TÉCNICO LISBOA

An algorithmic approach to the comparison of phylogenetic trees

António Pedro Paredes Silva Branco

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

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

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. Catia Raquel Jesus Vaz
Member of the Committee: Luís Manuel Silveira Russo

November 2023

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

The completion of this project was only possible due to the unwavering support of my supervisors, Prof. Alexandre -Francisco and Prof. Catia Vaz, to whom I am truly grateful. Therefore, I would like to emphasize the tireless way in which they consistently shared the necessary knowledge, motivation, and encouragement for the development of my work.

I also express my gratitude to Instituto Superior Técnico and all the faculty members I crossed paths with, who always knew how to provide me with the necessary tools to develop this work.

Moreover, I am sincerely thankful to my family for their enduring support and encouragement throughout my academic journey.

I also want to express my profound appreciation for the invaluable support of my friends. They have definitely enriched this journey with their support and camaraderie.

To each and every one of you – Thank you.

Abstract

There are several tools available to infer phylogenetic trees, which depict the evolutionary relationships among biological entities such as viral and bacterial strains in infectious outbreaks, or cancerous cells in tumor progression trees. These tools rely on several inference methods available to produce phylogenetic trees, with resulting trees not being unique. Thus, methods for comparing phylogenies that are capable of revealing where two phylogenetic trees agree or differ are required. There are several approaches to compute a similarity or dissimilarity measure between trees. Nevertheless, given the large and increasing volume of phylogenetic data, phylogenetic trees are becoming very large with hundreds of thousands of leafs. In this context, space requirements become an issue both while computing tree distances and while storing trees. In this thesis we propose an efficient implementation of the Robinson Foulds and Triplet comparison metrics over trees with succint representations. It is also demonstrated how these implementations extend the metrics to compare fully labeled trees. The Robinson Foulds implementation also extends the metric to compute the Weighted Robinson Foulds metric and to obtain additional information that can help evaluate the dissimilarities between trees. Experimental results show that the implementations achieves great performance with much lower memory usage. These implementations are available as an open-source tool for phylogenetic analysis in the git repository at https://github.com/pedroparedesbranco/TreeDiff.

Keywords

Phylogenetic Analysis, Comparative Metrics, Algorithms, Clusters, Bipartitions, Robinson Foulds, Triplets.

Resumo

Existem várias ferramentas disponíveis para inferir árvores filogenéticas, que descrevem as relações evolutivas entre entidades biológicas, como estirpes virais e bacterianas em surtos infecciosos, ou células cancerosas em árvores de progressão tumoral. Estas ferramentas baseiam-se em vários métodos de inferência disponíveis para produzir árvores filogenéticas, sendo que as árvores resultantes não são únicas. Assim, são necessários métodos de comparação de filogenias que sejam capazes de revelar onde duas árvores filogenéticas concordam ou diferem. Existem várias abordagens para calcular uma medida de similaridade ou dissimilaridade entre árvores. No entanto, dado o grande e crescente volume de dados filogenéticos, as árvores filogenéticas estão a tornar-se muito grandes, com centenas de milhares de folhas. Neste contexto, os requisitos de espaço tornam-se um problema, tanto no cálculo das distâncias entre árvores como no seu armazenamento. Nesta tese é proposta uma implementação eficiente das métricas de comparação Robinson Foulds e Triplet sobre representações sucintas de árvores. Também é demonstrado como estas implementações estendem as métricas para comparar árvores com informação em todos os nós. A implementação de Robinson Foulds também estende a métrica para calcular a métrica de Robinson Foulds para árvores com pesos e obter informações adicionais que podem ajudar a avaliar as dissimilaridades entre árvores. Os resultados experimentais mostram que as implementações atingem um ótimo desempenho com uma utilização de memória muito inferior. Estas implementações estão disponíveis como uma ferramenta de código aberto para análise filogenética no repositório git em `https://github.com/pedroparedesbranco/TreeDiff`.

Palavras Chave

Análise filogenética, métricas comparativas, algoritmos, clusters, bipartições, Robinson Foulds, Triplets.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

RF	Robinson Foulds
GRF	Generalized Robinson Foulds
WRF	Weighted Robinson Foulds
FLRF	Fully Labelled Robinson Foulds
FLWRF	Fully Labelled Weighted Robinson Foulds
FLT	Fully Labelled Triplets
lca	Lowest Common Ancestor

1

Introduction

Contents

It is essential to know the evolution of certain species or taxonomic groups such as SARS-CoV-2 to determine their origin, evolution and resistance patterns to the treatments under study. Phylogenetic analysis tries to understand this evolution by analysing the similarities and differences between those taxonomic groups [1]. For this analysis, they are often joined to form a tree where the distance between the nodes corresponds to the relation they have. For instance, the smaller the distance between two taxa, the more related they are. These trees are called phylogenetic trees.

Inferring phylogenetic trees can be very challenging for many reasons, like the limited and complex information available, the computational complexity, and the noise in the data. Different algorithms have been developed to infer phylogenetic trees. Some of them contain information only on the leaves, such as goeBURST [2], and others contain information on all nodes of the tree, like UPGMA [3]. However, depending on the inference method or even the input order, the phylogenetic tree obtained given a certain dataset could not always be the same.

Thus, comparing phylogenetic trees can give us much relevant information. For instance, if the goal is to know the best method to infer a phylogenetic tree given a particular dataset, the trees obtained from different methods can be compared to see how likely a specific topology is to be correct. In that way, one can try to determine which is the best inference method for that specific dataset. Most of the metrics that calculate how similar two phylogenetic trees are from each other only work if those trees only have data on the leaves.

There are several metrics for comparing phylogenetic trees. Some of them are based on rearrangements while others based on topology dissimilarity. The ones that compare the topologies between trees can also take into account the branch-length [4]. The ones that are based on rearrangements are based on finding the minimum number of rearrangements steps required to transform a tree into the other. Unfortunately this last ones are seldom used in practise for large studies as they costly to compute.

Otherwise, metrics that compare topologies are commonly used. One of the most used is the Robinson Foulds (RF) metric [5], given that there is a linear time algorithm for computing this metric proposed by Day [6]. There is also its branch-length variation, the Weighted Robinson Foulds (WRF) metric [7].

Given the increasingly large volume of phylogenetic data, the size of phylogenetic trees has grown substantially, often consisting of hundreds of thousands of leaf nodes. This brings significant challenges in terms of space requirements for computing tree distances, or even for storing trees. This leads to the requirement of developing implementations that are capable of computing these metrics using minimal memory usage.

1.1 Objectives

Given the large and increasing volume of phylogenetic data, this thesis is dedicated to introducing an implementation to compute the RF and Triplet metrics using succint data structures to represent the trees. Since there is a lack of metrics to compare fully labelled phylogenetic trees, another aim of this thesis is that these implementations are able to compare this labelled trees creating the Fully Labelled Robinson Foulds (FLRF) and the Fully Labelled Triplets (FLT) metrics. Finally, this thesis studies how to make these metrics return additional information that helps to evaluate the similarities and dissimilarities between trees. If possible, this information gives a certificate to guarantee that the distance is correct.

1.2 Document Structure

This thesis starts by giving an overview of some important concepts that are important to understand as long as a discussion of the main existing metrics to compare phylogenetic trees. The metrics that are discussed are the RF, Triplets and Quartets. It is explained their importance, supported by some examples and efficient computations.

Chapter 3 starts by explaining how it is possible to represent the trees in a succinct representation. Then it explains the approach taken to compute the RF and Triplets metric using that representation while it also shows how to extend those metrics to compute the FLRF, FLT metrics. It is also discussed how it is possible to obtain additional information when using those approaches.

In chapter 4 we discusse the details of the implementations.

Chapter 5 evaluates the theoretical complexity off the implementations while also doing a memory analysis. This chapter ends by presenting an experimental evaluation.

Finally, in chapter 6 it is presented some final remarks as long as some future work.

2

Background

Contents

When it comes to comparing and computing the distance between phylogenetic trees, several metrics can be used. Various aspects need to be considered to choose the best metric to use. The most important ones are running time and the discriminatory power in representing the distance between the trees. Usually there is a trade off for each metric between this two aspects. Therefore, there is a need to consider which of these aspects is more important in a particular context. This chapter will start by explaining some important concepts related to phylogenetics. Then it will be made a comparison between four metrics as well as efficient ways of computing them. Finally it will be shown some methods to represent phylogenetic trees.

2.1 Concepts

There are several concepts relevant to the understanding of the background, which are extensively described in the literature [1]. The most important ones will be introduced below.

Throughout this work it will be referred taxa as the taxonomic groups that phylogenetic trees will contain. Taxa will be denoted as $X = \{x_1, \cdots, x_n\}$ where each x_i will correspond to each taxonomic group (taxon).

It will also be denoted a tree as $T(V, E)$ where V and E will represent the set of vertices and edges present in T, respectively. Additionally, a set containing three taxa will be referred to as triplet and a set containing four taxa as quartet.

With these concepts introduced, a phylogenetic tree can be described as a tree $T(V, E)$ that given a certain X where $\lambda : X \to V$, assigns precisely one taxon to each leaf. The majority of them does not assign any taxon to the internal nodes, however some of them do it, which may lead them to be handled in a different way when applying comparison metrics, such as the ones referred earlier in this section.

These trees may be either unrooted or rooted, depending on some aspects. On one hand, unrooted phylogenetic trees (exemplified in Figure 2.1) need to have all V with a number of connected edges different than two (degree $\neq 2$), where the leaves will have equal to one and the internal nodes equal to 3. The reason for this is that if the degree equals to two, there is no inference happening in that specific vertex. On the other hand, rooted phylogenetic trees (exemplified in Figure 2.2) can have a vertex (the root) with degree equal to two, since the root is the start point and is only inferring two vertices.

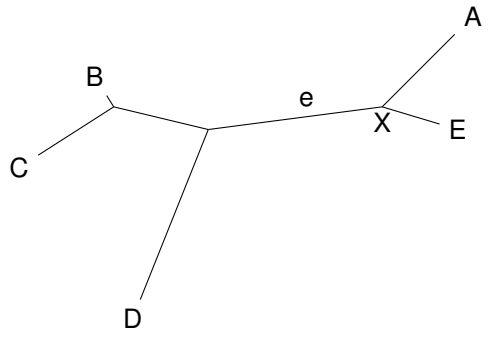
It is often necessary to transform an unrooted phylogenetic tree into a rooted phylogenetic tree. This can be important since there are some metrics that only work if the input contains rooted phylogenetic trees. For example, some metrics recieve as input the trees in a Newick format, which only can represent these types of trees. This is because the Newick format represents a tree starting from the root and then separates the two childs until it reaches a leaf. Below it can be seen an example of the Newick format applied to the tree in Figure 2.2(a).

$$\text{Newick format } = (((B, C), D), (A, E))$$

This transformation can be done simply by choosing any node to be the root of the tree and extending from there. Note that it will be needed to add an extra edge to the tree so that the tree will be a regular phylogenetic tree. If an internal node is chosen, this extra edge is needed so that the root only has two children. Conversely, if the choice is a leaf, the extra edge is needed so that the root does not contain any information.

For example, consider the tree presented in Figure 2.1 and assume the internal node X is chosen to transform it into a rooted phylogenetic tree. To accomplish this, it is needed to add the edge $e1$ so that the root does not have three children. The transformed tree can be seen in Figure 2.2(a).

However, if the choice is the leaf A, it is necessary to add the edge $e2$ so that the information that is present in that leaf continues in a leaf and not in the root of the transformed tree. The transformed tree can be seen in Figure 2.2(b).



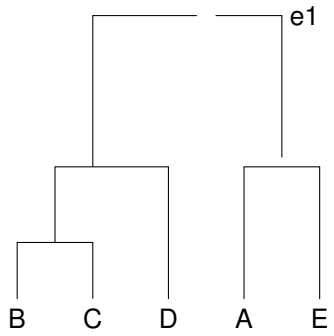
(a) T_1

Figure 2.1: Unrooted phylogenetic tree T example with five taxa.

