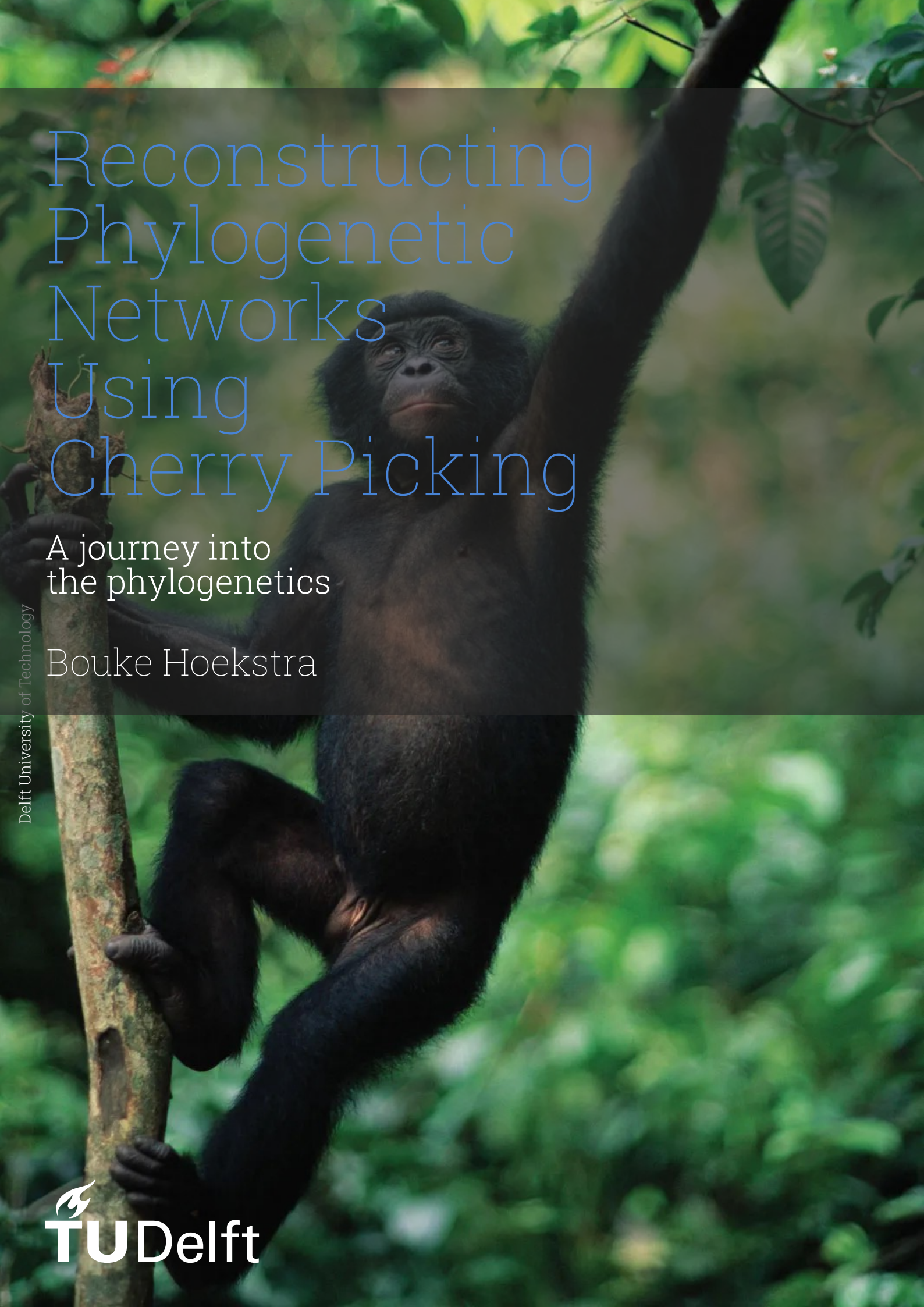


Reconstructing Phylogenetic Networks Using Cherry Picking

A journey into the phylogenetics

Bouke Hoekstra

Delft University of Technology

TU Delft

Reconstructing Phylogenetic Networks Using Cherry Picking

A journey into the phylogenetics

by

Bouke Hoekstra

Student Name	Student Number
Bouke Hoekstra	5412390

Instructors:	Leo van Iersel
	Esther Julien
Project Duration:	April, 2024 - July, 2024
Faculty:	EWI, Delft

TU Delft

Laymen's Summary

In the study of evolutionary biology, there exists a method called the “cherry picking algorithm” that produces the instructions needed to create a network that shows how different species are related. This report explores what happens when the algorithm starts with a wrong choice, or a “suboptimal cherry” for the first step of the algorithm, and how this affects the accuracy of the algorithm. Imagine you are trying to build a family tree for different species, but you start with a mistake. This research looks at how such initial mistakes can impact the accuracy of the entire family tree. We conducted this study using simulations of the algorithm that deliberately make an initial mistake, and afterwards continue as the algorithm would normally. The study found that starting with a wrong step in the algorithm usually makes the performance of the algorithm worse. Specifically, it lead to an average optimal performance decrease of 34,8% for networks relating a smaller number of species, and 11.3% for networks relating a larger number of species. Interestingly, the larger the number of species we are attempting to relate in the network produced by our algorithm, the less severe the impact of the initial mistake. We concluded that making an initial mistake negatively effects the average performance of the algorithm, and the extent of the effect varies with the number of species we are trying to relate in our network.

Summary

DNA is used as the primary tool of biological inheritance. DNA replication is the process by which DNA makes a copy of itself, and it occurs during reproduction among other things. During this replication of DNA, mutations can occur, We can model the mutation of DNA sequences using nucleotide substitution models, in Chapter 3 we discuss two of these: the Jukes-Cantor model and the Kimura-2 model. Many mutations of DNA sequences can ultimately give rise to the creation of new species.

Phylogenetics is the study of evolutionary history and relationships between groups of organisms, and these relationships are often determined by analyzing the DNA sequences and how they mutate from each other. The main tool used for displaying these evolutionary relationships are phylogenetic trees. In Chapter 4 we treat four algorithms for reconstructing trees: two distance-based methods and two character-based methods.

Phylogenetic trees however do not allow for more complex evolutionary processes, such as hybrid speciation or horizontal gene transfer for example, as they assume that species descend from only one ancestor. To address these kinds of processes, we use phylogenetic networks as an extension of traditional phylogenetic trees. These networks allow us to display more complex evolutionary processes, where we allow species to descend from multiple ancestors. In Chapter 5 we discuss a cherry picking algorithm that combines a set of phylogenetic trees into a phylogenetic network with the smallest number of reticulations, that displays all these trees. We discuss two different implementions of this algorithm: Rand and TrivialRand, and we analyze the differences of these two methods analytically.

After this, we researched the effect of picking a suboptimal cherry first in the cherry picking algorithm with TrivialRand, using several numerical simulations. Firstly, we researched the *average* output of the algorithm, and our simulations demonstrated that suboptimal picking led to an average increase of up to 11.3%, when we have a small optimal reticulation number. Then, we researched the effect of suboptimal picking on the *optimal* output, and we found that it led to an average increase of reticulations of up to 34.8%. Furthermore, our research found that there is a negative correlation between the effect of suboptimal picking and the optimal reticulation number. We found that in a small number of cases in our simulation, suboptimal picking did not have a negative effect on the amount of reticulations, this was due to limitations of the simulation, resulting in the simulation not picking a suboptimal cherry as a first cherry.

We concluded that suboptimal picking clearly has a negative impact on the performance of the algorithm, and the extent of this effect varies with the optimal reticulation number.

Contents

1

Introduction

A phylogenetic tree is a diagram that shows the lines of evolutionary descent of different species from a common ancestor. In Figure 1.1 we see an example of a phylogenetic tree. The leaves of the tree represent the species that we are trying to compare, and the interior nodes of the tree represent common ancesters. In this report we will be focussing on rooted binary trees, i.e. trees in which each node has at most two children, as we have in this example. In Chapter 4 we discuss different algorithms for reconstructing trees.

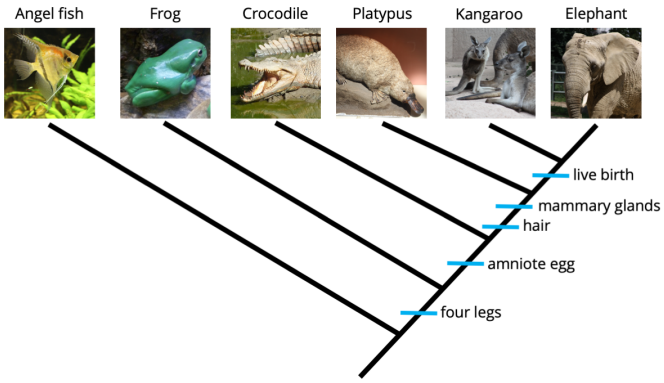


Figure 1.1: Rooted Phylogenetic tree. Figure by Paleontological Research Institution

Phylogenetic networks are a generalization of phylogenetic trees that allow for the representation of non-tree like evolutionary events, like hybrid speciation for example: where a new species is created from a combination of parental species. Thus in a network there can be nodes with multiple parents (in-degree > 1), which we will call *reticulations*. In this report we will only consider binary phylogenetic networks, where reticulations have exactly two parents, as in Figure 1.2.

A *rooted binary phylogenetic network* is a connected, directed acyclic graph where each vertex is either

1. the root : in-degree 0, out-degree 2
2. a tree node: in-degree 1, out-degree 2
3. a reticulation: in-degree 2, out-degree 1
4. a leaf : in-degree 1,out-degree 0

Where the in-degree is the number of edges incoming into a vertex, and the out-degree the number of edges leaving a vertex. Furthermore, each leaf is labeled uniquely by a taxon (species). A network is considered *tree-child* if each internal node has at least one child that is either another tree node or a leaf. So in a tree-child network a tree node can not have two reticulation as children, and a reticulations can not have another reticulation as a child.. A *normal network* is a type of tree-child network where,

additionally, the two parents of a reticulation node are always not comparable, one is not a descendant of the other. During the experiments conducted during our research, we focused on normal networks.

We will define the *parent* of a node v as the most recent ancestor of this node, and we will denote this by $P(v)$. A *cherry* in a network (or in a tree) is an ordered pair of leaves that share the same parent, i.e. (x, y) is a cherry if $P(x) = P(y)$. Notice that if (x, y) is a cherry, then so is (y, x). A *reticulated cherry* is an ordered pair of leaves (x, y), where the parent $P(x)$ of x is a reticulation, and the parent of $P(x)$ is a tree node, and the parent of y: $P(y)$.

In the network N_1 in Figure 1.2, we see that (d, c) and (d, e) are reticulated cherries. Notice that (x, y) is a reticulated cherry implies (y, x) is not a reticulated cherry. We will define reducing (picking) a cherry (x, y) in a network, as deleting x and the edge $(P(x), x)$, and replacing the edges $(P(P(x), P(x))$ and $(P(x), y)$ by a single edge, $(P(P(x)), y)$. Reducing a reticulated cherry (x, y) deletes the reticulation $P(x)$ and the edges $(P(y), P(x))$ and $(P(x), x)$, and replaces the other edge incoming to the reticulation $(z, P(x))$ with the edge (z, x). We say that a pair of leaves (x, y) is *reducible* in the network N if it is either a cherry or a reticulated cherry of N. In Figure 1.2, we see the action of reducing the cherry (b, a) from N_1, and then reducing the reticulated cherry (d, e) from N_2.

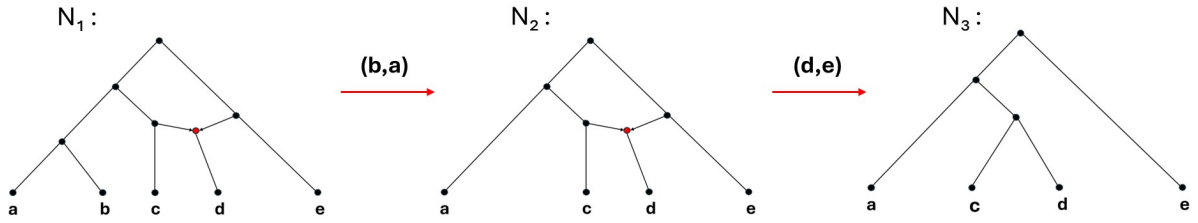


Figure 1.2: Picking cherries

We say that a sequence of cherries $S = (x_1, y_1), \ldots, (x_n, y_n)$, with $x_i \neq y_i$ for all i, is a *cherry-picking sequence* (CPS) if $y_i \in \{x_{i+1}, \ldots, x_n, y_n\}$ for all $i < n$, so each second leaf of a cherry is either a first leaf in a later pair, or the second leaf of the last cherry in the sequence. This is an important feature that will allow us to construct a network using our algorithm. Given a CPS S and a network N, we define N_S as the network obtained by reducing all cherries in S in order. Hence, $N_3 = N_{1(b,a)(d,e)}$ in Figure 1.2. We say that a CPS S *fully reduces* a network N, if N_S is a network consisting of only the root and one leaf. And we say that S fully reduces the set of networks (or trees) $\mathcal{N}$, if it fully reduces all networks $N \in \mathcal{N}$. Now we can define what it means for a network to display a set of trees. We say that the network N *displays* a set of trees $\mathcal{T}$ if a minimum-length CPS S that fully reduces N, also fully reduces $\mathcal{T}$. In this definition, minimum-length means that there does not exist a CPS of fewer cherries, that fully reduces N. This means that every reducible pair in N, must be a cherry in some $T \in \mathcal{T}$. Using this definition, a phylogenetic network can be interpreted as a tool to summarize a set of phylogenetic trees. In Figure 1.3(a) we see a network displaying a set of two trees, seen in Figure 1.3(b). Furthermore, for a tree set $\mathcal{T}$, a cherry (x, y) is defined as *trivial*, if it is a cherry in all trees that contain both x and y, and it's a cherry in at least one tree. We see that for the set of two trees in Figure 1.3(b), there are no trivial cherries.

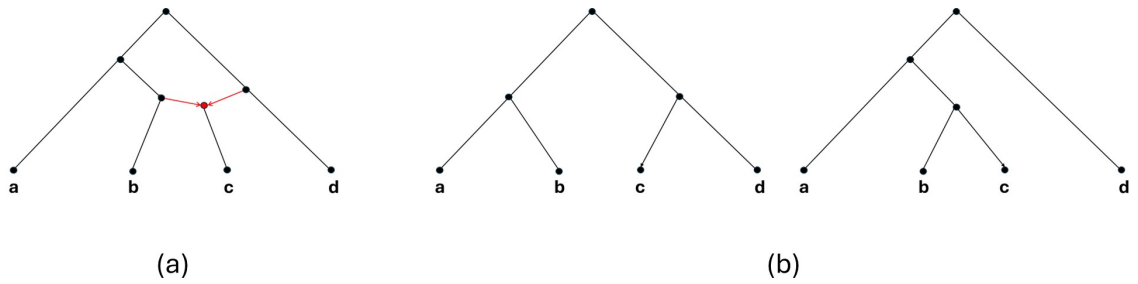


Figure 1.3: Phylogenetic network displaying two tree's

The **Hybridization problem** is the computational problem of combining a set of phylogenetic trees into a phylogenetic network, with the smallest number of reticulations, that displays this set. Solving the Hybridization problem is a major challenge in phylogenetics. We will discuss an algorithm that uses cherry

picking, in an attempt to solve this problem. It works by picking a CPS S, that fully reduces all the input trees, and then uniquely reconstructing the phylogenetic network, for which S is a minimum length CPS. This cherry picking heuristic algorithm, introduced by *G.Bernardini et al, in "Constructing phylogenetic networks via cherry picking and machine learning"* [1] has shown promise in producing a network with a (near) optimal reticulation number. We discuss this algorithm in detail in Chapter 5

Despite the effectiveness of the cherry picking algorithm, the impact of picking an suboptimal cherry as a first cherry, had not yet been investigated. Understanding this is important to understand and eventually improve the algorithm's reliability. Hence, the primary objective of our study was to investigate what the effect is of picking a suboptimal cherry as a first cherry, on the algorithm's performance. While there are more advanced and complex implementations of the cherry picking algorithm, we focused on the implementation of the cherry picking algorithm with TrivialRand (5.1.1), as we were interested in obtaining an initial understanding of the effect. In this implementation, the cherry picking heuristic algorithm picks trivial cherries if these exist, and otherwise pick a cherry uniformly at random. Our expectation was that initiating the cherry picking algorithm with a suboptimal cherry would increase the number of reticulations in the network, obtained using the CPS of the algorithm.

