Universitat de les Illes Balears

MASTER'S THESIS

OPTIMISATION AND IMPLEMENTATION OF ALGORITHMS ON PHYLOGENETIC NETWORKS

Narcís Rosselló Payeras

Master's Degree in Intelligent Systems (MUSI)

Specialisation: Artificial Intelligence and Data Science

Centre for Postgraduate Studies

Academic Year 2021-22

Tutor: Joan Carles Pons Mayol

OPTIMISATION AND IMPLEMENTATION OF ALGORITHMS ON PHYLOGENETIC NETWORKS

Narcís Rosselló Payeras

Master's Thesis
Centre for Postgraduate Studies
University of the Balearic Islands

Academic Year 2021-22

Key words:
Phylogenetic Networks, Tree-child Networks, Reconstruction, Algorithms.

Thesis Supervisor's Name: Joan Carles Pons Mayol

Optimisation and Implementation of Algorithms on Phylogenetic Networks

<Narcís Rosselló Payeras>
Tutor: <Joan Carles Pons Mayol>
End of Master's thesis in Intelligent Systems (MUSI)
University of the Balearic Islands
07122 Palma, Illes Balears, Espanya
<narcisrossello@gmail.com>

***Abstract*—In this paper we develop and implement two pseudo-code approaches to two models for the reconstruction of tree-child phylogenetic networks described in [13] and [3]. In order to feed and test these algorithms, a system for extracting the information necessary for the operation of the algorithm from the original networks is also developed. During the manuscript, the implemented functions are described, as well as different examples to see and understand how each of these approaches works internally. The code has been developed in Python through notebooks, and can be downloaded at the following link: https://github.com/narcisrossello/TreeChildReconstruction.**

***Index Terms*—Phylogenetic networks, tree-child networks, reconstruction, algorithms.**

I. Motivation

The classical way to represent the evolution of species or genes has been using the mathematical model of a phylogenetic tree, a rooted directed acyclic graph with the leaves labelled by the taxa under study. However, it has been seen that this model has some deficiencies when it comes to representing some events that occur in nature, such as horizontal gene transfer, recombination or hybridisation. For such cases, there are nodes that need to have more than one parent modelling the action of more than one entity involved in the generation of the new one, transforming the tree into a network [11].

One of the main issues about phylogenetic networks is how to reconstruct them from the available data, be it sequences, distances between sequences, trees, triplets, quartets, splits, clusters or even networks. Every possibility of input present can lead to a type of path or pathway from which a more or less efficient type of reconstruction can be made. This same problem can be derived to others, such as network visualisation, simulation, comparison, consensus and so on [1].

Based on this situation, different solutions have been found to the problems encountered. Some of these have been theoretical solutions, developing a mathematical and formal solution to the problem described. Other solutions have been closer to practice, resulting in solutions with possible computer implementations with polynomial costs, while many others are encompassed by NP-complete solutions [1]. In order to attack this problem, different implementations can be found in the literature and in computer libraries that solve some problems of treatment and management of the structures treated in a general way and others that try to give a solution to more specific problems. One of the main problems is not finding a common standardisation for these, as each one uses the informatic resources or programming languages closest to its environment, being in some cases outdated languages or difficult to use and access.

The main objective of this project is to develop an algorithm that solves a specific problem, in this case that of reconstruction. In this project, tree-child networks will be reconstructed from Maximum Lower-Level Subnetworks (MLLS) and from a distance matrix. In addition, with a vision closer to engineering, a solution has been given that optimises resources in the most appropriate way, as there is no point in having a solution if it is more costly than the problem itself. In addition, it is intended to give a forward-looking approach, for this reason, a programming language such as Python is used, which is one of the most widely used in programming today, and it is also essential that the access to the development is as transparent and simple as possible. For this purpose, the notebooks of this programming language fulfil this purpose appropriately. Finally, it is a clear objective to make maximum use of libraries and implementations that have already been implemented. In this case, we have chosen to make use of the network processing and manipulation library `networkx`, which is widely used in the world of networks. The use of such a widely used library allows greater ease of access and understanding of the project, as well as greater ease of connection with other types of formats such as eNewick or connection with other developments in possible future work.

The rest of the manuscript is organised as follows. Firstly, a brief overview and description of phylogenetic networks and more specifically tree-child networks is given in Section II. In addition, it is shown what kind of implementations and programmes can be found to attack from general to specific problems. Then, in Section III a common view of the two implementations is given, explaining the common structure developed and the technical procedure followed in the implementation. Furthermore, there are the explanatory points of the actual development of the two algorithms in Sections IV and V, explaining briefly the idea behind the one described in the original article and how this has been translated into a real and functional implementation. It also presents an explanation of real cases of networks reconstructed with the different algorithms, as well as a closer look to

the data processed to understand what happens and what is done internally in each of the algorithms. Moreover, in Section VI an observation is made of the different points in common, particularities and differences that each of the previously described and analysed algorithms have. Finally, some more personal conclusions and a possible view of future work are given in final Section VII.

II. Phylogenetic Networks

Phylogenetic networks are the main tool for describing and visualising the evolutionary and genetic relationships of some groups of species. This type of network is gaining in popularity as it allows for greater ease in describing reticulate events, which is not possible with phylogenetic trees. Over the years, the excessive simplicity of trees has led biologists and mathematicians to use networks [12].

In the transformation to networks, the leaves are labelled nodes representing species, genes or individuals, while the root of the network represents a common ancestor among them all. The internal nodes of the network with two or more outgoing connections represent those divergent events that have taken place, seeing a single lineage split into two or more lineages, while the internal nodes with one or more incoming and one outgoing connection represent the convergence of two or more lineages into one. The latter are known as reticulation evolutionary events, which include hybridisation, introgression and horizontal gene transfer. These nodes are known as reticulate nodes or network reticulations. It should be noted that a network without reticulations is a phylogenetic tree. In this way, it can be understood that phylogenetic networks are a more general model of representation and treatment of the evolution of species than the model offered by phylogenetic trees [10][15].

It is important to know the origin and evolution of phylogenetic networks. But it is also important to know the model, the restrictions, the scope and the possibilities that they can offer us. In this way, a formal definition extracted from [11] will be given in order to have a definitive idea of what they are and what they offer.

Definition II.1 (Rooted binary phylogenetic network)**.** A rooted binary phylogenetic network $N = (V, E)$ on X with root p is a rooted directed acyclic graph with no parallel arcs satisfying the following properties:

(i) the (unique) root p has in-degree zero and out-degree two;

(ii) a vertex with out-degree zero has in-degree one, and the set of vertices with out-degree zero is identified with X;

(iii) all other vertices have either in-degree one and out-degree two, or in-degree two and out-degree one.

Even if definition II.1 is taken as a way of understanding and specifying phylogenetic networks, it is still a definition that encompasses a large number of possibilities. When general definitions have to be translated into the computer world, this can sometimes become very costly works and processes. For this reason, from this starting point, different adaptations arise trying to facilitate the development and understanding of some specific problems. Some of these derivations are the following subclasses [11]: Time-consistent, Tree-child, Tree-sibling, Reticulation-visible, Genetically stable, Stack-free, Normal, Regular, Orchard, Tree-based or LGT networks, just to name a few.

A. Tree-child Networks

The class of tree-child networks is one of the most widely used classes of phylogenetic networks. Those were introduced to fit a complex biological reality into a computationally tractable environment. Biologically, these are networks where every non-existing species has some descendant by mutation. Mathematically, every tree node has at least one child that is a also a tree node [6][5].

Even so, tree-child networks are one of the most permissive types of phylogenetic networks and they are capable of modelling quite a few important scenarios, and those where perfect modelling is not possible, they do so with a fairly good approximation [5].

There are already a number of articles that solve problems and issues for this type of networks. Each of these provides a different solution, each with its own particularities, advantages and disadvantages. This is also the case and it is analysed in the algorithms developed in this project. In order to have some points of reference, we can find a first example of reconstruction from the trinets encoded in the tree-child network [17]. We can see another approach of reconstruction and comparison from their path multiplicity vectors, given as a practical solution developed in Java and Perl programming languages [7]. As a last example, a reconstruction by path-length distances between taxa can be found [2]. The idea behind this last example is similar to the one developed in the second algorithm (see Section V), which raises some particularly characteristic and relevant key points in the performance of the algorithm.

III. Procedure and Structure

In both developments made of `MLLS_Algorithm` and `Q_Algorithm` an internal structure is kept as similar as possible. The two relevant notebooks have their own differences which will be described in their respective points, but in general they follow a structure where each part is clearly defined.

The common diagram for both implementations shows where the different defined modules are fed from (see Figure 1). These modules are equivalent to function grouping layers, with the L being the Low-level Definitions layer, the M for Medium-level and the H for High-level. The extraction module stores those functions for obtaining the algorithm's input from original networks. Finally, the main module is the main code blocks that directly store the described and implemented algorithms. The names used in the different modules for all the different functions try to maintain an explanatory or intuitive meaning of the function performed. In addition, in the defined notebooks they maintain a minimal explanation of what each function performs, as well as what kind of output they have. Functions maintain a PascalCase format,

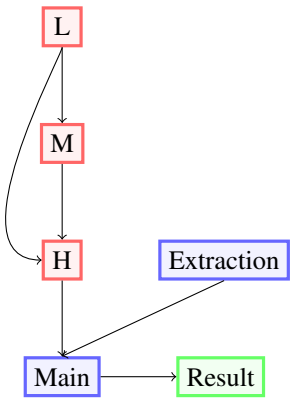

Figure 1. Implementation Diagram Scheme.

while variables follow the camelCase definition format. The different notebooks can be found at the following addresses:

- `MLLS_Algorithm`: https://bit.ly/3nLEm46.
- `Q_Algorithm`: https://bit.ly/3fwbo3K.

The different notebooks are written with Python, because of the usefulness of development on the available notebooks, as well as the possibility of using ready-made libraries like `networkx` [9] directly on the notebook. In both implementations, polynomial-time reconstruction algorithms are provided, depending on the size of the input networks [13].

IV. Reconstruction from reticulate-edge-deleted subnetworks

A. Paper

In [13] it is presented a way to reconstruct tree-child phylogenetic networks from their Maximum Lower-Level Subnetworks (MLLSs). These are extractions made from an original network. The reticular nodes, those nodes that have an input degree 2, play a major role in this process. Each MLLS is extracted by removing one of the two input edges from one of the reticulated nodes of each level-k biconnected component of the network. Looking at this, the higher the number of reticulated nodes, the higher the number of MLLS extractions. However, not all extractions are valid. It has to be fulfilled that by removing one of the parent edges of the reticulated node, there must be exactly two nodes and three edges less in the resulting network compared to the original one.

Knowing this, the algorithm is able to detect these reticulated nodes by comparing and contrasting the possible reticulated node deletions with the set of MLLSs it receives as input. This algorithm provides a unique way of reconstructing networks from their reticulate-edge-deleted subnetworks. Specifically, the analysed networks are tree-child networks and level-k networks. This fact is derived from the demonstration that tree-child networks contain either a Cherry (also under the name of Λ in the Figure 2) or a reticulated Cherry (also under the name of the H used in the algorithm but also found in the shapes K and A in the Figure 2) connection type. Efforts are put into reconstructing this second type H and it is shown how they can only be reconstructed from an exhaustive study of the case [13].