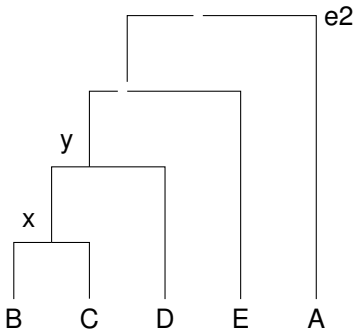
Another important concept to the understanding of the metrics that will be presented is the meaning of topology. Topology can be defined as the branching pattern that was inferred when generating a phylogenetic tree. The topology of a given triplet (X, Y, Z) can be determined by computing the Lowest Common Ancestor (lca) between each pair of them. If there is one pair that contains the lca lower then the others, for instance the $lca(X, Y)$ depth is lower than the $lca(Y, Z)$ and $lca(X, Z)$ it can be determined that the topology of the triplet (X, Y, Z) is $(X, Y|Z)$.

For example, considering the tree in Figure 2.2(b), the topology of the triplet (B, C, D) can be determined by computing the $lca(B, C)$, $lca(B, D)$ and $lca(C, D)$. Since $lca(B, C) = x$ and $lca(B, D) = lca(C, D) = y$ and the depth of x is lower than y, $topology(B, C, D) = (B, C|D)$.

Most of the metrics that compare phylogenetic trees accomplish it by analysing their topology. One



(a) T_1 by choosing X.



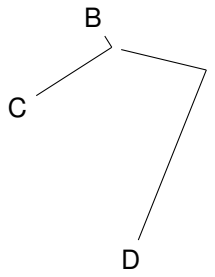
(b) T_1 by choosing A.

Figure 2.2: Rooted phylogenetic trees that resulted from transforming the unrooted phylogenetic tree of Figure 2.1 by selecting different nodes.

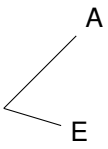
way of doing so is by using a split strategy.

A split $S = S_1|S_2$ can be defined as a bipartition of a certain set of taxa X into two subsets (S_1 and S_2) by retrieving an edge from the tree. This two subsets can not be empty as well as the intersection between them. It is important to note that a split can be applied to any phylogenetic tree, no matter if it is rooted or unrooted. A split in a certain edge e of a tree T will be represented as split(T, e).

For instance, considering the tree in Figure 2.1, to apply a split to T_1 by retrieving the edge e (split(T_1, e)), it would result in the two subsets $S_1 = B, C, D$ and $S_2 = A, B$ which corresponds to the subtrees T_1 and T_2 that can be seen in Figure 2.3.



(a) T_1



(b) T_2

Figure 2.3: Resulting trees after applying a split to the tree in Figure 2.1 to edge e.

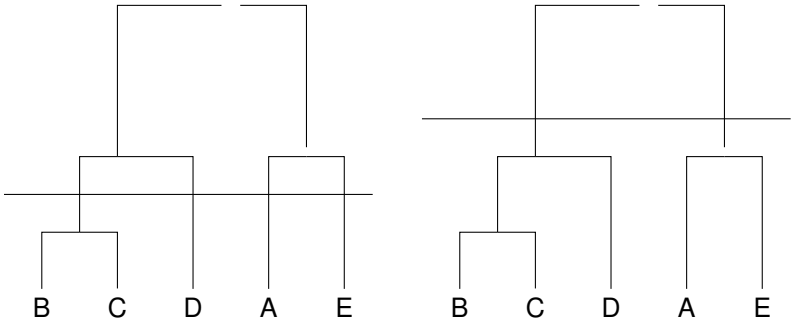
Another way of analysing their topology is by comparing the clusters present in the trees. A cluster can be defined as a any subset that is obtainable from a certain tree. Throughout this work a cluster obtained from a certain tree T will be represented as $C(T)$.

There are a lot of ways to obtain all clusters from a phylogenetic tree. The clusters that have only one taxon and the one which is the whole tree itself are considered trivial clusters since they are present

in all phylogenetic trees that contain set of taxa.

One method for getting all clusters is to apply a cut through all depths of the tree. For example, considering the tree present in Figure 2.2(a), applying the first cut would get the clusters $\{(B, C), D, A, E\}$ and the second cut, the clusters $\{((B, C), D), (A, E)\}$. Since the leaves are trivial clusters it is not necessary to apply a cut at the highest depth. Eventually, repeated clusters are removed if necessary. All clusters from the tree are obtained by adding the trivial clusters. This process can be visualized in Figure 2.4.

Unlike the split, this technique can only be applied to rooted phylogenetic trees since it relies on the depth of a tree.



(a) First cut. **(b)** Last cut.

Figure 2.4: Cuts applied to the tree in Figure 2.2(a) to get all the clusters.

2.2 Metrics

2.2.1 Robinson Foulds

The RF metric [5] consists in obtaining all the clusters present in both of the trees that are being compared and then counting the number of clusters that do not match in both of them. Thoughout this thesis it will be refered as the RF distance between two trees T_1 and T_2 as $rf(T_1, T_2)$. A possible implementation is described in Algorithm 2.1; the first step of this approach may differ in other implementations, since the original approach does not specify a concrete method to obtain all the clusters present in the phylogenetic trees.

In particular, consider the trees shown in Figure 2.5. To calculate the RF distance between these trees, the algorithm starts by obtaining all the clusters present in the trees. Then they are divided in three groups. The first one contains the ones that are common to both trees, the second one those that

Algorithm 2.1: Robinson Foulds approach.

Input: Two rooted phylogenetic trees T_1 and T_2.
Output: A value $rf(T_1, T_2)$ representing how different T_1 is from T_2.

Initialization: $clusters_1 = []$.
$clusters_2 = []$.

First step: For all depths in T_1 and T_2:
$clusters_1 += cut(T_1)$
$clusters_2 += cut(T_2)$

Second step: $C(T_1) = Unique(clusters_1)$
$C(T_2) = Unique(clusters_2)$

Third step: $rf(T_1, T_2) = \frac{1}{2}\left(|C(T_1) - C(T_2)| + |C(T_2) - C(T_1)|\right)$

Fourth step: Return $rf(T_1, T_2)$.

are only present in T_1 and the last one those that are only present in T_2. These groups can be observed in equation 2.1.

(a) T_1 **(b)** T_2

Figure 2.5: Examples of rooted phylogenetic trees.

$$\begin{aligned} &\text{Clusters present in } T_1 \text{ and } T_2 : \{(A),(B),(C),(D),(E),(A,B,C,D,E)\} \\ &\text{Clusters only present in } T_1 : \{(A,B),(A,B,C),(A,B,C,D)\} \\ &\text{Clusters only present in } T_2 : \{(B,C),(B,C,D),(A,E)\} \end{aligned} \tag{2.1}$$

Now that the groups are complete the distance distance between T_1 and T_2 can be calculated. This can be done by applying the formula present in the third step, which is done in equation 2.2.

$$rf(T_1, T_2) = \frac{1}{2}\left(|C(T_1) - C(T_2)| + |C(T_2) - C(T_1)|\right) = \frac{1}{2}(3+3) = 3 \tag{2.2}$$

That means that the distance between T_1 and T_2 is equal to 3.

In theory, for a tree that contains n nodes it contains at most $2n-1$ edges which means that it contains at most $2n-1$ clusters. If it is considered the n trivial clusters that correspond to each taxon alone and the cluster that correspond to the tree itself, it can be concluded that at least $n+1$ clusters are common to both trees. Thus, the maximum distance that the RF metric may return is $2n-1-(n+1) = n-2$. If it is applied to these specific trees, the maximum distance that the algorithm can obtain is $5-2=3$ which is exactly what was computed previously.

This being said, by observing the trees in question they are not different at all. The only difference is the position where the taxon A was inferred. In T_1 it was inferred to be next to B but if it were inferred to be next to E like it was in T_2, the two trees would be exactly the same. This is a great example to demonstrate that the RF metric does not have a big discriminatory power since one taxon inferred differently can induce into a big distance and it does not consider how different the clusters are.

2.2.1.A Generalized Approach

This is where the Generalized Robinson Foulds (GRF) metric [8] comes in. The main purpose of this metric is to try to mitigate the problem that the distance obtained can be high even though the difference between the trees is only a few leaves.

The key idea to this approach is to instead of counting the number of clusters that only belong to one tree, compare the most similar clusters and observe how different they are from each other. To do this it is used a metric called Jaccard-Robinson-Foulds. This metric receives as input two clusters and returns a value that indicates the difference between them. This value is obtained by doing the coefficient between the intersection and the union of the taxa present in both clusters and the equation that can be seen in Equation 2.3. In particular, k is an arbitrary constant set to regularize the similarity between the clusters with $k \geq 1$. Therefore, it returns a lower value if there are more leaves in common between the clusters.

$$JM(C_1, C_2) = 2 - 2 \times \left(\frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}\right)^k \tag{2.3}$$

However, there is a need to know how to pair the clusters that differ in both trees. For this, the chosen pairs are the ones that minimizes the distance obtained. Therefore, if the distance between the trees in Figure 2.5 is calculated using this approach, the pairings that minimizes the distance are the following

ones:

$$\{(A,B),(A,E)\}$$
$$\{(A,B,C),(B,C)\}$$
$$\{(A,B,C,D),(B,C,D)\}$$

Thus, by calculating the Jaccard-Robinson-Foulds metric for these pairings, the following values are obtained:

$$JM((A,B),(A,E)) = 2 - 2 \times \frac{1}{3} = \frac{4}{3}$$

$$JM((A,B,C),(B,C)) = 2 - 2 \times \frac{2}{3} = \frac{2}{3}$$

$$JM((A,B,C,D),(b,c,d)) = 2 - 2 \times \frac{3}{4} = \frac{2}{4}$$

And therefore the distance calculated by this approach would be:

$$rf(T_1,T_2) = \frac{4}{3} + \frac{2}{3} + \frac{2}{4} = \frac{10}{4}$$

Which means that the distance calculated between T_1 and T_2 is $\frac{10}{4}$, which is lower than the previous obtained distance, showing that this approach can mitigate the discriminatory power problem that the original one has. However, there is a downside to these approach. Given the fact that there is a need to find the pairings that minimizes the final distance this process is very costly. This problem it was shown to be NP-hard in [9].

2.2.1.B Day's Algorithm

To efficiently compute this metric it was presented an approach that could compute the RF distance in linear time [6]. This approach receives as input each tree as a table where each line contains information about one of the nodes. The nodes are presented in a pre-order traversal and each line contains two numbers, the first one represents the label of the node and the second one the number of nodes that are below them. Then, by applying the BUILD procedure 2.2 to the first tree, it creates a cluster table where it stores all the clusters present in it as well as the relation between index and label. This is done by assigning all the leafs an index from left to right so that if it stores the value x and y, the cluster represented contains all the leaves that have indexes between x and y. Finally, by applying the COMCLUSTER procedure to the second tree it is possible to traverse through all the clusters and verify

if they are present in the first tree by searching in the cluster table.

Algorithm 2.2: Build Procedure

```
begin
    clusters[num_leafs][3] = −1
    leafcode ← −1
    current ← 0
    while current < num_nodes(T1) do
        if T1size[current] == 0 then
            leafcode++
            clusters[current][2] ← right ← leafcode
            current++
        else
            left ← clusters[current − T1_size[current]][2]
            current++
            if T1size[current] == 0 then
                loc ← right
            else
                loc ← left
            clusters[loc][0] ← left
            clusters[loc][1] ← right
```

Considering the trees in Figure 2.5 and considering the following values for the labels: $B = 1, C = 2, A = 3, E = 4, D = 5$, it can be seen an example of how the input of the algorithm would look like in Table 2.1. Then by applying the Build procedure the cluster table present in Table 2.2 contains all clusters present in T_1. By traversing the second tree, it is possible to verify if the clusters are present in the first tree by searching in the lines with indexes equal to the left and right value; if the cluster is not in any of those two lines it can be concluded that the cluster is exclusive to the second tree.

2.2.1.C Robinson Foulds with weights

Even though the RF metric can give a very reliable distance between two trees in terms of their topology, sometimes it is also important to consider how far apart each node is from each other. To consider this, it was proposed an approach to compute the RF metric for phylogenetic trees with weights on the edges [10]. This metric will be referred as the WRF metric.

