A Graph-Neural-Network Approach for Reconstructing Temporal Networks

T.J.M. Broeders

TU Delft

A Graph-Neural-Network Approach for Reconstructing Temporal Networks

by

T.J.M. Broeders

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday March 27, 2024 at 11:00 AM.

Student number:	4174879	
Project duration:	December 21, 2022 – March 27, 2024	
Thesis committee:	Dr.ir. L.J.J. van Iersel,	supervisor
	E.A.T. Julien,	Daily Co-Supervisor
	Dr. E. Demirović,	TU-Delft

Abstract

Reconstructing a minimum reticulation network from phylogenetic trees is used in evolutionary studies. In this thesis, we focus on finding temporal networks using cherry-picking sequences for binary trees with all taxa. Finding such a minimum reticulation temporal network is NP-hard.

We introduce an algorithm to find a minimum reticulation network with a running time of $O(2^n \text{poly}(n, t))$. In addition, this study explores potential enhancements to the algorithm through the utilisation of branch and bound.

Additionally, we introduce a similar algorithm to determine the existence of a temporal phylogenetic network. This algorithm is improved upon by integrating a new concept called cherry growing. This leads to a notable speed-up in performance.

Furthermore, we examine the application of Graph Neural Networks (GNN) in heuristics to find a cherry-picking sequence which can be used to construct a network. This is done by classifying leaves into *good*, which leads to optimal solutions, and *bad* leaves. To assess this, two types of data were employed: one simulating evolutionary models and the other employing a fully random approach. The best-performing GNN model has a 97.4% accuracy for evolution-based data and a 79.1% accuracy for random-based data.

The GNN models are implemented as predictors in two classes of heuristics. The first generates a cherry-picking sequence by repeatedly picking leaves. The second class of heuristics is based on a tree search heuristic. This tree-search-based heuristic outperforms the cherry-picking-based heuristic. Furthermore, the GNN heuristics outperform their random variant, even for problems substantially larger than the GNN was trained on.

We also examine the use of GNN in predicting the existence of a phylogenetic temporal network given a set of trees. The best-performing GNN found for this problem has an accuracy of 80.3%.

Contents

1 Introduction

Life on this planet started with one-cell organisms. These organisms eventually evolved into the currently living species, including mammals. The study of evolutionary history is known as *phylogenetics*. Within this research area, trees have become a popular tool for visualising the evolutionary relationships of sets of organisms [32] or *taxa*. The different taxa are shown at the endpoints of the trees, which are called *leaves*.

However, trees can not always show the whole picture. This is because, for example, the definition of a tree limits each taxon to having only one closest ancestor. Not all evolutionary relationships can be correctly represented this way. For example, hybrid species have two closest ancestors. Phylogenetic networks are a solution to this problem [32]. They allow for the incorporation of more complex scenarios by including events such as hybridisation and lateral gene transfer. These events can not be represented by the traditional phylogenetic tree.

The complexity of a phylogenetic network can be measured by how close it is to a phylogenetic tree. This is measured by the *reticulation number*, which is the number of edges that need to be deleted from the network to obtain a tree. For binary networks, this is equal to the number of nodes with two incoming edges. Such nodes are called *reticulations* or *hybridisation nodes*.

A rooted phylogenetic network is a directed acyclic graph, which can, for example, be constructed by combining multiple phylogenetic trees. We say that a network *displays* a tree if the ancestral relationships described by the tree are contained within the network. Given a set of phylogenetic trees, there is generally not a unique network displaying the tree set. One option for deducing a plausible network is to find a network with minimum reticulation number [26]. For data sets where hybridisations are rare events, networks with fewer reticulations are more likely. From this, we get the following optimisation problem: Given a collection $\mathcal{T}$ of binary phylogenetic trees, what is the minimum reticulation number of a network displaying $\mathcal{T}$. This problem was found to be NP-hard [5].

One interesting variant of this problem is the restriction to *tree-child networks*. Such networks have the condition that each non-leaf node must have at least one *tree node* as a child, these tree nodes are nodes with only one parent. Tree-child networks have a certain kind of so-called *cherry-picking sequences* with the same weight as the network's reticulation number [25]. This translates to finding a minimum reticulation network being equivalent to finding a cherry-picking sequence with minimal weight.