The blob trees explored in the article are important in this process. These represent labelled trees obtained from the input networks by collapsing each biconnected component into a single node. These components are derived from nodes (also called top nodes of the biconnnected component) where their removal causes the disconnection of the network. This is seen in the different examples analysed in this section, as in Figure 9. By way of explanation, it can be seen how in this Figure 9 at Part 2 the focus is on the biconnected component derived from node 19 and this is collapsed or grouped in Part 3 of the same Figure. Finally, the different types of inter-leaf shapes in Figure 2 is essential when making the comparison between MLLS explained. By being able to have a number of specific shapes at the same point, the algorithm is able to ensure that at that point there has been a change from the original. It is thanks to this that the different missing reticulated nodes and the deleted edges are reconstructed.

B. Implementation Structure

In the developed notebook *TCM-LLS_Reconstruction_Algorithm.ipynb*, the idea described above is implemented, developing what the author proposes in the form of pseudo-code. As an introduction, the notebook is structured in different parts, which are made respecting as much as possible the initial idea of the author, but always trying to give as much efficiency as possible to the development. In this way, we can find the different points:

- **Initialization and variables declaration**: In this point the libraries used are imported and the constants used in the implementation are defined. The main library as well as the main data base is `networkx`. This is used to configure the networks that form part of the algorithm's input. In addition, its own methods are used, as well as its own specific implementations on the same network models.
- **Function Declaration**: This is where the methods involved in the operation of the main algorithm are defined. They are divided into layers according to their level of abstraction. See subsection IV-C.
- **Main algorithm**: At this point the reconstruction algorithm is developed, maintaining a structure as similar as possible to the one described in the article. See subsection IV-D.
- **Mlls extraction from original network**: In this section we work on obtaining an algorithm that performs the extraction of mlls from an original network. See subsection IV-E.
- **First Example**: In this point a first simple example is made to see how the algorithm works. In this example recursion is not necessary. See subsection IV-F1.
- **Second Example**: At this point a second, more complete example is carried out to see how the algorithm works. In addition, the extraction of mlls from a more complex network is also used. Recursion is needed to solve this problem. See subsection IV-F2.

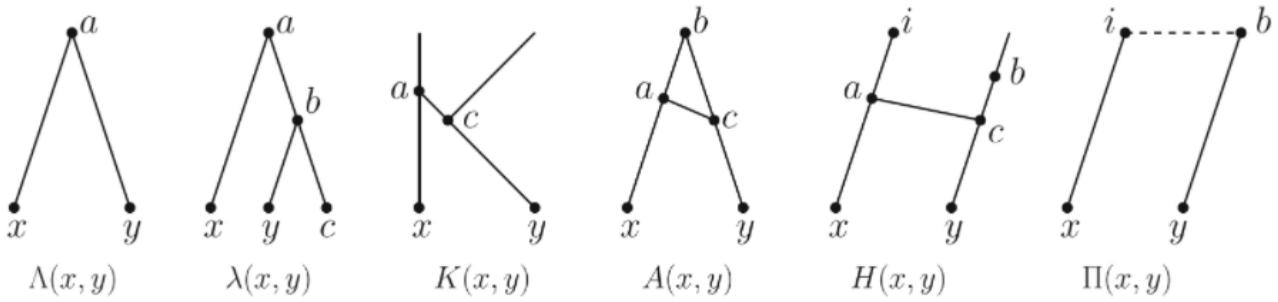

Figure 2. All possible shapes on two leaves $\{x, y\}$ in tree-child networks [13].

The main structures used are the following:

1) Networks with `networkx`. The networks provided by the public `networkx` library are used. These networks have a series of formal restrictions regarding their declaration. The root will be the node with value 0. Leaf nodes will have alpha-numeric values, while internal nodes will have positive numeric values. This point is critical as it conditions the good functioning of the algorithm from this library with the possibilities it attributes. With these restrictions, it is possible to formalise tree-child network structures.
2) Blob-tree in the form of a matrix of lists of lists. As a noticeable modification of the article definition, the blob-tree as such is not formalised in this case. Instead, lists of lists are formed. Each list is equivalent to a node, which has collapsed all leaf nodes. The list of lists is equivalent to the different nodes that form a blob tree. Finally, the array of list of lists equals the different blob-trees of all participating MLLSs.
3) Pointer to the top node. As an implementation optimisation, when forming the different blob-trees, the top nodes of the blob-trees from which they are formed are stored. This is to avoid having to traverse parts of the blob-trees again, which would be computationally expensive. In this way, saving the pointer can lead to significant savings in complexity in relatively large networks.

C. Methods

In this section the different methods involved in the main reconstruction algorithm are declared. The methods are divided into three main groups:

1) Low-Level Functions: In this group are the functions that are closest to data manipulation. They have very basic and concrete objectives. These include obtaining the internal nodes of the network in the form of a list, obtaining the network without leaves, obtaining the direct parent or parents of a node, obtaining the top node of a group of nodes or the comparison between foundation nodes. In these, the programming is clear and straightforward, so that the possibilities of input as well as output variables are kept to a minimum.

In addition to the functions related to the processing of the basic structures, this group includes the functions that identify the different shapes available between two leaves, which are described in the Figure 2. We will briefly describe how each of them works:

- `LambdaFU` (Lambda Function Upper-case): This function is equivalent to the type of connection between two leaves described by the letter Λ. In this function, the aim is to check whether the single parent of leaf x is equal to the single parent of leaf y. If so, the function will check whether the single parent of leaf x is equal to the single parent of leaf y. Thus, if so, it can only be the case for this type of connection.
- `LambdaFL` (Lambda Function Lower-case): This function is equivalent to the type of connection between two leaves described by the letter λ. In this function, the aim is to check whether the grandfather of leaf y (original notation with letter a) is equal to the parent of leaf x (a). In this way, the algorithm checks for all connections of the parent of y (b), whether there is a node that is equal to the parent of x. In addition, it checks that at node b there are exactly two leaves connected, which correspond to y and c. This procedure is unique, i.e., if you want to check the reverse case of the leaf order, you must call the function with the relevant changes in the leaf order.
- `H`: This function is equivalent to the type of connection between two leaves that the name of the method itself indicates. In this function, what is first examined is whether c is a reticulated node. If so, it is examined whether c has a parent connected node which is the same as the parent of a.
- `Pi`: This function is equivalent to the type of connection between two leaves described by the letter Π. This function checks only if the path distance between the two leaves is greater than or equal to 4.

2) Medium-Level Functions: In this group are declared those functions that need further development and with broader objectives than those of the first group. Even so, they are not found in the final declarations of the algorithm, so they are in this intermediate state. To better understand how all the different steps work and how they have been reached, the main ideas and functionalities of the different declared methods will be defined. The auxiliary functions in this group that do not provide further information are described only in the notebook.

- `BiconnectedNodes`.

Data: A set $\mathcal{T} = \mathcal{N}^{mlls}(N)$ for some level-k tree-child network N, where $k \geq 2$
Result: The network N

1 Update $\mathcal{T}$ by collapsing maximal common pendant subnetworks from every network in $\mathcal{T}$;
2 Find the blob tree for each network in $\mathcal{T}$;
3 Find a minimal set A that is a node of the blob tree of each network in $\mathcal{T}$;
4 Find a leaf pair $\{x, y\}$ where $x, y \in A$ such that distinct networks N_1, N_2, N_3 of $\mathcal{T}$ contain $\Lambda(x, y)$, $H(x, y)$, and one of $\lambda(x, y)$, $\lambda(y, x)$, or $\Pi(x, y)$, respectively;
5 Update N_3 by adding nodes a, c directly above x, y, respectively, and an edge (a, c);
6 Let N_A denote the pendant subnetwork rooted at the top node of this blob in N_3;
7 **for** $N^{mlls} \in \mathcal{T}$ **do**
8 Find the pure node p with leaf-descendant set A;
9 Replace the pendant subnetwork rooted at p by N_A;
10 Collapse N_A from N^{mlls};
11 **end**
12 **if** $\mathcal{T}$ *contains a single element* T **then**
13 $N' := T$;
14 **else**
15 $N' :=$ TCMLLS- RECONSTRUCTION($\mathcal{T}$);
16 **end**
17 Construct N from N' by appending the maximal common pendant subnetworks we have collapsed;
18 **return** N

Algorithm 1: Algorithm TCMLLS- RECONSTRUCTION($\mathcal{T}$)

```
def TCMLLS_RECONSTRUCTION(T, depth = 0):
  #0
  if len(T) == 0:
    return T
  #0.5
  if len(T) == 1:
    return T

  #1
  t, namesNewNodes = CollapsingMaximalCP(T)
  #2 - topNodes is optimization
  BT, topNodes = FindBlobTreeWithTopNodes(t)
  #3
  A, topNodeA = FindMinimalFoundationNodesWithTopNode(BT, topNodes)
  #4
  leaf1, leaf2, N3 = Check3Types(t, A)
  #5
  if N3 != -1 and leaf1 != -1 and leaf2 != -1:
    AddNodeAC(t[N3], leaf1, leaf2, depth)
    #6
    NA = GetNA(t[N3], topNodeA[N3])
    tprima = []
    for n, nmlls in enumerate(t):
      #8
      p = GetPureNode(nmlls, topNodeA[n])
      #9
      t[n] = ReplacePureNode(nmlls, topNodeA[n], p, NA)
      #10
      tprima.append(CollapseNA(t[n], p, topNodeA[n], NA, A))
  #11
  # Isomorphic reduction
  areIsomorphics = True
  while areIsomorphics:
    areIsomorphics = ThereAreIsomorphic(tprima)
  #12
  if len(tprima) == 1:
    Nprima = t[0]
  #14
  else:
    Nprima = TCMLLS_RECONSTRUCTION(tprima, depth + 1)
    return Nprima
    #15
    Nprima = UnCollapseNA(Nprima, topNodeA[N3], GetNodeA(A), NA)
  #17
  if len(namesNewNodes) == 0:
    return Nprima
  #18
  return UnCollapse(Nprima, namesNewNodes, depth)
```

Figure 3. MLLS_Reconstruction_Algorithm: Comparison between original algorithm (left side) and implemented one (right side).