This metric is very similar to the original one, the key idea is to associate to each cluster the weight of the edge that connects it to the tree. Then, when a cluster is exclusive to one tree, the weight associated to that cluster is added to the distance. However, when a cluster is present in both trees the value that is added to the distance is the absolute difference between the weights associated with both clusters. This means that the distance for this metric is equal to the sum of the clusters weights that are exclusive to one tree plus the sum of the absolute difference between all clusters that are not exclusive. One naïve

Algorithm 2.3: COMCLUSTER Procedure

```
begin
    equalClusters ← 0
    for current = 0; current < num_nodes(T2); current++ do
        if T2size[current] == 0 then
            s.push(< T1label[current], T1label[current], 1, 1 >)
        else
            size ← T2size(current) < L, R, N, W >←< ∞, 0, 0, 1 >
            while size ≠ 0 do
                < L*, R*, N*, W* >← s.pop()
                < L, R, N, W >←< min(L, L*), max(R, R*), N + N*, W + W* >
                size ← size − W*
            s.push(< L, R, N, W >)
            if N == r − l + 1 then
                if
                (L == clusters[L][0] & R == clusters[L][1]) or (L == clusters[R][0] & R == clusters[R][1])
                then
                    equalClusters++
```

Table 2.1: Day's algorithm input.

pre-order	T_1 label	T_1 size	T_2	T_2 size
1	3	0	1	0
2	1	0	2	0
3	6	2	6	2
4	2	0	5	0
5	7	4	7	4
6	5	0	3	0
7	8	6	4	0
8	4	0	8	2
9	9	8	9	8

implementation of this metric can be observed in Algorithm 2.4.

Algorithm 2.4: Weighted Robinson Foulds metric.

Input: Two weighted phylogenetic trees T_1 and T_2.
Output: $wrf\_distance$.

Initialization: $wrf\_distance = 0$.

First step: $\forall cluster \in C(T_1)$:
If $cluster \in T_2$:
$wrf\_distance$ += $|T_1 clusterweight - T_2 clusterweight|$
Else:
$wrf\_distance$ += $T_1 clusterweight$

Second step: $\forall cluster \in C(T_2)$:
If $cluster \notin T_1$:
$wrf\_distance$ += $T_2 clusterweight$

Second step: Return $wrf\_distance$

Table 2.2: Day's algorithm T_1 cluster table.

pre-order	T_1 label	T_1 size	T_2
1	-	-	2
2	1	2	3
3	1	3	1
4	1	4	5
5	1	5	4

Considering the trees present in Figure 2.4, they have the cluster A, B in common but the corresponding weights are different, in T_1 it is 2 and in T_2 is 1. In this case, the difference that would be considered to demonstrate how different they are in each tree would be $abs(2-1) = 1$. All the other clusters are exclusive to each tree so by adding this value to all their weights, the WRF distance would be obtained.

(a) T_1 **(b)** T_2

Figure 2.6: Examples of rooted phylogenetic trees.

2.2.2 Triplets

The triplets metric approach consists in obtaining all combinations of three taxon and comparing their topology in the trees that are being compared. That being said, a naïve implementation can be seen in Algorithm 2.5.

For example, by applying the first step of the algorithm to the tree represented in Figure 2.5, $\binom{5}{3} = 10$ combinations between the leafs are obtained, corresponding to all the combinations possible between the following set of taxa: (A, B, C, D, E). In Table 2.3 it can be seen the topology for all these combinations in both trees and if they are equal. Then it can be concluded that there are six triplets with different topologies and four with the same. Therefore, the triplets distance is 6. Given the fact that the maximum distance for a tree with 5 taxa was 10, this concludes that this metric can be more reliable.

Algorithm 2.5: Triplets approach.

Input: Two phylogenetic trees T_1 and T_2.
Output: A value $Trip(T_1, T_2)$ representing the number of triplets in common.

Initialization: $Trip(T_1, T_2) = 0$.
$n\_leafs = number\ of\ leafs$

First step: $\forall i \in \binom{n\_leafs}{3}$:
If $topology(T_1, i) \neq topology(T_2, i)$:
$Trip(T_1, T_2) += 1$

Second step: Return $Trip(T_1, T_2)$

Table 2.3: Comparison of topology between T_1 and T_2.

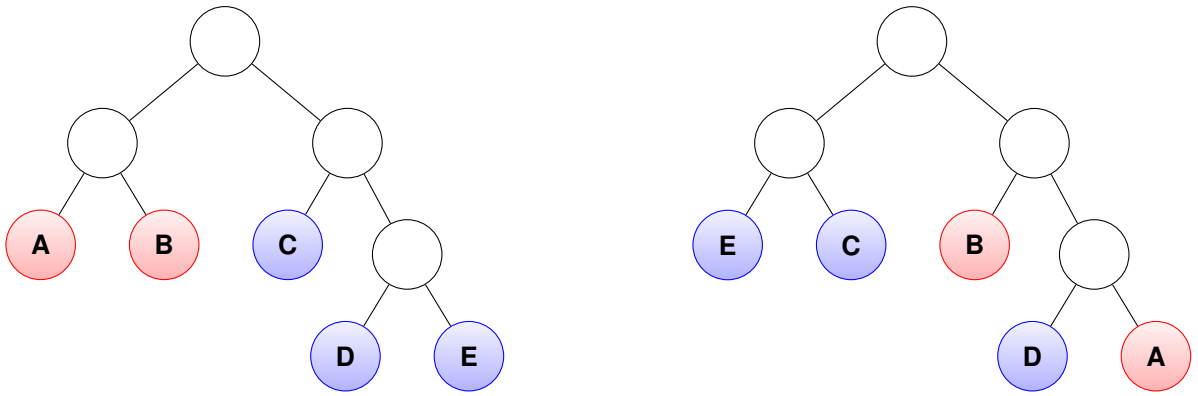
Triplets	Topology(T_1)	Topology(T_2)	Equal?
(A, B, C)	$AB\|C$	$A\|BC$	False
(A, B, D)	$AB\|D$	$A\|BD$	False
(A, B, E)	$AB\|E$	$AE\|B$	False
(A, C, D)	$AC\|D$	$A\|CD$	False
(A, C, E)	$AC\|E$	$AE\|C$	False
(A, D, E)	$AD\|E$	$AE\|D$	False
(B, C, D)	$BC\|D$	$BC\|D$	True
(B, C, E)	$BC\|E$	$BC\|E$	True
(B, D, E)	$BD\|E$	$BD\|E$	True
(C, D, E)	$CD\|E$	$CD\|E$	True

2.2.2.A Efficient computation

Even though this metric can give a very reliable representation of the difference between the trees, comparing the topology for all the combinations possible has a time complexity of $O(n^3)$ for trees with n taxa, given the need to go through all the combinations of triplets.

There have been many improvements which surpassed the naive implementation. In particular, there is one approach [11] that can reach a complexity of $O(n \log n)$. To overcome the need to go through all the combinations, the key idea is to only look for the triplets that have the same topology. Then by having this information and the total number of triplets that exist it is possible to know how many triplets have different topologies and therefore the triplets metric.

To verify if a triplet has the same topology, the approach consists in dividing a cluster from one of the trees with two colors, for instance blue and red. All the nodes that are to the left of the root are assigned with one color and to the right another color. Then, if in the other tree there is two nodes of one color to one side and one node of the other color in the other side, the triplet that consists of that those three taxa is considered to have the same topology. For example, in Figure 2.7 it can be seen an example of this process, in the first tree the triplet (A, B, C) was divided assigning the bodes A and B with the color red and C with the color blue. Then in the second tree it is found that there are 2 red nodes in the left and one blue node on the right. This means that the triplet (A, B, C) has the same topology in both of



(a) Tree_1 **(b)** Tree_2

Figure 2.7: Demonstration of colors assignment.

trees. It is important to note that it does not matter if the color is blue or red is found in the right or left and if the 2 nodes found are on the right or the left. Given this there are four ways to find clusters that have the same topology and it can be seen in the Equation 2.4.

$$\begin{aligned} &ComputeShared(leftred, leftblue, rightred, rightblue) = \\ &leftred \times \binom{rightblue}{2} + leftblue \times \binom{rightred}{2} + \\ &rightred \times \binom{leftblue}{2} + rightblue \times \binom{leftred}{2} \end{aligned} \tag{2.4}$$

This approach starts with all leaves from the first tree assigned with color red. Then, starting from the root it changes the color of the side that contains less leaves in it to blue and counts the number of shared triplets found on the other tree. Then it calls recursively on the largest side while assigning the smallest side with no color. When it ends it does the same thing for the smaller side. This is called the smaller-half trick and it guarantees that a given leaf changes color $O(log(n))$ which gives a total of $O(n \times log(n))$ color changes. The use of hierarchical decomposition trees (HDT) is relevant to reduce the complexity in counting the triplets. This structure is created to count the number of triplets in T_2 compatible with the coloring of the leaves in T_1. Each node in the hierarchical decomposition tree maintains a counter representing the number of triplets a certain part of T_2 contains. This tree can be constructed in linear time with a height of $O(\log n)$; the counters can be updated in linear time and the tree itself can be updated in $O(\log n)$ time after a color change during the transverse in T_1. Finally, by considering the $O(n \log n)$ complexity with the smaller half trick and by reducing the complexity in counting the triplets to $O(\log n)$, the complete method to compute triplets is $O(n \log n) \times O(\log n) = O(n \log^2 n)$. Summarising, this approach consists in applying the following steps:

1. T_1 starts with all leafs red.
2. Build the HDT for T_2.
3. Color the leafs in the smallest side as blue.
4. Compute triplets with same topology found in HDT.
5. Retrieve color from leafs in the smallest side.
6. Repeat from the third step for the largest side.
7. Color the leafs in the smallest side as red.
8. Repeat from the third step for the smallest side.
9. Return the number of triplets found that have the same topology.

2.2.2.B Quartets

The quartets approach is very similar to the triplets. The difference comes to the number of taxa that is being compared, that passes from three to four. This means that know it will be needed to compare the topology of every combination of four taxa. To view an implementation of this approach, it is only needed to modify the first step to run from the combination of four taxa.

Applying the first step to the quartets, there are obtained $\binom{5}{4} = 5$ combinations between the leafs, which corresponds to:

$$\{(A, B, C, D), (A, B, C, E), (A, B, D, E), (A, C, D, E), (B, C, D, E)\}$$

Table 2.4: Comparison of topology between T_1 and T_2

Quartets	Topology(T_1)	Topology(T_2)	Equal?
(A, B, C, D)	$ABCD$	$A\|BCD$	False
(A, B, C, E)	$ABC\|E$	$BC\|AE$	False
(A, B, D, E)	$AB\|DE$	$BD\|AE$	False
(A, C, D, E)	$ACDE$	$AE\|CD$	False
(B, C, D, E)	$BCDE$	$BCD\|E$	False

Thus, we can see that all quartets have different a different topology in each tree. This is due to the fact that the trees have 5 taxa, which means that it is impossible to have any two quartets with the same topology unless the trees are identical.

Table 2.5: Algorithms proposed to compare phylogenetic trees. (*Theoretical). .

Metric	Strategy	Proposed by	Interface	Time complexity*
RF	Clusters	D. F. ROBINSON and L. R. FOULDS [5]	TreeCmp	—
GRF	Clusters	Sebastian Bocker, Stefan Canzar and Gunnar W. Klau [8]	FastRF	NP-Hard
RF	Clusters	William H. E. Day [6]	—	$O(n)$
WRF	Clusters	D. F. ROBINSON and L. R. FOULDS [10]	TreeCmp	—
Triplets	Splits	DOUGLAS E. CRITCHLOW, DENNIS K. PEARL, AND CHUNLIN QIAN [11]	TreeCmp	$O(n^3)$
Triplets	Splits	Andreas Sand et al. [11]	—	$O(nlog^2(n))$
Quartets	Splits	George F. Estabrook, F. R. McMorris and Christopher A. Meacham [12]	Treecmp	$O(n^4)$

2.2.3 Analysis

All the metrics explained above have their advantages and disadvantages. For instance, the RF metric is one of the most used metrics since it is very simple and can achieve a sublinear complexity with high probability. Because of that, it is popular when large amounts of data is involved in the tasks. However, in the case that it is important to have a reliable distance metric, there are several other options that should be considered instead of this metric. In particular, the triplets and quartets are two of those approaches. It is also possible to generalize the RF metric, at a cost of it being considered NP-hard and, therefore, significantly slower than the original RF metric.