To conduct this study, we investigated the cherry picking algorithm analytically, to obtain a preliminary insight of what the potential impact could be of initiating the cherry picking algorithm with a suboptimal cherry. After this, we simulated scenarios where the cherry picking algorithm is firstly executed ordinarily and then deliberately guided to pick a suboptimal cherry first, and we compared these results.

2

Phylogenetics

Dna is the genetic information inside an organism's cells that makes any living organism unique. It contains the instructions needed for an organism to develop, survive and reproduce [2]. Your DNA determines what color eyes you have, how tall you are and how susceptible you are to various health conditions. DNA is composed of two linked strands that wind around each other in the form of a double helix [3]. Each strand consists of a sequence of four nucleotides – adenine, thymine, guanine and cytosine – forming pairs with their counterpart nucleotide in the other strand. In these linked strands, adenine is always linked to thymine, and guanine is always linked to cytosine, as can be seen in Figure 2.1.

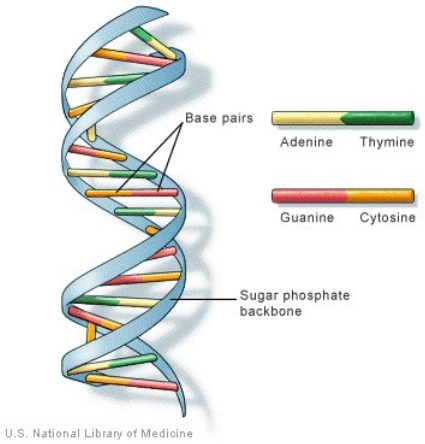


Figure 2.1: Structure of DNA. Figure by the National Library of Medicine (USA)

The order of the nucleotides, commonly referred to as bases, is the encoding that distinguishes each individual. The DNA of humans consist of around three million nucleotide bases, and more than 99 percent of these are the same for humans.

DNA does more than specify the structure of a living organism, it is also used as the primary tool of biological inheritance. During birth you inherit approximately half of the DNA of your mother, and half of the DNA of your father, by DNA replication. DNA replication is the process by which DNA makes a copy of itself, and it occurs during cell division. Therefore, it occurs during reproduction, but also during the repair of your body, for instance for the repair of damaged tissues. This replication of DNA doesn't occur flawlessly, it may happen that the wrong DNA is copied, or part of the DNA may be forgotten, these mistakes are called mutations and give rise to the creation of new species.

Phylogenetics is the study of evolutionary history and relationships within groups of organisms. These relations are often determined by looking at DNA sequences and how they mutate from each other. Phylogenetics is important because it gives us a better understanding of how species, or genes, evolve.

Through phylogenetics we don't only learn how the organisms have mutated and evolved to become what they are now, but it also enables us to predict how they will evolve in the future. These evolutionary relationships are usually shown in phylogenetic trees, which we introduced in Chapter 1. Every time the tree splits can be seen as the evolutionary process of DNA mutations, ultimately creating a new species. Therefore the modelling of DNA mutations is of vital importance. In the next Chapter, we will look into nucleotide substitution models, which model the mutation of DNA sequences over time.

3

Nucleotide Substitution Models

DNA replication is the process by which DNA makes a copy of itself, and it occurs during cell division, as stated in Chapter 2. During this replication of DNA sequences, a number of mutations can occur:

1. Substitutions: a nucleotide is replaced by another nucleotide.
   For example $ACT \longrightarrow AGT$.
2. Insertions: a nucleotide is inserted in the sequence.
   For example $ACT \longrightarrow ACGT$.
3. Deletions: a nucleotide is deleted from the sequence.
   For example $ACT \longrightarrow AT$.

These replication errors are one of the most important processes that drive evolution and genetic changes. In this chapter, we will study and analyze nucleotide substitution models, where we assume that there are no insertions or deletions during the replication process. The nucleotide substitution models we conmsider, are discrete-time markov models with the following assumptions:

- Only substitutions occur along the evolutionary process
- Each site in the sequence evolves independently of the other sites and with the same probabilities
- The substitution process is the same on each time step and does not depend on the past given the present

This last assumption is called the Markov assumption and mathematically is seen as

$$P(X_{n+1} = x_{n+1} \mid X_n = x_n, X_{n-1} = x_{n-1}, \ldots, X_0 = x_0) = P(X_{n+1} = x_{n+1} \mid X_n = x_n)$$

3.1. General nucleotide substution model

These models will be described with a transition matrix M, and an ancestral distribution vector p_0. These represent the probabilities of the substitutions and the ancestral sequence, respectively. Thus, the general nucleotide substitution model is given as:

$$M = \begin{bmatrix} p_{A,A} & p_{A,C} & p_{A,G} & p_{A,T} \\ p_{C,A} & p_{C,C} & p_{C,G} & p_{C,T} \\ p_{G,A} & p_{G,C} & p_{G,G} & p_{G,T} \\ p_{T,A} & p_{T,C} & p_{T,G} & p_{T,T} \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \qquad p_0 = \begin{bmatrix} p_A^0 \\ p_C^0 \\ p_G^0 \\ p_T^0 \end{bmatrix}$$

Where $p_{N,M}$ gives the probability of a substitution from base N to base M, and p_N^0 gives the probability of the base N being in the starting DNA sequence. In this case we don't assume any structure on the transition matrix or ancestral distribution vector. The only constraint of the general transition matrix is that the rows sum up to 1, as it is a matrix of probabilities: a base must either be replicated without errors, or substitute to another base. In the same manner, the column of the distribution vector must sum up to 1. In the rest of this chapter, we will discuss two nucleotide substitution models with a certain structure in the transition matrices and ancestral distribution vectors.

3.2. Jukes-Cantor model

The Jukes-Cantor model is a simple nucleotide substitution model that assumes that all substitutions between the bases have exactly the same probability $\frac{a}{3}$ of occurring. The model is given by the following transition matrix and ancestral distribution vector [4].

$$M = \begin{bmatrix} p_{A,A} & p_{A,C} & p_{A,G} & p_{A,T} \\ p_{C,A} & p_{C,C} & p_{C,G} & p_{C,T} \\ p_{G,A} & p_{G,C} & p_{G,G} & p_{G,T} \\ p_{T,A} & p_{T,C} & p_{T,G} & p_{T,T} \end{bmatrix} = \begin{bmatrix} 1-a & \frac{a}{3} & \frac{a}{3} & \frac{a}{3} \\ \frac{a}{3} & 1-a & \frac{a}{3} & \frac{a}{3} \\ \frac{a}{3} & \frac{a}{3} & 1-a & \frac{a}{3} \\ \frac{a}{3} & \frac{a}{3} & \frac{a}{3} & 1-a \end{bmatrix} \qquad p_0 = \begin{bmatrix} p_A^0 \\ p_C^0 \\ p_G^0 \\ p_T^0 \end{bmatrix} = \begin{bmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \end{bmatrix}$$

In this model, the transition matrix is called a Jukes-Cantor matrix, with parameter $a \in [0, 1]$. As you can see, every substitution has the same probability $\frac{a}{3}$, and the four bases are uniformly distributed, so we expect approximately an equal frequency of the four bases in the original DNA sequence. You can also easily check that the sum of the rows of the matrix and the sum of the column of the vector are equal to one, as stated in section 3.1. The probability of a mutation is considerably low, therefore in practice we expect the Jukes-Cantor parameter a to be close to 0. Furthermore, the Jukes-Cantor parameter a represents the probability of observing any substitution at a certain position after one time step. To see this, observe the following calculation:

$$P(\text{substitution}) = \sum_{\substack{N,M \in \Sigma \\ N \neq M}} p_N^0 \cdot P(N \to M) = \sum_{\substack{N,M \in \Sigma \\ N \neq M}} \frac{1}{4} \cdot \frac{a}{3} = 12 \cdot \frac{a}{12} = a$$

where $\Sigma = \{\text{A}, \text{C}, \text{G}, \text{T}\}$ is the set of nucleotide bases. Now that we know the structure of the Jukes-Cantor model, we will discuss an application on a DNA sequence.

Example.
Suppose that we have the following DNA sequence of 40 bases S_0 that has mutated to become the sequence S_1:

S_0 *TGTGCAGCATAACTGCGTGTATCCAGCTAGTATCATGACG*
S_1 *TGTCCAGCATAAAGGCGTGTATCCTGCTAGTATCAAGACG*

Notice that five (visible) substitutions have occurred:

1. $G \longrightarrow C$ *on site 4*
2. $C \longrightarrow A$ *on site 13*
3. $T \longrightarrow G$ *on site 14*
4. $A \longrightarrow T$ *on site 25*
5. $T \longrightarrow A$ *on site 36*

Thus, in five of the fourty sites of the DNA sequence, a substitution has occurred, and in the rest of the sites the DNA replication has been without errors. Therefore, we can estimate the Jukes-Cantor parameter a by the relative frequency of substitutions $\frac{5}{40}$, yielding the Jukes-Cantor matrix below to model our DNA sequence over time.

$$M = \begin{bmatrix} \frac{21}{24} & \frac{1}{24} & \frac{1}{24} & \frac{1}{24} \\ \frac{1}{24} & \frac{21}{24} & \frac{1}{24} & \frac{1}{24} \\ \frac{1}{24} & \frac{1}{24} & \frac{21}{24} & \frac{1}{24} \\ \frac{1}{24} & \frac{1}{24} & \frac{1}{24} & \frac{21}{24} \end{bmatrix}$$

Furthermore, notice that in our original DNA sequence, there are 10 G's and A's, 11 T's and 9 $C's$, so the bases are almost uniformly distributed over the fourty sites, in line with what we would expect according to the Jukes-Cantor ancestral distribution vector.

3.3. Kimura-2 model

In this section, we will be discussing a generalization of the Jukes-Cantor nucleotide substitution model: the Kimura 2 (parameter) substitution model [4]. It takes into account the structure of the DNA bases. We can divide the bases into two types based on their structure [5]:

1. Purines: consists of the DNA bases Adenine and Guanine, these have a double-ring structure
2. Pyrimidines: consists of Cytosine and Thymine (and Uracil), these have a single-ring structure

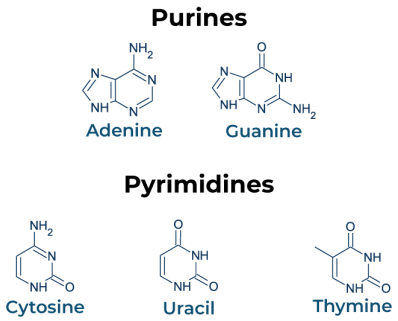


Figure 3.1: Purines and Pyrimidines structure, Figure by geeksforgeeks.org

These nucleotide types further give rise to two types of substitutions. Transitiitions [6] are substitutions between a purine and a purine ($A \leftrightarrow G$), and between a pyridine and a pyridine ($C \leftrightarrow T$). Transversions are substitutions between a purine and a pyridine, or vice-versa . In this model we assume that all transitions have the same probability: α, and all transversions have the same probability: β. In practice, transitions are observed to occur more frequently than transitions [6], therefore we would expect the transition parameter α to be larger than the transition parameter β

$$M = \begin{bmatrix} p_{A,A} & p_{A,C} & p_{A,G} & p_{A,T} \\ p_{C,A} & p_{C,C} & p_{C,G} & p_{C,T} \\ p_{G,A} & p_{G,C} & p_{G,G} & p_{G,T} \\ p_{T,A} & p_{T,C} & p_{T,G} & p_{T,T} \end{bmatrix} = \begin{bmatrix} 1-2\beta-\alpha & \beta & \alpha & \beta \\ \beta & 1-2\beta-\alpha & \beta & \alpha \\ \alpha & \beta & 1-2\beta-\alpha & \beta \\ \beta & \alpha & \beta & 1-2\beta-\alpha \end{bmatrix} \qquad p_0 = \begin{bmatrix} p_A^0 \\ p_C^0 \\ p_G^0 \\ p_T^0 \end{bmatrix} = \begin{bmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \end{bmatrix}$$

Example.
Suppose that we have the following DNA sequence of 40 bases S_0 that has mutated to become the sequence S_1:

S_0 *TGTGCAGCATAACTGCGTGTATCCAGCTAGTATCATGACG*
S_1 *TGTACAGCATAATGGCGTGTATCCGGCTAGTATCACGACG*

1. $G \longrightarrow A$ *transition*
2. $C \longrightarrow T$ *transition*
3. $T \longrightarrow G$ *transversion*
4. $A \longrightarrow G$ *transition*
5. $T \longrightarrow C$ *transition*

Notice that four transitions have occurred and one transversion. We can estimate the transition probability and the transversion probabilities by their relative frequencies, yielding the following transition matrix to model the mutation of the DNA sequence S_0 over time:

$$M = \begin{bmatrix} \frac{34}{40} & \frac{1}{40} & \frac{4}{40} & \frac{1}{40} \\ \frac{1}{40} & \frac{34}{40} & \frac{1}{40} & \frac{4}{40} \\ \frac{4}{40} & \frac{1}{40} & \frac{34}{40} & \frac{1}{40} \\ \frac{1}{40} & \frac{4}{40} & \frac{1}{40} & \frac{34}{40} \end{bmatrix}$$

Observe that the transition probability is larger than the transversion probability in this example, in line with what we would expect in practice.

4

Phylogenetic Tree Reconstruction Methods

As stated in Chapter 2, phylogenetic tree's are diagrams that show the lines of evolutionary descent of different species from a common ancestor. The nodes represent species (or other organisms) and the edges represent DNA mutation processes between the species. In this report we will only consider binary tree's, i.e. tree's in which each node has either zero children (leaves), or two children (internal nodes). There are various methods known for reconstructing phylogenetic trees. We will discuss two types of phylogenetic reconstruction methods: distance-based methods and character based methods.

4.1. Distance-based methods

Distance-based methods use a notion of genetic "distance" between species to infer their evolutionary relationships. These methods assume that the "further away" two species are, so how larger the genetic differences, the longer the time since their last common ancestor. These pairwise "distances" between species can be calculated in various manners, but they are often calculated based on the difference between DNA sequence alignments. Distance-based methods have a matrix of pairwise distances between species as input, and obviously a phylogenetic tree as an output. Distance-based methods are often very fast, as we reduce all the information of the difference between DNA sequences to a distance, and so the input data is relatively small, however this reduction of data to a distance could lead to the loss of information. We will be discussing two distance-based methods: The Neighbour-Joining algorithm and the UPGMA algorithm.

4.1.1. Neighbour-Joining algorithm

The Neighbour-Joining algorithm is a distance-based method that reconstructs trees by iteratively joining the closest pairs of organisms based on their genetic distance [7]. It starts by identifying the species pair with the smallest 'distance', and placing a common ancestor for these species. You then recalculate the distances between this common ancestor and the remaining species, to obtain a new distance matrix. After this, you repeat this process until there is only a single pair of species left.

Algorithm 1 Neighbor-Joining Algorithm

INPUT : distance matrix D
OUTPUT : unrooted binary phylogenetic tree

Step 1 : Initiation
Initialize the tree with each taxon as a separate node (leaf).

Step 2 : Iteration
Construct an $n \times n$ matrix, Q_d, whose entries are given by:

$$Q_D(i,j) = (n-2) \cdot d(i,j) - \sum_{k \neq i} d(i,k) - \sum_{k \neq j} d(j,k), \tag{4.1}$$

for each $i, j \in [n]$ with $i \neq j$, and set $Q_D(i,i) = \infty$. Identify the pair of leaves (x, y) that minimizes $Q_d(x, y)$. Define a new node z, and join the taxa x and y to z in the tree, using the following distances

$$\begin{cases} d(x,z) = \dfrac{1}{2} d(x,y) + \dfrac{1}{2(n-2)} \left(\displaystyle\sum_{k \neq x} d(x,k) - \sum_{k \neq y} d(y,k) \right) \\ d(y,z) = d(x,y) - d(x,z) \end{cases} \tag{4.2}$$

Calculate the distance matrix again, where we remove the leaves x and y from the matrix, and calculate the distance from z to each of the remaining taxa using:

$$d(z,u) = \frac{1}{2}(d(x,u) + d(y,u) - d(x,y)) \tag{4.3}$$

Therefore obtaining a new $(n-1) \times (n-1)$ distance matrix D_z.

