree it compares all the pairs of links incident there, and identifying those changes that are common to the two links, choosing to coalesce on a pair with the maximal number of common changes (*i.e.* that results in the largest reduction in the total length of the tree). If a cycle is present an attempt is made to break it by removing links which give the largest reduction in the total length of the tree. If there is more than one largest link, the cycle is left unbroken because it can subsequently be broken by later additions. The method terminates when all original species are connected. Although, if at the end the cycles are still unresolved, all the alternative trees are presented and the choice is left to the biologist who may wish to use some additional information.

For instance, assume you have the following three sequences: $S_1 = abcde$, $S_2 = bbdce$ and $S_3 = acdcd$, with distances: $d(S_1, S_2) = 3$, $d(S_1, S_3) = 4$ and $d(S_2, S_3) = 3$. We can see that S_1 differ from S_2 in the first, third and forth elements ($a \leftrightarrow b$, $c \leftrightarrow d$ and $d \leftrightarrow c$), S_1 differ from S_3 in the second, third, forth and fifth elements ($a \leftrightarrow b$, $c \leftrightarrow d$, $d \leftrightarrow c$ and $e \leftrightarrow d$) and finally, S_2 differ from S_3 in the first, second and fifth element ($b \leftrightarrow a$, $b \leftrightarrow c$ and $e \leftrightarrow d$). We now follow Kruskal, and search for the links of minimal length, not previously chosen, which, when added individually to the existing graph do not create a circuit, $d(S_1, S_2) = 3$ and $d(S_2, S_3) = 3$. These now give us a partial graph, illustrated in fig. 2.5 (a). We now attempt to reduce the graph by comparing all the pairs of links incident there, and identify those changes that are common to the two links, choosing to coalesce on a pair with the maximal number of common changes. In our example, S_2 is the only point with a pair of incident links and there is only one common change, ($a \leftrightarrow b$) at the first element. To represent this we introduce the Steiner point S_4, to represent the possible sequence: $S_4 = abcde$ (fig. 2.5 (b)). Now, as all points are connected to the tree the algorithm is complete, so we have a spanning tree.

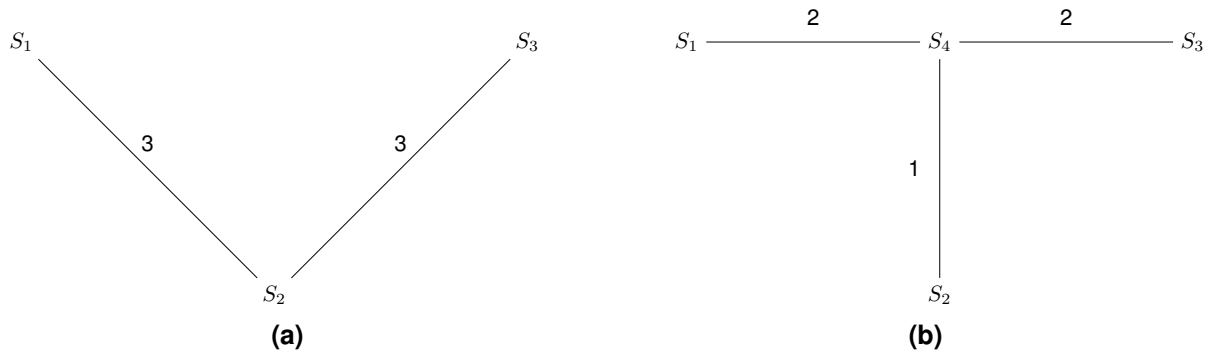

Figure 2.5: Step by step tree representation of FHP algorithm.

The present method overcomes the problem of differences in the rate of evolution on different links since all species are retained for further comparisons, even after they are linked into the graph. This is an important aspect of the method because it ensures that a particular specie is included in its optimal position. Another important feature of this method is that when new ancestral species are determined, they are compared with all the existing species. This means that the links are not in a fixed position while the tree is being determined.

Note that this method does not guarantee to build a minimum spanning tree and the accuracy of that tree is very much dependent upon the suitability and reliability of the data. A single error in the sequence entries could conceivably make a significant difference to the tree if it affected closely related sequences.

2.3.2.3.3 goeBURST

goeBURST algorithm [54] is a globally optimized implementation of the eBURST algorithm [68] that identifies alternative patterns of descent for several bacterial species. It implements the simplest model for the emergence of clonal complexes [69, 70] where a given genotype increases in frequency in the population, as a consequence of a fitness advantage or of random genetic drift, becoming a founder clone in the population, and this increase is accompanied by a gradual diversification of that genotype, by mutation and recombination, forming a cluster of phylogenetically closely related strains. This diversification of the "founding" genotype is reflected in the appearance of Sequence Type (ST) differing only in one housekeeping gene sequence from the founder genotype – Single Locus Variant (SLV). Further diversification of those SLVs will result in the appearance of variations of the original genotype with more than one difference in the allelic profile: Double Locus Variant (DLV), Triple Locus Variant (TLV), and so on. The result is a forest, a disjoint set of trees (acyclic graphs), where each tree corresponds to a clonal complex defined for an allelic distance of one (*i.e.* the minimum distance of any ST to at least one of the STs of the same clonal complex is a difference in a single, double or triple locus.).

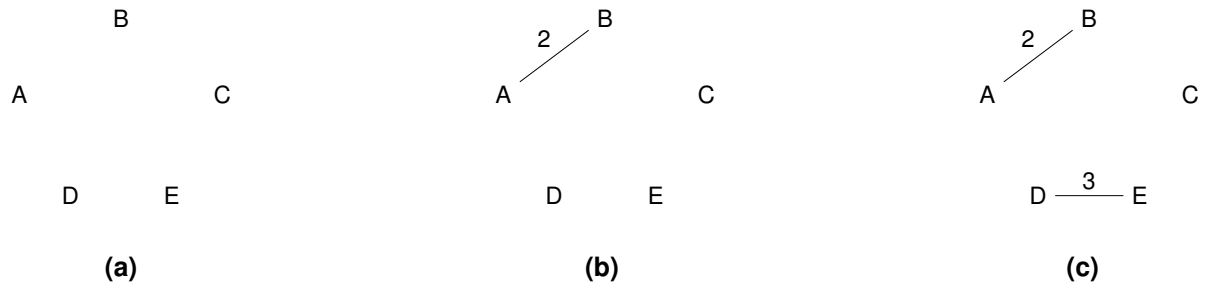

Figure 2.6: goeBURST phylogenetic trees according to matrix 2.1 with different allelic distances: (a) allelic distance of one, (b) allelic distance of two and (c) allelic distance of three.

This algorithm can be stated as finding the maximum weight forest or, depending on weight definition, as finding the minimum spanning tree. Therefore, the optimal solution can be provided by a greedy approach on identifying the optimal forest with respect to the defined partial order on the set of links between STs. Due to its desirable properties and ease of implementation, goeBURST uses Kruskal algorithm to achieve that goal.

This algorithm consists of building a spanning forest in a graph where each OTU is a node and two OTUs are connected if and only if they are at distance one (*i.e.* if they only are SLVs). Since this forest should be optimal with respect to link selection, the links between OTUs with higher number of SLVs must be selected. In case of a tie it should be considered the number of OTUs that are at distance two (*i.e.* number of DLVs), the number of OTUs that are at distance three (*i.e.* number of TLVs), the occurrence frequency of OTUs, and lastly the assigned OTU number (ID). Although this last tiebreak is rarely reached, this criterion is necessary to provide a consistent and unique solution to the problem as it will always provide a consistent tiebreak solution due to the uniqueness of the ID. Lower IDs take precedence over higher IDs because it is assumed that in a growing database with data of several contributing international studies, the more common OTUs are sampled first and will have lower ID than the subsequent studies that will add more OTUs to the database. These rules define a partial order on the set of edges between vertices [54] and therefore goeBURST can be considered a distance-based-MST algorithm.

goeBURST scheme is very similar to algorithm 2.1 but it skips the reduction step. In order to build a final optimal forest the algorithm starts with a forest of singleton trees (where each OTU is a tree) and then, iteratively, selects links connecting those in different trees with the minimum distance. This corresponds to selecting a pair of OTUs with maximum number of SLVs at step 1 and joining them at step 2. The final step is not necessary because it does not need to update the overall pairwise distances. goeBURST will always provide an optimal solution for the link assignment, since it performs a global optimization taking into consideration all possible ties at all levels between STs in the data set.

The final phylogenetic tree is graph-based because topologies like dendrograms frequently do not

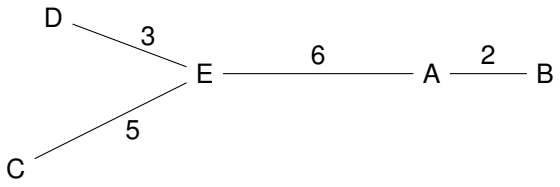

Figure 2.7: goeBURST Full MST phylogenetic tree according to matrix 2.1.

depict the precise evolutionary history and are particularly susceptible to the confounding effects of recombination. This kind of tree representation may more easily lead to false conclusions about the relationships between species whereas the graph-based tree representation should be preferred when recombination is high [71].

An example of goeBURST phylogenetic tree with allelic distances of one, two and three are presented in fig. 2.6 (a), (b) and (c) respectively, and according to distance matrix (matrix 2.1). As one can observe, goeBURST algorithm can construct different forests according to a defined allelic distance.

2.3.2.3.4 goeBURST Full MST

Using an extension of the goeBURST rules defined above up to nLV level (where n equals to the number of loci in a strain), a Minimum Spanning Tree-like structure can be computed. If one define n as one, two or three, the results of this algorithm will be equivalent to calculating goeBURST at those levels (SLV, DLV and TLV respectively).

For the full MST computation the Borůvka algorithm is used to build the tree, which is also a greedy algorithm for finding a minimum spanning tree in a graph but with the constraint of having all edge weights distinct fixed by ID [62, 72]. This algorithm is based on merging disjoint components.

At the beginning each vertex is considered as a separate component and in each step it merges every component with some other using strictly the cheapest edge of the given component (every edge must have a unique weight). This way it is guaranteed that no cycle may occur in the spanning tree. At each step this algorithm connects at least half of the currently unconnected components, therefore it is clear that the algorithm terminates in $O(\log V)$ steps. Because each step takes up to $O(E)$ operations to find the cheapest edge for all the components, the asymptotic complexity of Borůvka's algorithm is $O(E \log V)$, where E is the number of edges, and V is the number of vertices in the graph G.

The constraint of having distinct weights for all edges is not a problem for this algorithm. As mentioned before, in the case of two vertices having the same weight we can apply the tiebreak rules that will always provide a consistent and unique solution.

An example of goeBURST Full MST phylogenetic tree according to distance matrix (matrix 2.1) is presented in fig. 2.7.

2.3.3 Properties of reduction formulae

All clustering algorithms mentioned before use a specific reduction formula on their last step. In GCP algorithms this is the only thing that distinguish them from one another.

Gronau and Moran (2007) [73] demonstrated that if the reduction formula used by these algorithms on step 3 is both convex and commutative, then the selection criterion used on step 1 can be relaxed by joining in each stage a locally (rather than globally) closest cluster-pair. This means that each of the two clusters is closest to the other, but their dissimilarity is not necessarily minimal among all pairwise dissimilarities. This relaxed selection scheme is called the Locally Closest Pair (LCP) scheme and it was proven to be equivalent (*i.e.* same output tree) to GCP if those two properties of the reduction formula hold. They also state that this relaxation can be implemented using the Nearest-Neighbor-chain technique where the selection relies on a smaller set (called *NN-chain*) that contains only locally closest neighbors.

A *convex reduction formula* means that the distance between any cluster C_k to the new cluster C_u (where $C_u = C_i \cup C_j$) lies between the distance from that cluster C_k to C_i and C_j.

A reduction formula is said to be *commutative* if given four arbitrary clusters $\{C_1, C_2, C_3, C_4\}$, the dissimilarity matrix obtained by first joining $\{C_1, C_2\}$ and then joining $\{C_3, C_4\}$ is equal to the dissimilarity matrix obtained by first joining $\{C_3, C_4\}$ and then joining $\{C_1, C_2\}$.