Regarding the triplets metric, it has a great discriminatory power when compared to the RF metric. Although its naive implementation runs through all the combinations of triplets by reaching a complexity of $O(n^3)$, it is possible to compute it in a more efficient way that reduces the complexity to $O(n \log^2(n))$. In fact, this complexity can be reduced to a complexity of $O(nlog(n))$, despite not being covered in this document. [11].

Finally, the quartets are similar to the triplets, as the only difference is in the size of the sets. The computation of this metric tends to be slower than the computation of the triplets, with the benefit of providing better results. Such an aspect is due to the fact that quartets consider more information when compared to the triplets metric.

3

Approach

Contents

Given the large and increasing volume of phylogenetic data, phylogenetic trees are becoming very large with hundreds of thousands of leafs. In this section it will be shown how to compute the RF and Triplets metrics over trees represented through succinct data structures so that it is possible to compare big phylogenetic trees with reduced memory usage. It will also be demonstrated how to extend these metrics to consider fully labelled phylogenetic trees.

It will also be shown how to adapt the RF approach to compute the WRF metric so that it is possible to compare trees that contain weights in the edges.

Finally, it will be discussed how these approaches can keep the information about the similarities and dissimilarities between the trees.

3.1 Succint data representation

To represent the trees in a succinct data structure it was used a balanced parenthesis representation. This representation can represent a tree using only a bit vector of size $2n$ bits, where n is the number of nodes present in the tree. By representing each tree this way, it is possible to achieve a very compact representation that eliminates the need to store additional information.

This representation is extensively described in the literature [13], in this section it will be presented a summary of how it can represent a tree structure in so little space as well as some operations that can be used to efficiently traverse the tree.

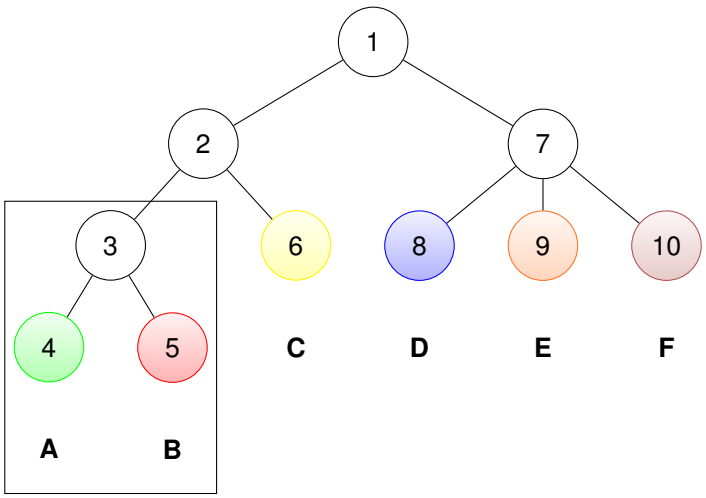


Figure 3.1: Tree T_1 and its balanced parenthesis representation.

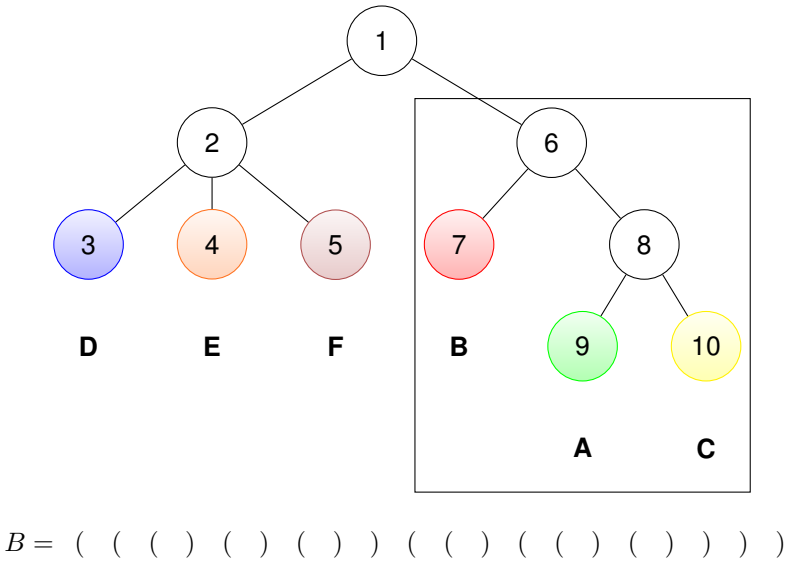


Figure 3.2: Tree T_2 and its balanced parenthesis representation.

3.1.1 Balanced Parentheses Representation

A balanced parentheses representation is a method frequently used to represent hierarchical relationships between nodes in a tree, closely related to the Newick format described before. In this representation, there are some key rules to understand how a tree can be represented by a sequence of parentheses.

The first one is that each node in the tree is represented by a pair of opening and closing parentheses. In the bit vector, the opening ones will be represented as a one and the closing ones as a zero. This is the reason why the size of the bit vector is two times the number of nodes.

The second rule is that for all nodes present in the tree, all their descendents must be after the opening parentheses and before the closing parentheses that represents them. For example, in Figure 3.1, the opening and closing parentheses that represent the third node are in positions 3 and 8, respectively. Then, it can be concluded that the parentheses that represents their descendents (nodes 4 and 5) are in positions 4, 5, 6, and 7, which are between 3 and 8.

The index assignment to each node is made with a pre-order traversal. This indexation can be seen in Figures 3.1 and 3.2. Throughout this thesis, a index of a node will be referred as i and a position in a bit vector as v.

3.1.2 Operations

The balanced parentheses representation contains several operations that are fundamental to manipulate this structure effectively. The most important ones in the present context are the operations rank,

select, excess, find open, find close, and enclose operations. Then, it was added the operations preOrderMap, preOrderSelect, postOrderSelect, isLeaf, lca, clusterSize, numLeaves, FirstChild and NextSibling that mostly use the fundamental operations just mentioned before. In Table 3.1 it is possible to see the list of operations as well as their meaning and their run time complexity.

Operation	**Meaning**	**Complexity**
Rank1(bv, v)	Given a bit vector bv and a position v, returns the number of occurrences of '1' until v.	$O(1)$
Rank10(bv, v)	Given a bit vector bv and a position v, returns the number of occurrences of the sequence '10' until v.	$O(1)$
Select1(bv, i)	Given a bit vector bv and a occurrence i, returns the position where the ith one is present.	$O(1)$
Select0(bv, i)	Given a bit vector bv and a occurrence i, returns the position where the ith zero is present.	$O(1)$
FindOpen(bv, v)	Given a bit vector bv and a position v, returns the position where the corresponding opening parentheses is located.	$O(log(n))$
FindClose(bv, v)	Given a bit vector bv and a position v, returns the position where the corresponding closing parentheses is located.	$O(log(n))$
Enclose(bv, v)	Given a bit vector bv and a position v, returns the position v where the smaller segment strictly containing v is located.	$O(log(n))$
Rmq(bv, l, r)	Given a bit vector bv and two positions l and r, returns the position v where the node with minimal excess in the range $[l..r]$ is located.	$O(log(n))$
PreOrderMap(bv, v)	Given a bit vector bv and a position v, returns the index of the node in pre-order located in v.	$O(1)$
PreOrderSelect(bv, i)	Given a bit vector bv and the index i of the node in pre-order traversal, returns the position in the bit vector bv where the node is located.	$O(1)$
PostOrderSelect(bv, i)	Given a bit vector bv and the index i of the node in post-order traversal, returns the position in the bit vector bv where the node is located.	$O(\log(n))$
FirstChild(bv, v)	Given a bit vector bv and a position v, returns the position where the first child of v is located.	$O(1)$
IsLeaf(bv, v)	Given a bit vector bv and a position v, returns True if v is a leaf and False otherwise.	$O(1)$

lca(bv, u, v)	Given a bit vector bv and two positions u and v, returns the position where the lowest common ancestor between u and v is located.	$O(\log(n))$
ClusterSize(bv, v)	Given a bit vector bv and a position v, returns the number of nodes that are below v.	$O(\log(n))$
NumLeaves(bv, v)	Given a bit vector bv and a position v, returns the number of leaves that are below v.	$O(\log(n))$

Table 3.1: Operations to manipulate trees.

3.1.3 Memory usage

To compare the topologies between two trees it is also necessary to have a relation between the indexation given to the nodes and their respective information, for example, when traversing to a node with taxon A in one of the trees it is crucial to know where that taxon A is located in the other tree. To resolve this problem it was used a vector of integers of size n, where the position of each integer represents the index on the first tree and the integer value represents the index of the second tree. This way it is possible to get the index of the node where a taxon is located in the second tree given the index of where it is located in the first tree. An example of this vector for the trees in Figures 3.1 and 3.2 can be seen in Equation 3.1. This being said, the algorithms that will be described in next sections will receive as input two bit vectors of size $2n$ and an integer vector of size n to represent the trees and the relationship between them.

$$[\ 0\ \ 0\ \ 0\ \ 9\ \ 7\ \ 10\ \ 0\ \ 3\ \ 4\ \ 5\ ]. \tag{3.1}$$

3.2 Robinson Foulds

In this section we discuss an approach to compute the RF metric using the data structures referred in section 3.1. It will also be discussed how to extend this approach to compute the WRF metric, to consider fully labelled trees and how to give more information about the difference between the trees being compared.

3.2.1 Robinson Foulds distance computation

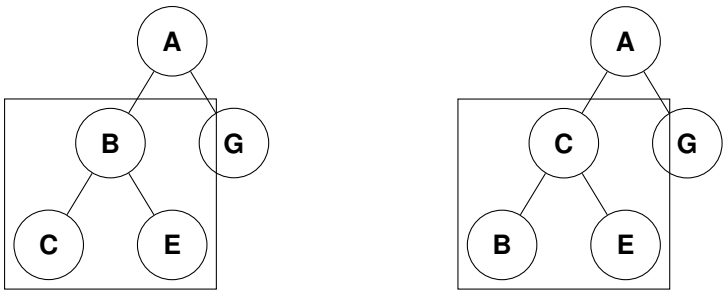
To compute the RF metric, the idea is to traverse all the clusters from one of the trees and verify for each of them if they are present in the other tree. Then, by knowing how many clusters are the common to both trees and the total number of clusters it is possible to infer how many are exclusive to one tree

and therefore get the RF distance. Thus, the key idea to verify if a cluster present in a given tree T_1 is present in other tree T_2, is to locate where the taxa present in the cluster are in T_2. Then, by computing the lca between those nodes it is obtained a node in T_2 that represents a cluster. If that cluster has the same number of leafs as the initial cluster, it can be concluded that the cluster is common to both trees. For example, consider T_1 and T_2 as the trees represented in Figures 3.1 and 3.2 respectively. To verify if the cluster highlighted in T_1, namely, A, B is present in T_2, we need first to calculate the indexation of both taxa A and B, which is 9 and 7 respectively. Then by computing the lca in T_2 between those nodes, it is obtained the index 6, which represents the cluster highlighted in T_2. Given that, this cluster contains three leafs and the original one two, and thus it can be concluded that cluster A, B is not present in T_2.

To efficiently compute this metric, the trees are traversed in a post-order traversal. This is done to guarantee that all nodes are accessed before their ancestors. This way it is possible to minimize the number of times that the lca operation is called. For example, considering T_1 as the tree represented in Figure 3.1, when computing the lca between the cluster (A, B) this value is saved so that when computing for the cluster (A, B, C), it only needs to compute the lca between the value obtained before and the node C. Otherwise it would need to compute the lca between A and B more than once. Summarising, this approach consists in applying the following steps:

1. Transverse T_1 in a post-order traversal.
2. If it finds a leaf, find the leaf with the same taxa in T_2.
3. Until it goes up, compute de lca between itself and the corresponding indexes of the siblings.
4. Verify if the cluster is present in T_2 by comparing the number of leafs.
5. Save the lca computed in the internal node .
6. Repeat this process until it reaches the root.