Step 3 : Termination
The iteration process is repeated until only two taxa remain. The distance between these final two taxa defines the last branch of the phylogenetic tree.

Example. *Suppose we have the following distance matrix between four species:*

$$D = \begin{bmatrix} 0 & 1.1 & 1.0 & 1.4 \\ 1.1 & 0 & 0.3 & 1.3 \\ 1.0 & 0.3 & 0 & 1.2 \\ 1.4 & 1.3 & 1.2 & 0 \end{bmatrix}$$

Consider matrix D, of dimension $n = 4$. For $1 \leq i, j \leq 4$, we compute $Q_D(i,j)$ as:

$$Q_D(i,j) = (n-2) \cdot d(i,j) - \sum_{k \neq i} d(i,k) - \sum_{k \neq j} d(j,k)$$

We have:

$$\begin{aligned} Q_D(1,2) &= 2 \cdot 1.1 - (1.1 + 1.0 + 1.4) - (1.1 + 0.3 + 1.3) = -4, \\ Q_D(1,3) &= 2 \cdot 1.0 - (1.0 + 1.1 + 1.4) - (1.0 + 0.3 + 1.2) = -4, \\ Q_D(1,4) &= 2 \cdot 1.4 - (1.0 + 1.1 + 1.4) - (1.4 + 1.3 + 1.2) = -4.6, \\ Q_D(2,3) &= 2 \cdot 0.3 - (1.1 + 0.3 + 1.3) - (1.0 + 0.3 + 1.2) = -4.6, \\ Q_D(2,4) &= 2 \cdot 1.3 - (1.1 + 0.3 + 1.3) - (1.4 + 1.3 + 1.2) = -4, \\ Q_D(3,4) &= 2 \cdot 1.2 - (1.0 + 0.3 + 1.2) - (1.4 + 1.3 + 1.2) = -4. \end{aligned}$$

We set $Q_D(i,i) = \infty$ for all i, in order to ensure that the minimum can never be found in the diagonal.

Therefore, the matrix Q_D is:

$$Q_D = \begin{bmatrix} \infty & -4 & -4 & -4.6 \\ -4 & \infty & -4.6 & -4 \\ -4 & -4.6 & \infty & -4 \\ -4.6 & -4 & -4 & \infty \end{bmatrix}$$

We can see that the minimum of the entries of this matrix corresponds to $Q_D(1,4)$ and $Q_D(2,3)$. Without loss of generality, we choose leaves 1 and 4 to be the first found cherry of the tree. To continue with the algorithm, we redefine leaves 1 and 4 into a new leaf $X_1 = (1,4)$ and compute the new distances from leaves 2 and 3 to X_1.

$$d(X_1, 2) = \frac{1}{2}\left(d(1,2) + d(4,2) - d(1,4)\right) = \frac{1}{2}(1.1 + 1.3 - 1.4) = 0.5$$

$$d(X_1, 3) = \frac{1}{2}\left(d(1,3) + d(4,3) - d(1,4)\right) = \frac{1}{2}(1.0 + 1.2 - 1.4) = 0.4$$

Moreover, if we calculate the branch length between the new node X_1 and the leaves 1 and 4, we have:

$$d(1, X_1) = \frac{1}{2}d(1,4) + \frac{1}{4}\left(\sum_{k\neq 1} d(1,k) - \sum_{k\neq 4} d(4,k)\right) = 0.6$$

$$d(4, X_1) = d(1,4) - d(1, X_1) = 1.4 - 0.6 = 0.8$$

Now, for the tree with nodes $X_1, 2, 3$ we have the new distance matrix:

$$D_{X_1} = \begin{bmatrix} 0 & 0.5 & 0.4 \\ 0.5 & 0 & 0.3 \\ 0.4 & 0.3 & 0 \end{bmatrix}$$

Let us now compute the off-diagonal values of the matrix Q_{X_1}:

$$\begin{aligned} Q_{X_1}(X_1, 2) &= 0.5 - (0.5 + 0.4) - (0.5 + 0.3) = -1.2, \\ Q_{X_1}(X_1, 3) &= 0.4 - (0.5 + 0.4) - (0.4 + 0.3) = -1.2, \\ Q_{X_1}(2, 3) &= 0.3 - (0.5 + 0.3) - (0.4 + 0.3) = -1.2. \end{aligned}$$

So,

$$Q_{X_1} = \begin{bmatrix} \infty & -1.2 & -1.2 \\ -1.2 & \infty & -1.2 \\ -1.2 & -1.2 & \infty \end{bmatrix}$$

Thus, we arbitrarily choose the cherry $(X_1, 2)$ as the following one. Let us denote this new node as Y_1, and calculate the remaining distances:

$$\begin{aligned} d(Y_1, 3) = \frac{1}{2}\left(d(X_1,3) + d(2,3) - d(X_1,2)\right) &= \frac{1}{2}(0.4 + 0.3 - 0.5) = 0.1 \\ d(X_1, Y_1) = \frac{1}{2}d(X_1,2) + \frac{1}{2}\left(\sum_{k\neq X_1} d(X_1,k) - \sum_{k\neq 2} d(2,k)\right) &= 0.3 \\ d(2, Y_1) = d(X_1,2) - d(X_1, Y_1) &= 0.2 \end{aligned}$$

With this, only two taxa Y_1 and 3 are left, and thus we implement the termination step, and we are ready to draw the phylogenetic tree corresponding to our initial distance matrix D_1.

The NJ-algorithm reconstructs the unrooted binary phylogenetic tree seen in figure 4.1. Notice that all the distances from our initial distance matrix D are correctly displayed in our tree.

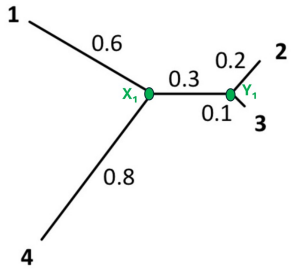


Figure 4.1: Output of the NJ-algorithm

4.1.2. UPGMA algorithm

The UPGMA (Unweighted Pair Group Method with Arithmetic mean) algorithm is a simple distance-based phylogenetic reconstruction method that works by iteratively joining groups of nodes, which we'll call 'clusters', that are closest to each other, using the average distance between the nodes in the clusters [8] . Thus, we define the distance between clusters C_1 and C_2 as:

$$d(C_i, C_j) = \frac{1}{|C_i| \cdot |C_j|} \sum_{p \in C_i, q \in C_j} d_{pq}, \tag{4.4}$$

where $|C_i|$ and $|C_j|$ are the amount of nodes in the clusters i and j, respectively. Using the UPGMA algorithm, we reconstruct rooted binary phylogenetic tree's from distance matrices. In the tree's constructed by the UPGMA algorithm, the only distance that we consider is the vertical distance between nodes, and we disregard the horizontal distance in the tree's.

Algorithm 2 UPGMA algorithm

INPUT : distance matrix D
OUTPUT : rooted binary phylogenetic tree

Step 1 : Initialisation
1. Assign each node i to its own cluster C_i, and place this leaf at height zero

Step 2 : Iteration
1. Determine the two clusters i, j for which the distance between these is the smallest
2. Define a new cluster k by $C_k = C_i \cup C_j$, and define d_{ku} for all u
3. Define a node k with descendants i and j and place this node at height $\frac{d_{ij}}{2}$
4. Replace clusters C_i and C_j with C_k

Step 3 : Termination
1. The algorithm terminates when only two clusters C_i and C_j remain, and then we place the root at height $\frac{d_{ij}}{2}$

Example. *Suppose that we have the following distance matrix between 4 species S_1 , S_2 , S_3 and S_4, and we want to compute the corresponding phylogenetic tree*

$$D = \begin{bmatrix} 0 & 1.3 & 1.2 & 1.4 \\ 1.3 & 0 & 0.3 & 0.8 \\ 1.2 & 0.3 & 0 & 1.6 \\ 1.4 & 0.8 & 1.6 & 0 \end{bmatrix}$$

The algorithm initiates by assigning all species to their own cluster and placing these at height zero as leaves:

After this, the algorithm determines the two clusters for which the distance between these are the smallest. In the distance matrix D, the smallest distance between clusters is 0.3, between clusters C_2 and C_3. Thus we define a new cluster $C_5 = C_2 \cup C_3$, and calculate the new distances:

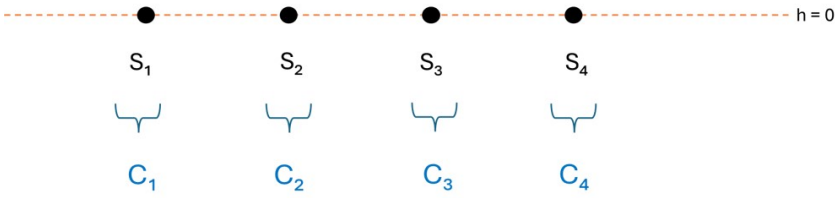


Figure 4.2: Initiation step UPGMA

- $d_{C_5,C_1} = \frac{d_{2,1}+d_{3,1}}{2} = 1.25$
- $d_{C_5,C_4} = \frac{d_{2,4}+d_{3,4}}{2} = 1.2$

Thus we place a node with descendants S_2 and S_3 at height $\frac{d_{1,2}}{2} = 0.15$, and obtain the following distance matrix

$$D = \begin{bmatrix} 0 & 1.25 & 1.4 \\ 1.25 & 0 & 1.2 \\ 1.4 & 1.2 & 0 \end{bmatrix}$$

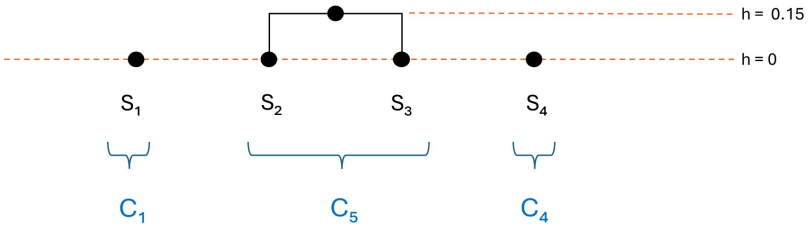


Figure 4.3: Iteration step UPGMA

Now we see that the smallest distance is 1.2, between clusters C_5 and C_4. Thus, we define a new cluster $C_6 = C_5 \cup C_4$, calculate the new distance

- $d_{C_6,C_1} = \frac{d_{2,1}+d_{3,1}=d_{4,1}}{3} = 1.3$

and place a node between clusters C_5 and C_4 at height $\frac{d_{C_5,C_4}}{2} = 0.6$

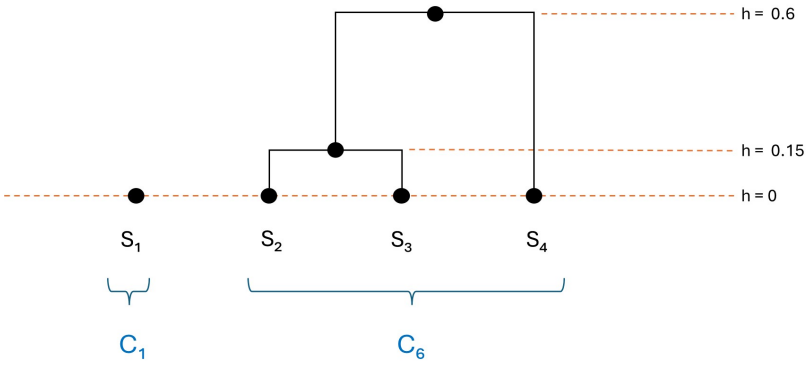


Figure 4.4: Iteration step UPGMA

Now only the clusters C_1 and C_6 remain, with distance matrix

$$D = \begin{bmatrix} 0 & 1.3 \\ 1.3 & 0 \end{bmatrix}$$

So the termination step begins, and we place a root at height $\frac{d_{C_1,C_6}}{2} = 0.65$ to obtain the following rooted phylogenetic tree

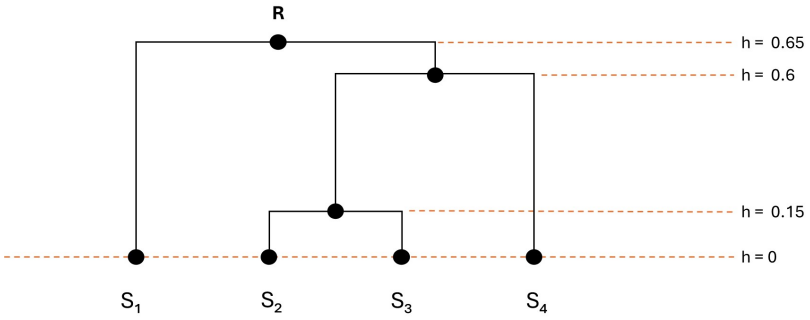


Figure 4.5: Termination step UPGMA

4.2. Character-based methods

Character-based methods for reconstructing phylogenetic trees analyze DNA sequence alignments directly, considering each individual position of the nucleotides in the sequence alignment. Using the whole DNA sequence gives us a more detailed and accurate tree, as we have more information to determine the evolutionary relationships on. However more information of course generally make these methods computionally more expensive; these methods are often slower than distance based methods, especially for large datasets.

4.2.1. Maximum Parsimony method

The Maximum Parsimony method (MP) is a simple character-based reconstruction method that aims to determine the evolutionary tree that minimizes the number of mutations in the DNA sequences. We will discuss the Fitch-Hartigan algorithm [9], which calculates the parsimony score of a specific tree topology.

The Maximum Parsimony method works as follows. Given a DNA alignment sequence of k species, each sequence consisting of n nucleotide bases. Firstly, we build all possible tree topology's on k leaves, and calculate the parsimony score of all these possible tree's using the Fitch-Hartigan algorithm. The MP method then supposes that the tree topology with the smallest parsimony score, therefore the least amount of mutations, is the most probable phylogenetic tree displaying the species from our sequence alignment.

Let $\mathcal{X}_i$ be the i'th column of the sequence alignment. So this corresponds to a vector $\mathcal{X}_i = (x_1, x_2, \ldots, x_n)$, where x_a is the i'th nucleotide of the a'th sequence.

Algorithm 3 Fitch-Hartigan algorithm

INPUT : DNA sequence alignment and a tree topology
OUTPUT : parsimony score of this tree topology

Step 1 : Initialisation

1. If the tree topology is unrooted, arbitrarily introduce a root to obtain a binary rooted tree

Step 2 : Iteration:

For each character alignment $\mathcal{X}_i$

1. Assign to each node v in the tree, a pair (A, n), where $A \in \{A, C, G, T\}$ and $m \in \mathbb{Z}_{\geq 0}$

   (a) To each leaf x in the tree, assign the pair $(\mathcal{X}(x), 0)$
   (b) Let u, v be the two children of v, with assigned pairs (A_1, n_1) of u and (A_2, n_2) of v, then assign to v the pair

$$(A, n) = \begin{cases} (A_1 \cup A_2, n_1 + n_2 + 1), & \text{if } A_1 \cap A_2 = \varnothing \\ (A_1 \cap A_2, n_1 + n_2), & \text{otherwise} \end{cases}$$

   Repeat this until all nodes have been assigned a pair. If the root has been assigned pair (A,n), then the parsimony score of the character alignment $\mathcal{X}_i$: $PS_{\mathcal{X}_i} = n$

Step 3 : Termination

1. If the parsimony score of all character alignments have been computed. The parsimony score of the tree is

$$PS(T) = \sum_i PS_{\mathcal{X}_i}$$

Example. *Let us construct the phylogenetic tree for 4 species S_1, S_2, S_3 and S_4, where we assume for simplicity that their DNA sequences are of length 4*

	$\mathcal{X}_1$	$\mathcal{X}_2$	$\mathcal{X}_3$	$\mathcal{X}_4$
S_1	*T*	*C*	*A*	*T*
S_2	*T*	*C*	*A*	*T*
S_3	*A*	*C*	*A*	*G*
S_4	*A*	*G*	*A*	*G*

The Maximum Parsimony method would proceed by firstly building all possible tree topologies on 4 leaves, two of which T_1 and T_2 can be seen in figure 4.6. Afterwards, it calculates the parsimony score of all the possible tree topologies, using the Fitch-Hartigan algorithm. In figure 4.6, you can see the implementation of the Fitch-Hartigan algorithm on the two trees T_1 and T_2

The parsimony score of

- $T_1 = PS_{\mathcal{X}_1} + PS_{\mathcal{X}_2} + PS_{\mathcal{X}_3} + PS_{\mathcal{X}_4} = 1 + 1 + 0 + 1 = 3$
- $T_2 = PS_{\mathcal{X}_1} + PS_{\mathcal{X}_2} + PS_{\mathcal{X}_3} + PS_{\mathcal{X}_4} = 2 + 1 + 0 + 2 = 5$

Therefore, The Maximum Parsimony method tells us that T_1 is the better tree topology. Notice that in T_1 species S_1 and S_2 are evolutionally 'closer', than in T_2, which agrees with what we would expect as species S_1 and S_2 have the same DNA squence. The MP method would proceed by calculating the parsimony score of all of the other possible tree topologies.

