Phylogenetic tree distance computation over succinct representations

António Pedro Branco[1,2], Cátia Vaz[1,3], and Alexandre P. Francisco[1,2]

[1] INESC-ID Lisboa
[2] Instituto Superior Técnico, Universidade de Lisboa
[3] Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa

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. An approach is then to compute a similarity or dissimilarity measure between trees, with Robinson and Foulds metric being one of the most used, and which can be computed in linear time and space. 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. We propose then an efficient implementation of the Robinson Foulds metric over trees succinct representations. Our implementation generalizes also the Robinson Foulds metrics to labelled phylogenetic trees, *i.e.*, trees containing labels on all nodes, instead of only on leaves. Experimental results show that we are able to still achieve linear time while requiring less space. Our implementation is available as an open-source tool for phylogenetic analysis.

Keywords: tree dissimilarity · succinct data structures · phylogenetic trees · Robinson–Foulds metric

1 Introduction

Comparative evaluation of differences, similarities, and distances between phylogenetic trees is a fundamental task in computational phylogenetics [5]. There are several measures for assessing differences between two phylogenetic trees, some of them based on rearrangements, others based on topology dissimilarity and in this case some of them take also into account the branch-length [8]. The rearrangement measures are based on finding the minimum number of rearrangement steps required to transform one tree into the other. Possible rearrangement steps include nearest-neighbor interchange (NNI), subtree pruning and regrafting (SPR), and tree bisection and reconnection (TBR). Unfortunately such measures are seldom used in practice for large studies as they are expensive

to calculate in general. NP-completeness has been shown for distances based on NNI [9], TBR [1], and SPR [2]. Measures based on topological dissimilarity are commonly used. One of the most used is the topological distance of Robinson and Foulds [12] (RF) and its branch-length variation [13]. The RF-like metrics, when applied to rooted trees are all based on the idea of shared clades, i.e., specific kinds of subtrees (for rooted trees) or branches defined by possession of exactly the same tips (the leaves of the tree). Namely, it quantifies the dissimilarity between two trees based on the differences in their clades, for rooted trees, or bipartitions if applied to unrooted trees. Formally, a clade or cluster $C(n)$ of a tree T is defined by the set of leaves (or nodes in fully labelled trees) that are a descendant from a particular internal node n.

The computation of the RF distance can be achieved in linear time and space. A linear time algorithm for calculating the RF distance was first proposed by Day [4]. It efficiently determines the number of bipartitions or clades that are present in one tree but not in the other. Furthermore, there have been advancements in the computation of the RF distance that achieve sublinear time complexity; Pattengale et al. [11] introduced an algorithm that can compute an approximation of the RF distance in sublinear time. However, it is important to note that the sublinear algorithm is not exact and may introduce some degree of error in the computed distances. Recently, it was proposed a linear time and space algorithm [3] that specifically addresses the labeled Robinson-Foulds (RF) distance problem, considering labels assigned to both internal and leaf nodes of the trees. For leaf-labeled trees the labeled RF distance reduces to the standard RF distance, where only the tree topology is taken into account.

Considering however 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 poses significant challenges in terms of space requirements for computing tree distances, or even for storing trees. Hence, we propose an efficient implementation of the Robinson-Foulds metric over succinct representations of trees. Our implementation not only addresses the standard Robinson-Foulds metric but it also extends it to handle labeled and weighted phylogenetic trees. By leveraging succinct representations, we are able to achieve practical linear time while significantly reducing space requirements. Experimental results demonstrate the effectiveness of our implementation in spite of expected trade-offs on running time overhead versus space requirements due to the use of succinct data structures.

2 Background

A phylogenetic tree, denoted as $T(V, E)$, is defined as a connected acyclic graph. The set V represents the vertices (nodes) of the tree, and E represents the edges connecting the vertices. The leaves of the tree, which are the vertices of degree one, are labelled with data that represents species, strains or even genes. A phylogenetic tree represents then the evolutionary relationships among taxonomical groups, or taxa.