- `IsBiconnected`.
- `VisitedLeaves`: This function is used to detect all those leaves that can be visited from a set of specific nodes. For each of these, all connections are visited. If it is a leaf, it is added to the final set. All those internal nodes not visited will be explored recursively in this same method.
- `GetLeaves`.
- `GetMax`.
- `GetSubNetworksWithTopNodes` and `GetSubNetworks`: These two functions have a very similar construction. They oversee calculating, from the list of biconnected nodes, all those subnetworks that are in a network. Each of these is obtained from the nodes that are not in the list of bi-connected nodes of the network. In this way, for those nodes that the elimination of one of their connections means the disconnection of the entire network, it means that they are a principle for one of these sub-networks. The difference between these methods is whether the node from which the subnetwork was removed, which is the top node of the subnetwork, is saved.
- `FindBlobTreeWithTopNodes` and `FindBlobTree`: These two functions have the same differences as the two previous ones. By saving or not the top node, they oversee generating for all the input networks a list of lists of nodes that are equivalent to the blob-trees.
- `GetNA`.
- `GetNANodes`.
- `GetP`.
- `GetPNodes`.
- `DeleteIntersectionNodes`: This function is used to solve an error in the construction of the network from the library used in `networkx`. The problem is that, although two subnets share the same leaves, they do not have to share the same internal nodes. Thus, when nodes are removed from the NA set, if there is a node that is not in this set, it is left behind in this network. As an example, if NA is a set formed by $\{1, 2\}$ and the NA' has the set formed by $\{1, 2, 3\}$, although they relate to the same leaves, when only 1 and 2 are eliminated, node 3 will remain in the network, but without connections, so it will cause errors. In this way, this function examines those nodes that are not in NA, but in NA' and eliminates them for these cases.
- `ReplacePureNode`.
- `FindFoundationNodes`.
- `ReduceLeafFN`.

3) High-Level Functions: In this group are declared the functions directed called from the algorithm as can be seen in the Figure 3.

- `UnCollapse`: This function un-collapses what has been collapsed during the algorithm. Specifically, here it un-

collapses the node created, which has as a compound name the following format: $\{leaf1, leaf2\}$. In this way, from this name it is possible to extract those initial leaves/nodes which have been collapsed. As can be seen in Figure 7, the created node $\{v, w\}$ is decomposed in 1 internal node and 2 leaves: $r0$, v and w. Node r will remain in the centre connecting with the parent of the previous node and with the two created leaves.

- `UnCollapseNA`: In a similar way to the previous un-collapse, this function un-collapses the node with the foundation node of A. In this way, the place where the node A was located is replaced by the NA network that existed before the collapse.
- `AddNodeAC`: This function adds two new nodes a and c in the network. These nodes have marked in their names the level of depth at which they have been created. In addition, a new edge is created between these two created nodes.
- `CheckOneEach`: This function checks that for the different types of inter-leaf connections required, there are a $\Lambda(x, y)$, an $H(x, y)$ and a $\lambda(x, y)$ or $\lambda(y, x)$ or $\Pi(x, y)$. Store the index of the last check.
- `CollapsingCP`: This function is composed of different parts. First, it checks all the blobs trees in the network. Secondly, it looks for the foundation nodes of these networks. Finally, for the smallest foundation node found, it collapses it by creating a new node.
- `FindMinimalFoundationNodes` and `FindMinimalFoundationNodesWithTopNode`: These two functions have a similar construction and purpose. They oversee searching for the minimum set of available foundation nodes. The difference is that in the second function the top node of the chosen foundation node is saved, using in this case the dedicated functions.
- `Check3Types`: This function checks for the three types of inter-leaves connections required. These checks are carried out for all possible combinations between the leaves. The order of the combination is decisive, as it is not the same to combine (x, y) as (y, x).
- `CollapseNA`: This function collapses the NA subnetwork within the network into a single node.
- `GetNodeA`: This function is an auxiliary function to generate the name of the new node A from the foundation node A.
- `ThereAreIsomorphic`: This function indicates whether two nets are isomorphic. If so, it deletes the second of these two. This check is done for all nets passed, but the function terminates when any or none of the nets are deleted.

D. Main Algorithm

The main process has been called in the same way as in the article: `TCMLLS_RECONSTRUCTION` (Figure 3). In addition to this, we have tried to respect as much as possible the different points that appear in this one, replicating the functionalities that are intended in each one. In this way, it is possible to compare and link each point with the one established in the development. In the same way, the different points dealt with are explained with comments on their equivalences. As a final consideration, the nomenclature used tries to give a maximum representation of the objective of each line.

The different points will be broken down and the different particularities or other concepts to be considered when understanding how it works will be explained. The first point to analyse is the inputs. As can be seen, there is the same input T as in the article, consisting of a list of MLLS from which the original network is to be reconstructed. In addition, an additional depth variable can be found. This variable, with a default value equal to 0, allows to be aware and place the execution in a certain depth, giving it the ability to generate an environment of variable names and other messages according to its iteration layer. In this way, it helps in the detection of errors by locating each part and being able to know at what depth each node of the final network has been reconstructed.

- **Point 0**: This first point is not covered in the article. In any algorithm there must be a minimum error control, so as it is not explicit in the article, it must also be included almost obligatorily. In this point, a minimum control is simply made, such as that the set of MLLS given as input is empty or not. If so, an error message is given, and the execution is terminated by returning a null value. It should be noted that this effect also applies in recursion.
- **Point 0.5**: As an additional point to the previous one, when having an input with only one network, it is not going to be possible to work on it, so the same variable T is going to be returned.
- **Point 1**: As the first point present in the algorithm of the article, as its name indicates, it oversees collapsing those maximal common pendant subnetworks for each one of the networks in T. In this way, for all those cherries present in the network, they are going to be collapsed in a single node. In this way, in case of any modification, all the names of the new nodes are returned in the form of a list. Likewise, the networks are returned as a t-variable, whether they have been modified or not. This preserves the original networks without affecting their possible dependencies inside or outside the algorithm.
- **Point 2**: In this second point, the different blob-trees are constructed. These are not going to be networks as such, but will have the form of a matrix of the different nodes present in each blob-tree. In addition, the different top nodes of each of these are extracted, so that they can be used in subsequent lines.
- **Point 3**: In this third point the minimal foundation node is obtained from the previous list of blob-trees. In addition, from the selected list, the relevant top node will be chosen, so that the different collapses can be assembled from it. The name of the original variable is considered, being this subnetwork with the foundation node called A, as well as its respective $topNodeA$ node.
- **Point 4**: In this fourth point, the existence of a pair of leaves $\{x, y\}$ belonging to the set A that meet the requirements is checked. It must be considered that if there are 3 networks that meet the three casuistry it

is enough. If an available combination is found, it is returned in two separate variables, as well as the index of network N3, which is going to be modified in the following point. If any of the cases is not fulfilled, -1 is returned.

- **Point 5**: As a fifth point, in case that some modification must be produced in the network N3 and all the variables to use are different to -1, the points from 5 to 10 are going to be executed. Otherwise, we will jump directly to point 11. When entering the if, the first thing to do is to add the new nodes a and c, with their respective connections in the $N3$ network.
- **Point 6**: Once the necessary changes have been made, we are going to extract the entire subnetwork whose collapsed node in the blob-tree is the one found previously with the variable A. In this function, it is helped by the previously saved variable of the top node, so the function must take only the network that hangs from this node $topNodeA$. In this way, a network with the name NA is finally obtained, with all the connections and nodes belonging to the subnetwork of $N3$.
- **Point 8**: As point number eight, for each of the MLLS, we will first have to know the node from which every NA hangs. Each network can have a different node directly above it, so this is a separate process for each of them. The extracted node is named p, as a similarity to the article, which is the parent of the node extracted previously as $topNodeA$.
- **Point 9**: Once we have the NA subnetwork to replace and the point from which p is going to hang, the only thing left to do is to do the directly obvious. For this, you must consider what is described in the respective functions used not to leave nodes that are neither in NP nor in NA, since the NA subnetwork is not the same for all networks, only the NA subnetwork of $N3$ is used. These changes are made directly on the networks stored in t.
- **Point 10**: Finally, to finish the modification with NA, it remains to collapse what has been hung on a single node. In this way, as has already been done in point 1, the whole set of leaves of A is collapsed into a single node, in this case without enclosing it by {}, separating them only by spaces. A new variable $tprima$ is created to store the information, so that the iteration over t is not altered.
- **Point 11**: This point is not present in the pseudocode of the article but its addition makes sense during the explanation of the article. It is necessary to somehow reduce the networks that are duplicated after the collapses that occur with the common pendant subnetworks and NA. In this way, if there are duplicate networks, they are eliminated one by one, until only one remains. The transformations are performed directly on the analysis variable $tprima$.
- **Point 12**: Key point in the termination of the execution of the algorithm. When enough isomorphic networks have been eliminated so that only one remains, it will mean that the algorithm has reached the end of the reconstruction process, having obtained the original network.
- **Point 14**: In the case that all the transformations have been carried out and the original network has not been obtained, it will mean that another iteration is needed. To carry out this step, another recursive call to this same function with the modified networks will be needed. In this case, the parameter $tprima$ will be passed, which have collapsed NA, and a depth value greater than the current one. The completion of the recursion will result in a $Nprima$, which will have only one net.
- **Point 15**: As point number fifteen, it will be necessary to uncollapse what has been previously collapsed. In this case, NA is going to be de-collapsed by the sub-network that it had. It must be considered that this NA network is going to be $N3$'s own network extracted previously. Once finished, the recovered information will be obtained in the same variable $Nprima$.
- **Point 17**: The penultimate point is to know if there has been any collapse in point 1. If not, $Nprima$ will be returned without having to make any additional modification.
- **Point 18**: As a final point, if it has arrived here, we have an $Nprima$ that has collapsed the common pendant subnetworks. In this way, for all possible collapsed CPs we are going to have to recover them with their original form. These, with form $\{leaf1, leaf2\}$ are going to have to be connected to a node created with the name of r plus the depth plus the reconstruction number. Finally, once everything necessary has been reconstructed, the resulting network will be returned.

E. MLLS Extraction

To form those networks that will be part of the input information of the algorithm, an additional function to the one implemented above is necessary. In this case, what we are interested in is to be able to form, from an original network, all its possible MLLS subnetworks. These will be formed thanks to the implemented function `CreateMLLSNetworks`. The purpose of this function is to detect those reticular nodes of the original network. From these, it will iterate over the connections they establish with their two parents. Once each edge is selected, it will be removed. The resulting network will be a valid MLLS if it meets two requirements: it must have exactly 3 edges and 2 less nodes. The functions used in this process are the following:

- `GetReticulatedNodes`.
- `GetNumberOfDeletedNodes`.
- `DeleteDegree2Nodes`.
- `CreateMLLSNetworks`.

F. Examples

To test the algorithm and all its different parts, two examples are performed. The first example, used in the article in an illustrative way, serves to get a better idea of how the algorithm must work. The problem with the first example, which makes it insufficient and requires an additional example, is that it does not need recursion. The algorithm solves this first network in

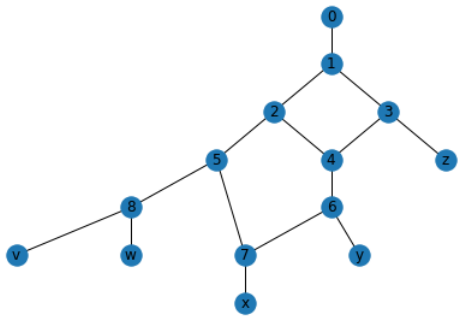

Figure 4. Original network 1 [13].

a single iteration. A second, more complex example, is used to test how well it works at different execution depths.