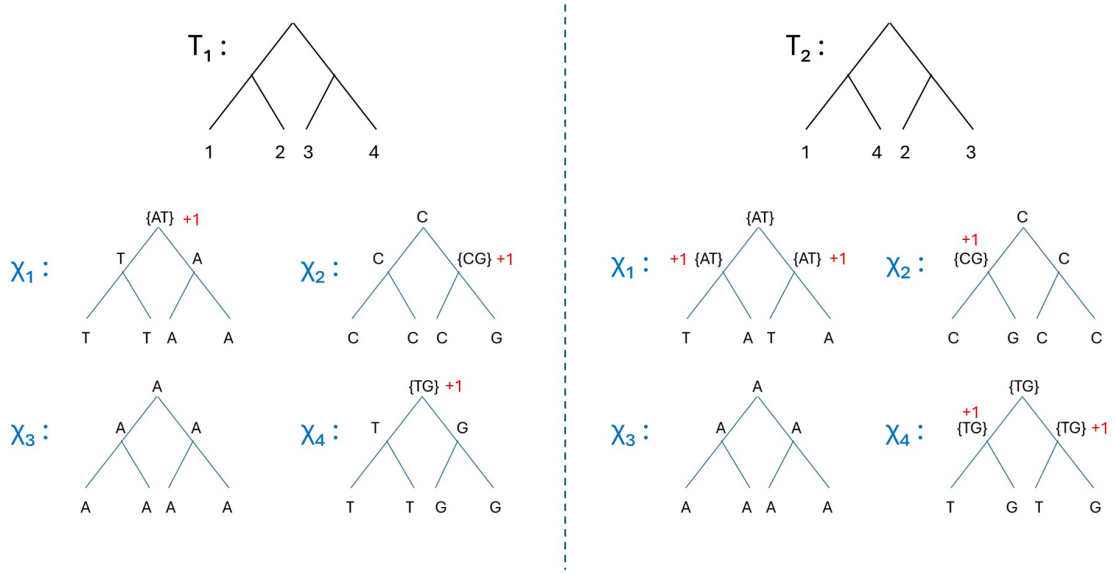


Figure 4.6: Fitch-Hartigan algorithm on two tree's T_1 and T_2

4.2.2. Maximum Likelihood Method

The maximum likelihood is a character-based reconstruction method that constructs a phylogenetic tree using the so-called Felsenstein Pruning Algorithm [9]. This algorithm computes the probability, or 'likelihood', of a specific DNA alignment. As in the Fitch-Hartigan algorithm of MP method, the vertices in the model will be assigned nucleotide bases, in the set $\{A, C, G, T\}$. To be able to calculate the likelihood of a certain alignment, the edges will contain transition matrices corresponding to a certain evolution model, e.g. a nucleotide substitution model, that calculate the probabilities of mutations, and the root r will have an ancestral distribution vector π_r, as in chapter 3.The Felsenstein Pruning algorithm works as follows; Given an alignment of n species with DNA sequences of m nucleotides,

$$\begin{aligned} S_1 &= s_1^1, s_1^2, \ldots, s_1^m \\ S_2 &= s_2^1, s_2^2, \ldots, s_2^m \\ &\vdots \\ S_n &= s_n^1, s_n^2, \ldots, s_n^m \end{aligned}$$

and given a specific tree topology T, root distrubution vector π_r, and transition matrices for each edge M_e, the Felsenstein Pruning algorithm computes the probability of the alignment in our tree

$$P(S_1, S_2, \ldots, S_n \mid T, \pi_r, M_e)$$

which we will henceforth denote as the likelihood $L(S_1, S_2, \ldots, S_n)$ of the alignment.

Let $\mathcal{X}_i$ once again be the i'th column of the sequence alignment: $\mathcal{X}_i = (s_1^i, s_2^i, \ldots, s_n^i)$ We will calculate the likelihood $L(S_1, S_2, \ldots, S_n)$ of the entire DNA alignment using the following assumption that we have already seen for nucleotide substituion models: all sites evolve independently of each other. Therefore the likelihood of the entire DNA sequence alignment is the product of the likelihood of each character alignment $\mathcal{X}_i$.

$$L(S_1, S_2, \ldots, S_n) = P(S_1, S_2, \ldots, S_n \mid T, \pi_r, M_e) = \prod_{i=1}^{m} P(\mathcal{X}_i \mid T, \pi_r, M_e)$$

Algorithm 4 Felsenstein Pruning algorithm

INPUT : DNA sequence alignment, tree topology, root distribution vector and transition matrices for each edge

OUTPUT : likelihood of this tree topology

Step 1 : Initialisation

For each character alignment $\mathcal{X}_i$

1. Label the leaves of the tree topology: x, corresponding to the associated nucleotide in the character alignment $\mathcal{X}_i(x)$
2. label each internal node, and label each edge with the corresponding transition matrix

Step 2 : Iteration:

For each character alignment $\mathcal{X}_i$

1. Calculate recursively the probabilities of the subtrees with root v_n, working from the leaves upwards to the root r:
   Let u, s be the children of v_n, where edge (v_n, u) has transition matrix M_u and edge v_n, s has matrix M_s
   (a) If u, s are leaves of nucleotides N, M:

$$L_i^{v_n}(x) = (M_u)_{x,N}(M_s)_{x,M}$$

   (b) if u is an internal node, and s is a leaf of nucleotide M:

$$L_i^{v_n}(x) = (\sum_{P\in\{A,C,G,T\}} (M_u)_{x,P}L_i^u(P)\ ) * (M_s)_{x,M}$$

   (c) if u, s are internal nodes:

$$L_i^{v_n}(x) = (\sum_{P\in\{A,C,G,T\}} (M_u)_{x,P}L_i^u(P)\ ) * (\sum_{P\in\{A,C,G,T\}} (M_s)_{x,P}L_i^s(P)\ )$$

2. After we have calculated the partial likelihood of the entire tree (subtree with root r). We say that the likelihood of the character alignment $\mathcal{X}_i$

$$L(\mathcal{X}_i) = \sum_{P\in\{A,C,G,T\}} L_i^r(P)\pi_r(P)$$

   where r is the root of the tree

Step 3 : Termination

1. The likelihood of the tree topology T

$$L(T) = \prod_i L(\mathcal{X}_i)$$

Example. *Consider the same DNA sequence alignment of 4 species S_1, S_2, S_3 and S_4 of the previous example*

	$\mathcal{X}_1$	$\mathcal{X}_2$	$\mathcal{X}_3$	$\mathcal{X}_4$
S_1	T	C	A	T
S_2	T	C	A	T
S_3	A	C	A	G
S_4	A	G	A	G

We start the initiation step by labeling all the leaves with the corresponding bases from the character alignments, and labelling each internal node and edge, as seen in figure 4.7 for the character alignments $\mathcal{X}_1$ and $\mathcal{X}_2$.

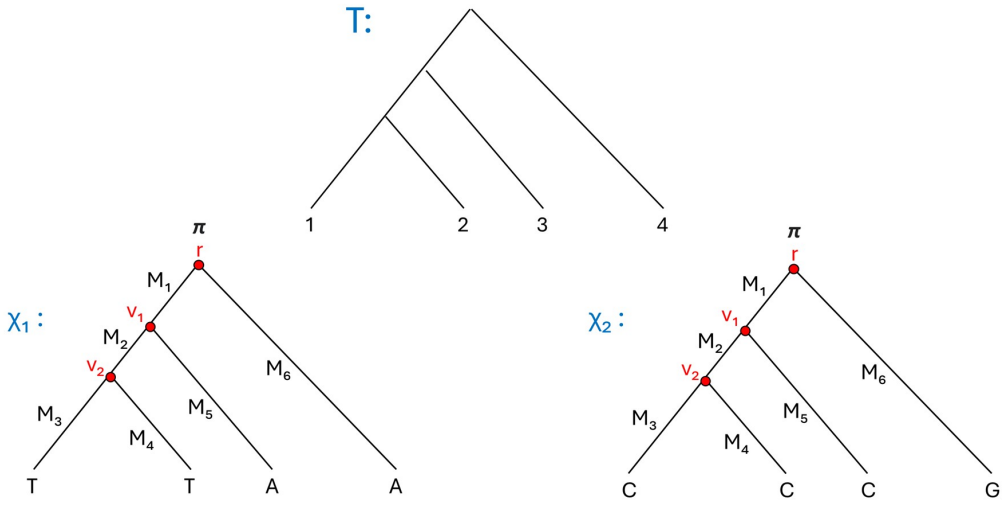


Figure 4.7: Felsenstein pruning algorithm algorithm on tree T

After this we need to calculate the partial conditional likelihood of all the alignments $L(\mathcal{X}_1)$, $L(\mathcal{X}_2)$, $L(\mathcal{X}_3)$ and $L(\mathcal{X}_4)$, and finally the likelihood of the entire tree T would be given by

$$L(T) = L(\mathcal{X}_1) * L(\mathcal{X}_2) * L(\mathcal{X}_3) * L(\mathcal{X}_4)$$

We calculate the likelihood of the character alignment $L(\mathcal{X}_1)$ as follows:

1. *Firstly, we calculate the likelihood of v_2, where both children of v_2 are leaves*

$$L_1^{v_2}(x) = (M_3)_{x,T}(M_4)_{x,T}$$

   for $x \in \{A, C, G, T\}$

2. *After this, we calculate the likelihood of v_1, where v_2 is an internal node, and A is a leaf*

$$L_1^{v_1}(x) = ((M_2)_{x,A}L_1^{v_2}(A) + (M_2)_{x,C}L_1^{v_2}(C) + (M_2)_{x,G}L_1^{v_2}(G) + (M_2)_{x,T}L_1^{v_2}(T)) * (M_5)_{x,A}$$

3. *Finally, we calculate the likelihood of r: the entire sequence alignment*

$$L_1^{r}(x) = ((M_1)_{x,A}L_1^{v_1}(A) + (M_1)_{x,C}L_1^{v_1}(C) + (M_1)_{x,G}L_1^{v_1}(G) + (M_1)_{x,T}L_1^{v_1}(T)) * (M_6)_{x,A}$$

Now if we were given the transition matrices M_i and the root ancestral distribution vector π_r, we could calculate the likelihood of the character alignment $L(\mathcal{X}_1)$

$$L(\mathcal{X}_1) = L_1^r(A)\pi_r(A) + L_1^r(C)\pi_r(C) + L_1^r(G)\pi_r(G) + L_1^r(T)\pi_r(T)$$

5

Reconstructing Phylogenetic Networks Using Cherry Picking

5.1. Solving the Hybridization problem

In this section, we will focus on solving the **Hybridization problem** discussed in Chapter 1, which is the computational problem of combining a set of phylogenetic tree's into a phylogenetic network, with the smallest possible number of reticulations, that displays this set of trees. To obtain this network, we use the cherry picking heuristic algorithm introduced in Chapter 1. The algorithm works by picking a CPS S that fully reduces all the input trees, and then uniquely reconstructing the phylogenetic network, for which S is a minimum length CPS. It constructs the network by processing the cherries in S in reverse order. In Section 5.1.2, we discuss the algorithm used to reconstruct the network corresponding to a CPS S.
As stated already, the cherry picking heuristic (CPH) algorithm seeks a cherry picking sequence S that fully reduces the set of input trees.

Algorithm 5 CPH algorithm [1]

INPUT : A set $\mathcal{T}$ of phylogenetic trees
Output : A CPS reducing $\mathcal{T}$

1. $S \longleftarrow \emptyset$
2. **while** there is a reducible pair in $\mathcal{T}$ **do**
3. $\quad (x, y) \longleftarrow$ PickNext($\mathcal{T}_S$)
4. $\quad S \longleftarrow S \circ (x, y)$
5. $\quad$ Reduce (x, y) in all trees of $\mathcal{T}_S$
6. $S \longleftarrow$ CompleteSeq(S)
7. **Return** S

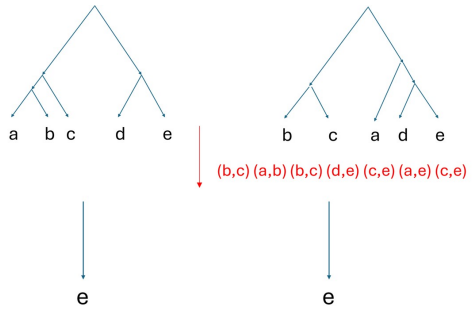
Notice that there is a function CompleteSeq in the algorithm, which turns a sequence of cherries S into a CPS if this is not the case, by adding cherries to S such that each second leaf is a first leaf in a later pair, as required in the definition of a CPS. This will be an important feature needed to construct the network corresponding to a CPS. Furthermore, notice that the function PickNext gives us different manners of picking which cherry to reduce first from $\mathcal{T}_S$.

5.1.1. PickNext function

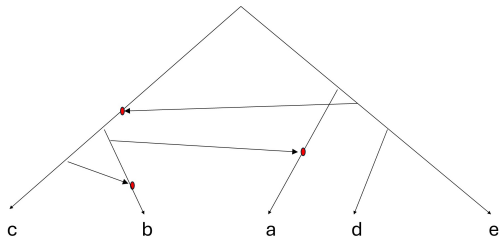
We will discuss two different implementations for the function PickNext.

1. **RAND**
   Function PickNext picks uniformly at random a cherry of $\mathcal{T}_S$
2. **TrivialRand**
   Function PickNext picks a trivial cherry if there exists one and otherwise picks a reducible pair according to RAND

Remember that a cherry (x, y) is defined as *trivial*, if it is a cherry in all tree's that contain both x and y, and it's a cherry in at least one tree. There exists another implementation of PickNext, that uses machine learning to pick a cherry at each step [1], that has the highest probability of leading to a network with the smallest number of reticulations possible, in Section 5.2 we will further elaborate on this implementation. During our research analytically we found that TrivialRand can give us up to 3 times fewer reticulations on a set of two trees with 5 leaves. In Figure 5.1a you can see an execution of CPH with RAND on a set of two trees with 5 leaves.



(a) CPS fully reducing set of two tree's

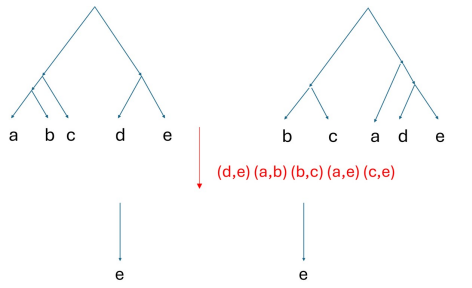


(b) Network reconstructed from this CPS

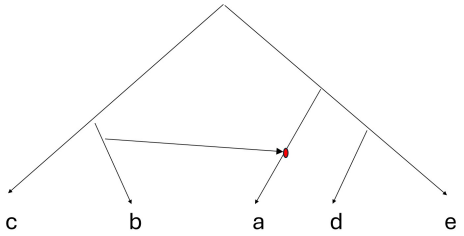
Figure 5.1: Implementation of RAND

By picking cherries randomly, we obtained a CPS $S = (b, c)(a, b)(b, c)(d, e)(c, e)(a, e)(c, e)$ that fully reduces our input set, as only the root and one leaf is left after reducing the cherries in S in order. In this case using RAND, we obtained a CPS containing 7 cherries, which reconstructed the network in Figure 5.1b with 3 reticulations.