This problem can be solved using a fixed-parameter tractable (FPT) algorithm with $O((8k)^k poly(n,t))$ running time [40], where n represents the number of leaves of each tree, t refers to the number of trees, and k denotes the reticulation number of the constructed network. The FPT algorithm can solve large problems with a small k in a feasible time.

In this thesis, we focus on finding a *temporal phylogenetic network* with minimum reticulation number. Temporal networks are tree-child networks that adhere to a time constraint. Every tree node must occur strictly later than its parent, while reticulation nodes must occur at the same time as all their parents. In phylogenetics, this translates to a more realistic network as evolution follows these time constraints. However, this is only true if there is no missing information due to taxa being extinct or not included in the experiment, which can be an issue in a practical setting.

Temporal networks have their own version of cherry-picking sequences [19] which is related to but not identical to the version used for tree-child networks. Finding a temporal phylogenetic network with a minimum reticulation number is equivalent to finding a minimum weight cherry-picking sequence. This gave rise to an FPT algorithm [6] with $O(5^k \cdot n \cdot t)$ running time.

The first goal of this thesis is to find faster algorithms to obtain an exact solution. We introduce an exact solver with $O(2^n \text{poly}(n, t))$ running time, by translating the minimum weight cherry-picking sequence problem to the shortest path problem. This is done by introducing a branching algorithm. In addition, we examine several bounds for branch-and-bound algorithms. Furthermore, we examine a few practical improvements by reducing the number of branches. However, these additions did not lead to a significant decrease in running time.

Given a set of trees, there always exists a tree-child network displaying them and hence the type of cherry-picking sequences used for tree-child networks always exists. However, this is not the case for the temporal network. For this variant of cherry-picking sequences, determining whether a cherry-picking sequence exists is NP-complete [13, 12]. Consequently, determining whether a temporal network displaying a set of trees exists is NP-hard. Moreover, note that these problems are already NP-hard for two trees.

In this thesis, we introduce an exact solver. Furthermore, we introduce the concept of cherry-growing algorithms, which, combined with a cherry-picking algorithm, significantly improves the running time.

With the exponential growth of the running time of these algorithms, there is a demand for polynomial time heuristics. However, this topic is mainly unexplored except for very recent work on the tree-child problem [3, 44]. These heuristics try to find low-weighted cherry-picking sequences. One of these heuristics [3] uses machine learning to help determine which cherries are good to pick.

We expand on this research by examining the temporal variant. For this variant, a cherry-picking sequence consists of leaves, while for the tree-child variant it consists of pairs of leaves. We compare random picking with machine learning to predict which to pick. Furthermore, we improve on these heuristics by implementing a network search, which is an adaptation of the Monte Carlo tree search. We also find that the machine-learning-based heuristic mainly outperforms its random counterpart, especially for practical problems.

In this thesis, we examine the use of machine learning in predicting which leaves are good to pick to solve the problem of finding a temporal phylogenetic network with minimum reticulation number displaying a set of trees. Machine Learning is a branch of artificial intelligence dedicated to crafting and exploring algorithms capable of learning from data. This field has made great strides, which has led to large language models [22] and facial recognition [24]. This is generally done with the use of Neural Networks (NN).

The use of machine learning for predictions in NP-hard problems has been done before with various successes [2, 27, 16]. A preferable capability of the model is that it can extrapolate the data. Neural networks are a strong candidate to be used for such predictions, as they can learn complex relationships [29]. An adaptation to this model is a Graph Neural Network (GNN) [35]. This method implements a graph to help distinguish relationships between data. Each node of this graph is its own NN, and neighbouring nodes can exchange information with each other if they are connected by an edge. Such exchanges are called *message passing*. GNN has the following useful properties. The first is that two graphs which are the same by permutation give the same result. Secondly, the input is a graph of undetermined size. This makes the model easily applicable to problems with different sizes. However, GNNs have their own problems. One is the limitation of passing information between far-away nodes [1]. Another is the inability to easily determine its structure [14, 11, 45].