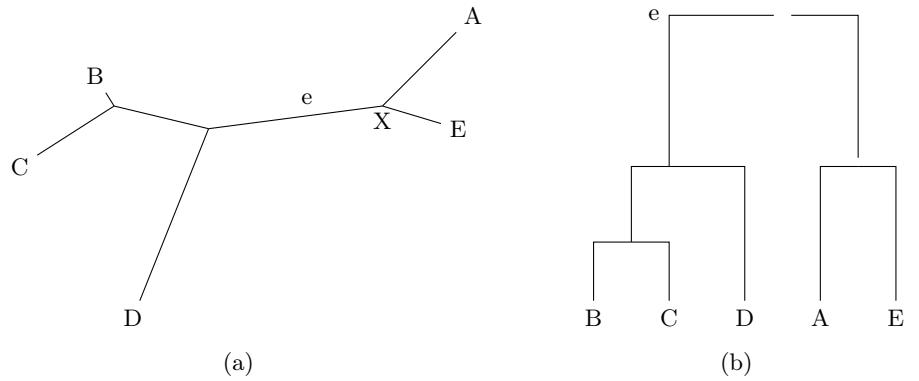


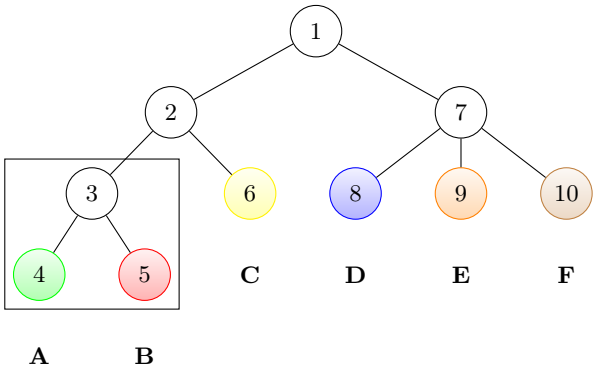
Fig. 1: An unrooted phylogenetic tree (a) and rooted phylogenetic tree (b) that resulted from transforming the unrooted one by selecting node X as root; e denotes the equivalent edge in both trees.

In most phylogenetic inference methods, the internal vertices of the tree represent hypothetical or inferred common ancestors of the entities represented by the leaves. These internal vertices do not typically have a specific label or represent direct data. However, there are some phylogenetic inference methods, like goeBURST [6], that infer a fully labelled phylogenetic tree. In such cases, the internal vertices may also have labels associated with them, providing additional information about the inferred common ancestors.

The inferred phylogenetic trees can be rooted or unrooted, depending on whether or not a specific root node is assigned. In the rooted ones, there exists a distinguished vertex called the root of T. To calculate the Robinson-Foulds (RF) metric on a rooted or unrooted tree, you need to compare the clades or bipartitions of the trees, respectively. The RF metric measures the dissimilarity between two trees by counting the differences in the clades they contain if rooted, or bipartitions, if unrooted. Nevertheless, we can convert an unrooted tree into rooted one by adding an arbitrary root node (see Figure 1). In this work, we are focused on rooted trees.

Phylogenetic trees are commonly represented and stored using the well known Newick tree format. For instance, the rooted tree in Figure 1 can be represented as `(((B, C), D),(A, E));`. In this example we have only labels on the leafs and we do not have edge weights. The Newick format supports however internal labelled vertices and edge weights, e.g., `(((B:0.2, C:0.2)W:0.3, D:0.5)X:0.6, (A:0.5, E:0.5)Y:0.6)Z;`. Assume by default that trees given as input in the experimental evaluation are represented with the Newick format.