After this we executed CPH with TrivialRand on the exact same set of two trees, to try to get a better understanding of how these two methods perform against each other.



(a) CPS fully reducing set of two tree's



(b) Network reconstructed from this CPS

Figure 5.2: Implementation of TrivialRand

In this case using TrivialRand we obtained the CPS $S = (d, e)(a, b)(b, c)(a, e)(c, e)$ containing 5 cherries, as you can see in Figure 5.2a, and we obtained the network in Figure 5.2b with one reticulation. Notice that the first cherry we pick (d, e) is indeed a trivial cherry, as the cherry is present in all trees containing the leaves d and e. We see that in this execution of TrivialRand we obtained a network with one reticulation, while the execution of RAND gave us a network with three reticulations. Remembering the hybridization problem and that we seek the network with the smallest amount of reticulations, clearly TrivialRand performed better during this iteration. Note that both of these methods make use of random choices, and thus both methods could have given us different results. In this case we were researching the worst-case theoretical performance of RAND, and the best case scenario for TrivialRand. During our numerical research we found that when we have as input for the CPH algorithm any set of two trees with five leaves, and the optimal reticulation number is one, executing CPH with TrivialRand gives us 3 reticulations worst-case, while Rand gives us 5 reticulations worst-case.

5.1.2. Reconstructing a network from a CPS

You might be wondering how we reconstructed the phylogenetic networks in Figure 5.1b and 5.2b. In this subsection we will be discussing the algorithm that we use to reconstruct the phylogenetic network from a cherry picking sequence S. We have told you before that we reconstruct the network by processing the cherries in S in reverse order, but what does that exactly mean?

Algorithm 6 Network reconstruction algorithm

INPUT : CPS S of length n
OUTPUT : rooted binary phylogenetic network N

We reconstruct the network by processing the cherries in S in reverse order
Step 1 : Initialisation
Start by processing the last cherry in S, say (a, b), and draw the network N with two leaves (a, b)

Step 2 : Iteration
Let $P(v)$ be the parent of node v. Process each cherry in S in reverse order, starting from the $(n-1)^{th}$ cherry, suppose this cherry is (c, d)

1. if c is not a leaf in N:
   draw an edge in $\mathcal{N}$ from the edge $(P(d), d)$ to a leaf, and label this leaf c

2. if c is a leaf in N:
   draw an edge in N from the edge $(P(d), d)$ to the edge $(P(c), c)$, thus creating a reticulation which becomes the parent of c

Step 3 : Termination

When all the cherries in S have been processed, the algorithm terminates

Return: network N

Notice that as S is a CPS, it is not possible in the iteration step that d is not a leaf, as in a CPS each second element must either be a first element in a later cherry, or the second element of the last cherry. An example of algorithm 6 is covered in Appendix A.

You might wonder if there is a way to predict how many reticulations a network might have. If you have a CPS S, and a phylogenetic network N on a set of leaves X, reconstructed from S, then for the number of reticulations in the network N, which we will henceforth denote as $r(N)$

Lemma 2 [1]:

$$r(N) = |S| - |X| + 1$$

Proof.
Let $|X|$ be the number of leaves in the network and R be the number of reticulations. As the network N is reconstructed from the CPS S, we know by the manner of construction that every element in S is reducible in N in order, and that S is a minumum length CPS fully reducing N. For each cherry $a \in S$, either

1. a is a cherry in N_{S^*}
2. a is a reticulated cherry in N_{S^*}

where S^* is the sequence of cherries in S up to a.
If a is a cherry in N_{S^*}, a will decrease the number of leaves in the network by one after reduction. If a is a reticulated cherry, it will decrease the number of reticulations by one after reduction. We know that S is a minimum length CPS fully reducing N, therefore S reduces the number of leaves by $|X| - 1$, as it reduces the network from $|X|$ leaves to one leaf, and S deletes all R reticulations in N. Consequently,

every element in S that does not reduce a cherry, deletes a reticulation, hence

$$|S| - (|X| - 1) = R$$

□

5.2. Performance when picking a suboptimal cherry first

In this section we will be researching the effect of picking a suboptimal cherry first in the cherry picking algorithm. But before getting into that we must first define what a suboptimal cherry is. Let a CPS S be such that it fully reduces a set of trees $\mathcal{T}$. We know that the network N, reconstructed from S, displays the tree set $\mathcal{T}$, but depending on S, this network might not be an optimal network displaying T, i.e. a network with the least number of reticulations.

Lemma 4 [1]:
A CPS S reducing a set of trees $\mathcal{T}$ reconstructs an optimal network N if and only if each cherry in S is successively reducible in the network N

Let $OPT(\mathcal{T})$ be the set of networks with the smallest possible number of reticulations that display a tree set $\mathcal{T}$. Suppose we were constructing a CPS S displaying $\mathcal{T}$ with the CPH algorithm. If at every iteration i of PickNext($\mathcal{T}_{S_i}$), we knew if a reducible pair (x, y) in $\mathcal{T}_{S_i}$, was reducible in some optimal network $N \in OPT(\mathcal{T})_{S_i}$, we could solve the Hybridization problem optimally according to Lemma 4, where S_i is the sequence of cherries constructed until iteration i. Therefore, in the Hybridization problem, we seek to predict whether a given cherry of $\mathcal{T}$ is reducible in an optimal network N, without knowing N. *G.Bernardini et al* introduce machine-learned heuristics for the CPH in [1]. They introduce an implementation of PickNext($\mathcal{T}_S$) that uses the information of the cherries in $\mathcal{T}_S$, to pick the cherry with the highest probability of being reducible in some (unknown) optimal network N_S. We will not discuss this implementation in detail, but this could be interesting to delve into in the future. As stated before, the aim in this section is to research the effect of picking a suboptimal cherry as a first cherry in the CPH algorithm with TrivialRand. According to Lemma 4, a suboptimal (wrong) cherry is a cherry that is not reducible in any optimal network. Therefore if we know the optimal networks, we know which cherries are suboptimal. We will use this property to conduct experiments analyzing the performance of the CPH algorithm with TrivialRand when it attempts to pick a suboptimal cherry first, but first we will discuss a method used to optimize the performance of TrivialRand.

5.2.1. Use of tree expansion

In Section 5.1.1, we saw during our numerical research that the worst-case scenario of TrivialRand, was (significantly) better than the worst-case scenario of Rand. A method is explained in [1] that improves the performance of TrivialRand quite a bit. This method, called tree expansion, works as follows. Suppose that the function PickNext($\mathcal{T}_S$) chooses the trivial pair (x, y) at some iteration. Therefore, for each tree in $\mathcal{T}_S$ that contains the leaves x and y, (x, y) is a cherry. Each other tree $T \in \mathcal{T}_S$ has exactly one of the following properties:

1. x is a leaf, but y is not
2. y is a leaf, but x is not
3. both x and y are not leaves

We will now add the following step to our CPH algorithm: before we reduce (x, y) in $\mathcal{T}_S$: in all trees with property 1, we replace the leaf x by the cherry (x, y). Therefore, during the reduction of the trivial pair (x, y), these trees will now also reduce this cherry, resulting in the relabeling of all leaves x by y in trees of property 1. This is desirable as now our reducted tree set no longer contains trees that contain x, therefore saving us a reduction step in the future needed to reduce x. An example of how tree expansion can lead to a reduction step less is seen in figure 5.3. As without tree expansion, we would still need to reduce the cherry (a, c) in a reduction step in the future.

In fact, the execution of TrivialRand with tree expansion can reduce the output reticulation number of the cherry picking algorithm by up to 40% [1]. We seek to investigate the accuracy of the CPH with TrivialRand, when it picks the wrong cherry first. To do this, it is interesting to compare the best iterations

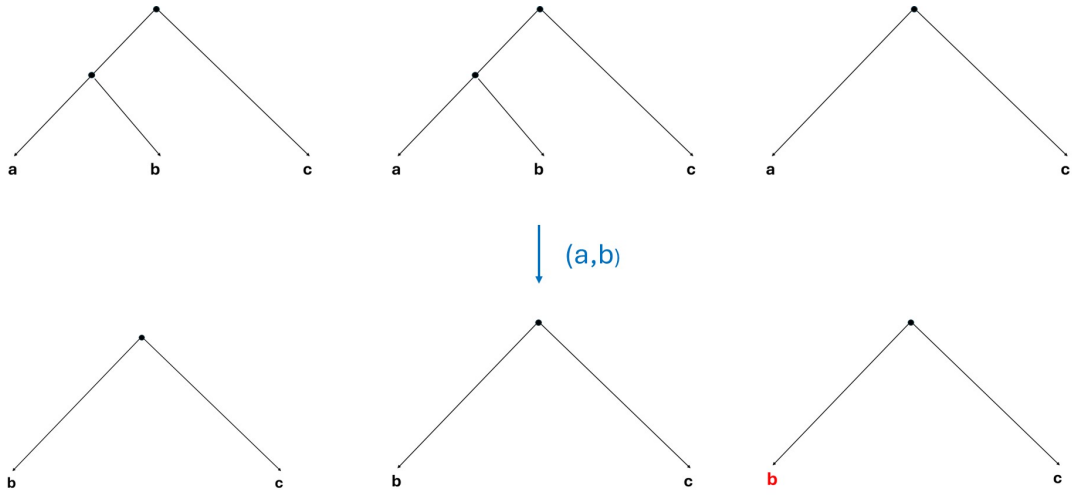


Figure 5.3: Example where tree expansion leads to a reduction step less

of the algorithm possible, therefore I wanted to check if tree expansion also gives us better results in the case that we pick a wrong cherry first. To do this, for each instance; we generate a normal network and the set of trees that it displays, we simulate 100 runs of TrivialRand, with picking a suboptimal cherry first, on this set of trees and calculate the average reticulation number out of all these runs, with and without tree expansion. We simulated 100 of these instances, for various reticulation and leaf numbers of the normal network, and took the average reticulation number over all these instances. The results are seen in figure 5.4

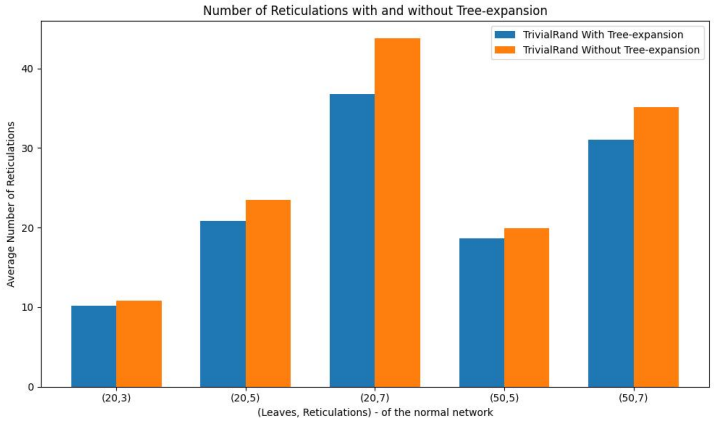


Figure 5.4: Simulation of TrivialRand

Notice that as we took the average over all the runs of TrivialRand, the average number of reticulations is relatively far from the optimal reticulation number (number of reticulations that the network displaying the trees has). As expected, tree-expansion also reduces the number of reticulations, if we pick the wrong cherry first. In fact, every single instance gave us fewer reticulations averaged over 100 runs, using tree-expansion. Therefore, to compare the best simulations of the algorithm, I have used tree expansion in all my simulations of TrivialRand that follow.

5.2.2. Simulation Results

In this section we will investigate the performance of the CPH algorithm when it picks a suboptimal (wrong) cherry in the first iteration of PickNext, i.e. a cherry that is not reducible in any optimal network. In an attempt to simulate this, we use the following idea: we create a normal network N and the set of trees that this network displays $\mathcal{T}$, then in the first iteration of the algorithm we pick a cherry that is reducible in $\mathcal{T}$, but not reducible in N, in an attempt to pick a suboptimal cherry. From now on we will call this *suboptimal picking*. To get a first insight of what the overall performance difference is between TrivialRand normally and with suboptimal picking, we carried out the following simulation. In

each instance, we created a normal network with a certain number of leaves and reticulations, and we generated the set of trees displayed in this network, then we ran TrivialRand normally 200 times on this tree set and calculated the average number of reticulations over all these runs, and did the same for TrivialRand with suboptimal picking. Finally, we repeated this process for 100 instances and calculated the average over all these instances to obtain the data in Figure 5.5 . For simplicity, henceforth we will denote a network with L leaves and R reticulations, as an (L, R) network.

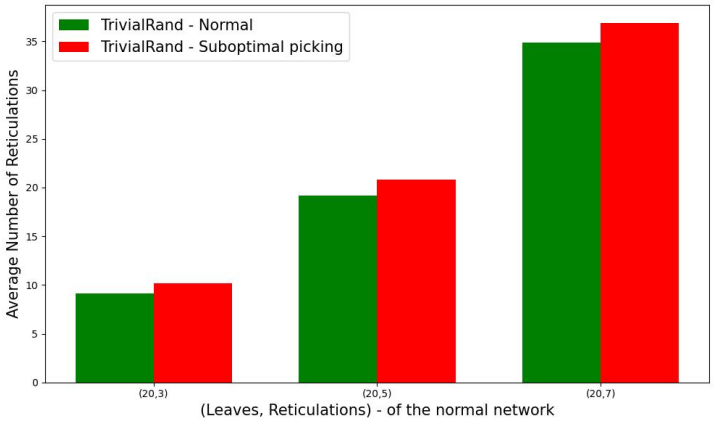


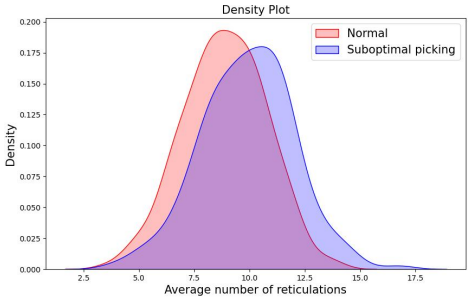
Figure 5.5: Average reticulation number over 200 runs, averaged over 100 instances

Notice that as expected, suboptimal picking gives us more reticulations on average than executing TrivialRand normally. I found that the simulation of TrivialRand with suboptimal picking, gave us on average:

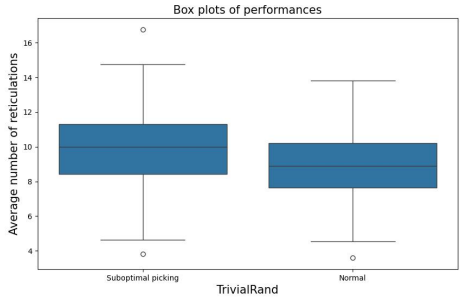
- 11,3% more reticulations on a $(20, 3)$ network
- 8,7% more reticulations on a $(20, 5)$ network
- 5,75% more reticulations on a $(20, 7)$ network

So there seems to be a negative correlation between the effect of suboptimal picking and the number of reticulations in the network.

As the simulation of TrivialRand with suboptimal picking gave us the largest difference on the tree set obtained from a $(20, 3)$ network, we shall further discuss this case. In Figure 5.6a, we see the density plot of the average number of reticulations of 200 instances, while running TrivialRand, with and without suboptimal picking. Observe that the density plot shows a shift to the right with suboptimal picking,



(a) Density plot - (20,3)



(b) Box Plots - (20,3)

Figure 5.6: Implementation of TrivialRand - 200 instances

reaffirming that the average reticulation number is larger with suboptimal picking. Furthermore, notice that the difference between the two density plots is not solely a horizontal shift (by one), therefore we conclude that suboptimal TrivialRand does not always give exactly one more reticulation than TrivialRand normally. The box plots in figure 5.6b reaffirm many things we have already seen. Notice that the interquartile range is slightly wider in the suboptimal picking, indicating a slightly larger variability in the

average number of reticulations.