In this thesis, we examine the use of GNN in predicting good leaves to pick and predicting if an instance has a temporal solution. For this, we implement two different graphs for the GNN. The first consists of reducing the tree set to a single graph, without loss of information. However, for this graph, leaves can be far away. This means that these leaves cannot communicate if we use the standard GNN. For this, we rediscovered the Directed Acyclic GNN (DAGNN) variant [38, 4]. The second input graph we examine is the complete graph on all leaves. Here, we use relationships between two nodes as edge features.

To determine the performance of the models in finding good leaves to pick, we generated two types of data. The first is biologically plausible, simulating evolutionary events [33, 28, 21]. For this type of data with 25 leaves, we found accuracies of 94.1% and 96.8% for the DAGNN models and the model using the complete graph respectively.

The second type of data contains more complex instances, as it consists of randomly generated trees. For this type of data with 20 leaves, we found an accuracy of 73.2% for the DAGNN and 77.7% for the method using the complete graph.

We also examined the capabilities of GNN in predicting if there exists a temporal phylogenetic network displaying a given set of trees. This resulted in an 80% accuracy for problems with 35 leaves.

The structure of this thesis is as follows. We start with the preliminaries, in which we introduce the minimum reticulation problem and if there exists a temporal phylogenetic network displaying a given set of trees. Furthermore, we also define the cherry-picking sequence. This chapter also describes the basis behind neural networks and graph neural networks.

In the third chapter, we examine methods for efficiently solving the minimum hybridisation problem and the corresponding problem to decide whether a temporal solution exists. This is done by first reducing the search space for the problem. We first introduce simple branching algorithms for both problems. After, we discuss possible improvements, like using lower bounds and finding good leaves. Furthermore, we examine substructures to determine if an instance has no temporal network. Additionally, we introduced the concept of cherry-growing, and we introduced a method to remove redundancies in the tree set. The chapter ends with the experimental results of the performance of the exact solvers of both problems.

Chapter 4 delves into DAGNN and explores possible models by examining various subproblems. In the first section, we examine the performance of the DAGNN compared to the general GNN and the recursive Neural Network. In the second section, we introduce the two different graphs which are used in this thesis for the GNN. Furthermore, we observe the performance of these models in finding substructures.

In Chapter 5 we predict good leaves using GNN. In the first section, we discuss how we generated the data used in this thesis. In the second section, we introduce the models used for predicting good leaves and examine the performance of small deviations to the models. The chapter concludes by examining the usefulness and computational cost of the features used in the models.

The usage of GNN in predicting if a problem has a temporal solution is discussed in Chapter 6. This chapter starts by explaining how the data was generated and ends with the experimental results.

The heuristics are discussed in Chapter 7, here we use the GNN models of Chapter 5 for the predictions and examine the practical performance. We start by introducing the heuristics for finding low-weighted cherry-picking sequences. Afterwards, we discuss the effect of adjusting the weights of the loss function and thereby prioritising the accuracy of either good or bad leaves. The chapter concludes with experimental results for the heuristic.

After these chapters, the thesis ends with a conclusion and a discussion chapter.

2 Preliminaries

In this chapter, we first introduce the definitions for phylogenetic trees and networks. In this thesis, we only examine binary phylogenetic trees. Then, we introduce the minimum temporal hybridisation number and the temporal decision problem. We conclude the phylogenetic theory with the introduction of the cherry-picking sequence. Here, we show that the minimum weight of such sequences is the same as finding the minimum temporal hybridisation number.

For the second part, we will describe Neural Networks and its adaption, Graph Neural Networks. These models will be used in this thesis to predict which leaves should be picked and to predict if a set of trees has a temporal solution or not.

2.1. Phylogenetic Network

A graph is a convenient and popular way of representing relationships between objects: in a graph, objects are represented by nodes, and relationships between objects are represented by edges. In this thesis, we only examine directed graphs. A directed graph $G = (V, E)$ consists of a finite set of nodes V and a finite set of directed edges E also known as *arcs*. Each edge $e \in E$ is of the form (v, w), with $v, w \in V$. In a directed graph we say that the arc $e = (v, w)$ is directed from v to w. We say that edge e is an out-edge of v and an in-edge of w. The indegree of a node u is the number of in-edges of u and the outdegree of u is the number of out-edges of u. The degree of u is the sum of its indegree and outdegree. These are denoted by $\deg(u)$, $\text{indeg}(u)$ and $\text{outdeg}(u)$ respectively.