Gronau and Moran (2007) [73] also state that every clustering algorithm that uses a reduction formula which is both convex and commutative can be locally optimized and can be implemented by the NN-chain technique.

Table 2.4 summarizes these properties for the clustering algorithms mentioned before. We can observe that except GME and BME, they all have a commutative reduction formula and only NJ and its variants cannot assure the convexity property. GME and BME cannot guarantee commutativity because the insertion of a new element in the tree is directly related with the length of the tree after that insertion, which involves changing the length of every other already existing edge. So, changing the order of the addition can lead to different trees. Regarding convexity, NJ and its variants cannot guarantee it because their reduction formula depends on the new branch lengths calculated for the two joined elements and the formula used to compute it do not guarantee convexity, leading to possible negative branch lengths.

2.4 Discussion

Until now we have discussed the most commonly used algorithms and their properties and although phylogenetic inference algorithms may seem rather different, the main difference among them resides on how one defines cluster proximity and on which optimization criterion is used.

The unified view of these algorithms that we provide is an important step not only to better understand

Table 2.4: Local optimization regarding reduction formula properties for different algorithms.

Algorithm	Reduction formula		Optimized locally
	Convex	Commutative	
GCP	Yes	Yes	Yes
NJ and variants	No	Yes	No
GME	Yes	No	No
BME	Yes	No	No
goeBURST	Yes	Yes	Yes
generic-distance-based-MST	Yes	Yes	Yes
FHP	Yes	Yes	Yes

such algorithms, but also to identify possible computational bottlenecks and improvements, such as dynamic updating of the algorithm's result whenever new data becomes available. By studying and analyzing all these algorithms we can address this issue and check if they can be dynamic updated or not just by looking at their properties.

Focusing on goeBURST Full MST algorithm, the dynamic update technique is based on an online algorithm for computing Minimum Spanning Trees (MSTs), allowing data to be continuously integrated and updated without requiring the computation of everything from the start [74]. This approach can be used in this method because it constructs MST like trees. It also guarantees that the resulting tree will be the same as before and, therefore, it will be optimal.

Despite our focus only being on goeBURST Full MST we briefly studied how other algorithms, namely GCP, could also solve this problem. This kind of algorithms construct dendrograms instead of graph-based trees, hence the techniques developed to solve this problem in goeBURST Full MST cannot be applied here. On the other hand, sufficient statistics [75, 76] have been widely use to solve many problems regarding data clustering [77–79]. The main goal here is to identify which are the statistics that summarize all of the information needed for allowing the addition of new data to a previous one, in such a way that the obtained result is the expected (*i.e.* the same as if it was obtain in a static way). Regarding GCP methods, this can be done if one have access to the list of edges ordered by the way the tree was created (*i.e.* insertion order of the edges in the tree) along with their weight. Then, iteratively by edge weight, we can check if an edge can be added to the tree or not by checking if they already exist in the tree that is currently being constructed. After each insertion of an edge on the tree all pairwise distances between the new element and the remaining need to be updated.

On the next chapter, we will discuss two different techniques that allow MST algorithms to be dynamically updated, namely generic-distance-based-MST and goeBURST Full MST. These techniques differ on the criterion they use to compare the edges to be added to a tree. They both choose always the least heavier edge to be added to the tree, however when two edges have the same weight, goeBURST Full MST selects the edge that connects two OTUs with higher number of SLVs, DLVs, and so on (according

to the tiebreak rules), while generic-distance-based-MST chooses it arbitrarily. Therefore, when we want to update a tree with a new OTU, we must realize how it would influence the choices made before, in order to obtain the same tree as if it was build from scratch.

3

Dynamic distance-based-MST algorithms

Contents

We may recall from the previous chapter that goeBURST algorithm consists of building a spanning forest in a graph where two STs are connected if and only if they are SLVs, whereas the goeBURST Full MST consists in building a minimum spanning tree that connects all STs up to nLV (where n equals the total number of loci in a strain). Since this tree should be optimal with respect to link selection, the search for a global optimal solution led to the formalization of the problem as a MST problem in such a way that the summed distance of all links of the tree is the shortest (minimum). Here, we will explain how the MST problem can be generalized by a matroid, namely a graphic matroid [65, 66], for which greedy algorithms that provide an optimal solution exist [80].

The chapter is divided into two main parts. On the first part, we define what is a matroid. Then, we describe how greedy algorithms like goeBURST Full MST is effective at finding minimum spanning trees using a graphic matroid approach. On the second half, we propose a dynamic version for generic-distance-based-MST and goeBURST Full MST algorithms that will allow data to be continuously integrated without requiring us to compute everything from the start.

3.1 goeBURST and graphic matroids

The Maximum Weight Forest (MSF) problem in goeBURST and the MST problem in generic-distance-based-MST and goeBURST Full MST are particular cases of graphic matroids [81]. Thus, finding a solution to represent the phylogenies by the use of MSTs consists of solving instances of graphic matroids [65, 81, 82] which can be optimally solved with a greedy approach [83].

A matroid is an ordered pair $M = (S, \mathcal{I})$ satisfying the following conditions:

1. S is a finite set.

2. $\mathcal{I}$ is a nonempty family of subsets of S, called the independent subsets of S, such that if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$. We say that $\mathcal{I}$ is hereditary if it satisfies this property. Note that the empty set $\emptyset$ is necessarily a member of $\mathcal{I}$.

3. If $A \in \mathcal{I}$, $B \in \mathcal{I}$ and $|A| < |B|$ then there exists some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$. We say that M satisfies the exchange property.

A graphic matroid is an example of matroids [66]. Consider $M_G = (S_G, I_G)$ defined in terms of a given undirected graph $G = (V, E)$ as follows:

- The set S_G is defined to be E, the set of edges of G.

- If A is a subset of E, then $A \in \mathcal{I}_G$ if and only if A is acyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a forest.

If the edges E in the graph G are weighted (which is the case in our study) then we can say that a matroid $M_G = (S_G, I_G)$ is weighted and that it is associated with a weight function w that assigns a strictly positive weight $w(x)$ to each element $x \in S$. The minimum spanning tree problem can be characterized as a special case of the weighted matroid optimization problem [84] in terms of finding a maximum-weight independent subset in a weighted matroid. That is, we are given a weighted matroid $M_G = (S_G, I_G)$ and we wish to find an independent set $A \in I$ such that $w(A)$ is maximized. Because the weight $w(x)$ of any element $x \in S$ is positive, an optimal subset is always a maximal independent subset. Consider now a weighted matroid $M_G = (S_G, I_G)$ with a weight function w', where $w'(x) = w_0 - w(x)$ and w_0 larger than the maximum length of any edge. In this weighted matroid, all weights are positive and an optimal subset is a spanning tree of minimum total length in the original graph. Therefore an independent subset that maximizes the quantity $w'(A)$ must minimize $w(A)$ [72].

Thus, given a graph G and the set $\mathcal{F}$ of all forests over G (*i.e.* a matroid), the optimization problem is to find an optimal forest. Note that by allowing any possible distance for link comparison and thereby fully generalizing link comparison, any problem of tree construction becomes a graphical matroid instance [85, 86], that can be solved by any algorithm that can find an optimal subset A in a matroid, such as the Borůvka algorithm [62], the Kruskal algorithm [63] or the Prim algorithm [64].

3.2 goeBURST Full MST

As mentioned above, this algorithm can be stated as finding the minimum spanning tree in a graph which can be optimally solved with a greedy approach. In this case, and unlike goeBURST, we use Borůvka's algorithm to provide an optimal solution for this problem class.

This static version of goeBURST Full MST starts by defining the maximum number of loci under analysis (*i.e.* maximum level) and then we compute for each OTU how many other OTUs are at distance one, two, etc. In other words, we compute the total number of SLVs, DLVs, TLVs and so on, till we reach the defined nLV level, for every single OTU. Finally, we run an adapted version of Borůvka's algorithm where the merging phase occurs only between components at the same level (*i.e.* equal weights).

Given $G = (V, E)$, it works as follows:

1. create a forest $\mathcal{F}$ where each $u \in V$ is a tree;
2. iterate over V and, for each edge $(u, v) \in E$ ordered by increasing weight, if u and v are in different trees, add (u, v) to $\mathcal{F}$ merging both trees as a single tree;
3. return $\mathcal{F}$.

The main difference between Borůvka's algorithm and its adapted version relies on the definition of cheapest edge. This means that instead of selecting the minimum weight edge for each OTU it starts by

Algorithm 3.1: Generic dynamic MST ADD operation scheme.

Input: $\mathcal{T}$, previously computed MST.
        u, OTU to be added to the tree $\mathcal{T}$.
        cmp, an edge comparator.
        sim, a similarity measure.
Output: Updated minimum spanning tree, $\mathcal{T}'$.

Pre-processing step begin
    $E \leftarrow \emptyset$
    if $u \in \mathcal{T}$ **then**
        finish
    foreach vertex $v \in \mathcal{T}$ **do**
        $d \leftarrow sim(u, v)$
        $E \leftarrow E \cup \{(u, v, d)\}$
    $E' \leftarrow sort(E, cmp)$
Add step begin
    $\mathcal{T}' \leftarrow$ dynamic-mst($E', \mathcal{T}, cmp$)
Finalize: Return the tree $\mathcal{T}'$.

selecting an edge that is at distance one from that OTU. This is done iteratively for each level till there are no more edges at that level. Once more, if there exists more than one edge for the same level, the goeBURST tiebreak rules are applied till no edges for that level remain.

This algorithm takes $O(V + E)$ space to store the nodes, links and all necessary information. It takes $O(V)$ time to create the initial forest and $O(E \log V)$ to build the optimal forest.

Since building the graph takes $O(V^2)$ and computing the connected components and the total number of SLVs, DLVs, TLVs and so on, till we reach the defined nLV level for all STs takes $O(V + E)$, the running time is dominated by the graph construction.

3.3 Dynamic MST algorithms

A distance-based-MST algorithm that wants to solve instances of weighted graphic matroids should support the following operations:

- ADD ($G(V, E)$, u, w): inserts the new element u into the graph G.
- UPDATE ($G(V, E)$, u, d): updates the new element u in the graph G with the new distance d.
- DELETE ($G(V, E)$, u): deletes an element u from the graph G.

Since we are solving instances of matroids, namely weighted graphic matroids, $G = (V, E)$ must be a undirected weighted graph defined by V and E as the set of vertices and edges, respectively. We must also define the weight function $w(e)$ to each edge $e \in E$ as a weight function $w(u, v)$ where u and

v are elements in V since distance-based-MST algorithms have the distances associated with each pair of vertices, and not on edge itself.

The dynamic MST algorithms that we present here only support the operation ADD, leaving the remaining two operations as a possible future work.

Focusing on operation ADD, generalized by algorithm 3.1, its goal is allow the addition of a new OTU to a previously computed minimum spanning tree without the need of running a static distance-based-MST algorithm from scratch. This operation can be divided mainly in two major steps: pre-processing and adding the new OTU. The pre-processing step is common to all algorithms and is responsible for creating all structures that are necessary for the addition step, while the latter refers to the specific process of adding the new OTU. Here, we will start by explaining the first step and later, the dynamic implemented algorithms that will detail the adding step for each algorithm and also, the associated weight function $w(u, v)$, that differs from one to another.

Regarding the pre-processing step, we start by checking if the new OTU, u, that we are going to add already exists. If it exists then we discard it and the algorithm finishes, otherwise, it continues by computing the distance from u to the other OTUs through a similarity measure (*e.g.* hamming distance) and create new edges, E, to represent it. Finally, and before going to the addition step, we sort those edges, $E' \leftarrow sort(E, cmp)$, based on a comparator, cmp, defined by the weight function $w(u, v)$ of each dynamic algorithm. This function defines the rules that will be applied every time a tie between two edges occurs.

This pre-processing step takes linear time in the size of the graph to process n edges and pairwise comparisons. In the next sections 3.3.1 and 3.3.2 we describe in detail the addition step for both dynamic versions.