Although the CPH algorithm with TrivialRand picks trivial cherries first an in attempt to find a network with the least amount of reticulations, it still contains randomness when these trivial cherries are not available. Hence, one could run TrivialRand many times and select the best run, to attempt to obtain an optimal network. In light of this, we conducted the following experiment to gain an understanding of the optimal performance difference between TrivialRand normally and with suboptimal picking. In each instance, we generated an (L, R) network and the set of trees displayed in this network, then we ran TrivialRand 200 times on this tree set and calculated the output of the optimal run (least reticulations). Finally, we repeated this process for 100 instances and calculated the average over all these instances to obtain the data in figure 5.7.

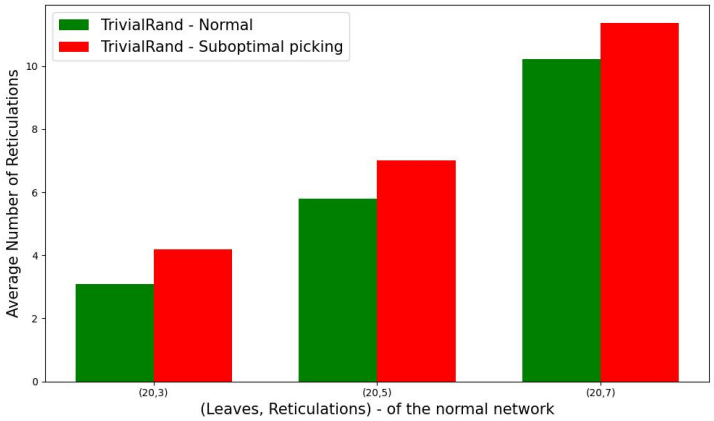


Figure 5.7: Average reticulation number over the best runs of 100 instances

As we now calculated the average of the *optimal* run of each instance, as opposed to the average of *all* the runs as in Figure 5.5, we notice that the average number of reticulations is much closer to the optimal reticulation number. The performance of 'normal' TrivialRand becomes worse as the number of reticulations in the optimal network grows (*G.Bernardini et al 2023, 13*) [1], observe that suboptimal TrivialRand also shares this property. The average of the best runs of our simulation with suboptimal picking gave:

- 34,8% more reticulations on a $(20, 3)$ network
- 20,9% more reticulations on a $(20, 5)$ network
- 11,3% more reticulations on a $(20, 7)$ network

As we have smaller numbers of reticulations now, the relative effect of suboptimal picking becomes larger, compared to our results from Figure 5.5. Once again we observe a negative correlation between the effect of the suboptimal picking and the number of reticulations in an optimal network. This can be explained as follows; the average number of reticulations of the output of CPH with TrivialRand grows significantly, as the number of reticulations in the optimal network increases. Therefore the extra reticulations gained in our simulations because of the suboptimal picking, will be of less impact, as the poor performance of TrivialRand is also giving us extra reticulations.

In Figure 5.8 we see the histogram of the best runs of 100 instances on our $(20, 3)$ network. The first thing that we notice is that in almost all the instances, the best run of CPH with TrivialRand normally gives us an optimal number of reticulations, and the small rest of the instances gave us exactly one reticulation above optimal, while the best run with suboptimal picking is a bit more varied.

Suboptimal picking makes sure that the CPS obtained does not reconstruct the optimal network that we based our simulation on, by picking a cherry first which is not reducible in this network. Therefore, we would expect that the best-case scenario of suboptimal picking would give us one reticulation more than the number of reticulations in the optimal network. Notice that the execution of suboptimal picking gave us one instance with an optimal number of reticulations. This is possible, as in our simulation, we pick a

cherry first that is not reducible in the network that we based our simulation on, but it might be reducible in some other optimal network. Therefore, in some cases the simulation of the CPH algorithm picks a CPS that recontructs another optimal network. We simulated 200 more instances with suboptimal picking and found that an optimal number of reticulations was found in 4,5% of these instances.

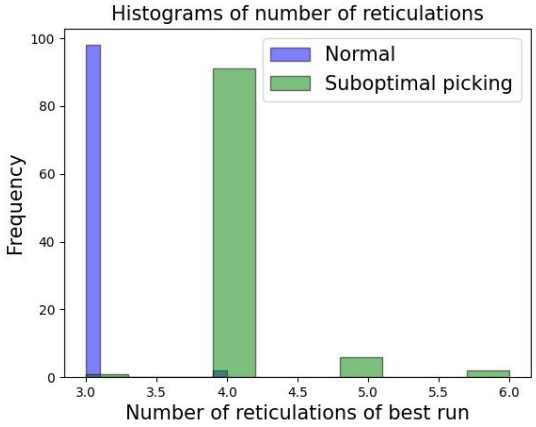


Figure 5.8: Histogram - $(20, 3)$ network

In the simulation we calculated the best run with suboptimal picking, and normally, Table 5.1 shows for each of the 100 instances in our simulations, how many more reticulations suboptimal picking gave us.

	$(20, 3)$	$(20, 5)$	$(20, 7)$
less reticulations	0	4	18
same amount	3	16	18
1 more	89	45	24
2 more	6	24	22
3 more	2	11	18

Table 5.1: Percentages of the instances that had x more reticulations with suboptimal picking

Notice that as the number of reticulations in an optimal network grows, the amount of times grows that suboptimal picking gives us a best run with less reticulations then executing TrivialRand normally. This once again has to do with the decrease in performance of TrivialRand. The average number of reticulations of the output of CPH with TrivialRand grows significantly, as the number of reticulations in the optimal network increases. Therefore the chance that the poor performance of TrivialRand outweighs the effect of suboptimal picking becomes larger.

6

Conclusion/Discussion

The study conducted on the effects of suboptimal picking on the CPH algorithm with TrivialRand provided some key insights into its performance. Our primary goal was to research how picking a suboptimal cherry in the first iteration impacts the algorithm's ability to pick a CPS that reconstructs a network with the smallest amount of reticulations, displaying a tree set $\mathcal{T}$

Our simulations demonstrated that suboptimal picking led to an increase in the average number of reticulations for different optimal network configurations. While researching the overall performance difference, suboptimal picking resulted on average in 11.3% more reticulations on a $(20, 3)$ network, 8,7% on a $(20, 5)$ network, and 5,75% on a $(20, 7)$ network.

After this we investigated the effect of suboptimal picking on the optimal output of the cherry picking algorithm. Our simulations demonstrated that the best runs resulted on average in 34,8% more reticulations on a $(20, 3)$ network, 20,9% on a $(20, 5)$ network, and 11,3% on a $(20, 7)$, indicating that the relative effect of suboptimal picking is larger considering optimal outputs of the cherry picking algorithm. Furthermore, in both cases our simulations indicated a negative correlation between the effect of suboptimal picking and the optimal number of reticulations; as the optimal number of reticulations grows, the performance reduction *due to suboptimal picking* becomes less noticeable.

We found that in a small number of cases suboptimal picking did not have a negative effect on the amount of reticulations, this was due to limitations of the simulation, resulting in the simulation not picking a suboptimal cherry as a first cherry.

This study opens avenues for further research, particularly in exploring the effects of suboptimal picking on the CPH that uses machine learned heuristics to pick the optimal cherry at each iteration, discussed in Section 5.2. Furthermore, it could be interesting to study the effect of picking suboptimal cherries in a different iteration than the first one.

In conclusion, picking a suboptimal cherry in the first iteration of the CPH algorithm with TrivialRand clearly has a negative effect on the number of reticulations of the network reconstructed by the picked CPS, and the extent of this effect varies with the optimal reticulation number. Moreover, future work focusing on machine learned heuristics could further refine our understanding of the effect of suboptimal picking.

References

[1] G. Bernardini, L. van Iersel, E. Julien, and L. Stougie. “Constructing phylogenetic networks via cherry picking and machine learning”. In: (2023).

[2] National Human Genome Research Institute. *Deoxyribonucleic Acid (DNA) Fact Sheet*. URL: https://www.genome.gov/about-genomics/fact-sheets/Deoxyribonucleic-Acid-Fact-Sheet#:~:text=What%20does%20DNA%20do%3F,the%20work%20in%20our%20bodies..

[3] National Library of Medicine, MedlinePlus. *What is DNA?* 2019. URL: https://medlineplus.gov/genetics/understanding/basics/dna/ (visited on 05/22/2024).

[4] D. Durand. *Computational Molecular Biology*. CMU School Of Computer Science, 2021, pp. 32–33.

[5] www.geeksforgeeks.org. *Difference Between Purines and Pyrimidines*. URL: https://www.geeksforgeeks.org/difference-between-purines-and-pyrimidines/.

[6] Z. Zhang and M. Gerstein. “Patterns of nucleotide substitution, insertion and deletion in the human genome inferred from pseudogenes”. In: (2003).

[7] Naruya Saitou and Masatoshi Nei. *The Neighbor-joining Method: A New Method for Reconstructing Phylogenetic Trees*. Center for Demographic and Population Genetics, The University of Texas Health Science Center at Houston, 1987.

[8] Li Yujian and Xu Liye. *Unweighted Multiple Group Method with Arithmetic Mean*. Institute of Electrical and Electronics Engineers, 2010.

[9] Joseph Felsenstein. *Inferring Phylogenies*. University of Washington, 2004, pp. 11–13, 248–255.

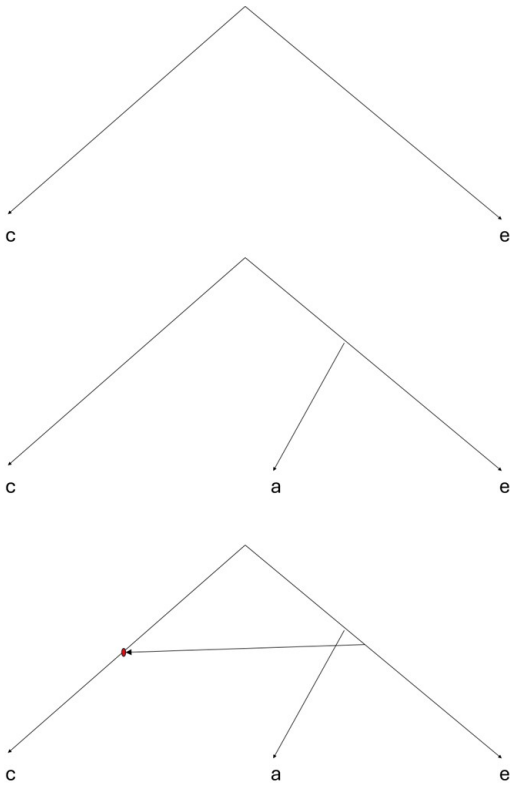
A

Appendix

An example of how we reconstruct the phylogenetic network corresponding to the CPS

$$S = (b, c)(a, b)(b, c)(d, e)(c, e)(a, e)(c, e)$$

We reconstruct the network by processing the cherries in S in reverse order, as follows:



S = (b,c)(a,b)(b,c)(d,e)(c,e)(a,e)(c,e)

S = (b,c)(a,b)(b,c)(d,e)(c,e)(a,e)(c,e)

S = (b,c)(a,b)(b,c)(d,e)(c,e)(a,e)(c,e)

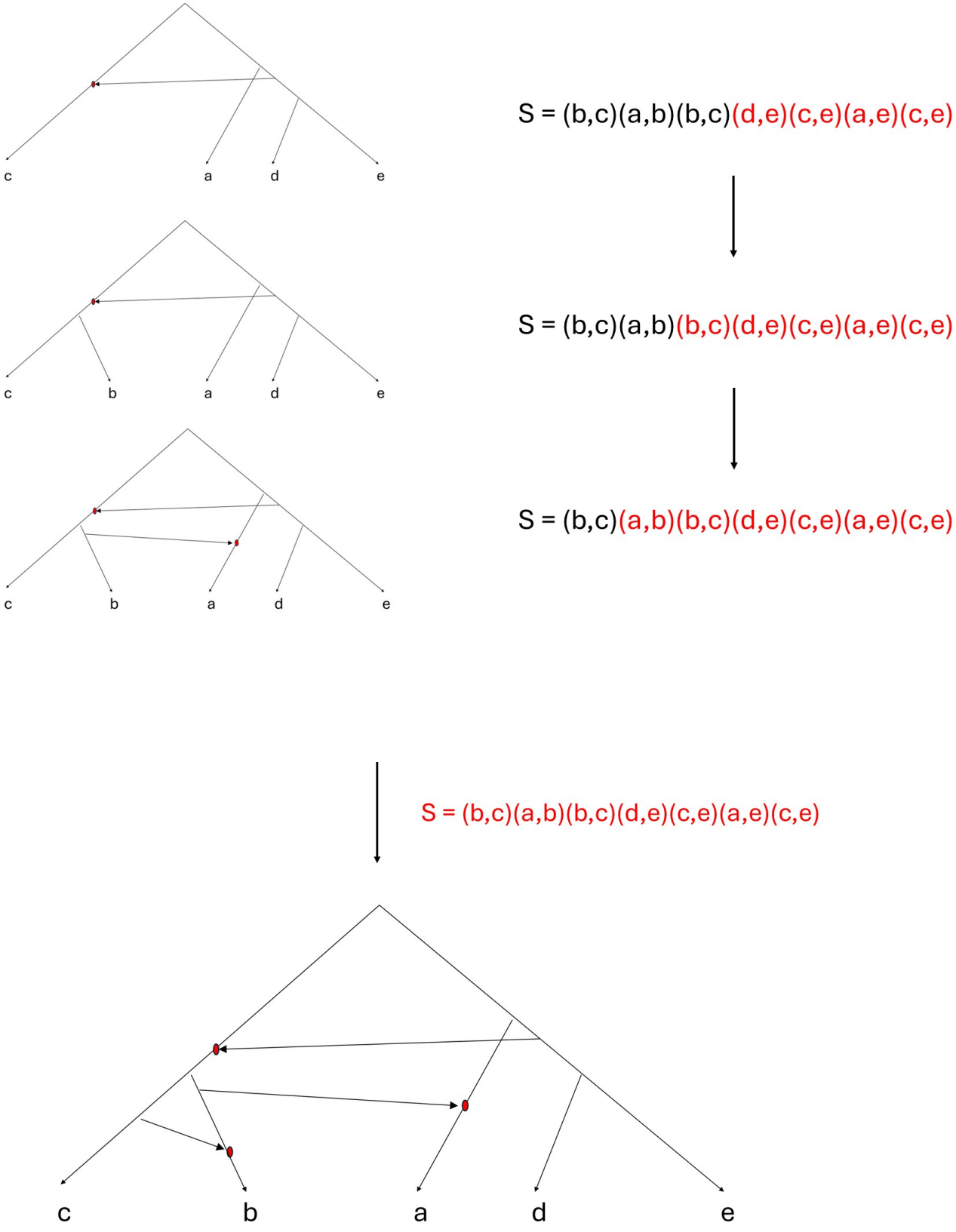


Figure A.1: Phylogenetic network reconstructed by S

B

Appendix

Table of the output of the average number of reticulations of 200 runs of TrivialRand per instance, for 200 instances. We do this for TrivialRand with suboptimal picking (wrong), and ordinary TrivialRand (Normal).