Given two trees T_1 and T_2, the Robinson Foulds distance computation consists in obtaining all the clusters present in both of the trees and then counting the number of clusters that do not occur in both trees. For instance, if we consider the trees in Figure 2 and 3, we can see that the only clusters that are unique to their trees are the ones inside the square. Since there is only one of



$$B = (\, (\, (\, (\,) \, (\,) \,) \, (\,) \,) \, (\, (\,) \, (\,) \, (\,) \,) \,)$$

Fig. 2: Tree T_1 and its balanced parenthesis representation.

them for each tree, we can compute the Robinson Foulds distance as follows,

$$RF(T_1, T_2) = \frac{1}{2} \left(|C(T_1) - C(T_2)| + |C(T_2) - C(T_1)| \right) = \frac{1}{2} \left(1 + 1 \right) = 1,$$

where $C(T)$ denotes the set of clusters in tree T.

3 Approach

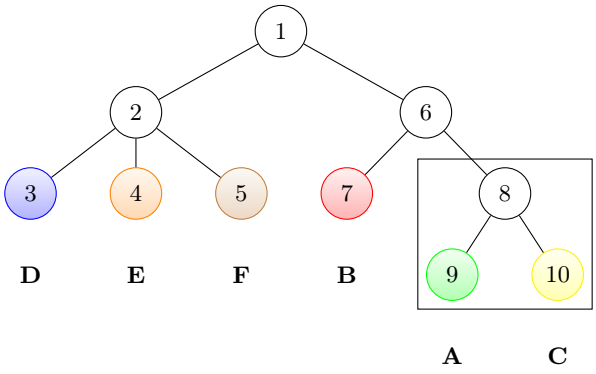
Our approach consists in encoding each tree using balanced parentheses and represent it as a bit vector. By representing each tree as a bit vector of size $2n$ bits, where n is the number of nodes present in the tree, we can achieve a very compact representation that eliminates the need to store additional information.

3.1 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 descendants must be after the opening parentheses and before the closing parentheses that represents them. For example, in Figure 2, the opening and closing parentheses that represent the third node are in positions 3 and 8, respectively. Then, we can



$$B = (\, (\, (\,) \, (\,) \, (\,) \,) \, (\,) \, (\, (\,) \, (\,) \,) \,) \,)$$

Fig. 3: Tree T_2 and its balanced parenthesis representation.

conclude that the parentheses that represents their descendants (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 2 and 3. Throughout this document we will refer to an index of a node as i and to a position in a bit vector as v.

3.2 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, we added operations preOrderMap, preOrderSelect, postOrderSelect, isLeaf, lca, clusterSize, and numLeaves that mostly use the fundamental operations just mentioned before. In Table 1 it is possible to see the list of operations as well as their meaning and their run time complexity; see the text by Navarro [10] for details on primitive operations.

3.3 Robinson Foulds distance computation

To compute the Robinson Foulds distance, our algorithm receives as input two bit vectors that represent the two trees being compared, as well as a vector of integers that does the correlation between taxa and indexation between the two trees. This vector has size n, and 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 stored in the second tree given the index of where it is stored in the first tree. An example of this vector for the trees in Figures 2 and 3 is

$$[\, 0\; 0\; 0\; 9\; 7\; 10\; 0\; 3\; 4\; 5\,]. \tag{1}$$

Table 1: Operations to manipulate trees (see [10] for details).

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, i)	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 an occurrence i, returns the position where the ith one is present.	$O(1)$
Select0(bv, i)	Given a bit vector bv and an 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 size of the cluster where v is the root.	$O(\log(n))$
NumLeaves(bv, v)	Given a bit vector bv and a position v, returns the number of leaves that are below v.	$O(\log(n))$