3.3.1 Dynamic generic-distance-based-MST algorithm

After finishing the pre-processing step we move on to the addition part, more specifically lines 15-16 in algorithm 3.1. This dynamic algorithm is the simplest because it defines $w(u, v)$ only based on the pairwise distances and ultimately, in case of a tie, in the increasing order of the vertices' IDs. This algorithm is defined in algorithm 3.2 and can be described as follows: given a new OTU, u, it starts by adding the first new edge, $(u, v) \in E'$, directly in the MST. Since we are adding u for the first time in the tree, it will not be necessary to check if it would form a cycle in the tree or not. Then, iteratively, we add each remaining new edge (in increasing order of the distance) and check if it will create a cycle by traversing the tree with a Breadth-First Search (BFS) approach. If that occurs and if the new edge weight (distance) is less than the weight of the highest weight edge in this cycle, then we create a lower weight MST by replacing that higher weight edge with the new edge. Otherwise, the current MST remains optimal and we discard that new edge.

To traverse a graph to detect if a cycle is present and because we are only dealing with undirected weighted graphs, we could either choose to use a BFS or a Depth-First Search (DFS) approach [72]. Both approaches are efficient for detecting cycles as they both run in linear time in the size of the graph $O(V + E)$ in the worst case to traverse an entire graph. However, the true (actual) running time of BFS and DFS is highly dependent on the structure of the graph and on the starting and ending points. So, if we want to choose right we must address these two questions in a wise manner. Therefore, and because the start and ending points are the same on both approaches, we only have to reason about its structure. We know that the graph is weighted, undirected and that we only want to search part of the tree as opposed to the whole tree, hence BFS is a better choice over DFS.

Overall, this algorithm takes linear time on average for real data to add a new OTU, being dominated by the time for pre-processing.

3.3.1.1 Example

Given the MST tree, T, built by generic-distance-based-MST (see fig. 3.1 (a)), and a new OTU u, we start by checking if it already exists in T. If exists, then the new OTU is discarded and the algorithm terminates. Otherwise, we compute all pairwise distances between the new OTU and the others on T, D_{uv}, using some similarity measure (assume $[1, 1, 3, 2, 4]$ to be the pairwise distances D_{uv} where $v = A, B, \ldots, E$ in T) and create new edges, (u, v), to represent them. The final step in the pre-processing phase is to sort those edges in increasing order of D_{uv}, resulting in the following list: $\{(u, A)$,(u, B), (u, D), (u, C), $(u, E)\}$.

Now, we must add the first edge, (u, A), to the updated MST tree, $T' \leftarrow T \cup (u, A)$, as shown in fig. 3.1 (b). Then, when we try to add edge (u, B) we see that it will form a cycle with the path $(u - A - B - u)$,

---

Algorithm 3.2: Dynamic generic-distance-based-MST addition step scheme.

Input: E', an ordered list of newly created edges.
$\mathcal{T}$, previously computed MST.
cmp, an edge comparator.
Output: Updated minimum spanning tree, $\mathcal{T}'$.

Initialization: Initialize the tree $\mathcal{T}'$ with the previous MST $\mathcal{T}$ and add the first edge $(u, v) \in E'$, $\mathcal{T}' \leftarrow \mathcal{T} \cup (u, v)$.

Loop: For each $(u, v) \in E'$ do:

1. **Cycle-finding:** Add current edge (u, v) to the MST $\mathcal{T}'$. Find a path p that forms a cycle in $\mathcal{T}'$.
2. **Cycle-removal:** Compare the heaviest edge (u', v') in p with the current edge (u, v) using cmp.
   If $(u', v') \leq (u, v)$: Remove (u, v) from $\mathcal{T}'$.
   Else: Remove (u', v') from $\mathcal{T}'$.

Finalize: Return the tree $\mathcal{T}'$.

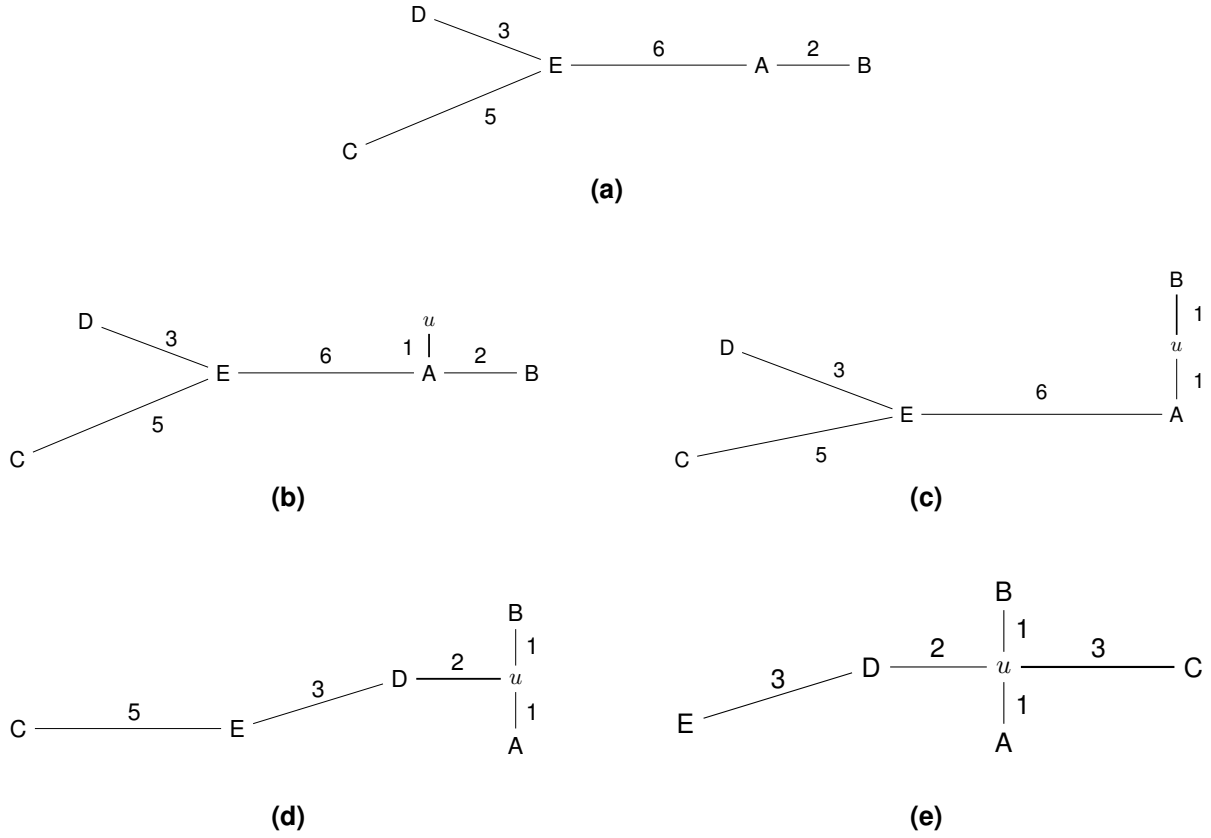

Figure 3.1: Step by step tree representation of dynamic generic-distance-based-MST addition of a new OTU to a previously computed minimum spanning tree.

so we must compare all edges from that cycle and remove the heaviest, that is (A, B), fig. 3.1 (c). The same happens with the two following edges (u, D) and (u, C), causing the deletion of (A, E) and (C, E) in (d) and (e), respectively. For the final edge, (u, E), we also get a cycle $(u - D - E - u)$ but unlike the last two edges, we will not remove any edge from $T' \setminus \{(u, E)\}$ since (u, E) is the heaviest edge on the cycle path. The final phylogenetic tree T' is then represent in fig. 3.1 (e).

3.3.2 Dynamic goeBURST Full MST

The addition step regarding dynamic goeBURST Full MST algorithm is more complex than the latter because it can change the entire tree. This is due to the fact that the overall number of locus variants for all OTUs will vary causing a possible change on the tiebreak rule applied before by goeBURST Full MST algorithm. Therefore, we need to start by updating those values for each OTU in the tree according to the distances to the new OTU, and sort them according to the specified comparator cmp. This comparator is based on the tiebreak rules defined in sec. 2.3.2.3.3, hence our weight function w defines a total order $\leq$ on E by comparing the number of SLVs, DLVs, TLVs, the occurrence frequency of the OTU and finally

Algorithm 3.3: Dynamic goeBURST Full MST addition step scheme.

Input: T, previously computed MST.
u, OTU to be added.
E', an ordered list of newly created edges.
L, maximum distance level.
cmp, an edge comparator.
Output: Updated minimum spanning tree, $\mathcal{T}'$.

Initialization: Initialize the tree $\mathcal{T}'$ with an empty set.

foreach OTU $v \in \mathcal{T}$ **do**
  updateLVs(u, v);
$E'' \leftarrow sort(E', cmp)$;

Loop: For each $\ell \in L$ do:

1. **Edges-selection:** Select all edges from the previously computed MST $\mathcal{T}$ that are at distance level ℓ. Select all new edges from E'' that are at distance level ℓ.
2. **Edges-sorting:** Sort all edges selected at step 1 using cmp.
3. **Tree-updating:** Add each sorted edge in the new tree $\mathcal{T}'$ if and only if it connects different subtrees in $\mathcal{T}'$.

Finalize: Return the tree $\mathcal{T}'$.

its unique ID. More formally, let $\mu : V \rightarrow N^5$ be a function such that $\mu(u)$ is a vector which components are the numbers of SLVs, DLVs, TLVs, the occurrence frequency and the ID of u. Then, given (u_1, v_1), $(u_2, v_2) \in E$, we say that $(u_1, v_1) \leq (u_2, v_2)$ if and only if ϵ is positive, where ϵ is computed as follows, starting with the first rule (SLV) $i = 1$:

1. $\epsilon \leftarrow max\{\mu_i(u_2), \mu_i(v_2)\} - max\{\mu_i(u_1), \mu_i(v_1)\}$.
2. if $\epsilon = 0$, then $w \leftarrow min\{\mu_i(u_2), \mu_i(v_2)\} - min\{\mu_i(u_2), \mu_i(v_2)\}$.
3. if $\epsilon = 0$ and $i < 5$, then set $i \leftarrow i + 1$ and go to step 1, otherwise return ϵ.

The application of this weight function by the comparator cmp will always provide a unique solution (unique minimum spanning tree) for this algorithm.

Then, iteratively and by distance level ℓ, we start adding the sorted edges to the new MST, T'. As we do not know if the tiebreak applied to the remaining edges has changed we also need to check and compare all edges $(u, v) \in E$ in the original MST, T, for that same level ℓ, and also all relevant edges of u and v. We define (u, v') as a relevant edge of u if $D_{uv} = D_{uv'}$ and $v \neq v'$, meaning that the algorithm that generate that MST had to use a tiebreaker to decide which edge to add. This is a very important detail in this algorithm in the sense that a previously deferred edge may now become much more relevant than the one that was chosen.

For instance, assume that while we are running goeBURST Full MST algorithm we have two edges (u_1, v_1) and (u_1, v_2) with the same distance, $\ell = 1$. In order to break this tie we must use the weight function defined above, and start by checking the number of SLVs of v_1 and v_2. If v_1 has more OTUs at distance level one than v_2 we choose the edge (u_1, v_1) and deffer (u_1, v_2). If v_2 has more OTUs at distance level one than v_1 we choose the edge (u_1, v_2) and deffer (u_1, v_1). In the case that they both have the same number of SLVs then we proceed to comparing the number of DLVs, TLVs, and so on. Assume now that we chose the edge (u_1, v_1) because v_1 has the same SLVs but more DLVs than v_2 and that the algorithm finishes and computes a MST $\mathcal{T}$. If we now run dynamic goeBURST Full MST to add a new OTU u in $\mathcal{T}$ we start by updating the overall number of locus variants for all OTUs according to u. If we assume that the new edge (u, v_1) has $\ell = 2$ and (u, v_2) has $\ell = 1$, then the edge (u_1, v_2) becomes more relevant than (u_1, v_1) since the number of SLVs in (v_2) increases, and hence the tiebreak rule applied before (number of DLVs) has changed to the number of SLVs. Therefore, we will opt to add (u_1, v_2) instead of (u_1, v_1) as opposed to before.