1) First Example: The first network is the one shown in the Figure 4. From this, a total of 4 sub-networks are formed, which are formed by eliminating the edges of the reticular nodes 4 and 7 with their respective parents. An example of this network is the one shown in Figure 6, which edge between nodes 7 and 6 has been eliminated. After eliminating this edge, the resulting nodes 7 and 6 have a degree 2, meaning that they have one input edge and one output edge. In this way, those nodes are eliminated, having eliminated a total of two nodes and three edges after the selection of this reticulated edge from the reticulated node 7. In the same way, the other three networks are formed to form the input set T of the reconstruction algorithm. In this way, from this input it is expected to obtain the same network as the original one.

Using the same subnetwork as an example, it can be seen at a glance how the algorithm could be expected to recognise and collapse the v and w leaves. This will happen if the same casuistry exists in all nets. Looking at the first paragraph when executing the code we see how exactly the same leaves as described are collapsed:

Starting Collapsing Part
Maximal CP to collapse: $[v, w]$
Node Created: $\{v, w\}$
Ended Collapsing Part

To see what the algorithm does internally, we can see how in the case of the second subnetwork formed in the Figure 5 Part 2 we have a node already collapsed with the name $\{v, w\}$. At this point, the different blob-trees must be formed. For this second case, we can observe that the non-biconnected nodes, that is to say, that with their elimination they disconnect the network, are nodes 1 and 2. Therefore, the two existing nodes to form the blob-trees will be formed from these. In this way, it can be seen how the algorithm does this:

$$[[x, \{v, w\}, y, z], [x, \{v, w\}, y]]$$
$$[1, 2]$$

Seeing that the above has been confirmed, we continue with the reconstruction. At this point we need to find the minimal foundation nodes (FN) in all the blob-trees. In other words, we need to find the minimum node of the blob-trees that is common to all of them. This case is fulfilled by the FN $[z, x, y, \{v, w\}]$, which is called the set A. Then, for all the leaves present in the set A, the check described above of the three types will be performed. In this case, the check is satisfied for the leaves y and x, resulting in the type $\Pi(y, x)$ having the subnetwork with index 3, the fourth network of the input. The selected leaves are not the only options, seeing as in the article the selected leaves are other equally valid ones such as $\{v, w\}$ and x. The selected nodes are already ordered according to the one that has the parent reticulated, in this case being the leaf x. This check has been done by checking for the other reticulated the H-form between the two leaves.

At this point, two new nodes, a and c, have to be added. This happens in the selected network, seeing how in the Figure 5 Part 3 nodes $a0$ and $c0$ are on the selected leaves. Finally, it remains to hang the subnetwork formed with these new nodes on all the other networks. In this way, the top nodes are selected, in this case being in all of them the root node 0. It remains to collapse this new sub-network, which can be seen in Figure 5 Part 3.

Finally, from these networks with collapsed A, those that are isomorphic are eliminated. In this case, with this first iteration we have managed to keep only one, which means that we have reached the original network, returning the network in the Figure 7 equivalent to the original one in the Figure 4.

2) Second Example: The second network is the one in the Figure 8. In this case, the leaves with original values a and c are converted to $aprima$ and $cprima$ to avoid internal conflicts with created internal nodes a and c. From this, a total of 14 sub-networks are formed, which are formed by eliminating the edges of the reticular nodes 4, 13, 14, 20, 21, 26 and 27 with their respective parents. An example of this network is the one shown in Figure 9 Part 1, which edge between nodes 2 and 4 has been eliminated. In this way, the resulting nodes 4 and 2 with the elimination of this edge become nodes with degree 2. For this reason, both nodes are eliminated, having eliminated a total of two nodes and three edges. In the same way, the other 13 networks are formed to form the input set T of the reconstruction algorithm. In this way, from this input it is expected to obtain the same network as the original one.

In a very similar way to the first example described above, the same reconstruction procedure is performed, but in this case a total of 3 complete runs at different recursion depths are needed. Thus, an idea of the procedure followed and the state of the network in each of these can be seen in Figure 9. In the first iteration, the subnetwork collapses the leaves e and f, forming for the case of the first subnetwork the collapsed node $\{e, f\}$ from the discovered cherry, which can be seen in Figure 9 Part 2. From this moment on, all the nodes that are part of the blob tree are collected. In the case of the observed MLLS we have the following, with their respective top nodes from which they are formed:

$$[k, l, m, n, g, h, i, d, \{e, f\}, aprima, b, cprima, j],$$
$$[k, l, m, n],$$
$$[aprima, b, cprima]$$

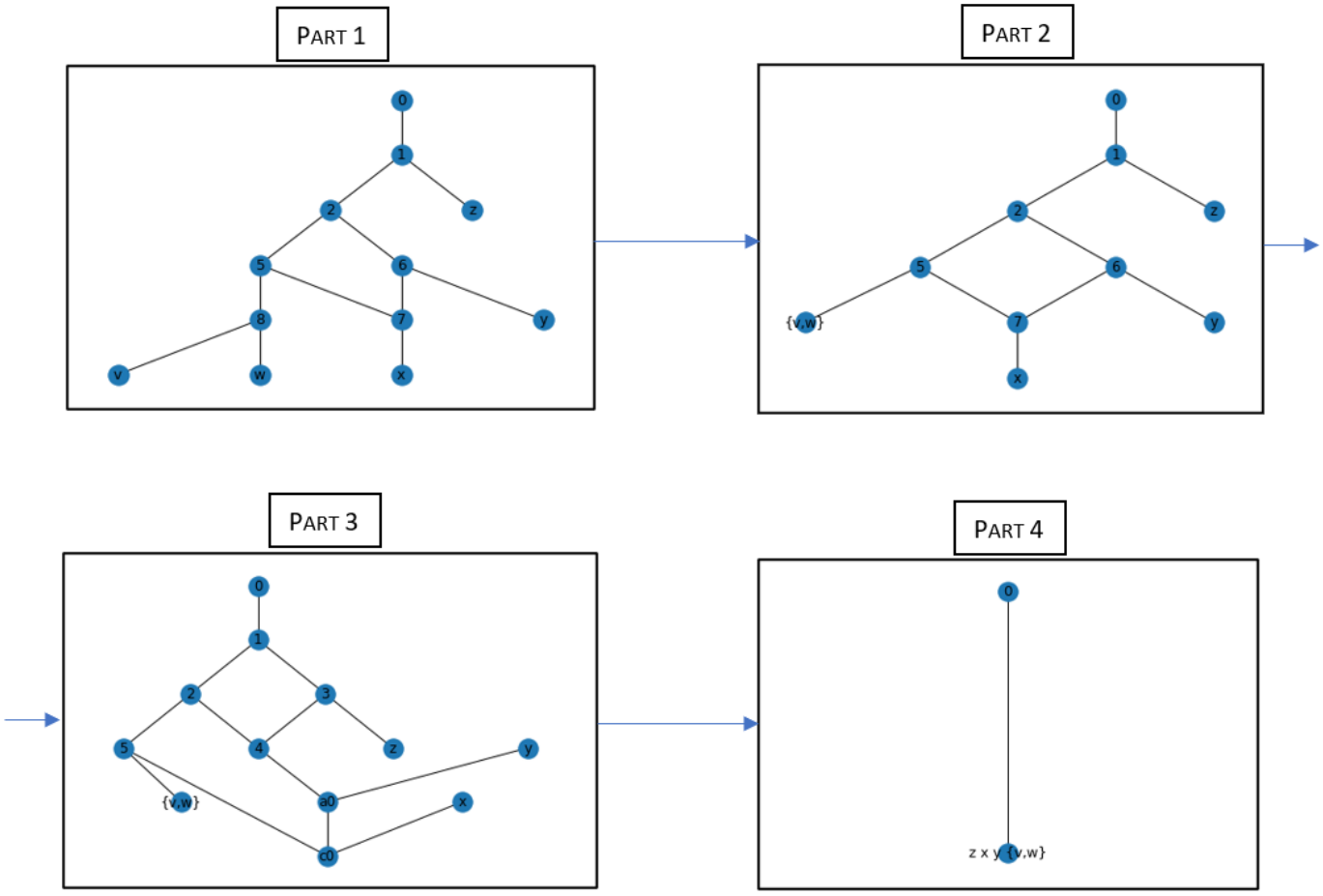

Figure 5. Part 1: Second MLLS extraction from original network in Figure 4. Part 2: Second MLLS extracted from Original Network 1 with collapsed leaves v and w as $\{w, v\}$. Part 3: Second subnetwork with addition of nodes a and c. Part 4: Second MLLS with collapsed subnetwork NA.

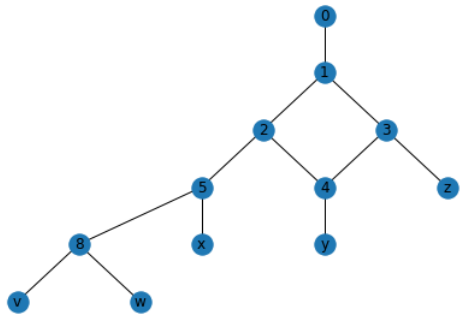

Figure 6. First MLLS extracted from original network 1 in Figure 4.

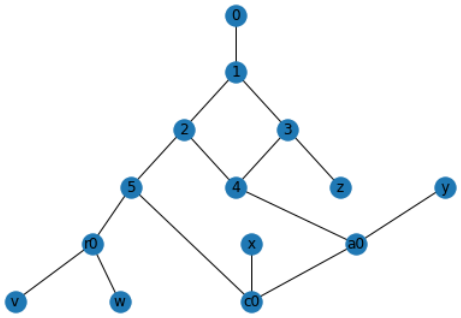

Figure 7. Reconstructed network from original network 1 in Figure 4.

$$[1, 7, 19]$$

Finally, for this first depth, it is found the minimum foundation node known as set A, which in this case is equivalent to the one formed by $[aprima, b, cprima]$. All the participating leaves of A are checked for those that meet the three types analysed. In this case, in the subnetwork 13 are added two new nodes (which are $a0$ and $c0$) for this depth above the leaves $aprima$ and b in this order, which can be seen in Figure 9 Part 3. In this way, it will be reduced in a subnetwork similar to the one observed for the first MLLS in Figure 9 Part 4, as all the part other than A can have its own particularities. With all the subnetworks with A collapsed and not having reached the global isomorphism, we proceed to a second iteration. In this moment, all the inputs have been modified for all the subnetworks, all the processes are passed again. In this case,

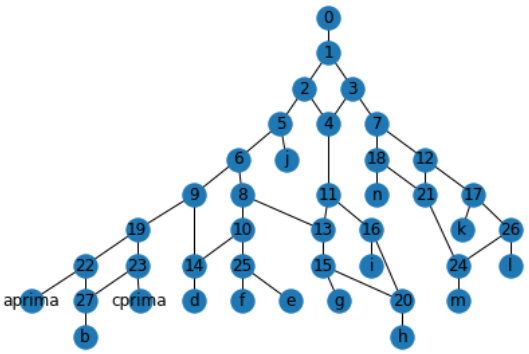

Figure 8. Original network 2 [13].