Instances	(20,3) - Wrong Cherry	(20,3) - Normal
1	7.21	6.23
2	11.66	11.05
3	16.76	13.58
4	10.05	8.56
5	11.2	10.0
6	9.49	8.7
7	10.33	9.68
8	13.0	11.77
9	10.91	9.43
10	8.15	7.12
11	9.17	8.4
12	10.83	9.87
13	11.85	9.51
14	5.64	4.78
15	9.55	8.24
16	10.78	10.87
17	9.14	7.9
18	13.52	12.33
19	11.43	10.29
20	9.96	8.09
21	6.87	6.13
22	9.88	9.46
23	11.46	10.93
24	11.08	10.74
25	7.94	6.09
26	12.63	11.91
27	9.57	7.96
28	11.3	10.04
29	9.09	8.18
30	9.35	8.67
31	10.02	9.21
32	9.38	8.79
33	10.31	9.04
34	13.11	11.48
35	8.82	7.91
36	11.92	10.78

Instances	(20,3) - Wrong Cherry	(20,3) - Normal
37	7.74	7.49
38	8.85	8.36
39	8.41	8.49
40	14.35	11.6
41	12.03	10.51
42	6.26	5.89
43	8.05	6.74
44	12.17	11.7
45	8.02	7.24
46	9.41	8.89
47	10.92	10.05
48	8.99	7.92
49	7.7	6.59
50	11.69	10.73
51	8.47	7.86
52	9.81	8.27
53	7.67	6.82
54	11.64	10.63
55	10.88	9.78
56	8.0	7.28
57	8.97	7.74
58	8.42	8.11
59	12.15	10.29
60	7.73	6.66
61	9.99	8.97
62	9.07	8.02
63	9.17	8.22
64	10.22	9.8
65	11.8	11.17
66	10.82	10.34
67	9.39	8.53
68	9.27	7.97
69	8.02	7.06
70	9.67	9.06
71	11.81	10.37
72	12.46	11.3
73	11.55	10.81
74	11.6	10.2
75	8.3	7.49
76	7.57	6.99
77	10.62	8.94
78	10.33	9.17
79	11.7	10.19
80	6.9	6.38
81	8.21	7.7
82	12.24	10.31
83	7.07	6.37
84	11.19	9.96
85	9.92	9.19
86	9.75	9.36
87	5.25	4.92
88	11.29	10.46
89	11.4	11.38
90	11.75	10.02
91	8.67	8.15
92	12.99	10.42
93	12.68	10.99

Instances	(20,3) - Wrong Cherry	(20,3) - Normal
94	9.18	7.29
95	6.49	6.19
96	11.49	11.75
97	14.18	11.95
98	8.65	8.45
99	5.48	5.2
100	11.7	10.96
101	10.95	7.9
102	10.51	9.47
103	11.88	11.48
104	13.15	11.5
105	10.25	8.62
106	13.77	11.62
107	9.35	8.42
108	8.35	7.2
109	11.19	9.92
110	3.81	3.6
111	4.94	4.54
112	9.55	9.48
113	10.5	10.01
114	10.4	8.79
115	11.62	10.28
116	14.52	11.45
117	8.38	7.1
118	7.38	6.77
119	11.77	10.37
120	11.05	10.36
121	8.59	8.21
122	9.96	9.13
123	12.94	11.06
124	11.02	9.37
125	10.77	9.2
126	10.37	9.24
127	8.45	6.96
128	8.35	8.01
129	9.33	8.03
130	8.3	7.75
131	10.48	9.07
132	12.5	10.93
133	8.29	7.22
134	10.52	10.55
135	11.44	9.75
136	9.89	8.52
137	10.19	8.5
138	7.13	6.74
139	10.12	8.89
140	11.06	9.44
141	8.43	7.95
142	9.82	8.8
143	6.28	5.58
144	7.6	6.48
145	12.42	11.89
146	10.56	10.03
147	6.73	5.88
148	8.04	7.96
149	6.42	6.07
150	7.32	6.34

Instances	(20,3) - Wrong Cherry	(20,3) - Normal
151	10.11	8.88
152	9.52	9.61
153	7.21	6.35
154	7.69	7.24
155	9.29	8.31
156	8.71	8.34
157	9.37	7.46
158	8.7	7.47
159	9.99	8.6
160	8.82	7.74
161	10.23	9.43
162	12.42	12.14
163	11.18	9.31
164	9.82	8.65
165	12.41	12.23
166	8.03	6.54
167	14.74	13.8
168	11.37	10.0
169	11.28	9.64
170	9.48	9.02
171	5.48	5.15
172	8.32	7.52
173	7.01	6.73
174	8.64	8.27
175	9.17	7.97
176	11.96	10.4
177	13.81	12.27
178	6.35	5.29
179	10.88	10.27
180	10.63	10.51
181	8.82	7.26
182	7.23	6.85
183	10.0	9.57
184	10.44	8.79
185	9.97	7.73
186	11.12	10.05
187	11.35	9.88
188	8.53	7.44
189	10.7	9.37
190	11.52	9.15
191	7.89	6.97
192	10.88	9.73
193	10.03	8.4
194	11.25	9.93
195	11.35	9.39
196	11.04	10.73
197	7.97	6.3
198	9.61	9.06
199	4.63	4.68
200	13.87	13.0

C

Appendix

Table showing the output of the best of 200 runs of TrivialRand, for 100 instances. We do this for TrivialRand with suboptimal picking (wrong), and ordinary TrivialRand.

Instances	(20,3) - Wrong	(20,3)	(20,5) - Wrong	(20,5)	(20,7) - Wrong	(20,7)
1	4	3	7	5	13	9
2	4	3	7	5	10	9
3	4	3	6	10	10	8
4	4	3	9	6	11	11
5	4	3	7	6	17	9
6	4	3	9	7	14	12
7	4	3	9	8	10	8
8	5	3	8	6	12	10
9	4	3	7	6	16	8
10	4	3	8	5	11	10
11	4	3	6	5	11	12
12	4	3	7	5	10	9
13	4	3	8	6	13	12
14	4	3	7	5	11	9
15	4	3	8	7	8	9
16	4	3	6	6	13	10
17	4	3	8	5	11	10
18	5	3	6	6	13	16
19	4	3	6	5	10	10
20	4	3	9	6	15	18
21	4	3	7	7	11	10
22	4	3	8	6	11	11
23	4	3	8	6	16	14
24	4	3	6	5	10	10
25	4	3	8	6	15	11
26	5	3	8	7	12	8
27	6	3	7	6	12	10
28	4	3	7	5	10	8
29	4	3	7	6	10	10
30	4	3	7	6	13	9
31	4	3	7	5	11	10
32	4	3	7	5	14	12
33	4	3	6	5	13	12
34	4	3	9	6	11	9
35	4	3	8	6	11	13
36	6	3	6	5	12	8
37	4	3	6	6	10	10

Instances	(20,3) - Wrong	(20,3)	(20,5) - Wrong	(20,5)	(20,7) - Wrong	(20,7)
38	4	3	7	6	12	11
39	4	3	7	6	10	7
40	4	3	7	6	9	9
41	4	3	6	5	11	10
42	4	3	8	7	12	8
43	4	3	6	5	10	11
44	4	3	6	5	10	9
45	5	4	7	7	9	10
46	4	3	7	5	9	7
47	4	3	7	6	8	9
48	4	3	6	5	10	10
49	4	3	10	5	11	10
50	4	3	8	5	13	10
51	4	3	8	7	11	15
52	4	3	7	5	11	9
53	4	3	5	5	12	9
54	4	3	9	5	14	10
55	4	3	7	8	13	9
56	4	3	8	5	10	9
57	4	3	7	6	9	9
58	4	3	6	6	10	9
59	4	3	9	7	12	13
60	4	3	6	7	9	7
61	4	4	5	5	12	10
62	4	3	7	9	14	12
63	4	3	7	6	11	9
64	4	3	6	5	11	9
65	4	4	6	6	16	13
66	4	3	6	5	10	8
67	4	3	7	6	11	11
68	4	3	7	5	10	11
69	4	3	6	6	14	16
70	4	3	7	5	10	9
71	4	3	6	6	12	12
72	5	3	7	5	12	11
73	4	3	6	5	11	9
74	4	3	7	6	10	12
75	4	3	7	5	13	10
76	4	3	8	6	10	11
77	4	3	6	5	11	8
78	4	3	6	6	13	9
79	4	3	6	5	10	11
80	4	3	9	7	12	14
81	4	3	7	5	9	8
82	4	3	6	6	8	9
83	4	3	6	6	14	11
84	4	3	7	5	9	14
85	4	3	7	5	14	12
86	4	3	8	6	10	10
87	4	3	6	6	12	11
88	4	3	5	5	9	11
89	4	3	8	6	10	9
90	4	3	6	5	14	8
91	5	3	7	5	12	14
92	4	3	7	6	10	10
93	4	3	7	5	11	8
94	3	3	8	6	10	11

Instances	(20,3) - Wrong	(20,3)	(20,5) - Wrong	(20,5)	(20,7) - Wrong	(20,7)
95	4	3	8	5	11	10
96	4	3	7	6	11	10
97	4	3	8	6	10	9
98	4	3	8	7	9	10
99	4	3	7	6	9	8
100	4	3	6	6	16	11

D

Source Code

These are the two Main python files I used to simulate the data, I extended the code seen in the section "Experiments" in "Constructing phylogenetic networks via cherry picking and machine learning" [1]

```
import os.path
from argparse import ArgumentParser
import pickle as pkl
import time


from NetworkGen.normal_network import simulation as normal_simulation
from NetworkGen.tree_to_newick import *
from NetworkGen.NetworkToTree import *

from CPH import CPHeuristic

def make_test_normal(net_num, l, ret):
    tree_info = f"_L{l}_R{ret}_normal"

    # MAKE NETWORK
    st = time.time()
    beta = 1
    distances = True
    n = l - 2 + ret

    # print info
    print(f"JOB {net_num}: Start creating NETWORK (Normal , L = {l}, R = {ret}, n = {n})")

    while True:
        if l <= 20:
            alpha = np.random.uniform(0.1, 0.5)
        elif l <= 50:
            alpha = np.random.uniform(0.1, 0.3)
        else:
            alpha = np.random.uniform(0.1, 0.2)
        net, ret_num = normal_simulation(n, alpha, 1, beta, net_num)
        num_leaves = len(leaves(net))
        if num_leaves == l and ret_num == ret:
            break

    # EXTRACT TREES
    net_nodes = int(len(net.nodes))

    while True:
        print(f"JOB {net_num}: Start creating TREE SET (Normal , L = {num_leaves}, R = {ret_num
            })")
        tree_set, tree_lvs, num_unique_leaves = net_to_tree(net, num_trees=None, distances=
            distances, net_lvs=num_leaves)
        if num_unique_leaves == num_leaves:
            break

    num_trees = 2 ** ret_num
    tree_to_newick_fun(tree_set, net_num, tree_info=tree_info)
```

```

    # SAVE INSTANCE
    metadata_index = ["network_type" , "rets" , "nodes", "net_leaves", "chers", "ret_chers", "trees", "n", "alpha",
                      "beta", "min_lvs", "mean_lvs", "max_lvs", "runtime"]

    net_cher, net_ret_cher = network_cherries(net)
    min_lvs = min(tree_lvs)
    mean_lvs = np.mean(tree_lvs)
    max_lvs = max(tree_lvs)
    metadata = pd.Series([0, ret_num, net_nodes, num_leaves, len(net_cher)/2, len(net_ret_cher),
                          num_trees, n, alpha, beta, min_lvs, mean_lvs, max_lvs,
                          time.time() - st],
                         index=metadata_index,
                         dtype=float)
    output = {"net": net, "forest": tree_set, "metadata": metadata}

    # ------------------------------------------------------------------------------------------

    net_Cher_and_retCher = net_cher | net_ret_cher

    cherries, reducible_pairs_trees = tree_cherries(tree_set)

    net_reducible_pairs = {tup: idx + 1 for idx, tup in enumerate(net_Cher_and_retCher)}
    first_cherry_to_pick = {k: v for k, v in reducible_pairs_trees.items() if k not in net_reducible_pairs}

    while len(first_cherry_to_pick) == 0:
        # --------------------------------------------Create a new network and tree set while there is no wrong cherry to pick
        tree_info = f"_L{l}_R{ret}_normal"

        # MAKE NETWORK
        st = time.time()
        beta = 1
        distances = True
        n = l - 2 + ret

        # print info
        print(f"JOB {net_num}: Start creating NETWORK (Normal, L = {l}, R = {ret}, n = {n})")

        while True:
            if l <= 20:
                alpha = np.random.uniform(0.1, 0.5)
            elif l <= 50:
                alpha = np.random.uniform(0.1, 0.3)
            else:
                alpha = np.random.uniform(0.1, 0.2)
            net, ret_num = normal_simulation(n, alpha, 1, beta, net_num)
            num_leaves = len(leaves(net))
            if num_leaves == l and ret_num == ret:
                break

        # EXTRACT TREES
        net_nodes = int(len(net.nodes))

        while True:
            print(f"JOB {net_num}: Start creating TREE SET (Normal, L = {num_leaves}, R = {ret_num})")
            tree_set, tree_lvs, num_unique_leaves = net_to_tree(net, num_trees=None, distances=distances,
                                                                net_lvs=num_leaves)
            if num_unique_leaves == num_leaves:
                break

        num_trees = 2 ** ret_num
        tree_to_newick_fun(tree_set, net_num, tree_info=tree_info)

        # SAVE INSTANCE
```

```
        metadata_index = ["network_type", "rets", "nodes", "net_leaves", "chers", "ret_chers", "trees", "n", "alpha",
                          "beta", "min_lvs", "mean_lvs", "max_lvs", "runtime"]

        net_cher, net_ret_cher = network_cherries(net)
        min_lvs = min(tree_lvs)
        mean_lvs = np.mean(tree_lvs)
        max_lvs = max(tree_lvs)
        metadata = pd.Series([0, ret_num, net_nodes, num_leaves, len(net_cher) / 2, len(net_ret_cher),
                              num_trees, n, alpha, beta, min_lvs, mean_lvs, max_lvs,
                              time.time() - st],
                             index=metadata_index,
                             dtype=float)
        output = {"net": net, "forest": tree_set, "metadata": metadata}

        net_Cher_and_retCher = net_cher | net_ret_cher

        cherries, reducible_pairs_trees = tree_cherries(tree_set)

        net_reducible_pairs = {tup: idx + 1 for idx, tup in enumerate(net_Cher_and_retCher)}
        first_cherry_to_pick = {k: v for k, v in reducible_pairs_trees.items() if k not in net_reducible_pairs}

        if len(first_cherry_to_pick) > 0:
            break

    # ---------------------------------------------------------------------------------------


    os.makedirs(f"data/network/instances_test", exist_ok=True)
    with open(
            f"data/network/instances_test/tree_data{tree_info}_{net_num}.pkl", "wb") as handle:
        pkl.dump(output, handle)

    print(f"JOB {net_num}: FINISHED in {np.round(time.time() - st, 3)}s (Normal, L = {num_leaves}, "
          f"R = {ret_num}, n = {n})")

#data_path = f"data/network/instances/tree_data_L10_R2_normal_{i}.pkl" (what was there)
def main(args):
    pass
    # once for each instance (change 1 to args.num_instances for a specific instance)

    data_best_runs = []
    data_avg_ret_number_L20_R5_runs50 = []
    data_worst_runs = []

    for i in range(1, args.num_instances+1):
        # make network + trees
        #os.makedirs("data/network/instances_test", exist_ok=True)
        data_path = f"data/network/instances_test/tree_data_L{args.num_leaves}_R{args.num_rets}_normal_{i}.pkl"
        #and len(first_cherry_to_pick) == 0??
        if not os.path.exists(data_path) :
            make_test_normal(i, args.num_leaves, args.num_rets)
        # load data
        data = pkl.load(open(data_path, "rb"))
        # simulation
        # get (reticulated) cherries of normal network
        net_cher, net_ret_cher = network_cherries(data["net"])


        #NEW set containing all reducible pairs of network
        net_Cher_and_retCher = net_cher | net_ret_cher


        # run CPH
        retics = dict()
        max_ret_number_instance = 0
```

```
        min_ret_number_instance = 1000
        total_ret_number = 0
        #suboptimal_solution_times = 0

        for s in range(1, args.num_runs+1):
            #print(f"Run {s}")
            # todo: give as input something such that it doesn't select a correct cherry in the first iteration
            cph = CPHeuristic(inst_num=i, tree_set=data["forest"], seed=s, verbose=args.verbose, pick_method=args.pick_method, tree_expansion=args.tree_expansion)
            seq = cph.run_heuristic(net_Cher_and_retCher)
            retics[s] = len(seq) - args.num_leaves + 1
            #print(f"Inst {i}, run {s}: reticulation number = {retics[s]}")
            total_ret_number += retics[s]
            if retics[s] > max_ret_number_instance:
                max_ret_number_instance = retics[s]

            if retics[s] < min_ret_number_instance:
                min_ret_number_instance = retics[s]

            #if retics[s] == args.num_rets+1 :
                #suboptimal_solution_times += 1

        average_ret_number = total_ret_number/(args.num_runs)
        data_avg_ret_number_L20_R5_runs50.append(average_ret_number)
        data_best_runs.append(min_ret_number_instance)
        data_worst_runs.append(max_ret_number_instance)

        print(f'\n Instance {i}: \n Largest Reticulation number in these runs is: {max_ret_number_instance} \n the smallest is: {min_ret_number_instance} '
              f'\n Average reticulation number is: {average_ret_number}')
    print()
    #print("The average ret. number of these instances are: ", data_avg_ret_number_L20_R5_runs50)
    #print(f"Average ret number over all these instances is {sum(data_avg_ret_number_L20_R5_runs50)/(args.num_instances)}")
    print(f'The best run of these instances are {data_best_runs} \n Average best run over all these instances is {sum(data_best_runs)/(args.num_instances)}' )
    #print(f'The worst run of these instances are {data_worst_runs} and the worst overall run is {min(data_worst_runs)}')

            #f'\n In {(suboptimal_solution_times/args.num_runs)*100}% of the runs we find a (sub)optimal solution' )

            # todo: get best reticulation per instance


if __name__ == "__main__":
    parser = ArgumentParser()
    # input data
    parser.add_argument('--num_instances', type=int, default=1)
    parser.add_argument('--num_leaves', type=int, default=10)
    parser.add_argument('--num_rets', type=int, default=2)

    # simulation
    parser.add_argument('--num_runs', type=int, default=10)

    # heuristic settings
    parser.add_argument('--pick_method', type=str, default="trivial", choices=["random", "trivial"])
    parser.add_argument('--tree_expansion', type=int, default=0, choices=[0, 1])

    parser.add_argument('--verbose', type=int, default=1)

    args = parser.parse_args()

    main(args)
```

```

from copy import deepcopy
import networkx as nx
import numpy as np