For instance in T_1 the algorithm would start by going through the nodes 4 and 5 and would find that the corresponding indexes in T_2 are 9 and 7 respectively. Then it would compute the lca between those indexes and would obtain node 6. The cluster that would be represented by this node, which is the one highlighted in Figure 3.2, would be compared with the cluster A, B in T_1. Since they have different number of leafs, T_2 does not contain that cluster which means that A, B is exclusive to T_1. After the algorithm goes up to node 3 and it would save there the lca between the taxa below so that it does not need to compute it again in the future. This process is repeated for node 6 and 2 which would compute the lca between node 6 and 10 instead of 9, 7 and 10. In this case the lca would remain node 6 and this time the clusters would be the same since each of them have three leafs. Finally, the algorithm would conclude that cluster D, E, F is common to both trees as well and when it reaches the root it stops.



(a) Cluster 1 **(b)** Cluster 2

Figure 3.3: Fully labelled trees.

3.2.2 Weighted trees

It is also possible to adapt the previous approach so that it can compute the WRF metric. To achieve this it is needed to save the total sum of the weights of both trees that are being compared before starting the algorithm. By having that information, that value will be the distance between the trees if there are no clusters that are common to both trees. Then by running the algorithm explained before to discover which clusters are present in both trees, instead of counting that cluster as equal, the value is corrected to consider that cluster as present in both trees. This can be done by removing the weights of the edges that correspond to the cluster in both trees and adding the absolute difference between the values of the weights. This approach can be seen as first considering that all clusters are present in only one tree and then correcting for the clusters that are found in both trees.

3.2.3 Fully labelled trees

To extend the RF metric, it was also implemented a way to compare phylogenetic trees that contain taxa in all nodes. To do this, a cluster is considered equal to another if the taxa present in both is exactly the same regardless of the depth. For example in Figure 3.3, the clusters highlighted are considered to be the same even though in cluster 1 the taxon C is in the leaf but in cluster 2 is not. This can be achieved by using an approach very similar to the ones explained in section 3.2.1. The only thing that needs to be done is to also consider the information inside the internal nodes. This means, before comparing the clusters, computing the lca value obtained in step 2 with the corresponding index where the internal node taxa is located in T_2.

3.2.4 Information about the clusters

One of the main advantages of this approach is that it is possible to save additional information about the difference between the trees. This is done by whenever it is found a cluster that is present in both

trees, the indexes where that cluster is present in both trees are stored. This way it is possible to know which clusters are exclusive from both trees by searching if the index of the node that represents them were stored. This can be very useful not only to verify if the distance obtained by the metric represents well the difference between the trees, but also it gives a certificate that confirms the distance obtained is actually correct.

3.3 Triplets

Unlike the previous approach, this one only considers phylogenetic trees that are binary. In this section it will be discussed an approach to compute the Triplets metric using the data structures referred in section 3.1.

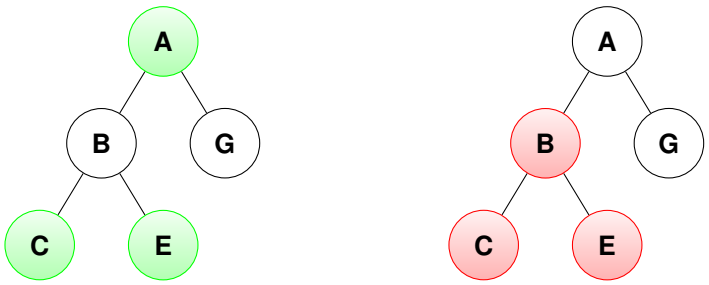
3.3.1 Triplets distance computation

The idea used in this approach to verify if a given triplet has the same topology in both trees is to use the $ComputeShared$ function explained in section 2.2.2.A.

The approach starts by traversing T_2 and, for each node, assigns red to all nodes to its left and blue to all nodes to its right. To do this it suffices to use the pre-order indexation of the boundaries, in this case the left child with highest index l and the right child with highest index r and the node itself m. With these values it is possible to verify the color depending on the index of a given node being between two values. If it is between m and l is red, if it is between l and r is blue.

Then for each assignment, it traverses T_1 and for every node searches how many blue and red nodes it finds to the left and to the right. With these four values it uses the $ComputeShared$ equation to count how many triplets it found to have the same topology. Applying this process between all combinations of nodes in T_1 and T_2 it is possible to find the number of triplets that share the same topology by adding all the $ComputeShared$ returns. Therefore it is possible to find how many triplets do not share the same topology by subtracting the number of triplets with same topology to the total number of triplets. Summarising, this approach consists in applying the following steps:

1. Transverse T_2 in a post-order traversal.
2. For each internal node in T_2 assign red to all nodes to its left and blue to all nodes to its right.
3. For each assignment traverse T_1 in a post-order traversal.
4. For each internal node in T_1 count the number of left reds, left blues, right reds and right blues.
5. Apply the $ComputeShared$ equation for each combination of internal nodes between the first and second tree.



(a) Resolved triplet. **(b)** Unresolved triplet.

Figure 3.4: Fully labelled trees.

6. Sum all the values obtained by the $ComputeShared$ equation to obtain number of triplets with shared topology.

7. Return the total number of triplets minus the value previously obtained.

3.3.2 Fully labelled trees

To extend the metric to compare fully labelled trees it is important to understand what was considered as triplets having the same topology when comparing triplets that contain information in internal nodes. When comparing binary trees that only contain information on leaves, it is always possible to separate a triplet in two taxa that are always closest to the third one. Given that now the internal nodes also contain information it is possible to have a case where a triplet (X, Y, Z) has $lca(X, Y) = lca(X, Z) = lca(Y, Z)$. For example, in Figure 3.4(b), the triplet (B, C, E) can not be separated since $lca(B, C) = lca(B, E) = lca(C, E) = B$. In these cases, the triplet will be assigned as unresolved. However it is also possible to have triplets containing internal nodes that can be separated. In Figure 3.4(a) it is possible to see with that the triplet (A, C, E) is possible to separate since $lca(C, E) = B$, $lca(C, A) = lca(E, A) = A$ and B depth is lower than A. In the cases where the triplet can be separated, it will be assigned as resolved.

For the extension approach a triplet topology is only considered equal when the triplet is resolved in both trees. This means that if a triplet is unresolved in any of the trees it is considered to have a different topology.

To extend the metric to also consider internal nodes, given that the indexation of the trees representation is already in preorder, the colors division process does not need to change since the internal nodes will always be inside the boundaries referred in section 3.3.1. To consider the internal nodes in the $ComputeShared$ equation, there is a need to also look for color nodes when traversing T_2.

3.3.3 information discussion

For this metric, keeping additional information is a lot more complex than in the RF metric due to the combinatorial explosion of triplet combinations. It would be possible to change the implementation to keep all the triplets that are in common but the tradeoff between efficiency and the benefits that saving all combinations of triplets would give in terms of readability of the data would not compensate. Even more storing all the triplets with the same topology would lead to high memory requirements which would cancel the advantage that our algorithm gives.

4

Implementation

Contents

All algorithm implementations in this thesis uses the C++ programming language and are available in the git repository at https://github.com/pedroparedesbranco/TreeDiff. To represent the bit vectors, the implementation uses the Succinct Data Structure Library (SDSL) [14] which contains three implementations to compute the fundamental operations mentioned in Section 3.1.2. The one that it was chosen to extend was the `bp_support_sada.hpp` since it was the one that obtained the best results in terms of time and space requirements. To represent the vector that correlates the taxa between both trees it was used 32bit integers.

4.1 Parsing phase

The parsing phase receives two trees in Newick format. Then the goal is to parse this format, creating the two bit vectors and the vector of integers that the algorithm receives as input. In this phase, when parsing the Newick format, the construction of the bit vectors is straightforward since the Newick format already has the parenthesis in order to represent a balanced parenthesis bit vector. The only detail that needs to be taken into account is that when it encounters a leaf, it needs to add two parentheses, the first one open and the second one closed. To create the vector of integers it was used a temporary hash table that associated the labels present in the newick format with an index that was assigned according to the indexation explained in section3.1. When a labelled node is found while traversing the first tree, it is added to the hash table the mapping between the labelled found and the index where it is located. Then, when a labelled node is found in the second tree, the algorithm looks for that label in the hash table to find the index where it is located in the first tree. By having the indexes where that specific label is located in both trees, the only thing that needs to be done to build the integers vector is to store the second tree location in the position corresponding to the first tree location. Doing this to all the labelled nodes creates the final integers vector.

To compute the WRF metric there is also necessary to create two float vectors with the same size of the integers vector to save the weights that correspond to a given label for each tree. These vectors can be achieved by simply saving the weights in the position that corresponds to the index where they are located in each tree. This process can be seen in Algorithm 4.1.

4.2 Added operations

The succint data structures library already provided an implementation of the rank1, select1, findOpen, findClose, enclose and rmq operations on `bp_support_sada.hpp` file. To extend this library it was created a new file `bp_support_sada_extended.hpp` that added an implementation to the operations needed to compute the metrics and were not implemented that. These operations are the PreOrderMap, Pre-

Algorithm 4.1: Parsing phase

```
Input: T_1, T_2 in newick format.
Output: bv_1, bv_2, w_1, w_2, CodeMap, weightsSum
HashTable ← null
weightsSum ← 0
for i = 1; i < 3; i++ do
    bv_i ← null
    index ← 0
    s ← null                                   ▷ s is a stack of integers
    while c ← getChar(T_i) ≠ ';' do
        if c = '(' then
            push(s, index)
            index++
            push_back(bv_i, 1)
        else if c == ',' then
            continue
        else if c = ')' then
            c ← getChar(T_i)
            if not (c = ')' or c = '(' or c = ',' or c = ';') then
                if c = ':' then
                    c ← getChar()
                    weight ← getWeight(T_i)
                    w_i[index] ← weight
                    weightsSum ← weightsSum + weight
                else
                    label ← getLabel(T_i)
                    if i = 1 then
                        HashTable[label] ← index
                    else
                        CodeMap[HashTable[label]] ← count
            else
                c ← ungetChar(T_i)
            pop(s)
            push_back(bv_i, 0)
        else
            push_back(bv_i, 1)
            push_back(bv_i, 0)
            label ← getLabel(T_i)
            if i = 1 then
                HashTable[label] ← index
            else
                CodeMap[HashTable[label]] ← count
free(HashTable)
return bv_1, bv_2, w_1, w_2, CodeMap, weightsSum
```

OrderSelect, PostOrderSelect, IsLeaf, lca, ClusterSize and NumLeaves and their implementation can be seen in Algorithm 4.2. Moreover, it was added two operations that were already in the SDSL library but not present in the `bp_support_sada.hpp` file. These operations were the rank10 operation to enable us to count the number of leaves, and the select0 operation to enable us to go through the tree in a post order traversal.

Algorithm 4.2: Operations added

int preOrderMap(bv, v**):**
    return rank1(bv, v)

int preOrderSelect(bv, i**):**
    return select1(bv, i)

int postOrderSelect(bv, i**):**
    return findOpen(select0(bv, i))

int isLeaf(bv, i**):**
    return bv$[i + 1] == 0$

int lca(bv, u, v**):**
    if u $>$ v:
        u $\leftrightarrow$ v
    return enclose(bv,rmq(bv, u, v) + 1)

int clusterSize(bv, v**):**
    return (findClose(bv, v) - v + 1) / 2

int numLeaves(bv, u, v**):**
    return rank10(bv, findClose(bv, v)) - rank10(bv, v) + 1

4.3 Robinson Foulds

For the RF metric, it was implemented two approaches to make sure that the algorithm traverses the tree in a post-order traversal. The first one is the `rf_postorder` and it takes advantage of the PostOrderSelect operation while the second one is the `rf_nextsibling` and takes advantage of the NextSibling and FirstChild operations.