no maximal common pendant is found to collapse, so the algorithm is continued. As a first step, the new blob trees need to be formalised, which following the case of the first MLLS, its new blob tree and top nodes are as follows:

$$[k, l, m, n, g, h, i, d, \{e, f\}, aprima\ b\ cprima, j],$$
$$[k, l, m, n]$$
$$[1, 7]$$

It can be seen in the first node formed above that there is a leaf which is the collapse performed in the previous recursion. Thus, the foundation node minimal for these new blob trees is sought, which is the one formed by $[k, l, m, n]$. Being this the new set A, the three types for all the leaves participating in this subnetwork A will be searched again. In this case, leaves k and l are found to be valid, which will receive two additional nodes a and c for subnetwork number 11. Once the new subnetwork is found in Figure 9 Part 5, it will be substituted by all the subnetworks A of all the other subnetworks. In this way, the last point to do is to collapse these new ones into a single node, the resulting modified subnetworks are going to be the input for the next recursion as the global isomorphism is not yet found. Even so, 4 equivalent networks are found, which are eliminated from the study to optimise the cost of the algorithm. As a last recursion for this network, we have as input for the observed case the subnetwork observed in the Figure 9 Part 6. For this depth there are no maximal common pendant subnetworks to collapse. Again, the blob trees will be formed, being for this first case the following with its upper nodes:

$$[g, h, i, klmn, d, \{e, f\}, aprima\ b\ cprima, j]$$
$$[1]$$

As in the previous cases, it can be seen how the A subnets are kept collapsed in the form of leaves. In this case, as there is only one node left in the blob tree, this will have to be the common one for all the subnetworks, as shown by the algorithm. For this last case, the subnetwork selected to add nodes a and c is number 7, with leaves g and h, observing the result in Figure 9 Part 7. Finally, this new subnetwork A is added and collapsed in all MLLS. Seeing as a result in Figure 9 Part 8, all the subnetworks are isomorphic, so all of them are eliminated leaving only one, which is returned as equivalent to the original one before the extraction of all its MLLS, as shown in Figure 10.

V. Reconstruction from pairwise leaf distances

A. Paper

In [3] it is presented a reconstruction of an original network made from a distance matrix. This matrix is obtained from the distances between the leaves that form the network. Because of the existence of reticular nodes, there can be more than one possible distance between two leaves. This characteristic will be fundamental in differentiating whether between two leaves there was a Cherry type or a reticulated Cherry.

In this case, the network contains positive weights or distances associated with each edge. In its definition, there is a strong constraint. For all reticulated nodes, the input edges of the reticulated nodes must have the same weight. In the same way they will be reconstructed. It should be noted that the reconstructed networks will have the same number of nodes, the same number of edges and the same connections, but they could have different weights for each edge comparing with the original one. The main difference is that the input edges of the reticulated nodes will have a weight equal to 0. Even so, the distance matrix that can be obtained from the reconstructed one is the same as the one obtained from the original one, meaning that both networks are equivalent. This can be seen and compared in original network in Figure 14 and in its reconstruction in Figure 16 used in the example of this Section.

The idea in this algorithm is to classify each pair of leaves into one type or the other. This process follows an order according to the distance from an introduced element called outgroup. This is a kind of leaf with unique characteristics. It is directly connected to the root node, and the distances to all the other leaves are calculated in the same way. This element will be fundamental to be able to know, calculate and reconstruct the distances and above all the reticular shapes. The order followed to iterate and select the leaves to analysed is followed by the $Qscore$ inspired by the Neighbour-Joining algorithm [14], to determine those local structures that are of major importance in the construction of the network.

B. Implementation Structure

In the developed notebook *Q_Reconstruction_Algorithm.ipynb*, which can be found in https://bit.ly/3fwbo3K, the idea described above is implemented, developing what the authors propose in the form of pseudo-code. The defined points try to keep a common structure with the `MLLS_Reconstruction_Algorithm` algorithm. In this way, the different points can be found:

- **Initialization and variables declaration**: In this point the libraries used are imported. In this case, none of the main ones is the main one, being all of them used in an auxiliary way. The `networkx` library is only used to configure the networks that form part of the algorithm input.
- **Function Declaration**: At this point, the methods that participate in the operation of the main algorithm are

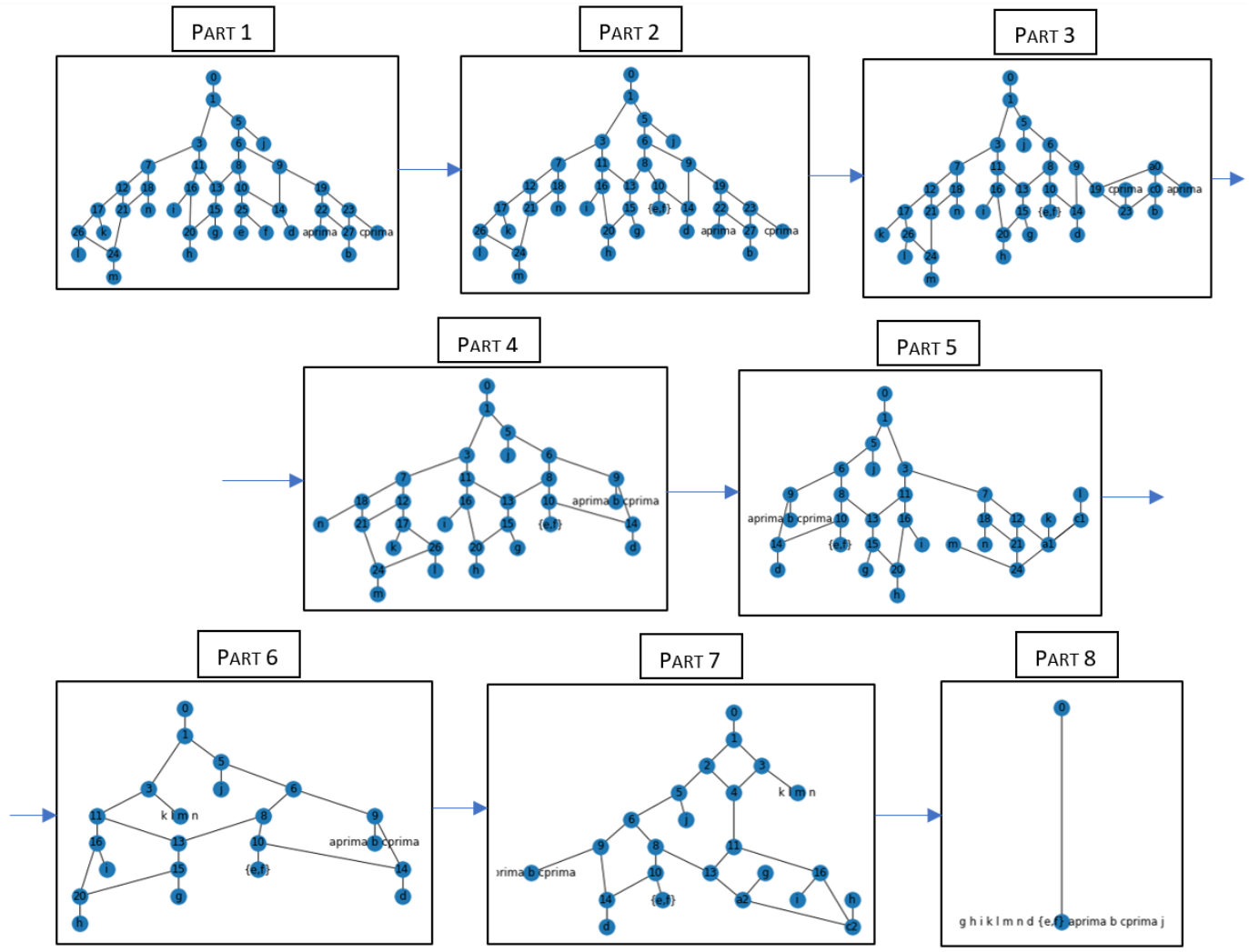

Figure 9. Internal transformation of the first subnetwork in the reconstruction performed by the algorithm. Part 1: First MLLS extracted from original network 2 in Figure 8 as input. Part 2: Collapsing part generating node $\{e, f\}$ as a result. Part 3: Addition of nodes $a0$ and $c0$. Part 4: Collapsed subnetwork NA. Part 5: First recursive level and adding new nodes $a1$ and $c1$. Part 6: Collapsed NA. Part 7: Second recursive level and adding new nodes $a2$ and $c2$. Part 8: Final collapse of subnetwork NA and final step of the reconstruction.

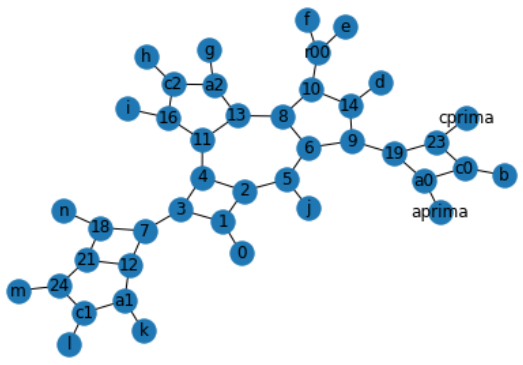

Figure 10. Reconstructed network from original network 2 in Figure 8.

defined. They are divided into layers according to their level of abstraction. See subsection V-C.

- **Main algorithm**: At this point the reconstruction algorithm is developed, maintaining a structure as similar as possible to the one described in the reference paper. See subsection V-D.
- **Distance matrix extraction from original network**: In this section we work on obtaining an algorithm that performs the extraction of the distance matrix from an original network. See subsection V-E.
- **Example**: In this last section, a complete example of the algorithm is given. In this case, only one example is used as it accesses different levels of recursion. See subsection V-F.

The main structures used are the following:

1) Networks with `networkx`. The networks provided by the free `networkx` library are used. These networks have a series of formal restrictions regarding their declaration. The root will be the node with value 0. Leaf nodes will have alphanumeric values, while internal nodes will have positive numeric values. The edges between them will

1. If $|X| = 1$, then return the phylogenetic network $(\mathcal{N}_0, w_0)$ consisting of the single vertex r.
2. If $|X| = 2$, say $X = \{r, s\}$, then return the phylogenetic network $(\mathcal{N}_0, w_0)$ consisting of leaves r and s adjoined to the root ρ with (ρ, r) weighted the single value in $\mathcal{D}_{r,s}$ and (ρ, s) weighted 0.
3. Else, find a 2-element subset $\{s, t\}$ of X such that

$$\mathcal{Q}_r(s,t) = \max\{\mathcal{Q}_r(x,y) : x, y \in X - \{r\}\}.$$

(a) If $|\mathcal{D}_{s,t}| = 1$ (in which case, $\{s, t\}$ is a 0-reticulated cherry), then
(i) Reduce t in $\mathcal{D}$ to give the multi-set distance matrix $\mathcal{D}'$ on $X' = X - \{t\}$.
(ii) Apply $\mathcal{Q}$-REDUCTION to input X', $\mathcal{D}'$, and r. Construct $(\mathcal{N}_0, w_0)$ from the returned network $(\mathcal{N}'_0, w'_0)$ on X' by reversing the reduction on t. In particular, if u is the parent of s in $(\mathcal{N}'_0, w'_0)$, then subdivide (u, s) with a new vertex v, add a new leaf t and adjoin it with the new edge (v, t), assign weights $w_0(u, v)$ and $w_0(v, s)$ so that

$$\mathcal{Q}_r(s,t) = \max\{\mathcal{Q}_r(x,y) : x, y \in X - \{r\}\}$$

and

$$w_0(u,v) + w_0(v,s) = w'_0(u,s),$$

and assign weight $w_0(v, t)$ so that $d_{\min}(s,t) = w_0(v,s) + w_0(v,t)$. Return $(\mathcal{N}_0, w_0)$.