Definition 2.1.1 (Rooted DAG)**.** A *directed acyclic graph* (DAG) is a directed graph without cycles. A DAG is called *rooted* if it contains precisely one node with an indegree of 0, called the *root*.

Given a DAG, we call a node v a *leaf* if $\text{outdeg}(v) = 0$. We call a node an *internal* node if it is not a leaf, i.e., if $\text{outdeg}(v) > 0$. The root is an internal node. We can now define the rooted phylogenetic tree as a subclass of DAG with the property that it has no internal nodes with indegree 1 and outdegree 1.

Definition 2.1.2 (Rooted Phylogenetic Tree)**.** Given a set of taxa $\mathcal{X}$, a *phylogenetic tree* T on $\mathcal{X}$ consists of a rooted tree $T = (V, E)$, in which every node v besides the root has $\text{degree}(v) \neq 2$, together with a taxon labelling $\sigma : \mathcal{X} \to V$ that assigns exactly one taxon to every leaf and none to any internal node. We say that T is an $\mathcal{X}$-tree or a tree on $\mathcal{X}$

If a rooted tree contains non-root nodes with a degree of 2, a phylogenetic tree can be obtained by contacting these nodes. This is done for a node n with edges (a, n), (n, b) by replacing the node and edges with a single edge (a, b). We call a rooted phylogenetic tree *binary* if all internal nodes have outdegree 2. In this thesis, we only examine binary trees. All trees are thus assumed binary except when explicitly mentioned.

Let T be an $\mathcal{X}$-tree and $\mathcal{X}' = \{x_1, x_2, ..., x_k\}$ a subset of $\mathcal{X}$. We refer to $T \backslash \mathcal{X}'$ as the phylogenetic tree obtained by deleting the leaves in $\mathcal{X}'$ and repeatedly contracting all nodes with both indegree 1 and outdegree 1. We also write $T\left[-x_1, x_2, ..., x_k\right]$ or $T[-\mathcal{X}]$.

In this thesis, we examine rooted binary phylogenetic trees. We call a rooted acyclic digraph a phylogenetic network $\mathcal{N}$ on $\mathcal{X}$ if it satisfies the following properties:

- The *root* has an indegree of 0 and an outdegree of at least 2;
- $\mathcal{X}$ is the set of *leaves* of the network. These nodes have an indegree of 1 and an outdegree of 0. If a node is not a leaf then we call it an *internal* node.
- All remaining nodes have either an indegree of 1 and an outdegree of at least 2 or are *hybridisation* nodes that have an indegree of at least 2 and an outdegree of 1.

Edges ending in a hybridisation node are called *hybridisation arcs*, all other arcs are *tree arcs*. A node is called a *tree node* if it is not a hybridisation node. A network is called a *tree child* if every internal node has at least one tree arc.

Definition 2.1.3 (Temporal Network)**.** A phylogenetic network $\mathcal{N} = (V, E)$ is called *temporal* if it is tree-child and there exists a map $t : V \to \mathbb{R}^+$, called the temporal labelling, such that for all nodes $u, v \in V$ we have $t(u) = t(v)$ when (u, v) is a hybridisation arc and $t(u) < t(v)$ when (u, v) is a tree arc.

Let $\mathcal{N}$ be a phylogenetic $\mathcal{X}$-network and $\mathcal{T}$ a set of phylogenetic $\mathcal{X}$-trees. We say that $\mathcal{N}$ displays $\mathcal{T}$ if each tree in $\mathcal{T}$ can be obtained from $\mathcal{N}$ by a sequence of hybridisation arc deletions and degree-2 node contractions. An example is given in Figure 2.1.

For a temporal network $\mathcal{N}$, the hybridisation number $h(\mathcal{N})$ is also called the reticulation number, and it is defined as

$$h_t(\mathcal{N}) := \sum_{v \neq \rho} (\text{indeg}(v) - 1)$$

with ρ the root. This number is equivalent to the number of edges that needs to be deleted to get a tree. With this, we define the following optimization problem:

Problem: Minimum Temporal hybridisation Number.
Instance: A set $\mathcal{T}$ of phylogenetic binary trees on $\mathcal{X}$;
Question: What is the smallest reticulation number for a temporal network on $\mathcal{X}$ which displays $\mathcal{T}$.