4.3.1 Robinson Foulds using PostOrderSelect

To make sure that the tree is traversed in a post-order traversal, this implementation simply does a loop from 1 to n and calls the function PreOrderSelect for all the values. This implementation also uses a very similar strategy than the one used in Day's algorithm explained in section 2.2.1.B to keep the lca results computed earlier. This strategy consists in using a stack that keeps track of two informations about the nodes, the index of the corresponding cluster from the second tree and the size of the cluster from the

first tree. Then, whenever it is found a leaf it is added the corresponding index as well as the size of the leaf which is one. When the node found is not a leaf, it goes through the stack and finds the last nodes that where added until the sum of their sizes is equal to the size of the cluster. With these nodes it is computed the lca between them while removing them from the stack. When this process is done it is possible to verify if that cluster is present in the other tree and then add the information of that cluster to the stack so that the computations of the lca that where made are not done again. An implementation of this process can be seen in Algorithm 4.3.

Algorithm 4.3: `rf_postorder` implementation

```
begin
    equalClusters ⟵ 0
    for i ← 1 to N do
        p ← PostOrderSelect(v1, i)
        if IsLeaf(v1, p) then
            lca ← PreOrderSelect(Code[PreOrderMap(v1, p) - 1] + 1)
            size ← 1
            push(< lca, size >, s)
        else
            cs ← ClusterSize(v1, p)
            while cs ≠ 0 do
                < lca, size >← pop(s)
                lcas ← lca(v2, lca,)
                if lcas = null then
                    < lcas, size >← pop(s)
                else
                    < lca, size >← pop(s)
                    lcas ← lca(v2, lcas, lca)
                cs = cs − size
            if NumLeaves(v1, p) = NumLeaves(v2, lcas) then
                equalClusters ← equalClusters + 1
            push(< lcas, ClusterSize(v1, p) + 1 >, s)
            lcas ← null
    distance ← (numInternalNodesv1 + numInternalNodesv2 - equalClusters*2) / 2
    return distance
```

4.3.2 Robinson Foulds using NextSibling and FirstChild

To make sure that the tree is traversed in a post-order traversal, this implementation takes a slighly different approach than the first one. It uses a recursive function that is called for the first time for the root node. Then it verifies if the given node is not a leaf and if that is the case it calls the function to the first child node. Then for all the calls it goes through all the siblings of the given node and computes the lca between all of them. Whenever the node in question is not a leaf, the algorithm uses the lca value

computed and the operation NumLeaves to verify if the cluster is common to both clusters. The function returns the index of the lca obtained so that in this implementation there is no need to keep a stack to reuse the lca values computed throughout the algorithm. An implementation of this process can be seen in Algorithm 4.4.

Algorithm 4.4: `rf_nextsibling` implementation

```
rf(current)
begin
    if IsLeaf(v1, current) then
        lcas ← PreOrderSelect(Code[PreOrderMap(v1, current) - 1] + 1
    else
        lcas ← rf(FirstChild(v1, current))
        if NumLeaves(v1, current) = NumLeaves(v2, lcas) then
            equalClusters ← equalClusters + 1
    while NextSibling(v1, current) do
        current ← NextSibling(v1, current)
        if IsLeaf(v1, current) then
            lcas ← lca(v2, lcas, PreOrderSelect(Code[PreOrderMap(v1, current) - 1] + 1)
        else
            lcas_aux ← rf(FirstChild(v1, current))
            lcas ← lca(v2, lcas, lcas_aux)
            if NumLeaves(v1, current) = NumLeaves(v2, lcas) then
                equalClusters ← equalClusters + 1
    return lcas
```

4.4 Triplets

For the Triplets metric, it was implemented the `trip_treediff` program. This implementation uses two recursive functions, the DivideColors and the CountTriplets. The first one traverses the second tree and it calls the CountTriplets function for each node it traverses given three variables as arguments: $init$, mid and $last$ that represents the indexes of the node itself, the last left child and the last right child. To know these indexes without any additional computation, the tree is traversed in a post-order traversal and this process can be seen in Algorithm 4.5. The CountTriplets function traverses the first tree in post-order as well and when it goes through a leaf, it verifies which is the associated color. Then, while it goes up, for each internal node saves how many right reds, right blues, left reds, left blues where found below it and apply the $ComputeShared$ equation described in 2.4. The CountTriplets function process can be observed in Algorithm 4.6. Finally it returns the total number of triplets minus the number of triplets found to have the same topology.

Algorithm 4.5: Divide colors

```
begin
    if IsLeaf(v1, root) then
        return 1
    else
        mid ← trip(FirstChild(root))
        last ← mid + trip(root + mid)
        countTriplets(root, root + mid, root + last)
        return last
```

Algorithm 4.6: Count Triplets

```
begin
    if IsLeaf(v1, root) then
        results ←< 1, 0, 0, 0, 0 >
        if init < code[PreOrderMap(root) - 1] + 1 < mid then
            results[branch * 2 + 1] = 1
        else if init < code[PreOrderMap(root) - 1] + 1 < mid then
            results[branch * 2 + 1] = 1
        return results
    else
        results1 ← CountNodes(FirstChild(root))
         results2 ← CountNodes(PreOrderSelect(PreOrderMap(root) + results[0] + 1)
        results[0] ← results1[0] + results2[0] + 1
        results[1] = results1[1] + results1[3]
        results[2] = results1[2] + results1[4]
        results[3] = results2[1] + results2[3]
        results[4] = results2[2] + results2[4]
        triplets_equal ←
         triplets_equal + computeShared(results[1], results[2], results[3], results[4], )
        return results
```

4.5 Baseline implementations

To compare the `rf_postorder`, `rf_nextsibling` and `trip_treediff`, it was implemented both the Day's algorithm described in section 2.2.1.B and the approach that computes the Triplets metric described in section 2.2.2.A. The first one it is called `rf_day` and it also contains a parsing phase that transforms the newick format in the table that is received as input in the Day's approach. This process uses a recursive function and it can be seen in Algorithm 4.7 where the variables $size$ and $leafCount$ starts with value 0 in the first call. This process also uses a hash table to assign the clusters an index but this time the index is assigned by the order the labels appear in the first tree. This means that in the cluster table the third column is not necessary since it would only save the number of the line where it is located. The rest of the algorithm is implemented the same way it was previously described.

For the second one it is called `trip_sht` and it implements the smaller half trick exactly like it was

Algorithm 4.7: Day's algorithm parsing phase

```
begin
    while c ← getChar() ≠; do
        if c == ( then
            size = size + dayParse(size = 0)
        else if c ==, then
            continue
        else if c ==) then
            treeSize.pushback(size)
            treeLabel.pushback(−1)
            return size + 1
        else
            label ← getLabel()
            if firstTree then
                HashTable[label] ← leafCount
                treeLabel.pushback(leafCount)
            else
                treeLabel.pushback(HashTable[label])
            leafCount++
            size++
    return-1
```

explained in section 2.2.2.A. However, instead of implementing the hierarchical decomposition tree, the counting of triplets was applied in the original tree since its efficiency is similar in practise.

5

Evaluation

Contents

In this section it will be discussed both the theoretical and empirical aspects of the implementations' complexities, in part by comparing them with the baseline implementations.

5.1 Theoretical evaluation

This section will start by comparing the time complexity of the `rf_postorder`, `rf_nextsibling` and `trip_treediff` implementations. Then it ends by comparing the memory requirements of these implementations, comparing also with the baseline approaches.

5.1.1 Running time complexity

The complexity of both parsing phases are $O(n)$ since they loop through all the nodes.

In relation to the RF metric, the complexity analysis is separated between the number of times the lca, NumLeaves, ClusterSize, PostOrderSelect and NextSibling are called and which depends on n.

For the RF computed with the `rf_postorder` implementation, it is called the PostOrderSelect for each node therefore n times. The number of times the lca is called is equivalent to the number of leafs minus the number of first leafs which in the worst case is $n-2$. This case would be a tree that only contains the root as an internal node. For the NumLeaves operations, it is called two times for each internal node and the worst case is $2 \times (n-1)$. This case is when a tree only has one leaf. It is also called the ClusterSize function one time for each internal node which would result in $n-1$ times for the same case as before. In total, the complexity would be $(n+(n-2)+(2\times(n-1))+(n-1)) \times O(\log(n)) = (5n-5) \times O(\log(n))$. However, the worst case for the lca operations is the best case for the NumLeaves and ClusterSize and vice-versa. This means that in the worst case, the lca operation would be applied 0 times and the complexity would be $(4n-3) \times O(\log(n)))$.

For the RF computed with the `rf_nextsibling` implementation, the number of calls for the lca and NumLeaves are the same as before. The number of NextSibling operation calls is the same as the lca operation. This means that the complexity for this case is $((n-2)+(n-2)+(2\times(n-1))) \times O(\log(n)) = (4n-5) \times O(\log(n))$, however for the same reason as before it can be reduced to $(2n-2) \times O(\log(n))$.

For the WRF metric computed by both the `rf_postorder` and `rf_nextsibling` implementations, the number of times this operations are called is the same as the original RF.

For the FLRF computed by both implementations the only difference is that the lca operation is called also for all internal nodes. This means that the number of times the lca operation is applied is equivalent to the total number of nodes minus the number of first leafs. This would change the complexity in both of the implementations. In `rf_postorder` it would change to $(5n-4) \times O(\log(n))$ since the number of lca calls passed from 0 to $n-1$ in the worst case. In `rf_nextsibling` it would change to $(3n-3) \times O(\log(n))$. All implementations referred above have then a running time complexity of $O(n \log(n))$.

Now looking into the Triplets metric, the complexity of the `trip_treediff` implementation does not depend of any operation since all that are being used have complexity equal to $O(1)$. Given that the algorithm computes the $ComputeShared$ equation for all combinations of nodes between the first and second tree, the final complexity is $O(n^2)$.

The summary of the complexities for all the implementations referred above can be observed in Table 5.1.

5.1.2 Memory comparison

In terms of memory usage, the RF computed by both the `rf_nextsibling` and `rf_postorder` implementations only uses two bit vectors of size $2n$ and a vector of 32bit integers with size n so the total of bits used is $36n$ bits, however the last one needs an additional stack to save the lca values. The `rf_day` implementation needs four vectors of 32bit integers with size n as well as two vectors of 32bit integers with the size equal to the number of leafs which is equal to $n - 1$ in the worst case. This results in $32 \times 4n + 32 \times 2(n - 1) = 192n - 64$ bits.

In relation to the triplets metric, the `trip_treediff` implementation only uses the same two bit vectors and the integer vector mentioned before and therefore $36n$ bits. The `trip_sht` implementation also needs to save how many red, blue and white nodes each node contains below them as well as how many triplets where found at that point. Given the combinatorial explosion related to the number of triplets there is a need to store that last value has a 64bit integer. This implementation needs in total $3 * 32n + 64n = 160n$ bits.

5.2 Experimental evaluation

In this section it is discussed the performance of the `rf_postorder`, `rf_nextsibling` and `trip_treediff` implementations in comparison to their baselines and extensions. To evaluate the implementations we used the ETE Python library to create a random generator of trees in Newick format. To evaluate the RF metric, we generated ten trees starting with 10000 leaves and going to trees 100000 leaves all having the same interval between them. Given the fact that the Triplets metric is more time consuming, for this metric we generated ten trees starting with 1000 leaves and going to 10000 leaves also with the same interval between them. It was genarated not only five pairs of trees with information only on the leaves for all the sizes tested but also for fully labelled trees. To analyse the memory usage it was used the valgrind massif tool [15].

First it was analysed the memory used throughout the algorithm for a tree that contains 100000 leaves. Figure 5.1 shows that for the first phase, the memory consumption is higher since a hash table has to be used to create the vector of integers that correlates the taxa. In the transition to the second

Table 5.1: Implementations theoretical complexities.