(b) Else ($\{s, t\}$ is a 1-reticulated cherry, in which case it has reticulation leaf t if, for all $x \in X - \{s, t\}$,

$$\{d + c : d \in \mathcal{D}_{s,x}\} \subsetneq \mathcal{D}_{t,x},$$

where $c = d_{\max}(r,t) - d_{\max}(r,s)$),
(i) Cut $\{s, t\}$ in $\mathcal{D}$ to give the multi-set distance matrix $\mathcal{D}'$ on X.
(ii) Apply $\mathcal{Q}$-REDUCTION to input X, $\mathcal{D}'$, and r. Construct $(\mathcal{N}_0, w_0)$ from the returned network $(\mathcal{N}'_0, w'_0)$ on X by reversing the cutting of $\{s, t\}$. In particular, if u_1 and u_2 denote the parents of s and t, respectively, in $(\mathcal{N}'_0, w'_0)$, then subdivide (u_1, s) and (u_2, t) with new vertices v_1 and v_2, respectively, adjoin v_1 and v_2 with the new edge (v_1, v_2), assign weight $w_0(u_1, v_1)$ so that

$$\mathcal{Q}_r(s,t) = \max\{\mathcal{Q}_r(x,y) : x, y \in X\},$$

assign weight $w_0(v_1, s)$ so that

$$w_0(u_1,v_1) + w_0(v_1,s) = w'_0(u_1,s),$$

and assign weight 0 to (v_1, v_2) and (u_2, v_2), and weight $w'_0(u_2, t)$ to (v_2, t). Return $(\mathcal{N}_0, w_0)$.

```python
def Q_Algorithm(X, D, r, recursivity_depth = 0):
  finalNetwork = nx.Graph()
  #0
  if len(X) == 0:
    return finalNetwork
  #1
  elif len(X) == 1:
    if r == X[0]:
      return finalNetwork.add_node(r)
    return finalNetwork
  #2
  elif len(X) == 2:
    CreateDoubleVertexNet(finalNetwork, X, D, r)
    return finalNetwork
  #3
  else:
    #Find max distance subset - {r}
    subset, max_distance = Find2ElementSubset(X, D, r)
    #Case 0-reticulated cherry
    if len(GetDictValue(D, subset[0], subset[1])) == 1:
      #i. Reduce
      Dprima, Xprima = UpdateD0(X, D, subset[0], subset[1])
      #ii. Q-Algorithm(X', D', r)
      finalNetwork = Q_Algorithm(Xprima, Dprima, r, recursivity_depth+1)
      if recursivity_depth >= 0:
        CherryReconstruction(finalNetwork, subset[0], subset[1], D, r, recursivity_depth)
    #Case 1-reticulated cherry
    else:
      if ReticulationTest(D, X, subset[0], subset[1]):
        s = subset[0]
        t = subset[1]

      elif ReticulationTest(D, X, subset[1], subset[0]):
        s = subset[1]
        t = subset[0]
      #i. Cut
      Dprima = UpdateD1(X, D, s, t, r)
      #ii. Q-Algorithm(X, D', r)
      finalNetwork = Q_Algorithm(X, Dprima, r, recursivity_depth+1)
      if recursivity_depth >= 0:
        HReconstruction(finalNetwork, s, t, D, recursivity_depth)
  return finalNetwork
```

Figure 11. Q_Reconstruction_Algorithm: Comparison between original algorithm (left side) and implemented one (right side).

have positive weight. In this case, in order to maintain the same defined structure, a new outgroup node is added with the name r connected only to the root node. This point is critical as it conditions the good functioning of the algorithm from this library with the possibilities that it attributes. With these restrictions, it is possible to formalise tree-child network structures.

2) Distance matrix in dictionary form. For the definition of the distance matrix defined in the paper, a structure offered by the dictionary is chosen. This makes it easier to iterate over the different sheets, as well as to delete, modify or perform other operations. It must be considered that the distances appear only once, that is to say, even if from x to y and from y to x there is the same distance, it will only be saved in one of these two pairings.

C. Methods

In this section the different methods involved in the main reconstruction algorithm are declared. The methods are divided into three main groups:

1) Low-Level Functions: In this group are the functions that are closest to data manipulation. They have very basic and concrete objectives. Among these are the functions that manipulate the data structures used: the dictionary and the network itself. With these four functions, all the necessary operations on the dictionary are achieved. The operation of these functions is intuitive, yet the description of their development is available in the notebook.

- `GetDictValue`.
- `RemoveDictValue`.
- `UpdateDictValue`.
- `CleanDictValue`.

In addition, a few auxiliary functions are needed for the extraction of the distance matrix from the original network, such as the `GetLeaves` function and the `GetValue` function (described in the previous algorithm).

2) Medium-Level Functions: In this group are declared those functions that need further development and with broader objectives than those of the first group. Even so, they are not found in the final declarations of the algorithm, so they are in this intermediate state. In order to better understand how all the different steps, work and how they have been reached, the main ideas and functionalities of the different declared methods will be defined.

- `Q_distance`: This function oversees calculating the distance Q for two given leaves. It applies the mathematical function of the $Qscore$ from the distance data stored in

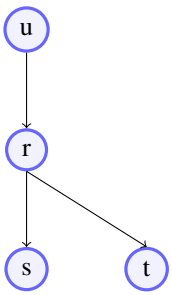

Figure 12. Cherry reconstruction structure.

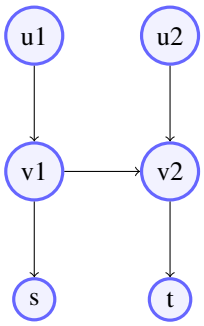

Figure 13. Reticulated cherry (H) reconstruction structure.

the dictionary.

- `D_max`.
- `D_min`.
- `FindCherryWeights`: This function calculates the distances in the reconstruction between u and v, between v and s, and between v and t. A value c is calculated from the difference of weights between the two weigths pending of r. This function works for both positive c and negative c, when the path to s is greater than to t. This structure and its nomenclature can be seen in Figure 12.
- `FindHWeights`: This function calculates the weights for the edges between the u, v and s nodes. This calculation is performed for the reconstruction of the network with the element H. This structure and its nomenclature can be seen in Figure 13.
- `GetSubset`: This function returns true or false depending on whether or not $d1$ is a subset of $d2$, comparing the first subset to the second one summed to the value c.
- `CutST`: This function eliminates the central path of the case H for the specific case of the leaves s and t. For this case the smallest distance between these two leaves is eliminated.
- `GetOutJoin`: This function returns those distances between two leaves that do not pass through the central edge of H. For this, it is checked for all distances plus c whether they lie within the other set of distances.

3) High-Level Functions: In this group are declared the functions directed called from the algorithm as can be seen in the figure 11.

- `GetCutDistances`: This function obtains the distances between leaf s and leaf t that do not pass through the central edge of the form H, which we have already seen in the Figure 2, between them. For this, the value of c is needed and from this the distances that do not pass through this central edge are obtained.
- `ReticulationTest`: This function says, for case of the form H between two leaves, the leaf t passed by parameters has the parent as a reticular node.
- `Reduce`: This function updates the weights for the first case, cherry shape between two leaves. For this case all distances to leaf t will disappear, as this leaf is removed from the list. The same happens in the list of available leaves.
- `Cut`: This function updates the weights for the second case, form H between two sheets. For this case no leaf is removed. The distances that used the central path will disappear. Moreover, only the distances to the leaf with the lattice parent of the H-shape, i.e. with t, are modified.
- `HReconstruction`: This function reconstructs the H element of the network, see Figure 13. It modifies the network by adding the new element and assigns the relevant weights to the affected edges. The modification occurs directly in the input network.
- `CherryReconstruction`: This function reconstructs the cherry element of the network, see Figure 12. It modifies the network by adding the new element and assigns the relevant weights to the affected edges. The modification occurs directly in the input network.
- `CreateDoubleVertexNet`: This function creates a defined and concrete type of structure. All the weight found in D, there being only one, is assigned to the edge between the root and the leaf in X, while the weight between the root and r will be 0.
- `Find2ElementSubset`: This function returns a pair of leaves s and t, which have the maximum value of the formula Q score. In case there is more than one pair of leaves with the same maximum, the first one is chosen.

D. Main Algorithm

The main process has been called in the same way as in the article: `Q_Algorithm` (Figure 11). The same guidelines and procedures in terms of nomenclature and structure have been maintained as in the previous algorithm.

The first point to analyse is the inputs. As can be seen, there are the same three inputs as the article, namely X, D and r. The first of these contains the list of leaves present in the network. The second variable contains the distance matrix. This, as we have seen, also implicitly contains the leaves in the network, although it is fully complementary to the first variable. Finally, the last variable is r, which indicates the name of the outgroup present in the list of leaves. In addition, as in the previous algorithm, there is a final additional variable `recursivity_depth` to indicate and be aware of the depth of the recursion of the execution in which it is found. This variable is initialised to 0 by default, so that it is not used externally, allowing a correct increment when recursion is performed.

- **Point 0**: In this first one it is not contemplated in the article. Even so, it must be present as an error control

point or simply as an indication of an input which does not give the option of generating a reconstruction from it. In this way, it is controlled when the list of leaves X comes without any element, that, by default, the distance matrix will be empty as well as the r variable. In this case, execution is stopped and an empty network is returned.