Algorithm 1 Treediff Robinson Foulds

```
1: Input: v1, v2
2: equalClusters ← 0
3: for i ← 1 to N do
4:     p ← PostOrderSelect(v1, i)
5:     if IsLeaf(p)) then
6:         lca ← NodeSelect(Code[NodeMap(v1, p) - 1] + 1)
7:         size ← 1
8:         push(< lca, size >, s)
9:     else
10:        cs ← ClusterSize(v1, p)
11:        while cs ≠ 0 do
12:            < lca, size >← pop(s)
13:            lcas ← Lca(v2, lca,)
14:            if lcas = null then
15:                < lcas, size >← pop(s)
16:            else
17:                < lca, size >← pop(s)
18:                lcas ← Lca(v2, lcas, lca)
19:            end if
20:            cs = cs − size
21:        end while
22:        if NumLeaves(v1, p) = NumLeaves(v2, lcas) then
23:            equalClusters ← equalClusters + 1
24:        end if
25:        push(< lcas, ClusterSize(v1, p) + 1 >, s)
26:        lcas ← null
27:    end if
28: end for
29: distance ← (numInternalNodesv1 + numInternalNodesv2 - equalClusters*2) / 2
30: return distance
```

Then the algorithm counts the number of clusters present in both trees. This is achieved by going through the first tree in a post-order traversal and verifying, for all clusters, if they are present in the second tree. To verify if a cluster from the first tree is present in the second tree, the key idea is to obtain the cluster from the second tree that is described by the lowest common ancestor between all the taxa present in that cluster. Then if the number of taxa present in the cluster that was obtained is equal, we can guarantee that the cluster is present in both trees.

Then, knowing the number of clusters from both trees and the number of clusters that are present in both trees, we can conclude how many clusters are not common to both trees, therefore knowing the Robinson Foulds distance. This process can be seen with more detail in Algorithm 1.

Given that the algorithm goes through each node in the first tree and needs to compute the lca, the post order index and the number of leaves from each

cluster, the theoretical complexity of the algorithm is $O(n \log(n))$, with n the number of nodes in each tree.

3.4 Fully labelled and weighted trees

Our algorithm can also compute the Robinson Foulds distance for trees with taxa in internal nodes and with weights on edges. For internal labelled nodes, the approach is very similar to the previous one. The only difference is that it is also necessary to compute the lca for the taxa inside the internal nodes. Then, to compare if the clusters are the same, instead of comparing the number of leaves, we compare if the size of the clusters are the same.

Computing the weighted Robinson Foulds metric can be done by storing the total sum of the weights of both trees. Then, we verify if each cluster is present in the second tree; if that is the case we subtract the weights of the cluster in both trees and add the absolute difference between the two of them. 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.

The theoretical complexity remains $O(n \log(n))$ in both cases. We only add the lca operation also for internal nodes when these are labelled; and the cost is exactly the same for weighted trees.

3.5 Information about the clusters

Applying the algorithms described above, it is also possible to store the clusters that are detected in both trees. This can be done through adding to a vector the indexes of the cluster in the first and second tree when concluding that they are the same. Doing this allows us to confirm if the distance returned by the algorithm is correct and if the difference between the trees is well represented by the distance computed.

4 Implementation

Our algorithm is implemented in C++ and it is divided in two phases. The first phase of the algorithm 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. The second phase of the algorithm is to compute the Robinson Foulds distance. To represent the bit vectors we used the Succinct Data Structure Library (SDSL) [7] which contains three implementations to compute the fundamental operations mentioned in Section 3.2. The one that we chose was the `bp_support_sada.hpp` since it was the one that obtained the best results in terms of time and space requirements.

In the first 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 we need to take into account is that when we encounter a leaf, we need to add two

Algorithm 2 Create bit vector

```
1: Input: T                                          ▷ Tree in newick format
2: bv ← null
3: while c ≠; do
4:     c ← next string
5:     if c == ( then
6:         bv.push_back(1)
7:     else if c ==, then
8:         continue
9:     else if c ==) then
10:        bv.push_back(0)
11:    else
12:        bv.push_back(1)
13:        bv.push_back(0)
14:    end if
15: end while
16: return bv
```

parentheses, the first one open and the second one closed (see Algorithm 2). The vector exemplified in Equation 1 is also built in the first phase. For this process, we need to have a temporary hash table that associates the labels found in the Newick representation to the indexes of the first tree. Then when the second tree is parsed, whenever a label is found, the corresponding value x is stored in the vector position v, where v is the value stored in the hash table and x is the corresponding index in the first tree. The first phase takes linear time with respect to the number of labelled nodes.