An optimization is made after this process when T contains edges that have a lower distance level when comparing to the minimum level in E''. Thus, all edges at that minimum distance are added in T', directly.

This algorithm takes linear time on average for real data to add a new OTU, being also dominated by the time for pre-processing, and it can be defined as in algorithm 3.3.

3.3.2.1 Example

Given the MST tree, T, built by goeBURST Full MST (see fig. 2.7), and a new OTU u, we start by checking if it already exists in T. If exists, then the new OTU is discarded and the algorithm terminates. Otherwise, we compute all pairwise distances between the new OTU and the others on T, D_{uv}, using some similarity measure (assume $[1, 1, 3, 2, 4]$ to be the pairwise distances D_{uv} where $v = A, B, \ldots, E$ in T) and create new edges, (u, v), to represent them. Then, we sort those edges in increasing order of D_{uv}, resulting in the following list: $\{(u, A)$,(u, B), (u, D), (u, C), $(u, E)\}$. Till this point, this process has been exactly like in dynamic generic-distance-based-MST algorithm (section 3.3.1) because they share a common pre-processing step. Then, we must update the overall number of locus variants for all OTUs and sort the edges.

Now, we update the MST T by selecting all edges that are at distance level one ($\ell = 1$). By observing fig. 2.7 we know that no level one edges exist, hence we will only consider the new edges $(u, v) \in E''$, *i.e.*, $\{(u, A)$,$(u, B)\}$, that are at that exact level. Figure 3.2 (a) represents the tree $T' \leftarrow \{(u, A) \cup (u, B)\}$ of level one. Then, for level two, we have $(A, B) \in E$ and $(u, D) \in E''$ as possible edges. Because A and B are already in the tree T', we only update T' with (u, D), fig. 3.2 (b). At level three we terminate this algorithm since after updating T' with edges $(D, E) \in E$ and $(u, C) \in E''$ of distance level three, we

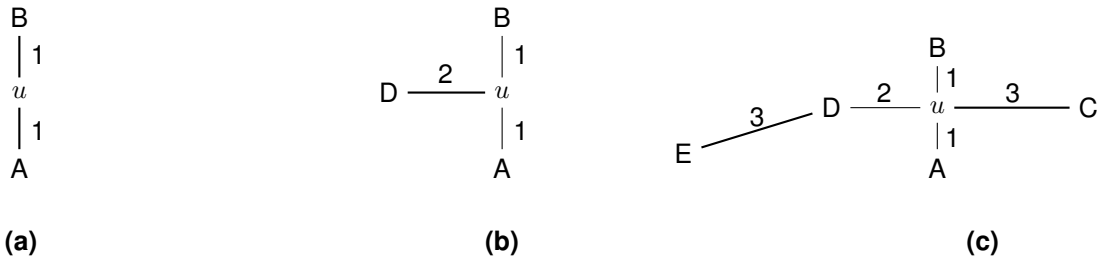

Figure 3.2: Step by step tree representation of dynamic goeBURST Full MST addition of a new OTU to a previously computed minimum spanning tree (see fig. 2.7).

no longer have any more OTUs left to be added to T'.

3.4 Discussion

In this chapter we defined what is a matroid and how greedy algorithms like goeBURST Full MST is effective at finding minimum spanning trees using a graphic matroid approach. Then, we propose an algorithm based on Borůvka approach, where given a graph $G = (V, E)$, with V vertices and E edges, it tries to build an optimal minimum spanning tree by merging different subtrees according to a specified level.

However, the main contributions of this chapter are the propose dynamic algorithms, dynamic generic-distance-based-MST and dynamic goeBURST Full MST, that will allow data to be continuously integrated without requiring us to compute everything from the start.

Next in chapter 4, we describe the implementation details of the proposed methods and we also identify possible computational bottlenecks and improvements that can be made to the implementation.

4

Implementation

Contents

As we previously discussed, generic-distance-based-MST and goeBURST Full MST algorithms are classified as distance matrix methods but, unlike the others, they follow a graph theoretical approach to infer phylogenies, hence building optimal minimum spanning trees as result. We implemented the latter using a greedy approach, *i.e.*, Borůvka algorithm, where each ST is considered as a separate tree and for each level (SLV, DLV, ..., nLV) we merge every tree with some other by selecting the cheapest (minimum allelic distance) edge of the given tree that is at that exact level.

Concerning dynamic graphs, several techniques have been evolving in recent years, which represents a major advantage to our study since the proposed dynamic algorithms rely on some of those techniques. The basic process of adding a new edge to a generic-distance-based-MST can be done by checking if that edge creates a cycle on the graph and, in that case, remove the heaviest edge from that same cycle. This way it is guaranteed that no cycle may occur in the spanning tree. However, each ST affects the overall number of locus variants for all STs, hence affecting the tiebreak rule applied on tied edges. These rules differ between the proposed algorithms in the sense that generic-distance-based-MST breaks the ties only according to the increasing order of the vertices' IDs, while dynamic goeBURST Full MST uses a more vast set of rules, as defined in goeBURST [11]. The latter dynamic algorithm solves this problem by updating first the number of locus variants for all STs and then, the entire MST. That may require the deletion and/or addition of previous deferred edges.

As you may have noticed, both dynamic algorithms depend on previously computed results, namely the MST. However, for the dynamic goeBURST Full MST we also need all edges that were deferred upon the application of a tiebreak rule, since the rule may have changed with the addition of new data. Thus, and in order to implement these algorithms efficiently we highly depend on efficient data structures, specially for dynamic algorithms since we want to integrate new data and update inferred evolutionary patterns faster than running the static algorithms from the scratch.

Therefore, in this chapter, we reason about our solutions for the proposed algorithms and discuss the implementation details for each one. We will start by briefly introducing part of the framework that gives support to these algorithms and then we describe the implementation of each one of them. Finally, we provide a tool that will allow you to run these algorithms.

All algorithms were implemented in *Java*.

4.1 Framework

goeBURST Full MST algorithm is classified as distance matrix method and therefore in order to implement it as we described in the last chapter we need to have a matrix as input. This matrix can be provided directly or generated from allelic profiles. This means that the pairwise dissimilarity can be obtained either by a distance vector that represents directly the distance from one element to the others or through several profile comparisons using some similarity measure (*e.g.* hamming's distance).

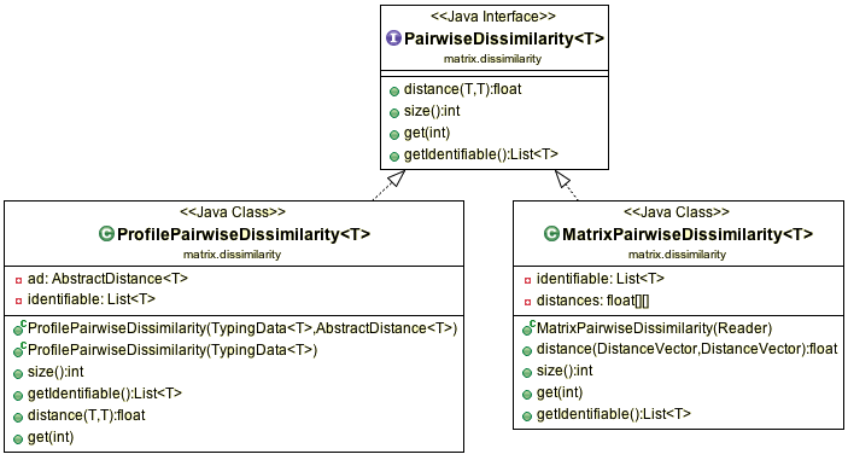

Figure 4.1: UML representation of `PairwiseDissimilarity` data structure that supports different input representations.

To allow these two types of input we created a generic interface called `PairwiseDissimilarity<T>`, where `T` corresponds to a specific `Identifiable`. This type is represented by an interface and can correspond either to a `DistanceVector` or to a `Profile` (see fig. 4.1). Hence, if we receive the input as a matrix we create a `MatrixPairwiseDissimilarity<T>` where we have direct access to the distances, otherwise we create a `ProfilePairwiseDissimilarity<T>` that contains the profiles and a defined similarity measure (`AbstractDistance<T>`) to provide the distances between them.

Now that we have obtained all distances regardless of the type of input provided, we can now build the final matrix that will support our algorithm. As one may recall from algorithm 2.1, namely step 2 and 3, this matrix needs to be dynamic in the sense that it must allow the insertion and removal of some of its elements and also allow distances to be updated. This structure was already implemented efficiently in [13] however it only supported profiles as input. Thus, we simply changed it to support both inputs instead of one, like it was explained before. This modification has no negative impact on its previous performance, quite the opposite, it can improve it. This relies on the fact that, if we receive a distance matrix instead of the profiles as input, we take no time on computing the distances between them as they are already computed. This will reflect highly on the algorithm's performance when we are analyzing thousands of profiles.

Now that we have explained how this framework gives support to the proposed algorithms, we will proceed with their implementations.

4.2 Static goeBURST Full MST

This algorithm starts by defining the maximum number of loci under analysis (*i.e.* maximum distance level) and then it computes for each OTU how many other OTUs are at distance one, two, etc. Then, it

runs Borůvka's algorithm adapted version by merging only components that are at the same level (*i.e.* equal weights). This last phase is described in algorithm 4.1.

It receives as input a matrix M, explained earlier, an integer value representing the maximum number of loci that we want to analyze and a graph G that represents the adjacency lists of all OTUs in the matrix M. These adjacencies are implemented using a map (`java.util.HashMap`) that give us a set of OTUs for a specific ID in constant time.

Starting at level one, $\ell = 1$, it iterates over all OTUs to obtain all edges that are at distance ℓ. Then, for each edge $(u, v) \in E$ if u and v are in different trees, we add that edge (u, v) to a new set E'. This new set must be an ordered set since it will have multiple edges with the same distance that need to be prioritized according to a set of tiebreak rules. To efficiently sort those edges we use a self-balancing binary search tree, more specifically a red-black tree (`java.util.TreeSet`), that uses a comparator based on those rules. Then, for each edge (u, v) in E' if u and v are in different trees, we add (u, v) to the MST $\mathcal{T}$. To trace the OTUs in each tree efficiently we use disjoint sets data structure [72]. This process repeats till all OTUs are in the final MST or the maximum level is reached.

The core of the algorithm is explained. However, if we look at algorithm 4.1 in detail we observe that there are some parts left to explain, namely lines 18-19 and 23. Briefly, they provide information about the order of construction of the MST (line 23) and also about all edges that were deferred by the comparator (lines 18-19). These lines will support the dynamic algorithms, as we shall see next on section 4.3, and they do not affect the full comprehension of this algorithm.

4.3 Dynamic MST algorithms

These dynamic algorithms allow the addition of a new OTU to a previously computed minimum spanning tree, T, without the need of running goeBURST Full MST algorithm from scratch. In order to do this, and for now, we will only need that previous MST, T. In this chapter, like in section 3.3, we will explain how we implemented the ADD operation, starting by explaining the pre-processing step, common to both algorithms, and then we provide details about the implementation of the adding steps for each one.

The pre-processing step is responsible for creating all structures that are necessary for the addition step. We start by creating a graph G to represent the MST T. Then, we need to check if the new OTU, u, that we are going to add already exists in T. This is done efficiently in constant time once we have the graph created. If it exists then we discard it and the algorithm finishes, otherwise, it continues by computing the distance from u to the other OTUs in T. Then, we create a list of new edges, E, for u. Finally, and before going to the addition step, we sort those edges, $E' \leftarrow sort(E, cmp)$, based on a comparator, cmp, defined by each algorithm.