- **Point 1**: First point covered in the paper. In this case it is considered the case that in the list of leaves X there is only one element. For this case, if the remaining element is equal to the leaf present in the outgroup variable r, it means that a particular network has to be created. This shall be formed only by the node r, which shall be returned. In case the equality between the remaining node and the outgroup is not fulfilled, a null value shall be returned.
- **Point 2**: As a second point, following the previous idea, we intend to give a solution to a possible end of the recursion. In this case, the end is contemplated when there are two elements left in the list of leaves. By definition of the algorithm itself, one of these two remaining leaves must be equal to the outgroup r, so this check is not performed. In addition, there will only be two leaves in the distance matrix, and there is only one distance for these two leaves. At this point, a specific basis for the final network is constructed. This will consist of three nodes, a root connected to a leaf r and at the same time connected to the other remaining leaf in list X. Moreover, the distribution of the weights is also very specific, with all the weight of the distance available in the matrix being dumped onto the edge between the root and the leaf other than r. In this way, the distance between the root and the leaf r will be equal to 0.
- **Point 3**: As a third point, having passed the various checks above, we have a number of leaves greater than 2. For this case, we must continue with the recursion and obtain a next group of type cherry or type H between two leaves. The selection of these two candidate leaves must be done through the application of the Q score function. In this way, for all the leaves available at this depth, the first two with a maximum value of Q score will be selected. These two will be stored in the list of two subset elements (s, t), in addition to the value of q_distance obtained in the max_distance variable. The latter will be used only for external display for understanding in the execution of this.
  - Case 0: For the two available types of participating network structures, it is examined at this point whether the one found is of type cherry. To be part of this structure in the distances for the two leaves found in the previous point there must be only one distance available.
    - Case 0, (i): After finding a substructure between the two selected cherry leaves, the current distance matrix must be reduced. This process abstractly eliminates the existence of the second leaf, i.e., the leaf t. All leaves that have a distance connection to this leaf will be eliminated, as well as all entries of t. For this point a different distance matrix will be obtained, which will be named $Dprima$, in addition to the elimination of the leaf t in the list of leaves, which will be named $Xprima$.
    - Case 0, (ii): As an end point of the first cherry case found, the function must be recursively called again with the previously reduced information. As a result of this, a network with all leaves and distances present in the newly formed raw variables will be returned. As a final part, the information that has been found at this depth level in the network returned in the recursive call has to be reconstructed. For this, the relevant distances between the connections between the s- and t-sheets and their parent must be correctly inscribed.
  - Case 1: For the two available types of participating network structures, at this point it is examined whether the one found is of type reticulated cherry (H). To be part of this structure in the distances for the two leaves found in point 3 there must be more than two available. This means that more than two paths are present between one leaf and the other. Before proceeding further, it is essential to check which of the two leaves has the latticed parent, as they cannot be reversed. In this way, the `ReticulationTest` function is used to examine whether the first subset element is equivalent to s or t.
    - Case 1, (i): Once the structure H and the order of the leaves have been found, the central connection of the formed H is cut in this line. This will involve the elimination of certain distances in the distance matrix D. It should be stressed that this process does not eliminate any leaves, so the list of leaves remains identical. Thus, as a result of this call, a new variable $Dprima$ is obtained to store the new information of the distance matrix.
    - Case 1, (ii): As the end point of the second case H found and the last point of the algorithm, this same function must be called recursively

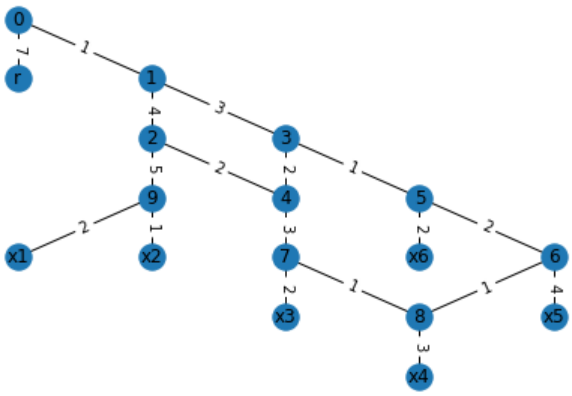

Figure 14. Original Network for Q_Algorithm [3].

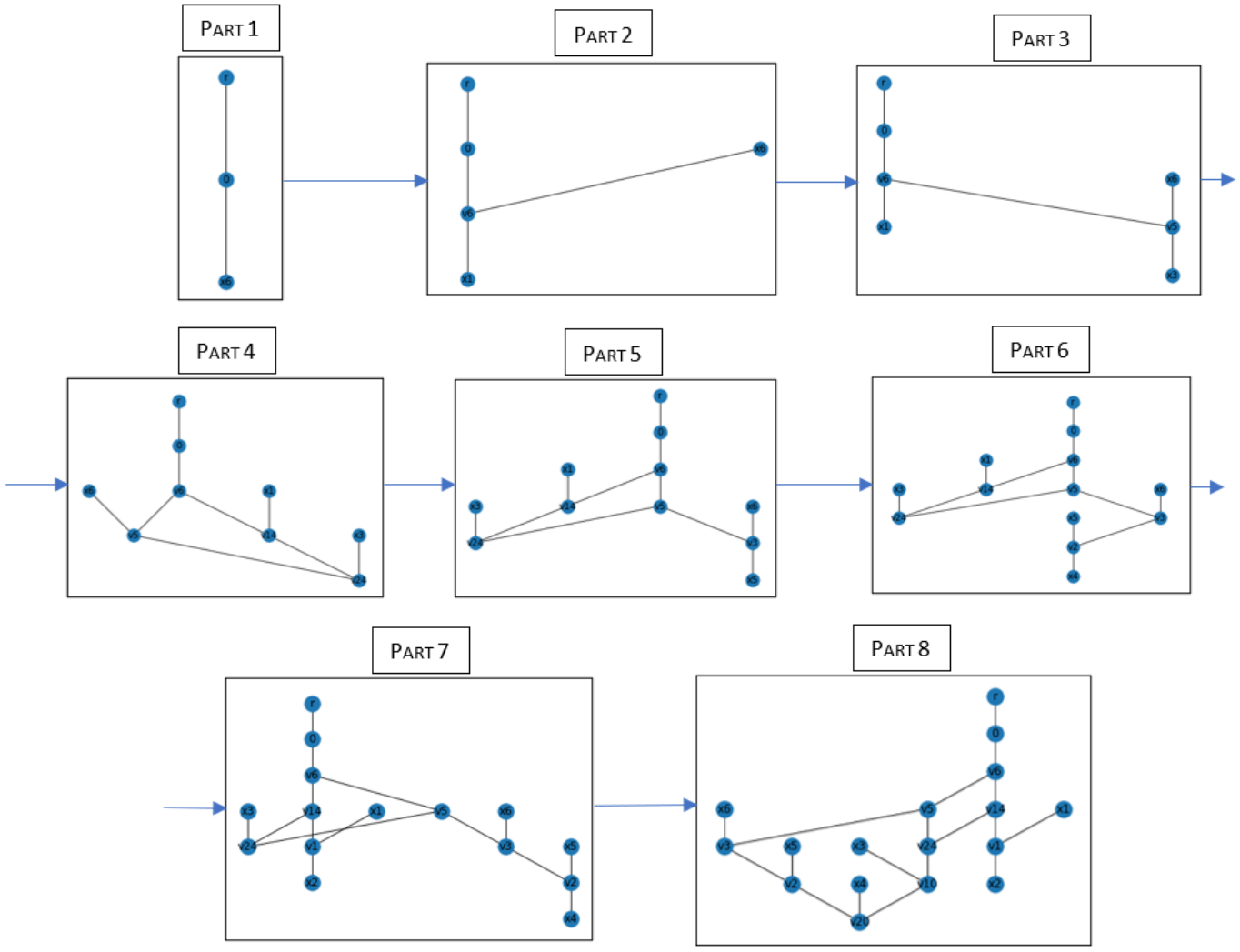

Figure 15. Internal transformation of the subnetwork in the reconstruction performed by the `Q_algorithm`. Part 1: Basic structure in depth 7. Part 2: Reconstructed Cherry in depth 6. Part 3: Reconstructed Cherry in depth 5. Part 4: Reconstructed H in depth 4. Part 5: Reconstructed Cherry in depth 3. Part 6: Reconstructed Cherry in depth 2. Part 7: Reconstructed Cherry in depth 1. Part 8: Reconstructed H in depth 0.

with the information cut from the previous point. As a result, a network with all the leaves and distances present in the newly formed raw variable will be returned. As a final part, the information that has been found at this depth level in the network returned in the recursive call must be reconstructed. For this, the relevant distances between the connections between the s- and t-leaves and the connections between these must be correctly inscribed. It should be noted that the double connection between the parents of the lattice parent of t will have a weight equal to 0.

E. Distance Matrix Construction

For the construction of the distance matrix, a series of own implementations are used. For this construction, all possible paths between one leaf and another are traversed with a series of established limitations. These limitations, which can be followed in a very specific way in the code, are, on the one hand, the non-repetition of paths already visited and, on the other hand, always following an up-down path. The latter limitation means that when there is a change of direction, it cannot be changed again, i.e., once you go down, you cannot go up again. With this, it is possible to limit the possibilities and obtain the paths that are really going to be used in the reconstruction of the network. The following functions are used:

- `GetDistance`.
- `GetDirection`.
- `GetDistanceMatrix`.

F. Example

For the complete testing and execution of the whole algorithm, only one example is used in this case. This runs through all the different parts, as well as different levels of recursion. The source network used is the one shown

Table I
DISTANCES MATRIX EXTRACTED FROM ORIGINAL NETWORK IN FIGURE 14

	r	x1	x2	x3	x4	x5	x6
r	-	-	-	-	-	-	-
x1	[19]	-	-	-	-	-	-
x2	[18]	[3]	-	-	-	-	-
x3	[19, 18]	[14, 21]	[13, 20]	-	-	-	-
x4	[21, 20, 18]	[21, 16, 23]	[20, 15, 22]	[21, 14, 6]	-	-	-
x5	[18]	[21]	[20]	[21, 14]	[23, 16, 8]	-	-
x6	[14]	[17]	[16]	[17, 10]	[19, 12, 8]	[8]	-

in the Figure 14. The first step to perform before making use of the reconstruction algorithm is to obtain the distance matrix between all the different leaves. Using the implemented function, the following distances are reported in Table I.

In this table it should be noted that the distances are stored in a unique way, so that only one triangle of the total table is retained. In addition, the outgroup node is also obtained in the form of variable r and the list of leaves in the form of variable X:

$$[r, x6, x5, x3, x4, x1, x2]$$

Once we have all the necessary variables, we proceed to the reconstruction of the analysed network. For each iteration, the one with the highest q_score is calculated for all possible combinations of leaves. In this first case, it can be observed that leaves $x3$ and $x4$ are selected with a q_score value of 17. As there is more than one distance available between these two leaves, it is equivalent that in the original network there must be an H shape between the two, which can be seen in the Figure 14 that this is the case. In this way, once the structure found has been selected, we proceed to do a `Cut` of the distance matrix. This procedure tries to eliminate those distances that pass through the central connection of the H form. The resulting table is as shown in the Table II.

Table II
DISTANCE MATRIX ALGORITHM DEPTH 0.

	r	x1	x2	x3	x4	x5	x6
r	-	-	-	-	-	-	-
x1	[19]	-	-	-	-	-	-
x2	[18]	[3]	-	-	-	-	-
x3	[19, 18]	[14, 21]	[13, 20]	-	-	-	-
x4	[18]	[21]	[20]	[21, 14]	-	-	-
x5	[18]	[21]	[20]	[21, 14]	[8]	-	-
x6	[14]	[17]	[16]	[17, 10]	[8]	[8]	-

Having made the necessary adjustments, we proceed to the next level of recursion. For this second level, a pair of leaves $x1$ and $x2$ is found with the same value of q_score as in the first case. As they have the same value, they could have been produced in reverse without altering the results. For this case, as there is only one distance, it is a cherry structure formed between the two leaves, as can be seen in the Figure 14. In this case, the method `Reduce` has to be performed on the matrix, which consists of eliminating all paths leading to $x2$. Thus, the resulting matrix is shown in the Table III.