The second phase is an implementation of Algorithm 1, and we extended the functionality of SDSL for that purpose. The library `bp_support_sada.hpp` provided already implementations for the operations rank1, select1, findOpen, findClose, enclose and rmq operations. For our algorithm, we added 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. We also implemented other operations, most of them using the ones already implemented. Their implementation can be seen in Algorithm 3.

We also implemented the Day algorithm, so that we could compare it with our implementation in terms of running time and memory usage. This algorithm was also implemented in C++ and stores two vectors of integers with size n for each tree and two vectors of integers with size $n/2$ (the number of leaves). The complexity of the algorithm is $O(n)$ with respect to both time and space.

5 Experimental evaluation and discussion

To evaluate our implementation we used the ETE Python library to create a random generator of trees in Newick format. We created 5 pairs of trees for

Algorithm 3 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 ↔ 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
```

each size, namely trees having 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000 and 100000 leaves.

First, we analysed the memory used throughout the algorithm for a tree that contains 100000 leaves. Figure 4 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 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. Note that by serializing and storing the trees as their bit vector representations, we can avoid the memory required for parsing the Newick format.

Secondly, we compared the memory peak usage during the second phase of our algorithm with that of Day algorithm implementation. Figure 5 shows the comparison between both algorithms in terms of their memory peak usage. Our algorithm exhibits a lower memory peak compared to Day algorithm.

Then, we compared the running time between the two algorithms for the first and second phases: Figure 6. We observe that the time differences for the first phase are not significant, indicating similar performances between the two algorithms. In the second phase, we notice an interesting trend for our algorithm: while the theoretical complexity is $O(n \log(n))$, the practical complexity seems to be $O(n)$. This finding suggests that our algorithm effectively manages the computational complexity associated with a growing number of leaves.

As expected, our implementation is slower than the Day algorithm implementation, because we have the overhead of using a succinct representation for trees. But it is noteworthy that our implementation offers a significant advantage in