This pre-processing step takes linear time in the size of the graph to process n edges and pairwise

Algorithm 4.1: Static goeBURST Borůvka's phase pseudocode.

```
1  Input: A matrix M representing all pairwise dissimilarities between OTUs.
2         G, Graph representing the adjacency list for all OTUs in M.
3         maxLevel, maximum distance level.
4  Output: A newly computed MSTResult, R.
5
6  begin
7      set ← newDisjointSet()
8      T ← newMST()
9      E' ← newTreeSet()
10     ℓ ← 1
11     while T.size() < M.size() − 1 && ℓ ≤ maxLevel do
12         foreach u ∈ M do
13             E ← G.getNeighborsEdges(u, ℓ)
14             foreach (u, v) ∈ E do
15                 if notSameSet(set, u, v) then
16                     E' ← E' ∪ (u, v)
17         foreach (u, v) ∈ E' do
18             addCluster(R, ℓ, getEdges(u, ℓ))
19             addCluster(R, ℓ, getEdges(v, ℓ))
20             if notSameSet(set, u, v) then
21                 T ← T ∪ (u, v)
22                 union(set, u, v)
23                 mergeClusters(R, ℓ, u, v)
24         E' ← newTreeSet()
25     addTree(R, T)
26     return R
27 Finalize: Return MSTResult R.
```

comparisons.

As we discussed, the operation ADD as two steps, one that is shared between both dynamic algorithms (*i.e.* pre-processing step) and another that is dependent on each one. Thus, to support this ADD operation we implemented the abstract class `MSTDynamicDistanceBasedRunner`. This class only contains the pre-processing step and delegates on concrete dynamic implementations the step of adding the new OTU. Therefore, two classes were created, `MSTGenericDynamicDistanceBasedRunner` and `MSTDynamicgoeBURSTRunner`, that extend the abstract class, in order to implement their particular adding step.

Next, we provide details about the implementation of the adding steps for both methods.

4.3.1 Dynamic generic-distance-based-MST algorithm

As we can see in algorithm 4.2, the process of adding a new OTU to a MST is simple because we only have to detect if a cycle is present or not. This can be done using a queue (`java.util.Queue`) to

Algorithm 4.2: Dynamic generic-distance-based-MST's BFS pseudocode.

```
1  Input: (u, v) ∈ E', new edge added to the tree T'.
2         G, Graph representing the adjacency list for all OTUs.
3  Output: Path that forms the cycle, p.
4
5  begin
6      Q ← newQueue()
7      add(Q, u)
8      root ← u
9      while notEmpty(Q) do
10         u ← poll(Q)
11         setVisited(u)
12         N ← G.getNeighbors(u)
13         foreach nb ∈ N do
14             if nb == v then
15                 while nb ≠ root do
16                     add(p, nb)
17                     nb ← parent(nb)
18                 return p
19             if notVisited(nb) then
20                 add(Q, nb)
21                 parent(nb) ← u
22     return p
23 Finalize: Return path p.
```

store all OTUs (neighbors) that were still not visited by BFS, an array of boolean values (`boolean[]`) to mark each OTU as visited or not and an array of integers (`int[]`) to store the parent ID of each OTU. This last array will be very useful to form the path soon as we find the cycle.

Suppose that we add an edge $(u, v) \in E'$ to the tree. We start by defining u as the root of our tree and then we look for v by going through all neighbors of u that were not visited yet and enqueuing them. These neighbors can be found efficiently using the graph G data structure that represents the adjacency lists of all OTUs using an `HashMap` and that was created in the pre-processing step. If we cannot find v, then we restart this process for every enqueued neighbor, otherwise, we backtrace the all process through the parents' array adding each to the cycle path. This process pseudocode is described in algorithm 4.2.

The adding stage for this dynamic algorithm only depends on the time took to detect the presence of a cycle. For that reason it takes linear time in the size of the graph to add a new OTU in generic-distance-based-MST.

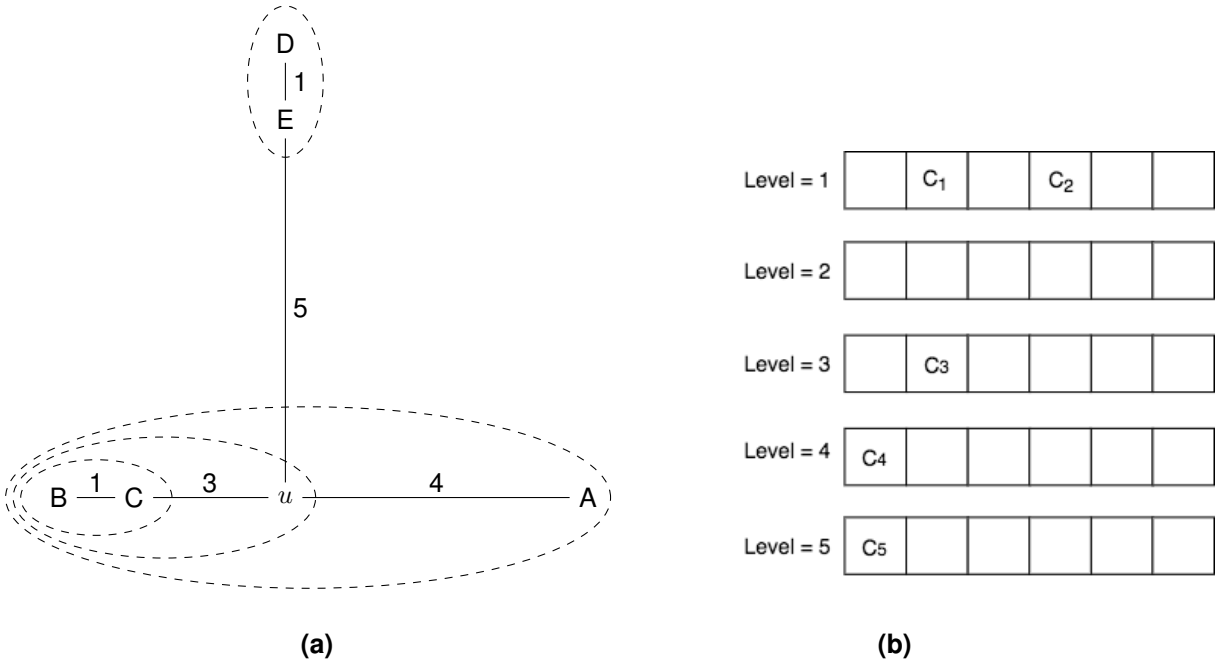

Figure 4.2: Phylogenetic tree represented by different clusters, where $C_1 = \{(B, C)\}$, $C_2 = \{(D, E)\}$, $C_3 = C_1 \cup \{(C, u)\}$, $C_4 = S_3 \cup \{(A, u)\}$ and $C_5 = C_4 \cup \{(E, u)\}$.

4.3.2 Dynamic goeBURST Full MST

The implementation of this algorithm is a little bit more complex than the generic-distance-based-MST one. By going over algorithm 4.3 we see that it receives as input an ordered list of edges E' and a graph G that were created in the pre-processing step, a maximum distance level and also, a `MSTResult`, R. Till this point, we have only needed from a previous computation a MST. However, we have mentioned that this algorithm also need to know how this tree was created, *i.e.*, for each level which were the subtrees that were clustered. Therefore, to represent each subtree, we created a new data structure called `MSTCluster` that contains a list of edges. Not only the ones that represent it but also all edges that were deferred at each level. This is due to the fact that the overall number of locus variants of the OTUs in the MST could vary according to the new distances to the new OTU, causing a possible change on the tiebreak rule. So, the `MSTResult` data structure that we receive as input contains not only the minimum spanning tree as a collection of edges but also a collection of clusters. These clusters are represented in fig. 4.2 (b), according to the example in (a), after the execution of goeBURST Full MST, for all distance levels ℓ. We can observe that the set of clusters $\{C_1, C_2, \ldots, C_5\}$ represent exactly how the tree was built. Starting at $\ell = 1$, we have two clusters, C_1 and C_2, containing the edges $\{(B, C)\}$ and $\{(D, E)\}$, respectively. At $\ell = 2$ we did not add any edge to any cluster. At $\ell = 3$, we add $\{(C, u)\}$ to C_1 in C_3, at $\ell = 4$ $C_4 \leftarrow C_3 \cup \{(A, u)\}$ and finally, for $\ell = 5$, $C_5 \leftarrow C_4 \cup \{(E, u)\}$.

The algorithm starts by updating the overall number of locus variants for each OTU. Then, iteratively

Algorithm 4.3: Dynamic goeBURST Full MST addition step pseudocode.

```
1  Input: E', an ordered list of newly created edges.
2         u, the new OTU to be added.
3         R, MSTResult from a previous execution.
4         G, Graph representing the adjacency list for all OTUs.
5         cmp, weight edges comparator.
6         maxLevel, maximum distance level.
7  Output: A newly computed MSTResult, R'.
8
9  begin
10   foreach OTUv ∈ getMST(R) do
11     updateLVs(u, v)
12   E' ← sort(E', cmp)
13
14   set ← newDisjointSet()
15   T' ← newMST()
16   for ℓ = 1; ℓ ≤ maxLevel; ℓ++ do
17     E'' ← getEdges(R, ℓ) ∪ getEdges(E', ℓ)
18     foreach (u, v) ∈ E'' do
19       addCluster(R', ℓ, G.getNeighborsEdges(u, ℓ))
20       addCluster(R', ℓ, G.getNeighborsEdges(v, ℓ))
21       if notSameSet(set, u, v) then
22         T' ← T' ∪ (u, v)
23         union(set, u, v)
24         mergeClusters(R', ℓ, u, v)
25   addTree(R', T')
26   return R'
27 Finalize: Return MSTResult R'.
```

and by distance level ℓ, selects all edges from all clusters that are at distance level ℓ and all new edges from E' that are at the same distance and adds them on a new set E''. For each edge $(u, v) \in E''$, we update the clusters for u and v with all edges that are at distance ℓ from u and v, respectively, and if it connects different subtrees in $\mathcal{T}'$ we add it in the new tree $\mathcal{T}'$ and we merge the previous two clusters into one.

Now we can better understand why the `MSTResult` data structure is so important when we want to add a new OTU to a MST and maintain its structure exactly like if it was created by static goeBURST Full MST algorithm. This also explains the purpose of lines 18-19 and 23 of algorithm 4.1. This algorithm uses that data structure and also computes a new one in order to allow a possible second addition of a new OTU to the resulting tree. In other words, instead of only using the structure that represents the result of running static goeBURST Full MST algorithm on a undirected weighted graph for n OTUs to add a new OTU, it also computes a new structure that represents the updated graph for $n + 1$ OTUs for a future dynamic addition. If we have only chosen to use and not to update the previous `MSTResult`, then we would have to run static goeBURST Full MST from scratch for those $n + 1$ OTUs and then this

algorithm for that second addition or running just the static version for $n + 2$ OTUs, which was not a desirable solution.

The adding stage for this dynamic algorithm depends on the time took to get all edges for each level, the time to sort them and also the time took to create and update the clusters. These operations on clusters, `addCluster` and `mergeClusters` depend on the total number of edges they contain. For all these reasons this algorithm takes a quasilinear time in the worst case, $O(n \log n)$ in the size of the graph to add a new OTU in a minimum spanning tree.

4.4 Data persistence

After running all the proposed algorithms they output a phylogenetic tree in newick format. However, goeBURST Full MST and dynamic goeBURST Full MST algorithms also output a `.result` file that corresponds to the serialization of the *Java Object* `MSTResult`. This file can then be used by any dynamic algorithm, through deserialization, to allow the addition of a new element to it.

This solution for loading data can be discussed since dynamic generic-distance-based-MST algorithm may not need the information of the entire tree and consequently only loading into memory the information regarding the path found by breadth-first-search. This can be made by considering other memory-based solutions like memory-mapping although they are not trivial and do not bring any greater benefit to our problem because all these are offline algorithms and they all use secondary memory to persist their results.

4.5 Tool

We implemented these algorithms independently in *Java* to allow portability and an easy integration with PHYLOViZ framework by adding to it the resulting *jars* as modules. Therefore, in order to execute them you just have to run the corresponding command:

Run goeBURST Full MST:

```
> java -jar phyloviz-goeBURSTFullMST.jar -i <input-data> -l <level>
```

Run dynamic generic-distance-based-MST:

```
> java -jar phyloviz-dynamic-generic-distancebased-mst.jar -i <input-data> -r <base_tree>
```

Run dynamic goeBURST Full MST:

```
> java -jar phyloviz-dynamic-goeBURSTFullMST.jar -i <input-data> -r <base_tree> -l <level>
```

For instance, if you wish to run a dynamic algorithm you need to have the computed result from a previous execution in a file with extension `.result`, otherwise you will need to run goeBURST Full MST first. Then, you just need to provide that file to the dynamic algorithm by replacing `<base_tree>` with the corresponding file (*e.g.* `goeburst-full-mst.result`) along with the element you want to add in `<input-data>` to get the updated tree.