Implementation	Metric	Complexity
`rf_postorder`	RF	$O(n \log(n))$
`rf_postorder`	WRF	$O(n \log(n))$
`rf_postorder`	WRF	$O(n \log(n))$
`rf_postorder`	FLRF	$O(n \log(n))$
`rf_postorder`	Fully Labelled Weighted Robinson Foulds (FLWRF)	$O(n \log(n))$
`rf_nextsibling`	RF	$O(n \log(n))$
`rf_nextsibling`	WRF	$O(n \log(n))$
`rf_nextsibling`	FLRF	$O(n \log(n))$
`rf_nextsibling`	FLWRF	$O(n \log(n))$
`Day's algorithm`	RF	$O(n)$
`trip_treediff`	Triplets	$O(n^2)$
`trip_treediff`	FLT	$O(n^2)$
`Triplets efficiently`	Triplets	$O(n \log(n))$

phase, since the hash table is no longer needed and, as such, the memory it required is freed, the memory consumption falls abruptly. In the second phase of the algorithm, the memory consumption is lower since this phase only uses the information stored in the two bit vectors and related data structures, and in the vector that correlates the two trees.

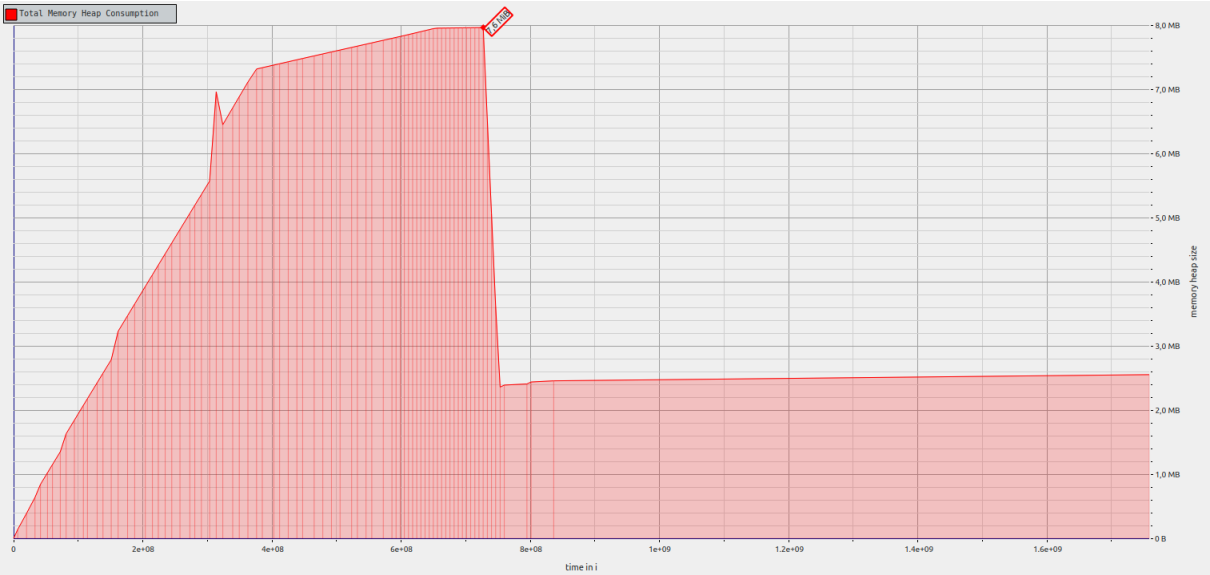


Figure 5.1: Heap allocation profile for two trees with 100000 leaves in `rf_postorder` implementation.

Note that by serializing and storing the trees as their bit vector representations, it can be avoid the memory required for parsing the Newick format.

Secondly, it was compared the memory peak usage during the second phase of the `rf_postorder` and `rf_day` implementations. Figure 5.2 shows the comparison between both algorithms in terms of their memory peak usage. The `rf_postorder` exhibits a significant lower memory peak compared to the `rf_day` algorithm.

Then, it was compared the running time between the two algorithms for the parse and algorithm phases. For the parse phase only compares the `rf_postorder` with the `rf_day` since the `rf_nextsibling` has the exact same parsing phase has `rf_postorder`. It can be observed in Figure 5.3(b) that the differences between the running time of the two implementations are not significant. In relation to the algorithm phase, the comparison was made between all three implementations. In Figure 5.3(a), it can be observed that both the `rf_postorder` and `rf_nextsibling` implementations presented almost the same results. Even though the theoretical complexity of those implementations are $O(n\log(n))$, in practise it seems to be almost linear. This finding suggests that the operations used to traverse the tree tend to be almost $O(1)$ in practise even though they have a theoretical complexity of $O(\log(n))$. The difference in the constant between these implementation and the `rf_day` one can be explained by the number of operations that are computed for each node as explained in Section 5.1.

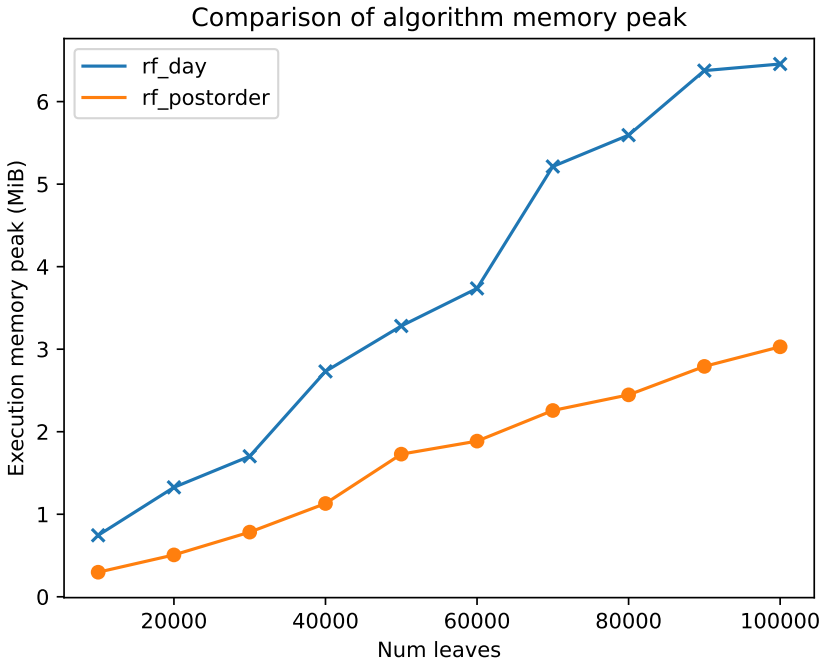


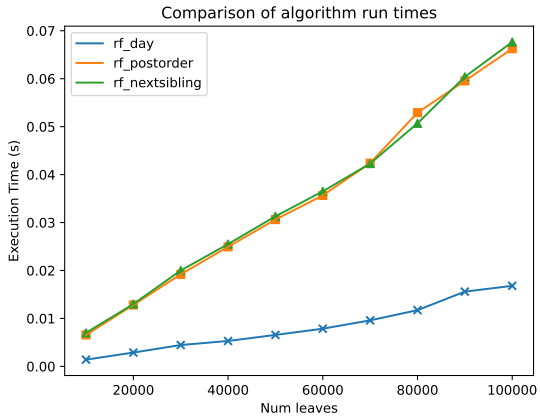
Figure 5.2: Memory usage peak comparison.

Next, it was compared the run time for the RF and FLRF metrics. In both of the `rf_postorder` and `rf_nextsibling` implementations, the FLRF metric seems to also be linear in practise. However in the `rf_nextsibling` the difference between constants seems to be much insignificant than the difference in the `rf_postorder`. These results are consistent with the theoretical analysis since in the `rf_nextsibling` implementation the number of times each operation is applied in the worst case is $(3n - 3)$ and in the `rf_postorder` is $(5n - 4)$. This concludes that the `rf_nextsibling` implementation gives better results for most trees when it is used to compute the distance for fully labelled trees.

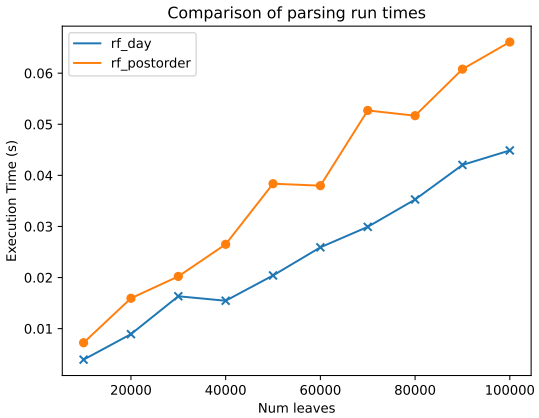
To analyse the `trip_treediff` implementation, the same process was done between this implementation and the `trip_sht` one. To start, in Figure 5.5 it can be seen a comparison of the run time between these two implementations and it concludes that in practise the performance of both implementations corresponds to the theoretical complexities. This means that in practise `trip_treediff` is $O(n^2)$ and `trip_sht` is $O(n \log(n))$.

Then it was analysed the peak of memory peak usage between both implementations. Figure 5.6 shows that the `trip_treediff` implementations exhibits a significant lower memory peak compared to the `rf_day` algorithm.

Finally, it was made a comparison between the Triplets and the FLT metric. In Figure 5.7, it can be seen that the run times are almost identical, this makes sense since the FLT metric implementation does not need to compute any additional operation in relation to the Triplets one.

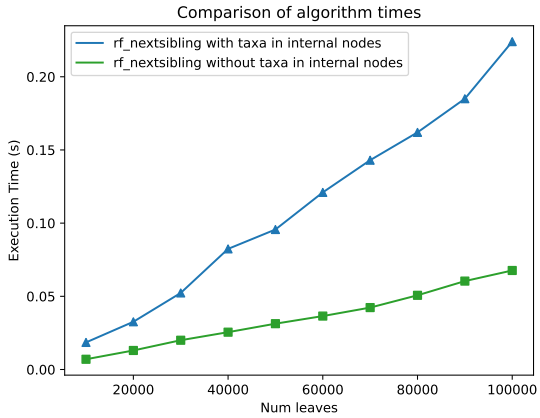


(a) Algorithm phase

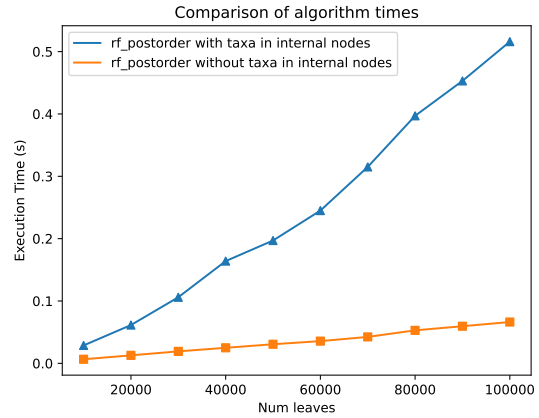


(b) Parsing phase

Figure 5.3: Run time for trees with different sizes.



(a) rfTreediff using NextSibling for labbeled trees.



(b) rfTreediff using PostOrderSelect for labbeled trees.

Figure 5.4: Run time comparison.