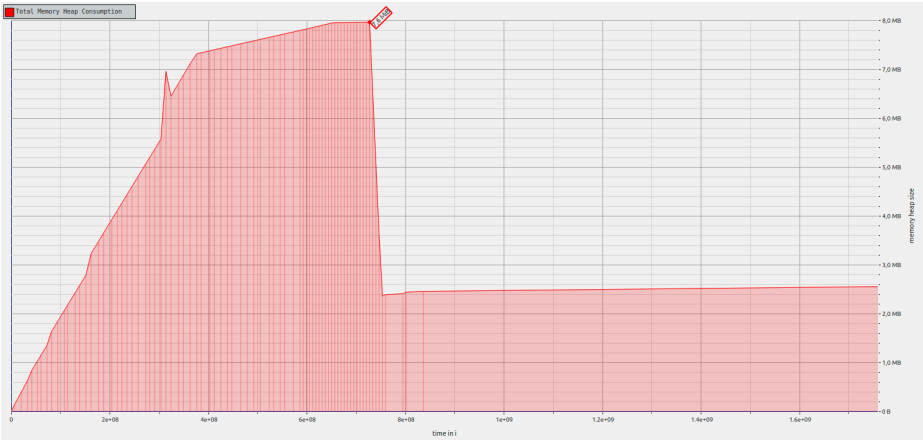


Fig. 4: Heap allocation profile for two trees with 100000 leaves.

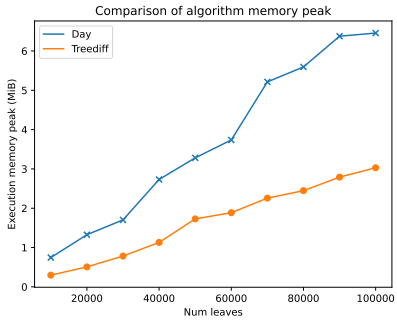


Fig. 5: Memory peak of the second phase for trees with different sizes.

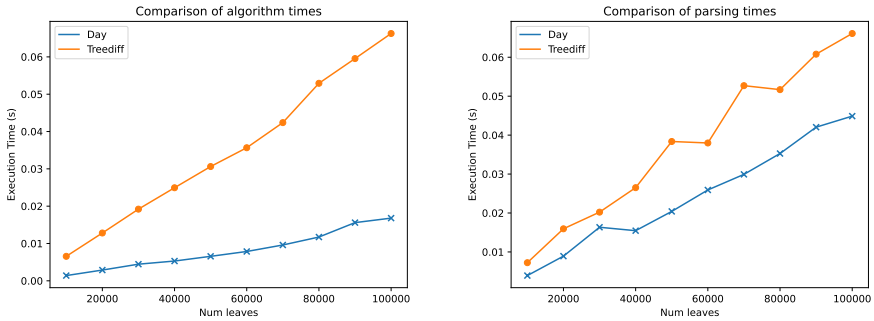
terms of memory usage, namely when trees are succinctly serialized and stored. This trade-off between memory usage and running time is common in this setting, with succinct data structures being of practical interest when dealing with large data.

Acknowledgements

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references DSAIPA/DS/0118/2020, UIDB/50021/2020 and LA/P/0078/2020.

References

1. Allen, B.L., Steel, M.: Subtree transfer operations and their induced metrics on evolutionary trees. Annals of combinatorics **5**, 1–15 (2001)



(a) Second phase (b) First phase

Fig. 6: Run time for trees with different sizes.

2. Bordewich, M., Semple, C.: On the computational complexity of the rooted subtree prune and regraft distance. Annals of combinatorics **8**, 409–423 (2005)
3. Briand, S., Dessimoz, C., El-Mabrouk, N., Nevers, Y.: A linear time solution to the labeled robinson–foulds distance problem. Systematic Biology **71**(6), 1391–1403 (2022)
4. Day, W.H.: Optimal algorithms for comparing trees with labeled leaves. Journal of classification **2**(1), 7–28 (1985)
5. Felsenstein, J., Felenstein, J.: Inferring phylogenies, vol. 2. Sinauer associates Sunderland, MA (2004)
6. Francisco, A.P., Bugalho, M., Ramirez, M., Carriço, J.A.: Global optimal eburst analysis of multilocus typing data using a graphic matroid approach. BMC bioinformatics **10**(1), 1–15 (2009)
7. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: 13th International Symposium on Experimental Algorithms, (SEA 2014). pp. 326–337 (2014)
8. Kuhner, M.K., Yamato, J.: Practical performance of tree comparison metrics. Systematic Biology **64**(2), 205–214 (2015)
9. Li, M., Zhang, L.: Twist–rotation transformations of binary trees and arithmetic expressions. Journal of Algorithms **32**(2), 155–166 (1999)
10. Navarro, G.: Compact data structures: A practical approach. Cambridge University Press (2016)
11. Pattengale, N.D., Gottlieb, E.J., Moret, B.M.: Efficiently computing the robinson-foulds metric. Journal of computational biology **14**(6), 724–735 (2007)
12. Robinson, D.F., Foulds, L.R.: Comparison of phylogenetic trees. Mathematical Biosciences **53**(1-2), 131–147 (1981)
13. Robinson, D.F., Foulds, L.R.: Comparison of weighted labelled trees. In: Combinatorial Mathematics VI: Proceedings of the Sixth Australian Conference on Combinatorial Mathematics, Armidale, Australia, August 1978. pp. 119–126. Springer (2006)