Assume that you have a file `new-element.txt` that contains a new element to be dynamically added (either a profile or a distance-vector) and we have the file `goeburst-full-mst.result` representing a previous execution. We now can execute a dynamic algorithm by running one of the following commands:

- `> java -jar phyloviz-dynamic-generic-distancebased-mst.jar -i new-element.txt -r goeburst-full-mst.result`.
- `> java -jar phyloviz-dynamic-goeBURSTFullMST.jar -i new-element.txt  -l 5 -r goeburst-full-mst.result`.

For a full description of these commands just run `java -jar phyloviz-<method>.jar -h`, according to the chosen `<method>`. These algorithms are also available in https://gitlab.com/martanascimento/dynamic-phyloviz/wikis/home.

4.6 Discussion

In this chapter we detailed the implementation of three algorithms, goeBURST Full MST, dynamic generic-distance-based-MST and dynamic goeBURST Full MST. We developed several data structures to implement these algorithms efficiently, along with a stable framework to support our algorithms, taking always into consideration both time and memory requirements.

These structures were based on an existing implementation of Holm, Rotenberg, and Wulff-Nilsen [74] for updating a fully dynamic minimum spanning forest. However, their data structure was design to support insertion and deletion of edges, unlike our implementation, that only supports insertions. They start by defining $G = (V, E)$ as a dynamic graph and its data structure maintains, for each edge, a level which is an integer between 0 and $\log n$. As we have seen, we also associate each edge with a level, but it represents the distance between the two vertices, unlike Holm's, that defines this level as the depth of the tree in which that edge exists. Then, is defines the subtrees as clusters, for each level, containing all edges that form those subtrees at that same level. It also stores for each vertex the weight of the cheapest edge that is linked to it so that when it inserts an edge $e = (u, v)$, where u and v belong to the same clusters, it can easily find the path that connects both of them.

Although there are well known dynamic algorithms for computing and updating minimum spanning trees, they are not directly usable in this context due to distance updating and tie breaking rules based on locus variants.

Therefore, we contributed with practical and scalable implementations of the proposed methods providing always an optimal solution.

In the next chapter we evaluate the performance of the dynamic algorithms to check if they bring a great advantage on integrating new data and updating inferred evolutionary patterns.

5

Experimental Evaluation

Contents

In this chapter we consider three implementations in *Java* of the described algorithms, concerning both time and memory requirements.

To evaluate the performance of the dynamic algorithms, we ran them against our implementation of goeBURST Full MST where we can observe the efficiency of our solutions. We performed the tests for the *Streptococcus pneumoniae* MLST dataset available in PubMLST[1], with the number of OTUs varying from $n = 10$ to $n = 1000$ and running times averaged over 500 executions. We also used two Single Nucleotide Polymorphism (SNP) [87] datasets, one with random data and *Salonella typhi*, with the number of OTUs varying from $n = 10$ to $n = 400$ and $n = 10$ to $n = 1000$, respectively, and running times averaged over 100 executions.

MSLT dataset was specially chosen because it is part of several published studies and also because its access is public, which will facilitate the interpretation of the results. Regarding SNP datasets, we chose *Salonella typhi* to represent real data, with profile length of 22143 and a simulated one with a profile length of 200, instead of 7 like the MLST dataset.

All experiments were performed on a machine with the following specifications: Intel Core i5 2.7 GHz quad core processor with 8 GB of memory.

5.1 Time

In this section, we are going to start by analyzing the MLST dataset and its properties and then we will move on to the SNP datasets. To analyze the running time of the algorithms we will start by providing a time-table and some graphical comparisons of those results.

5.1.1 *Streptococcus pneumoniae* MLST dataset

This dataset contains a profile length of 7 and a total of 1000 profiles (OTUs). If we look at table 5.1, that represents a time comparison between the three different algorithms, we can observe the time, in milliseconds, that the static algorithm takes to add n OTUs against the time that both dynamic algorithms take to add a new OTU to a MST with $n-1$ OTUs. For instance, goeBURST Full MST takes about ≈ 400 ms to add 500 OTUs, while dynamic generic-distance-based-MST and dynamic goeBURST Full MST take much less time (about ≈ 10 ms and ≈ 30 ms, respectively) to add a new OTU to a previously computed 499 OTUs MST. This clearly shows the great advantage of the dynamic algorithms on integrating new data and updating inferred evolutionary patterns.

We can also make a direct comparison between both dynamic algorithms, represented by fig. 5.1(a). We can observe that generic-distance-based-MST is faster than the dynamic goeBURST Full MST. This analysis supports our study since the dynamic generic-distance-based-MST process of updating

[1] https://pubmlst.org

Table 5.1: Time analysis, in milliseconds, of *Streptococcus pneumoniae* MLST dataset for the proposed algorithms.

n	goeBURST Full MST	Dynamic generic-distance-based-MST	Dynamic goeBURST Full MST
10	0.352	0.02	0.1
20	0.672	0.052	0.156
50	2.848	0.204	0.914
100	8.836	0.836	1.938
200	42.786	3.34	6.14
300	109.476	5.65	12.34
400	229.586	7.72	18.25
500	389.086	10.344	29.5
600	561.156	14.792	35.3
700	860.421	34.836	42.6
800	1109.334	33.78	48.1
900	1488.936	39.274	58.2
1000	1871.678	44.61	77.9

a minimum spanning tree is much simpler because it only depends on the pairwise distances between OTUs. This algorithm just checks if an edge creates a cycle on the graph and, if that is the case, it removes the heaviest edge from that same cycle. However, this difference is not substantial when we compare it with goeBURST Full MST because any of them is much better than the latter on integrating new data.

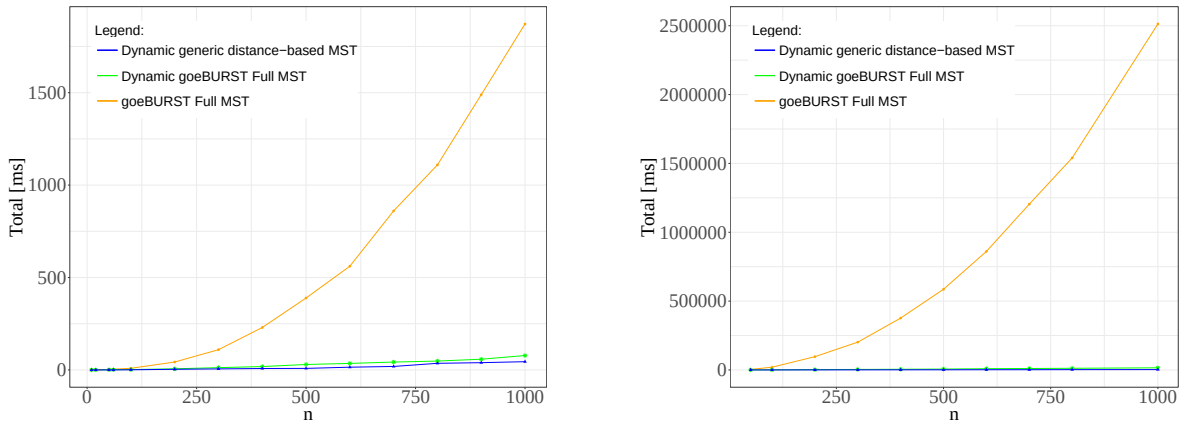

Figure 5.1: Dynamic vs static algorithms running time comparison as new OTUs are incrementally integrated for: (a) *Streptococcus pneumoniae* MLST dataset and (b) *Salonella typhi* SNPs dataset.

Table 5.2: Time analysis, in milliseconds, of simulated SNPs dataset for the proposed algorithms.

n	goeBURST Full MST	Dynamic generic-distance-based-MST	Dynamic goeBURST Full MST
10	1.05	0.143	0.586
20	3.897	0.276	1.307
30	10.53	0.424	2.501
40	21.123	0.614	3.753
50	35.398	0.812	5.95
60	51.79	0.992	7.235
70	72.374	1.272	9.663
80	92.769	1.468	11.956
90	131.731	1.866	14.519
100	160.214	2.142	17.699
150	402.83	3.674	36.64
200	786.802	4.918	64.992
250	1420.148	6.72	101.853
300	1871.72	1871.72	142.963
350	2645.7	9.72	205.914
400	3600.32	11.28	251.163

5.1.2 SNP datasets

Salonella typhi dataset have a profile length of 22143 while the one that represents randomized sequences of single nucleotide polymorphism have a profile length of 200. This means that in order to compute the distances between them we must do a comparison between 22143 and 200 loci, respectively, for each pair of OTUs. These pairs grow quadratically with the number of OTUs, which can be computationally intensive in terms of time and memory for large data sets. We can observe the impact that it causes specially in the pre-processing step of the dynamic algorithms, when comparing this step in MLST and *Salonella typhi* datasets, fig. 5.2. We can see that for the same algorithm, the time that takes to create all structures necessary to the adding step increase dramatically with the total number of loci under analysis.

By observing table 5.2 that fully details the computational time of the algorithms on the simulated SNPs dataset, and also fig. 5.1(b), we can conclude that despite the increase in the overall time, these dynamic algorithms still represent an improvement over the goeBURST Full MST algorithm.

5.2 Memory

In this section we analyze *Streptococcus pneumoniae* dataset in terms of memory requirements (table 5.3).

The results in the table were obtained using the *Java* interface `MemoryPoolMXBean`. It represents a management interface of the memory resources managed by *Java* virtual machine. This way we can

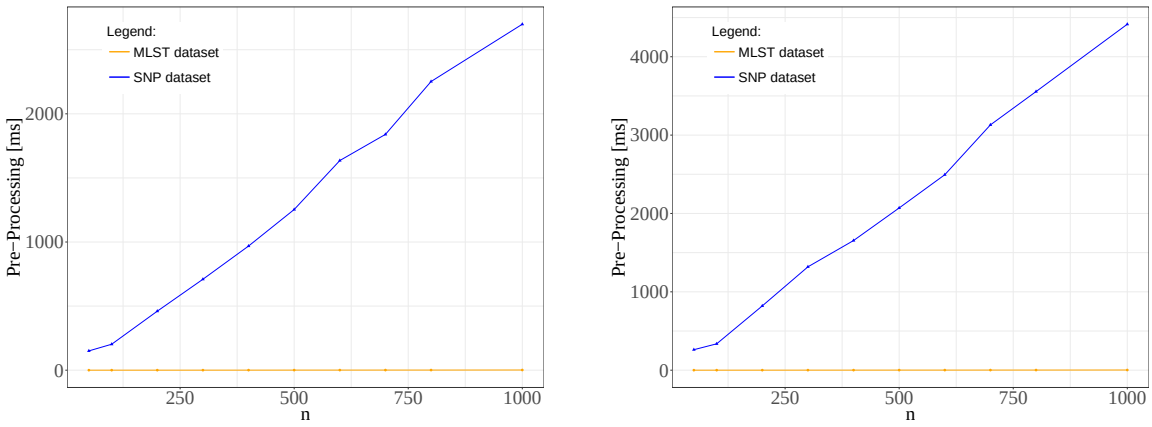

Figure 5.2: *Streptococcus pneumoniae* MLST vs *Salonella typhi* SNPs dataset for pre-processing n OTUs by: (a) Dynamic generic-distance-based-MST and (b) dynamic goeBURST Full MST.

get the peak of memory usage of a memory pool since the virtual machine was started.

Both dynamic algorithms use about the same amount of resources but more when comparing to goeBURST Full MST. This is due to all the structures that they need beforehand to execute, unlike the static algorithm that starts from scratch. So, it is interesting to note that, although it is more or less the double, it grows linearly, which is acceptable since we are supporting a dynamic structure.