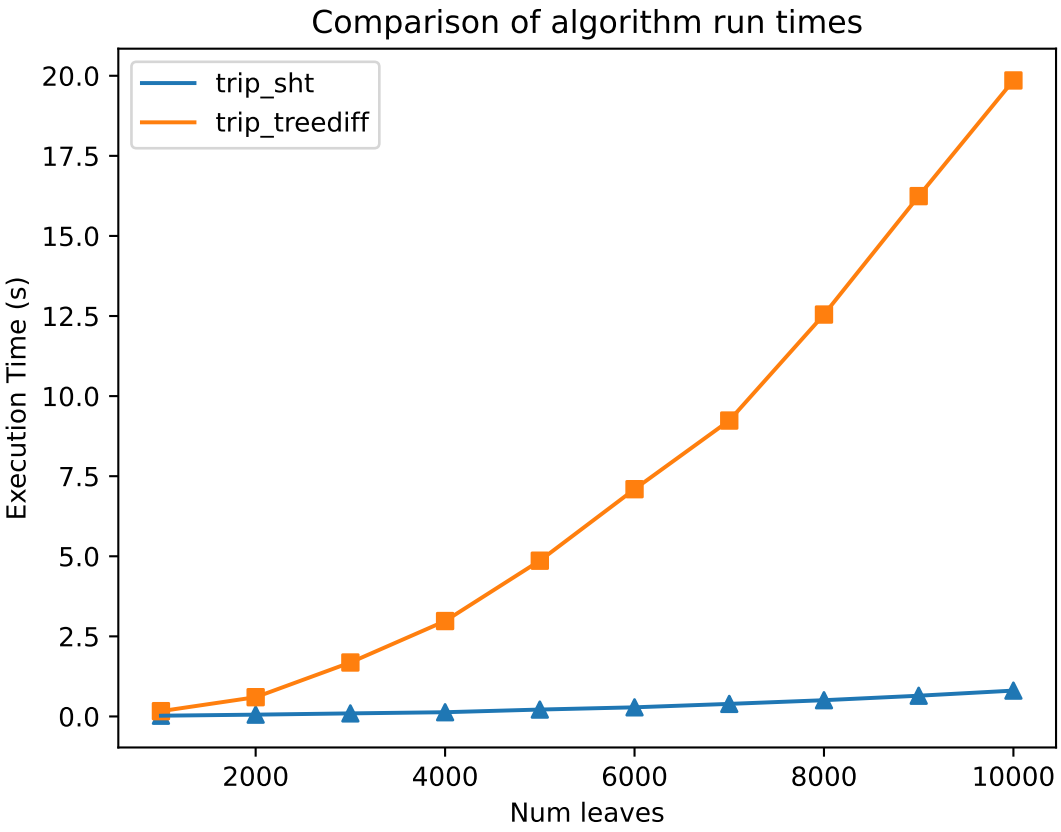


Figure 5.5: Run time comparison between `trip_treedif` and `trip_sht`.

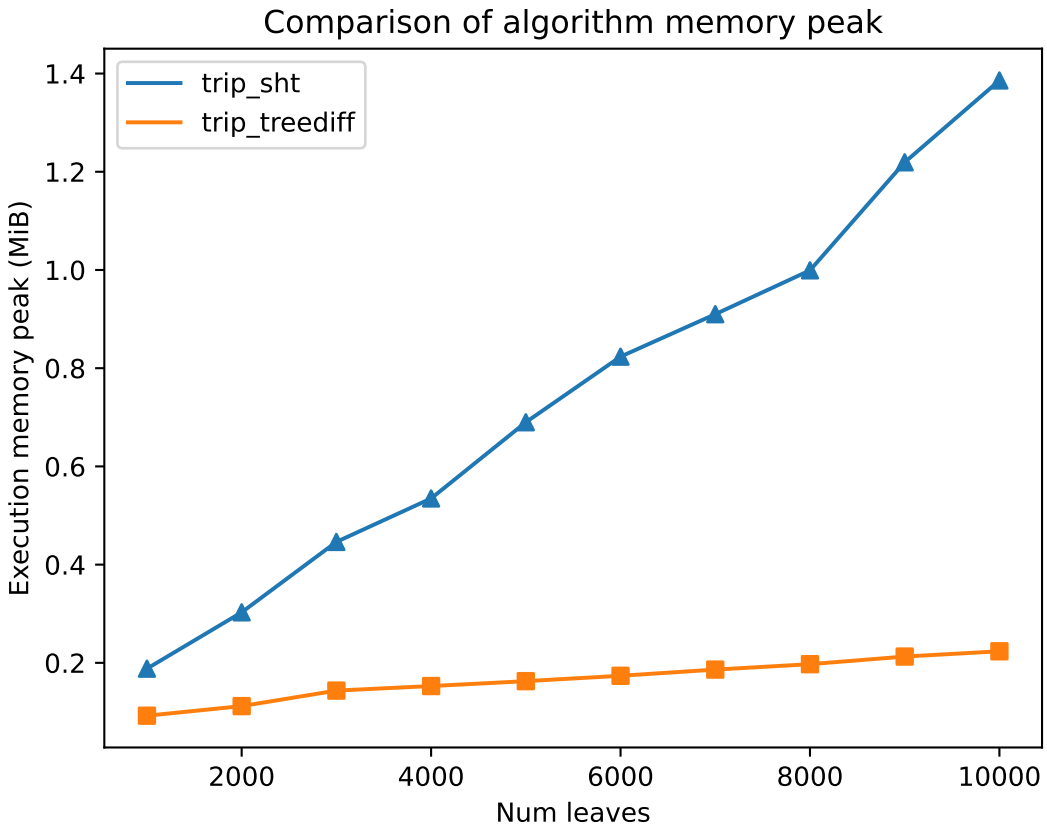


Figure 5.6: Memory usage peak comparison between `trip_treedif` and `trip_sht`.

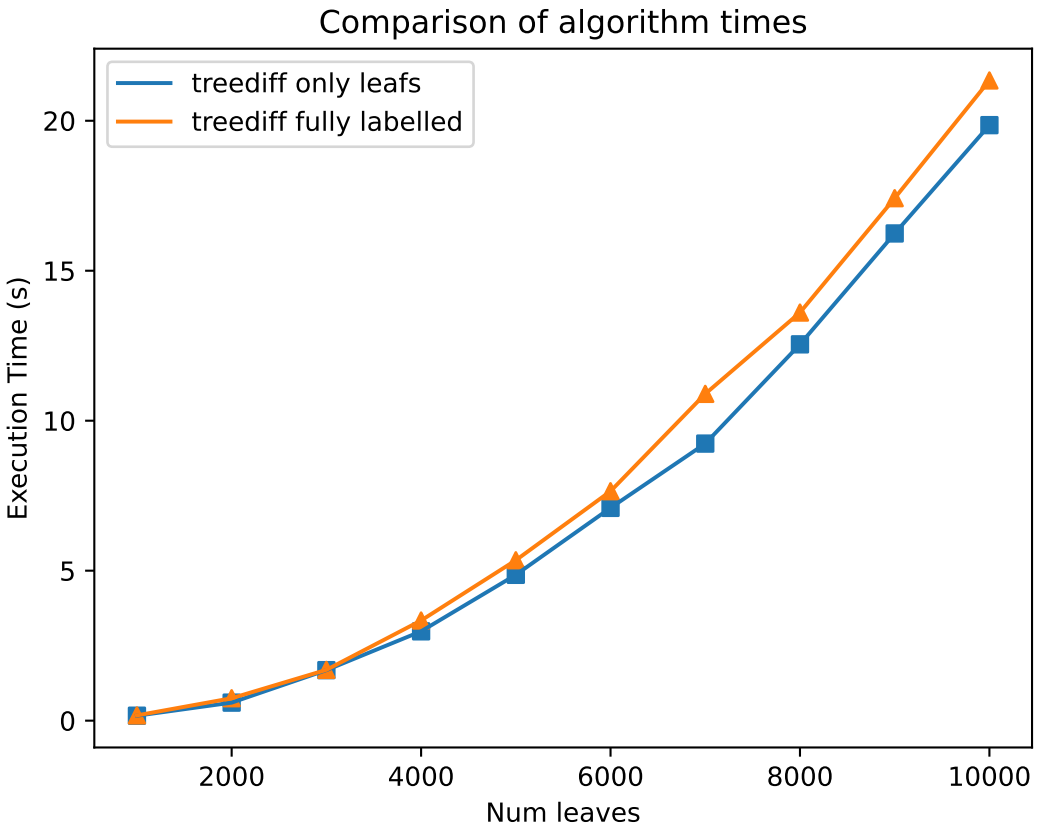


Figure 5.7: Run time comparison for fully labelled trees using `trip_treediff`.

6

Conclusion

Contents

It is essential to know the evolution of certain species or taxonomic groups such as SARS-CoV-2 to determine their origin, evolution and resistance patterns to the treatments under study. These are examples among several applications that can be modeled through phylogenetic trees. There are several methods to infer these tree, some of them infer trees that only contains information on the leafs while others infer trees that contain information in all nodes.

Comparing phylogenetic trees can give us a lot of relevant information about which inferring methods did the best job to represent the original dataset. However most of the metrics that exists only compare trees that contain information only on the leafs. Another problem is that, given the large and increasing volume of phylogenetic data, space requirements become an issue both while computing tree distances and while storing trees.

6.1 Achievements

In this thesis it is provided an approach to compute the RF and Triplets metric using succinct data structures. This work also demonstrates how the approach can be applied to extend those metrics. The first extension to the RF metric considered was to consider fully labelled trees and this metric was given the name FLRF. It was also implemented the WRF metric and this implementation consider weighted trees. Finally it was demonstrated a way to retrieve additional information to help identifying the similarities and dissimilarities between the trees. Concerning the Triplets metric it was also extended to consider fully labelled trees and this metric was given the name FLT. After the experimental evaluation, it was concluded that the RF, FLRF and WRF metrics implementations, in practise, achieved a linear time complexity in relation to the number of nodes in the tree while having very low memory consumption. For the Triplets and FLT metrics, it was achieved a complexity of $O(n^2)$ and can also be computed using very low memory consumption.

6.2 Future Work

There are several things that can be done to continue the work presented in this thesis.

For the RF metric, it would be interesting to explore a dynamic update approach that could recompute the distance of a given tree when a specific node is added.

For the Triplets metric, it would be beneficial to study how to adapt the structures used to compare non-binary tree and how would it impact in terms of the performance of the algorithm. Another thing that can be done in relation to this metric is to explore different methods to store information about the Triplets distance other than saving all triplets with different topologies. It would also be very interesting to implement a metric that also consider unresolved triplets to have the same topology when they are

unresolved in both trees and compare the results of this implementation with the `trip_treediff` one.

Finally it would be worth to expand the approach to compute other metrics like the Quartets one.

Bibliography

[1] D. H. Huson, R. Rupp, and C. Scornavacca, *Phylogenetic Networks: Concepts, Algorithms and Applications*. Cambridge University Press, 2010.

[2] 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, pp. 1–15, 2009.

[3] *UPGMA (unweighted pair group method with arithmetic means)*. Dordrecht: Springer Netherlands, 2008, pp. 2068–2068. [Online]. Available: https://doi.org/10.1007/978-1-4020-6754-9_17806

[4] M. K. Kuhner and J. Yamato, "Practical performance of tree comparison metrics," *Systematic Biology*, vol. 64, no. 2, pp. 205–214, 2015.

[5] D. F. Robinson and L. R. Foulds, "Comparison of phylogenetic trees," *Mathematical biosciences*, vol. 53, no. 1-2, pp. 131–147, 1981.

[6] W. H. Day, "Optimal algorithms for comparing trees with labeled leaves," *Journal of classification*, vol. 2, pp. 7–28, 1985.

[7] D. F. Robinson and L. R. Foulds, "Comparison of weighted labelled trees," in *Combinatorial Mathematics VI: Proceedings of the Sixth Australian Conference on Combinatorial Mathematics, Armidale, Australia, August 1978*. Springer, 2006, pp. 119–126.

[8] S. Böcker, S. Canzar, and G. W. Klau, "The generalized robinson-foulds metric," in *International Workshop on Algorithms in Bioinformatics*. Springer, 2013, pp. 156–169.

[9] O. Dubois, "On the r, s-sat satisfiability problem and a conjecture of tovey," *Discrete Applied Mathematics*, vol. 26, no. 1, pp. 51–60, 1990.

[10] D. F. Robinson and L. R. Foulds, "Comparison of weighted labelled trees," in *Combinatorial mathematics VI*. Springer, 1979, pp. 119–126.

[11] G. S. Brodal, R. Fagerberg, T. Mailund, C. N. Pedersen, and A. Sand, “Efficient algorithms for computing the triplet and quartet distance between trees of arbitrary degree,” in *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 2013, pp. 1814–1832.

[12] G. F. Estabrook, F. McMorris, and C. A. Meacham, “Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units,” *Systematic Zoology*, vol. 34, no. 2, pp. 193–200, 1985.

[13] G. Navarro, *Compact data structures: A practical approach*. Cambridge University Press, 2016.

[14] S. Gog, T. Beller, A. Moffat, and M. Petri, “From theory to practice: Plug and play with succinct data structures,” in *13th International Symposium on Experimental Algorithms, (SEA 2014)*, 2014, pp. 326–337.

[15] N. Nethercote and J. Seward, “Valgrind: A program supervision framework,” *Electronic notes in theoretical computer science*, vol. 89, no. 2, pp. 44–66, 2003.