```

```
6
7  class CPHeuristic:
8      def __init__(self, inst_num, tree_set, seed=1, verbose=1, pick_method="trivial",
           tree_expansion=0):
9          # data
10         self.inst_num = inst_num
11         copy_tree_set = deepcopy(tree_set)
12         self.trees = {t: PhT(tree) for t, tree in copy_tree_set.items()}
13
14         self.seed = seed
15         self.verbose = verbose
16         self.rng = np.random.RandomState(seed)
17
18         # set picking method
19         if pick_method == "trivial":
20             pick_triv = True
21             pick_random = False
22         else:
23             pick_triv = False
24             pick_random = True
25
26         if tree_expansion:
27             relabel = True
28         else:
29             relabel = False
30
31         self.pick_triv = pick_triv
32         self.pick_random = pick_random
33         self.relabel = relabel
34
35     def run_heuristic(self, net_Cher_and_retCher):
36         # Works in a copy of the input trees, copy_of_inputs, because trees have to be reduced
               somewhere.
37         CPS = []
38         reduced_trees = []
39
40         # Make dict of reducible pairs
41         reducible_pairs = self.find_all_pairs()
42
43         # Pick the wrong cherry first
               ----------------------------------------------------------------------------------------------
44
45         # Make dict of reticulated cherries/cherries in the network
46         net_reducible_pairs = {tup: idx + 1 for idx, tup in enumerate(net_Cher_and_retCher)}
47
48         # first_cherry_to_pick = {cherries that are in reducible_pairs but not in
               Net_cher_and_RetCher}
49         first_cherry_to_pick = {k: v for k, v in reducible_pairs.items() if k not in
               net_reducible_pairs}
50
51         # Run the algorithm once only for the first Cherry to pick, out of
               first_cherry_to_pick
52
53         pick_random = self.pick_random
54         triv_picked = False
55         # pick cherry
56         if self.pick_triv:
57             chosen_cherry, triv_picked = self.pick_trivial(first_cherry_to_pick)
58             if chosen_cherry is None:
59                 pick_random = True
60             else:
61                 pick_random = False
62         if pick_random:
63             random_cherry_num = self.rng.choice(len(first_cherry_to_pick))
64             chosen_cherry = list(first_cherry_to_pick)[random_cherry_num]
65
66         CPS += [chosen_cherry]
67
68         if self.verbose and not pick_random:
69             print(f"Instance␣{self.inst_num}:␣TRIVIAL␣chosen␣cherry␣=␣{chosen_cherry}")
70         elif self.verbose:
71             print(f"Instance␣{self.inst_num}:␣RANDOM␣chosen␣cherry␣=␣{chosen_cherry}")
```

```
72
73          # REDUCE CHOSEN CHERRY FROM FOREST
74          new_reduced = self.reduce_pair_in_all(chosen_cherry, reducible_pairs=reducible_pairs)
75          reducible_pairs = self.update_reducible_pairs(reducible_pairs, new_reduced)
76          reduced_trees += [new_reduced]
77
78          #
                ----------------------------------------------------------------------------------------------------------------
79
80          # START ALGORITHM
81          pick_random = self.pick_random
82          while self.trees:
83              triv_picked = False
84              # pick cherry
85              if self.pick_triv:
86                  chosen_cherry, triv_picked = self.pick_trivial(reducible_pairs)
87                  if chosen_cherry is None:
88                      pick_random = True
89                  else:
90                      pick_random = False
91              if pick_random:
92                  random_cherry_num = self.rng.choice(len(reducible_pairs))
93                  chosen_cherry = list(reducible_pairs)[random_cherry_num]
94
95              # TREE EXPANSION
96              if triv_picked:
97                  relabel_needed, chosen_cherry = self.pick_order(*chosen_cherry,
98                                                                  reducible_pairs[chosen_cherry
                                                                      ],
99                                                                  return_relabel_needed=True)
100                 if self.relabel and relabel_needed:
101                     if self.verbose:
102                         print(f"Instance␣{self.inst_num}:␣RELABEL␣chosen␣cherry␣=␣{
                                chosen_cherry}")
103                     reducible_pairs, merged_cherries = self.relabel_trivial(*chosen_cherry,
                            reducible_pairs)
104
105             CPS += [chosen_cherry]
106
107             if self.verbose and not pick_random:
108                 print(f"Instance␣{self.inst_num}:␣TRIVIAL␣chosen␣cherry␣=␣{chosen_cherry}")
109             elif self.verbose:
110                 print(f"Instance␣{self.inst_num}:␣RANDOM␣chosen␣cherry␣=␣{chosen_cherry}")
111
112             # REDUCE CHOSEN CHERRY FROM FOREST
113             new_reduced = self.reduce_pair_in_all(chosen_cherry, reducible_pairs=
                    reducible_pairs)
114             reducible_pairs = self.update_reducible_pairs(reducible_pairs, new_reduced)
115             reduced_trees += [new_reduced]
116
117             if len(self.trees) == 0:
118                 break
119
120         # finish heuristic
121         return self.sequence_add_roots(CPS)
122
123     def find_all_pairs(self):
124         reducible_pairs = dict()
125         for i, t in self.trees.items():
126             red_pairs_t = t.find_all_reducible_pairs()
127             for pair in red_pairs_t:
128                 if pair in reducible_pairs:
129                     reducible_pairs[pair].add(i)
130                 else:
131                     reducible_pairs[pair] = {i}
132         return reducible_pairs
133
134     def find_all_reducible_pairs(self):
135         red_pairs = set()
136         for l in self.leaves:
137             red_pairs = red_pairs.union(self.find_pairs_with_first(l))
138         return red_pairs
```

```
139
140
141     def reduce_pair_in_all(self, pair, reducible_pairs=None):
142         if not len(reducible_pairs):
143             print("no␣reducible␣pairs")
144         if reducible_pairs is None:
145             reducible_pairs = dict()
146         reduced_trees_for_pair = []
147         if pair in reducible_pairs:
148             trees_to_reduce = reducible_pairs[pair]
149         else:
150             trees_to_reduce = deepcopy(self.trees)
151         for t in trees_to_reduce:
152             if t in self.trees:
153                 tree = self.trees[t]
154                 # print(t, tree.leaves)
155                 if tree.reduce_pair(*pair):
156                     reduced_trees_for_pair += [t]
157                     if (self.trees[t].root == 0 and len(tree.nw.edges()) <= 1) or \
158                             (self.trees[t].root == 2 and len(tree.nw.edges()) <= 2):
159                         # print(t, pair, tree.leaves)
160                         del self.trees[t]
161         return set(reduced_trees_for_pair)
162
163     def update_reducible_pairs(self, reducible_pairs, new_red_trees):
164         # Remove trees to update from all pairs
165         pair_del = []
166         for pair, trees in reducible_pairs.items():
167             trees.difference_update(new_red_trees)
168             if len(trees) == 0:
169                 pair_del.append(pair)
170         for pair in pair_del:
171             del reducible_pairs[pair]
172         # Add the trees to the right pairs again
173         for index in new_red_trees:
174             if index in self.trees:
175                 t = self.trees[index]
176                 red_pairs_t = t.find_all_reducible_pairs()
177                 for pair in red_pairs_t:
178                     if pair in reducible_pairs:
179                         reducible_pairs[pair].add(index)
180                     else:
181                         reducible_pairs[pair] = {index}
182         return reducible_pairs
183
184     # TRIVIAL CHERRY PICKING
185     def pick_trivial(self, reducible_pairs):
186         trivial_cherries = []
187         trivial_in_all_cherries = []
188         for c, trees in reducible_pairs.items():
189             if len(trees) == len(self.trees):
190                 trivial_in_all_cherries.append(c)
191                 continue
192             trivial_check = self.trivial_check(c, trees)
193             if trivial_check:
194                 trivial_cherries.append(c)
195
196         if trivial_in_all_cherries:
197             chosen_cherry = trivial_in_all_cherries[self.rng.choice(len(
                    trivial_in_all_cherries))]
198             triv_picked = False
199         elif trivial_cherries:
200             chosen_cherry = trivial_cherries[self.rng.choice(len(trivial_cherries))]
201             triv_picked = True
202         else:
203             chosen_cherry = None
204             triv_picked = False
205
206         return chosen_cherry, triv_picked
207
208     def trivial_check(self, c, trees):
209         if len(trees) == len(self.trees):
210             return False
```

```
        return len([t for t, tree in self.trees.items() if (set(c).issubset(tree.leaves) and t
             not in trees)]) == 0

    # TREE EXPANSION
    def relabel_trivial(self, x, y, reducible_pairs):
        # print(f"Cherry = {(x, y)}: RELABEL X = {x} to Y = {y}")
        merged_cherries = set()
        new_cherries = set()
        for t, tree in self.trees.items():
            if t in reducible_pairs[(x, y)]:
                continue
            if x in tree.leaves:
                # change leaf set
                tree.leaves.remove(x)
                tree.leaves.add(y)

                # relabel x to y
                tree.nw = nx.relabel_nodes(tree.nw, {x: y})

                # check if we have a new cherry now
                for p in tree.nw.predecessors(y):
                    for c in tree.nw.successors(p):
                        if c == y:
                            continue
                        if c not in tree.leaves:
                            continue
                        if (c, y) in reducible_pairs:
                            reducible_pairs[(c, y)].add(t)
                            reducible_pairs[(y, c)].add(t)
                            try:
                                del reducible_pairs[(c, x)], reducible_pairs[(x, c)]
                                merged_cherries.add((x, c))
                            except KeyError:
                                pass
                        else:
                            # add to reducible_pairs?
                            reducible_pairs[(c, y)] = {t}
                            reducible_pairs[(y, c)] = {t}
                            new_cherries.add((c, y))
                            try:
                                del reducible_pairs[(c, x)], reducible_pairs[(x, c)]
                            except KeyError:
                                pass
        return reducible_pairs, merged_cherries

    def pick_order(self, x, y, new_reduced, return_cherry=False, return_relabel_needed=False):
        leaf_x_left = 0
        leaf_y_left = 0
        for t, tree in self.trees.items():
            if t in new_reduced:
                continue
            if x in tree.leaves:
                leaf_x_left += 1
            if y in tree.leaves:
                leaf_y_left += 1
        if return_cherry:
            # FAVOR X, Y OVER Y, X
            if leaf_x_left <= leaf_y_left:
                return x, y
            else:
                return y, x
        elif return_relabel_needed:
            if leaf_x_left == 0:
                return False, (x, y)
            elif leaf_y_left == 0:
                return False, (y, x)
            elif leaf_x_left <= leaf_y_left:
                return True, (x, y)
            else:
                return True, (y, x)
        else:
            return leaf_x_left, leaf_y_left

```

```
283     def pick_cherry(self, triv_cherries, reducible_pairs):
284         leaf_left = dict()
285         for x, y in triv_cherries:
286             leaf_x_left, leaf_y_left = self.pick_order(x, y, reducible_pairs[x, y],
                    return_cherry=False)
287             leaf_left[(x, y)] = leaf_x_left
288             leaf_left[(y, x)] = leaf_y_left
289         best_cherry_id = np.argmin(list(leaf_left.values()))
290         return list(leaf_left)[best_cherry_id]
291 
292     # FINISH HEURISTIC
293     @staticmethod
294     def sequence_add_roots(seq):
295         leaves_encountered = set()
296         roots = set()
297         # The roots can be found by going back through the sequence and finding pairs where
                the second element has not been
298         # encountered in the sequence yet
299         for pair in reversed(seq):
300             if pair[1] not in leaves_encountered:
301                 roots.add(pair[1])
302             leaves_encountered.add(pair[0])
303             leaves_encountered.add(pair[1])
304         roots = list(roots)
305         # Now add some pairs to make sure each second element is already part of some pair in
                the sequence read backwards,
306         # except for the last pair in the sequence
307         for i in range(len(roots) - 1):
308             seq.append((roots[i], roots[i + 1]))
309         return seq
310 
311 
312 class PhT:
313     def __init__(self, tree):
314         self.nw = tree
315         self.root = 0
316         self.leaves = get_leaves(self.nw)
317 
318     # Checks whether the pair (x,y) forms a cherry in the tree
319     def is_cherry(self, x, y):
320         if (x not in self.leaves) or (y not in self.leaves):
321             return False
322         px = -1
323         py = -1
324         for p in self.nw.predecessors(x):
325             px = p
326         for p in self.nw.predecessors(y):
327             py = p
328         return px == py
329 
330     # the new arc has length length(p,v)+length(v,c)
331     # returns false if v is not a degree-2 node
332     def clean_node(self, v):
333         if self.nw.out_degree(v) == 1 and self.nw.in_degree(v) == 1:
334             pv = -1
335             for p in self.nw.predecessors(v):
336                 pv = p
337             cv = -1
338             for c in self.nw.successors(v):
339                 cv = c
340             self.nw.add_edges_from([(pv, cv, self.nw[pv][v])])
341             if 'length' in self.nw[pv][v] and 'length' in self.nw[v][cv]:
342                 self.nw[pv][cv]['length'] = self.nw[pv][v]['length'] + self.nw[v][cv]['length'
                        ]
343             self.nw.remove_node(v)
344             return True
345         return False
346 
347     # reduces the pair (x,y) in the tree if it is present as cherry
348     # i.e., removes the leaf x and its incoming arc, and then cleans up its parent node.
349     # note that if px, and py have different lengths, the length of px is lost in the new
            network.
350     # returns true if successful and false otherwise
```

```
351     def reduce_pair(self, x, y):
352         if x not in self.leaves or y not in self.leaves:
353             return False
354         px = - 1
355         py = - 1
356         for p in self.nw.predecessors(x):
357             px = p
358         for p in self.nw.predecessors(y):
359             py = p
360         if self.is_cherry(x, y):
361             self.nw.remove_node(x)
362             self.leaves.remove(x)
363             self.clean_node(py)
364             return True
365         return False
366 
367     # Returns all reducible pairs in the tree involving x, where x is the first element
368     def find_pairs_with_first(self, x):
369         pairs = set()
370         px = -1
371         for p in self.nw.predecessors(x):
372             px = p
373         if self.nw.out_degree(px) > 1:
374             for cpx in self.nw.successors(px):
375                 if cpx in self.leaves:
376                     if cpx == x:
377                         continue
378                     pairs.add((x, cpx))
379         return pairs - {x, x}
380 
381     # Returns all reducible pairs in the tree
382     def find_all_reducible_pairs(self):
383         red_pairs = set()
384         for l in self.leaves:
385             red_pairs = red_pairs.union(self.find_pairs_with_first(l))
386         return red_pairs
387 
388 
389 def get_leaves(net):
390     return {u for u in net.nodes() if net.out_degree(u) == 0}
```