With this new distance matrix, we move to the next level.

Table III
DISTANCE MATRIX ALGORITHM DEPTH 1.

	r	x1	x3	x4	x5	x6
r	-	-	-	-	-	-
x1	[19]	-	-	-	-	-
x3	[19, 18]	[14, 21]	-	-	-	-
x4	[18]	[21]	[21, 14]	-	-	-
x5	[18]	[21]	[21, 14]	[8]	-	-
x6	[14]	[17]	[17, 10]	[8]	[8]	-

Here, with a q_score value of 14, a pair of $x5$ and $x4$ leaves is found. These are equivalent in the same way as the previous one to a cherry. At first glance, this structure is not visible in the original network. It should be noted that the connection between nodes 7 and 8 of the original network in Figure 14 does not exist for the first step, so that from node 6 hangs a cherry between the leaves found. The resulting matrix has its leaf $x4$ removed as shown in the Table IV.

Table IV
DISTANCE MATRIX ALGORITHM DEPTH 2.

	r	x1	x3	x5	x6
r	-	-	-	-	-
x1	[19]	-	-	-	-
x3	[19, 18]	[14, 21]	-	-	-
x5	[18]	[21]	[21, 14]	-	-
x6	[14]	[17]	[17, 10]	[8]	-

As the next recursion level, another cherry is found again between the leaves $x6$ and $x5$, in this case with a q_score value of 12. The resulting matrix has the connections to $x5$ removed as shown in the Table V.

Table V
DISTANCE MATRIX ALGORITHM DEPTH 3.

	r	x1	x3	x6
r	-	-	-	-
x1	[19]	-	-	-
x3	[19, 18]	[14, 21]	-	-
x6	[14]	[17]	[17, 10]	-

As the antepenultimate case, a second H-structure is detected between leaves $x1$ and $x3$ with a q_score value of 12. In this case, it must be taken into account that the cross-linked parent is the one of leaf $x3$ and not the other way round, since in that case the reconstruction cannot be performed. In

this case the method `Reduce` is performed again, so that the matrix is as follows without seeing any of its leaves eliminated as shown in the Table VI.

Table VI
DISTANCE MATRIX ALGORITHM DEPTH 4.

	r	x1	x3	x6
r	-	-	-	-
x1	[19]	-	-	-
x3	[18]	[21]	-	-
x6	[14]	[17]	[10]	-

As a penultimate case, another cherry is found again for the $x6$ and $x3$ leaves found with a q_score value of 11. The distance matrix is as follows as shown in the Table VII.

Table VII
DISTANCE MATRIX ALGORITHM DEPTH 5.

	r	x1	x6
r	-	-	-
x1	[19]	-	-
x6	[14]	[17]	-

Finally, the last recursion level finds a cherry with a q_score value of 8 for the combination of leaves $x6$ and $x1$. By eliminating the latter, only $x6$ and r will remain in the list of leaves, so in the next iteration it will form the second final basic structure from the following distance matrix in the Table VIII.

Table VIII
DISTANCE MATRIX ALGORITHM DEPTH 6.

	r	x6
r	-	-
x6	[14]	-

For the network reconstruction process, the algorithm will reconstruct the network from the most basic structure found above. As can be seen in the Figure 15 it is possible to see how for each depth the found structures are added. In this way, for Parts 4 and 8 in this same Figure, H structures are added, while in all the other Parts cherries are added. In the case of Part 1, it is the basic structure found in point 2 described in the algorithm. Finally, the recovered structure equivalent to the original one is formed in the Part 8 seen in the process with all the weights, as can be seen in Figure 16.

The reconstructed network consists of the same number of nodes and number of edges, as well as the same connections between the nodes that form it. Even so, the weights of the edges are not the same as the original one, having a total sum of the weights of 43 compared to the sum of 46 of the original network. This difference is derived from the weights $1 + 2$ of the parent edges of the reticulated nodes 4 and 8 respectively, which are equal to 0 in the reconstructed network. Seeing this, the networks are not equal but equivalent, being able to generate from these the same distance matrix.

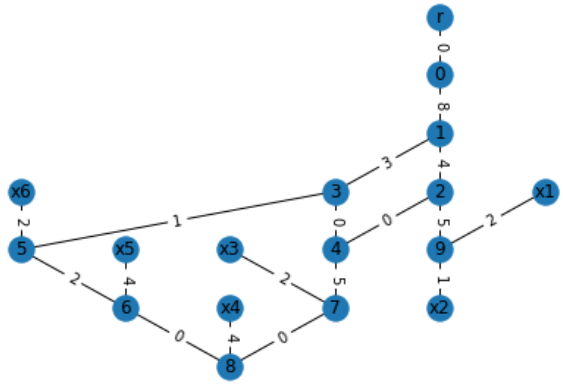

Figure 16. Reconstructed network with weights from original network in Figure 14.

VI. MAIN SIMILARITIES AND DIFFERENCES

As we have seen in section IV and in section V, two algorithms are developed that address the same problem from different points of view. In this way, the two, although they embark on their reconstruction journey from different starting points, share parts of the path to reach the same goal, to achieve a total reconstruction of a phylogenetic network of tree-child class. Looking at this it is clear that there will be differences and similarities between the two approaches.

The main similarity is the structure of the network itself. The two developments exploit and play around its construction in order to define the reconstruction. The shapes defined between two leaves in the Figure 2 is key in both the first and the second. For the reconstruction from MLLS it is necessary to find and reconstruct those H structures, with special interest in the reticulated parent. In the extraction of the subnetworks, they lose this characteristic, so finding them is the main point of this one. In the second algorithm analysed, the H structure is of vital importance, as is the cherry (Λ) structure. Another similarity is the computational cost in polynomial time. This implies that the larger the input, the higher the cost of resources and time used in the two algorithms.

The main difference is the origin of the reconstruction itself. While the first algorithm requires a relatively large number of sub-networks from the original one, the second one only requires a matrix of distances from a single network. This implies that in some situations this may be a differential issue. Still, the algorithms can work in a complementary way, as they both address the same problem. In addition, another difference that can be extracted in the operation of the algorithms themselves is that in the first one, the complexity and computational cost of solving the reconstruction is a function of the number of lattice nodes in the original network. The more nodes there are, the more MLLS appear for the algorithm's input. Thus, in the second algorithm, the computational cost is a function only of the size of the network and the number of leaves it contains.

In view of the above, the type of network and the amount of information available may be an important factor to take into

account when using one type of reconstruction or another, as it may involve a computational cost or even the impossibility of being able to carry out the total reconstruction depending on the data available and those that are not available.

VII. Conclusion

I hope that the realisation and implementation of these two algorithms will be a useful contribution to the world of computational phylogenetics. In part, the choice of a language such as `Python` has been to facilitate and share the work done in a greater way, so that it can be easily reproducible and understandable through the notebooks. Some of the basic network manipulation methods used in the algorithms are implemented in other languages such as `Julia` or `Java` [16], and parts in `Python` [4]. The problem with these is that in order to use possibly useful methods, the format of the network has to be adapted to the expected one, thus complicating the need to apply additional methods not originally implemented. Even so, the benefit of the possible methods against the difficulty of adaptation has not outweighed the gains that this brings, so we have opted for an implementation close to the problem addressed, optimising the operation based on what is expected to be obtained.

Along the same lines, the same problem has been seen with the `networkx` visualisation and data processing library. Better visualisations such as the eNewick [8] format allow a better visualisation of the network but make it difficult to work with it. However, as a possible future work, it is possible to transform a network in eNewick format into a format that the algorithm can work with, and the result can be transformed back into eNewick format.

As a possible future work that could be interesting in terms of packaging everything that has been done is the creation of a programme with a graphical interface in which different options can be obtained from the data. These could be reconstruction in the case of having parts of an original network, being able to compare, visualise, etc. In short, a hub in which the input method can be facilitated, and the output mode facilitated, so that we have a tool in which the difficult part is understanding the network and not using it. The algorithms developed are part of this idea, in which, depending on the data available, it can be reconstructed in one way or another.

References

[1] T. Agarwal, P. Gambette, and D. Morrison. Who is who in phylogenetic networks: Articles, authors and programs. urlhttp://phylnet.univ-mlv.fr, 2016.

[2] M. Bordewich and C. Semple. Determining phylogenetic networks from inter-taxa distances. *Journal of Mathematical Biology*, 73:283–303, 2016.

[3] M. Bordewich, C. Semple, and N. Tokac. Constructing tree-child networks from distance matrices. *Algorithmica*, 80(8):2240–2259, Aug. 2018.

[4] G. Cardona. Phylonetwork 2.1. urlhttps://pypi.org/project/phylonetwork/, 2020.

[5] G. Cardona, J. C. Pons, and C. Scornavacca. Correction: Generation of binary Tree-Child phylogenetic networks. *PLoS Comput. Biol.*, 15(10):e1007440, Oct. 2019.

[6] G. Cardona, F. Rossello, and G. Valiente. Comparison of tree-child phylogenetic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(4):552–569, 2009.

[7] G. Cardona, F. Rossello, and G. Valiente. Comparison of tree-child phylogenetic networks. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 6(4):552–569, oct 2009.

[8] G. Cardona and D. Sanchez. Phylonetwork's documentation. urlhttps://pythonhosted.org/phylonetwork/index.html, 2012.

[9] N. developers. Networkx. urlhttps://networkx.org/, 2021.

[10] L. Jetten and L. van Iersel. Nonbinary tree-based phylogenetic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 15(1):205–217, 2018.

[11] S. Kong, J. Carles Pons, L. Kubatko, and K. Wicke. Classes of Explicit Phylogenetic Networks and their Biological and Mathematical Significance. *arXiv e-prints*, page arXiv:2109.10251, Sept. 2021.

[12] B. Moret, L. Nakhleh, T. Warnow, C. Linder, A. Tholse, A. Padolina, J. Sun, and R. Timme. Phylogenetic networks: modeling, reconstructibility, and accuracy. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(1):13–23, 2004.

[13] Y. Murakami, L. van Iersel, R. Janssen, M. Jones, and V. Moulton. Reconstructing tree-child networks from reticulate-edge-deleted subnetworks. *Bull. Math. Biol.*, 81(10):3823–3863, Oct. 2019.

[14] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.*, 4(4):406–425, July 1987.

[15] C. Solís-Lemus, P. Bastide, and C. Ané. PhyloNetworks: A package for phylogenetic networks. *Mol. Biol. Evol.*, 34(12):3292–3298, Dec. 2017.

[16] C. Solís-Lemus and C. Ané. Phylonetworks.jl. urlhttps://juliapackages.com/p/phylonetworks, 2021.

[17] L. Van Iersel and V. Moulton. Trinets encode tree-child and level-2 phylogenetic networks. *Journal of Mathematical Biology*, 68(7):1707–1729, June 2014.