5.3 Discussion

In this chapter we discussed the implications of our solutions in terms of time and memory requirements. The most important results are related with the dynamic algorithms' computational cost when

Table 5.3: Memory analysis, in megabytes, of *Streptococcus pneumoniae* dataset for the proposed algorithms.

n	goeBURST Full MST	Dynamic generic-distance-based-MST	Dynamic goeBURST Full MST
10	3.250382423	3.900232315	3.90038681
100	4.55062294	5.200351715	5.200494766
200	8.450428963	12.35045147	11.05061436
300	13.1957159	28.60437965	25.35048485
400	19.82765388	34.25072193	34.25197029
500	28.47127533	35.16017723	34.80991364
600	52.11545753	35.31921577	35.30359268
700	51.84895802	68.43165588	68.42102814
800	52.27539063	119.0787468	90.58642387
900	73.31301022	137.144062	137.2128458
1000	74.9512825	187.713028	156.5607862

comparing to goeBURST Full MST since the firsts represent a notable improvement over the latter.

A major issue in graph mining is the efficiency of algorithms and data structures, in particular when we process large data. We consider our implementations efficient in the way that they provide a clear improvement over state of the art graph clustering methods concerning dynamic graphs. However, the running time of these algorithms can still be improved by creating new data structures better suitable to graph representations, namely for our computed results.

6

Conclusion

Contents

Large epidemiological studies on pathogen populations start to emerge as sequencing technologies become commodity, continuously generating huge volumes of typing data, and also ancillary data. And there is no doubt about the importance of such studies for the surveillance of infectious diseases and the understanding of pathogen population genetics and evolution. There are still however a number of challenges. Phylogenetic analysis is one of the main tools used in this context and, although there are many phylogenetic inference methods as we saw before, their differences and similarities are not clear most of the time. On the other hand, given the huge volume of ever growing data to analyse, many methods are becoming unpractical due to their computational complexity.

6.1 Conclusions

We provide in this paper an unifying view on most well known phylogenetic inference methods suitable for processing typing data. As we discussed, these methods share a common algorithmic background and differ only on optimization criteria and related evolution models. Taking this observations into account, one can better understand the difference among these methods and, from a computational point of view, can address simultaneously several challenges in common with all algorithms. Moreover, given that datasets are not only huge but are growing continuously, dynamic updating is becoming mandatory. And given the strict relation between clustering and these methods, we believe that well known techniques developed in last years for clustering large data can be useful in this context.

Based on the study presented here, we implemented goeBURST Full MST method and two dynamic updating techniques: dynamic generic-distance-based-MST and dynamic goeBURST Full MST algorithms. Although there are well known dynamic algorithms for computing and updating minimum spanning trees, they are not directly usable in this context due to distance updating and tie breaking rules based on locus variants.

The main contributions of this thesis are the unified view of the most commonly used inference methods and the improvements over state of the art graph clustering methods in what concerns dynamic graphs. In particular we contributed with practical and scalable implementations of the proposed methods providing always an optimal solution.

6.2 Future Work

There are several possible continuations of the work done in this thesis. One could extend this study to several different algorithms. We have already provided an explication on how globally closest pairs methods can be dynamically updated, and therefore it could be a good starting point to try to apply some of the techniques that were explained in that context to other algorithms. It will also be interesting to extend this work by studying and implementing the operations UPDATE and DELETE on the dynamic

algorithms, instead of just the ADD.

Most of the discussed hierarchical clustering methods were developed a long time ago, hence studying different ways to optimize these methods could have a major impact on their performances and, thereby, their usability.

Another interesting work would be how would different evolution models impact on the results of an algorithm in computing the phylogenetic tree and how close their results are when comparing to the real tree.

However, exploring new methods and new dynamic techniques are, in our opinion, the most interesting directions for future work.

Bibliography

[1] J. A. Reuter, D. V. Spacek, and M. P. Snyder, "High-Throughput Sequencing Technologies", *Molecular Cell*, vol. 58, no. 4, pp. 586–597, 2015, ISSN: 1097-2765. DOI: 10.1016/j.molcel.2015.05.004.

[2] M. C. J. Maiden, J. A. Bygraves, E. Feil, G. Morelli, J. E. Russell, R. Urwin, Q. Zhang, J. Zhou, K. Zurth, and D. A. Caugant, "Multilocus sequence typing: A portable approach to the identification of clones within populations of pathogenic microorganisms", *Proceedings of the National Academy of Sciences*, vol. 95, 1998. DOI: 10.1073/pnas.95.6.3140.

[3] B. Lindstedt, "Multiple-locus variable number tandem repeats analysis for genetic fingerprinting of pathogenic bacteria", *ELECTROPHORESIS*, vol. 26, no. 13, pp. 2567–2582, 2005, ISSN: 1522-2683. DOI: 10.1002/elps.200500096.

[4] *A collaborative learning space for science - SNP*. [Online]. Available: http://www.nature.com/scitable/definition/single-nucleotide-polymorphism-snp-295.

[5] R. M. Bush, W. M. Fitch, C. A. Bender, and N. J. Cox, "Positive selection on the H3 hemagglutinin gene of human influenza virus A.", *Molecular Biology and Evolution*, vol. 16, no. 11, pp. 1457–1465, 1999.

[6] D. M. Hillis, J. P. Huelsenbeck, C. W. Cunningham, *et al.*, "Application and accuracy of molecular phylogenies", *Science-AAAS-Weekly Paper Edition-including Guide to Scientific Information*, vol. 264, no. 5159, pp. 671–676, 1994.

[7] M. L. Metzker, D. P. Mindell, X.-M. Liu, R. G. Ptak, R. A. Gibbs, and D. M. Hillis, "Molecular evidence of HIV-1 transmission in a criminal case", *Proceedings of the National Academy of Sciences*, vol. 99, no. 22, pp. 14 292–14 297, 2002. DOI: 10.1073/pnas.222522599.

[8] K. A. Crandall, O. R. P. Bininda-Emonds, G. M. Mace, and R. K. Wayne, "Considering evolutionary processes in conservation biology", *Trends in Ecology & Evolution*, vol. 15, no. 7, pp. 290–295, Jul. 2000, ISSN: 0169-5347. DOI: 10.1016/S0169-5347(00)01876-0.

[9] E. P. Martins, *Phylogenies and the comparative method in animal behavior*. Oxford University Press on Demand, 1996.

[10] D. Darriba, M. Weiß, and A. Stamatakis, "Prediction of missing sequences and branch lengths in phylogenomic data", *Bioinformatics*, vol. 32, no. 9, pp. 1331–1337, 2016.

[11] *Francisco AP, Bugalho MF, Ramirez M, Carri{ç}o JA: Global optimal eBURST analysis of multilocus typing data using a graphic matroid approach. BMC Bioinf 2009., 10(152):*

[12] A. P. Francisco, C. Vaz, P. T. Monteiro, J. Melo-Cristino, M. Ramirez, and J. A. Carriço, "PHYLOViZ: phylogenetic inference and data visualization for sequence based typing methods", *BMC Bioinformatics*, vol. 13, no. 1, p. 87, 2012, ISSN: 1471-2105. DOI: 10.1186/1471-2105-13-87.

[13] M. Nascimento, A. Sousa, M. Ramirez, A. P. Francisco, J. A. Carriço, and C. Vaz, "PHYLOViZ 2.0: providing scalable data integration and visualization for multiple phylogenetic inference methods.", *Bioinformatics (Oxford, England)*, btw582, 2016, ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btw582.

[14] A. P. Francisco, M. Nascimento, and C. Vaz, "Dynamic phylogenetic inference for sequence-based typing data", in *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, ser. ACM-BCB '17, Boston, Massachusetts, USA: ACM, 2017, pp. 604–604, ISBN: 978-1-4503-4722-8. DOI: 10.1145/3107411.3108214. [Online]. Available: http://doi.acm.org/10.1145/3107411.3108214.

[15] N. Saitou, *Introduction to evolutionary genomics*. Springer, 2013.

[16] S. H. Bokhari and D. Janies, "Reassortment networks for investigating the evolution of segmented viruses", *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 7, no. 2, pp. 288–298, 2010.

[17] D. Huson, R. Rupp, and C. Scornavacca, *Phylogenetic Networks: Concepts, Algorithms and Applications*. 2011, vol. 1, p. 362, ISBN: 9780521755962.

[18] R. W. Hamming, *Error Detecting and Error Correcting Codes*, 1950. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[19] S. Fortunato, "Community detection in graphs", *Physics Reports*, vol. 486, no. 3-5, pp. 75–174, 2010, ISSN: 03701573. DOI: 10.1016/j.physrep.2009.11.002. arXiv: 0906.0612.

[20] A. K. Jain and S. Maheswari, "Survey of Recent Clustering Techniques in Data Mining", *International Archive of Applied Sciences and Technology*, vol. 3, no. June, pp. 68–74, 2012.

[21] J. Hein, M. Schierup, and C. Wiuf, *Gene genealogies, variation and evolution: A primer in coalescent theory*. Oxford University Press, USA, 2004.

[22] M. Kimura and J. F. Crow, “The number of alleles that can be maintained in a finite population”, *Genetics*, vol. 49, no. 4, p. 725, 1964.

[23] T. Ohta and M. Kimura, “A model of mutation appropriate to estimate the number of electrophoretically detectable alleles in a finite population”, *Genetics Research*, vol. 22, no. 2, pp. 201–204, 1973.

[24] T. H. Jukes and C. R. Cantor, “Evolution of protein molecules”, *Mammalian protein metabolism*, vol. 3, no. 21, p. 132, 1969.

[25] M. Kimura, “A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences”, *Journal of molecular evolution*, vol. 16, no. 2, pp. 111–120, 1980.

[26] J. Felsenstein, “Evolutionary trees from dna sequences: A maximum likelihood approach”, *Journal of molecular evolution*, vol. 17, no. 6, pp. 368–376, 1981.

[27] M. Hasegawa, H. Kishino, and T.-a. Yano, “Dating of the human-ape splitting by a molecular clock of mitochondrial dna”, *Journal of molecular evolution*, vol. 22, no. 2, pp. 160–174, 1985.

[28] C. A. Stewart, *Computational Biology*. 2003, ISBN: 9781447153030. [Online]. Available: http://hdl.handle.net/2022/14000.

[29] N. Saitou and T. Imanishi, “Relative efficiencies of the Fitch-Margoliash, maximum-parsimony, maximum-likelihood, minimum-evolution, and neighbor-joining methods of phylogenetic tree construction in obtaining the correct tree”, *Mol. Biol. Evol*, vol. 6, no. 5, pp. 514–525, 1989, ISSN: 0737-4038.

[30] N. Saitou, “Genomu Shinkagaku Nyumon”, *Tokio: Kyoritsu-Shuppan (in Japanese)*, 2007.

[31] W. M. Fitch and E. Margoliash, “Construction of phylogenetic trees”, *Science*, no. 155, pp. 279–284, 1967.

[32] A. W. F. Edwards and L. L. Cavalli-Sforza, “A method for cluster analysis”, *Biometrics*, vol. 21, no. 2, pp. 362–375, 1965. DOI: 10.2307/2528096.

[33] N. Saitou and M. Nei, “The neighbour-joining method: a new method for reconstructing phylogenetic trees”, *Mol Biol Evo*, vol. 4, no. 4, pp. 406–425, 1987.

[34] a. Rzhetsky and M. Nei, “A Simple Method for Estimating and Testing Minimum-Evolution Trees”, *Molecular Biology and Evolution*, vol. 9, no. 5, pp. 945–967, 1992, ISSN: 07374038.

[35] J. H. Camin and R. R. Sokal, “A Method for Deducing Branching Sequences in Phylogeny”, *Evolution*, vol. 19, no. 3, pp. 311–326, 1965, ISSN: 00143820, 15585646. DOI: 10.2307/2406441.

[36] J. Felsenstein, “Evolutionary trees from DNA sequences: A maximum likelihood approach”, *Journal of Molecular Evolution*, vol. 17, no. 6, pp. 368–376, 1981, ISSN: 1432-1432. DOI: 10.1007/BF01734359.

[37] R. Sokal and C. Michener, *A Statistical Method for Evaluating Systematic Relationships*, 38th ed. University of Kansas Science Bulletin, 1958, pp. 1409–1438.

[38] R. Sibson, “SLINK: An optimally efficient algorithm for the single-link cluster method”, *The Computer Journal*, vol. 16, no. 1, pp. 30–34, 1973. DOI: 10.1093/comjnl/16.1.30.

[39] D. Defays, “An efficient algorithm for complete link method”, *The Computer Journal*, vol. 20, no. 4, pp. 364–266, 1977. DOI: 10.1093/comjnl/20.4.364.

[40] P. H. A. Sneath and R. R. Sokal, “Numerical taxonomy. The principles and practices of numerical classification”, *WF Freeman and Co., San Francisco*, vol. 573, 1973.

[41] J. Studier and K. Kepler, “A note on the neighbour-joining method of Saitou and Nei.”, *Molecular Biology and Evolution*, vol. 5, no. 6, pp. 729–731, 1988.

[42] O. Gascuel, “BIONJ: An Improved Version of the NJ Algorithm Based on a Simple Model of Sequence Data”, *Mol. Biol. Evol*, vol. 14, pp. 685–695, 1997.

[43] ——, “Concerning the NJ algorithm and its unweighted version, UNJ”, *Mathematical hierarchies and biology*, vol. 37, pp. 149–171, 1997.

[44] M. Simonsen, T. Mailund, and C. N. S. Pedersen, “Rapid Neighbour-Joining”, in *Algorithms in Bioinformatics: 8th International Workshop, WABI 2008, Karlsruhe, Germany, September 15-19, 2008. Proceedings*, K. A. Crandall and J. Lagergren, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 113–122, ISBN: 978-3-540-87361-7. DOI: 10.1007/978-3-540-87361-7_10.

[45] M. Simonsen, T. Mailund, and C. N. S. C. N. S. Pedersen, “Building Very Large Neighbour-Joining Trees”, *Proceedings of the First International Conference on Bioinformatics*, pp. 26–32, 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.154.1594%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[46] K. Howe, A. Bateman, and R. Durbin, “QuickTree: building huge Neighbour-Joining trees of protein sequences.”, *Bioinformatics (Oxford, England)*, vol. 18, no. 11, pp. 1546–7, 2002, ISSN: 1367-4803. DOI: 10.1093/bioinformatics/18.11.1546.

[47] T. Mailund and C. N. S. Pedersen, “QuickJoin — Fast Neighbour-Joining Tree Reconstruction”, *Bioinformatics*, vol. 20, no. 17, pp. 3261–3262, 2004. DOI: 10.1093/bioinformatics/bth359.

[48] T. J. Wheeler, “Large-Scale Neighbor-Joining with NINJA”, in *Algorithms in Bioinformatics: 9th International Workshop, WABI 2009, Philadelphia, PA, USA, September 12-13, 2009. Proceedings*, S. L. Salzberg and T. Warnow, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 375–389, ISBN: 978-3-642-04241-6. DOI: 10.1007/978-3-642-04241-6_31.

[49] J. Wang, M. Guo, and L. L. Xing, “FastJoin, an improved neighbor-joining algorithm”, *Genetics and Molecular Research*, vol. 11, no. 3, pp. 1909–1922, 2012.

[50] L. Sheneman, J. Evans, and J. A. Foster, “Clearcut: A fast implementation of relaxed neighbor joining”, *Bioinformatics*, vol. 22, no. 22, pp. 2823–2824, 2006, ISSN: 13674803. DOI: 10.1093/bioinformatics/btl478.

[51] V. Lefort, R. Desper, and O. Gascuel, “FastME 2.0: A comprehensive, accurate, and fast distance-based phylogeny inference program”, *Molecular Biology and Evolution*, vol. 32, no. 10, pp. 2798–2800, 2015, ISSN: 15371719. DOI: 10.1093/molbev/msv150.

[52] I. Elias and J. Lagergren, “Fast neighbor joining”, *Theoretical Computer Science*, vol. 410, no. 21-23, pp. 1993–2000, 2009, ISSN: 03043975. DOI: 10.1016/j.tcs.2008.12.040.

[53] R. Desper and O. Gascuel, “Fast and accurate phylogeny minimum-evolution principle”, *Journal of computational biology*, vol. 9, no. 5, pp. 687–705, 2002, ISSN: 1066-5277. DOI: 10.1089/106652702761034136.

[54] A. P. Francisco, M. Bugalho, M. Ramirez, and J. A. Carriço, “Global optimal eBURST analysis of multilocus typing data using a graphic matroid approach”, *BMC Bioinformatics*, vol. 10, no. 1, p. 152, 2009, ISSN: 1471-2105. DOI: 10.1186/1471-2105-10-152.

[55] L. Foulds, M. Hendy, and D. Penny, “A graph theoretic approach to the development of minimal phylogenetic trees”, *Journal of molecular Evolution*, vol. 13, no. 2, pp. 127–149, 1979.

[56] L. Liu and S. V. Edwards, “Phylogenetic analysis in the anomaly zone”, *Systematic Biology*, vol. 58, no. 4, pp. 452–460, 2009, ISSN: 10635157. DOI: 10.1093/sysbio/syp034.

[57] W. Vach and P. O. Degens, “Least-squares approximation of additive trees to dissimilarities-characterizations and algorithms.”, *Computational Statistics Quarterly*, vol. 3, pp. 203–218, 1991.

[58] F. Pardi and O. Gascuel, “Distance-based methods in phylogenetics”, in *Encyclopedia of Evolutionary Biology*, ser. 1st Edition, K. R., Ed., 2016, pp. 458–465. [Online]. Available: https://hal-lirmm.ccsd.cnrs.fr/lirmm-01386569.

[59] K. Atteson, “The performance of neighbor-joining algorithms of phylogeny reconstruction BT - Computing and Combinatorics: Third Annual International Conference, COCOON ’97 Shanghai, China, August 20–22, 1997 Proceedings”, in, T. Jiang and D. T. Lee, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 101–110, ISBN: 978-3-540-69522-6. DOI: 10.1007/BFb0045077.

[60] R. Mihaescu, D. Levy, and L. Pachter, “Why Neighbor-Joining Works”, *Algorithmica*, vol. 54, no. 1, pp. 1–24, 2009, ISSN: 1432-0541. DOI: 10.1007/s00453-007-9116-4.

[61] F. Harary, S. Hedetniemi, and R. Robinson, “Uniquely colorable graphs”, *Journal of Combinatorial Theory*, vol. 6, no. 3, pp. 264–270, 1969.

[62] O. Borůvka, “O jistém problému minimálním”, 1926.

[63] J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem”, *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[64] R. C. Prim, “Shortest connection networks and some generalizations”, *Bell Labs Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[65] H. Whitney, “On the abstract properties of linear dependence”, *American Journal of Mathematics*, vol. 57, 1935. DOI: 10.2307/2371182.

[66] W. T. Tutte, “Lectures on matroids”, *J. Res. Nat. Bur. Standards Sect. B*, vol. 69, no. 1-47, p. 468, 1965.

[67] S. E. Dreyfus and R. A. Wagner, “The steiner problem in graphs”, *Networks*, vol. 1, no. 3, pp. 195–207, 1971.

[68] E. J. Feil, B. C. Li, D. M. Aanensen, W. P. Hanage, and B. G. Spratt, “eBURST: Inferring Patterns of Evolutionary Descent among Clusters of Related Bacterial Genotypes from Multilocus Sequence Typing Data”, *Journal of Bacteriology*, vol. 186, 2004. DOI: 10.1128/JB.186.5.1518-1530.2004.

[69] E. J. Feil, E. C. Holmes, D. E. Bessen, M.-S. Chan, N. P. J. Day, M. C. Enright, R. Goldstein, D. W. Hood, A. Kalia, C. E. Moore, J. Zhou, and B. G. Spratt, “Recombination within natural populations of pathogenic bacteria: Short-term empirical estimates and long-term phylogenetic consequences”, *Proceedings of the National Academy of Sciences*, vol. 98, no. 1, pp. 182–187, 2001. DOI: 10.1073/pnas.98.1.182. eprint: http://www.pnas.org/content/98/1/182.full.pdf. [Online]. Available: http://www.pnas.org/content/98/1/182.abstract.

[70] J. M. Smith, E. J. Feil, and N. H. Smith, “Population structure and evolutionay dynamics of pathogenic bacteria”, *BioEssays*, vol. 22, 2000. DOI: 3.0.CO;2-R.

[71] M. Achtman and M. Wagner, “Microbial diversity and the genetic nature of microbial species”, *Nature Reviews Microbiology*, vol. 6, 2008.

[72] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[73] I. Gronau and S. Moran, “Technion - Computer Science Department - Tehnical Report CS-2007-06 - 2007 Optimal Implementations of UPGMA and Other Common Clustering Algorithms Technion - Computer Science Department - Tehnical Report CS-2007-06 - 2007”, no. Step 3, pp. 1–9, 2007.

[74] J. Holm, E. Rotenberg, and C. Wulff-Nilsen, “Faster Fully-Dynamic Minimum Spanning Forest”, in *Algorithms - ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, N. Bansal and I. Finocchi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 742–753, ISBN: 978-3-662-48350-3. DOI: 10.1007/978-3-662-48350-3_62.

[75] R. V. Hogg and A. T. Craig, *Introduction to mathematical statistics.(5””edition)*. Upper Saddle River, New Jersey: Prentice Hall, 1995.

[76] G. Hulten, L. Spencer, and P. Domingos, “Mining time-changing data streams”, in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’01, San Francisco, California: ACM, 2001, pp. 97–106, ISBN: 1-58113-391-X. DOI: 10.1145/502512.502529. [Online]. Available: http://doi.acm.org/10.1145/502512.502529.

[77] T. Zhang, R. Ramakrishnan, and M. Livny, “Birch: An efficient data clustering method for very large databases”, in *ACM Sigmod Record*, ACM, vol. 25, 1996, pp. 103–114.

[78] P. S. Bradley and U. M. Fayyad, “Refining initial points for k-means clustering.”, in *ICML*, Citeseer, vol. 98, 1998, pp. 91–99.

[79] P. S. Bradley, U. M. Fayyad, C. Reina, *et al.*, “Scaling clustering algorithms to large databases.”, in *KDD*, 1998, pp. 9–15.

[80] J. Edmonds, “Matroids and the greedy algorithm”, *Mathematical programming*, vol. 1, no. 1, pp. 127–136, 1971.

[81] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization. Dover*. 1998.

[82] W. T. Tutte, “Lectures on matroids”, *J Res Nat Bur Standards Sect B*, vol. 69, 1965. DOI: 10.6028/jres.069B.001.

[83] J. Edmonds, “Matroids and the greedy algorithm”, *Mathematical Programming*, vol. 1, 1971. DOI: 10.1007/BF01584082.

[84] J. G. Oxley, *Matroid theory. 1992*, 1992.

[85] E. J. Feil, E. C. Holmes, D. E. Bessen, M. S. Chan, N. P. J. Day, M. C. Enright, R. Goldstein, D. W. Hood, A. Kalia, and C. E. Moore, “Recombination within natural populations of pathogenic bacteria: Short-term empirical estimates and long-term phylogenetic consequences”, *Proceedings of the National Academy of Sciences*, vol. 98, 2001. DOI: 10.1073/pnas.98.1.182.

[86] P. Ruiz-Garbajosa, M. J. Bonten, D. A. Robinson, J. Top, S. R. Nallapareddy, C. Torres, T. M. Coque, R. Canton, F. Baquero, B. E. Murray, R. del Campo, and R. J. Willems, “Multilocus sequence typing scheme for Entrococcus faecalis reveals hospital-adapted genetic complexes in a background of high rates of recombination”, *Journal of Clinical Microbiology*, vol. 44, 2006. DOI: 10.1128/JCM.02596-05.

[87] A. J. Page, B. Taylor, A. J. Delaney, J. Soares, T. Seemann, J. A. Keane, and S. R. Harris, “SNP-sites: rapid efficient extraction of SNPs from multi-FASTA alignments”, *Microbial Genomics*, vol. 2, no. 4, 2016. [Online]. Available: http://mgen.microbiologyresearch.org/content/journal/mgen/10.1099/mgen.0.000056.