The minimum temporal hybridisation problem can also be written as:

$$\inf \{h_t(\mathcal{N}) : \mathcal{N} \text{ is a temporal network on } \mathcal{X} \text{ that displays } \mathcal{T}\}.$$

This problem was shown to be NP-hard [5]. The infimum is taken because there is not always a temporal network that displays a tree set. This leads to the following decision problem:

Problem: Temporal Network.
Instance: A set $\mathcal{T}$ of phylogenetic binary trees on $\mathcal{X}$;
Question: Does there exist a network $\mathcal{N}$ on $\mathcal{X}$ that displays $\mathcal{T}$.

This problem was shown to be NP-complete [12, 13].

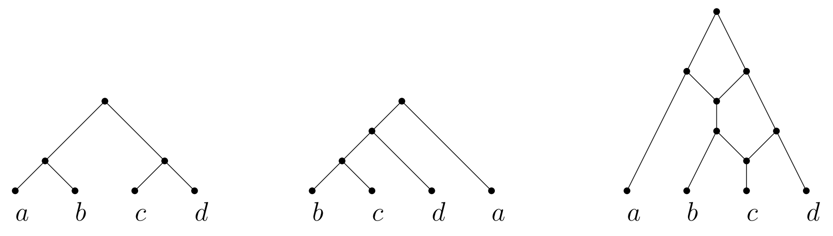


Figure 2.1: The two trees on the left are displayed in the temporal network on the right.

2.2. Cherry-Picking Sequence

Let $\mathcal{T}$ be a set of $\mathcal{X}$-trees. A pair of distinct leaves $\{a, b\}$ is called a *cherry* if they have the same parent in a tree. We say $\{a, b\} \in T$ if it is a cherry in tree T, and we say $\{a, b\} \in \mathcal{T}$ if there is a $T \in \mathcal{T}$ with

$\{a, b\} \in T$. We say that a leaf is *pickable* if it is in a cherry in each tree in $\mathcal{T}$. Furthermore, we call a cherry *trivial* if it is a cherry in all trees.

Let $(x_1, x_2, ..., x_n)$ be an ordering of all leaves in $\mathcal{X}$. This sequence is a *cherry-picking sequence* for a set of phylogenetic trees $\mathcal{T}$ precisely if each x_i with $i \in [n-1]$ is a leaf of cherry in each tree contained in $\mathcal{T}[-x_1, x_2, ..., x_{i-1}]$. This leads to the following two theorems.

Theorem 2.2.1. [19] Let $\mathcal{T}$ be a set of trees on $\mathcal{X}$. There exists a temporal network $\mathcal{N}$ that displays $\mathcal{T}$ if and only if there exists a cherry-picking sequence for $\mathcal{T}$

Definition 2.2.2. Given a set of phylogenetic trees $\mathcal{T}$ and tree $T \in \mathcal{T}$. $N_T(x) := \{y \in \mathcal{X} : (x, y) \in T\}$ is the *neighbour* of leaf x in tree T. We also define $N_{\mathcal{T}}(x) := \{y \in \mathcal{X} : (x, y) \in \mathcal{T}\}$ as the *neighbourhood* of x.

Definition 2.2.3. For a set of binary trees $\mathcal{T}$ containing a leaf x, we define $w_{\mathcal{T}}(x) = |N_{\mathcal{T}}(x)| - 1$ as the weight of x in $\mathcal{T}$.

Definition 2.2.4. Given a set of trees $\mathcal{T}$ and a cherry-picking sequence $s = (s_1, s_2, ..., s_n)$ for this set, we define $w_{\mathcal{T}}(s) := \sum_{i=1}^{n-1} w_{\mathcal{T}[-s_1,s_2,...,s_{i-1}]}(s_i)$ as the *weight* of the cherry-picking sequence. Furthermore, we define $s(\mathcal{T})$ as the minimum weight of all possible cherry-picking sequences of $\mathcal{T}$.

Theorem 2.2.5. [19] Let $\mathcal{T}$ be a set of phylogenetic trees on $\mathcal{X}$, such that there exists a temporal network that displays $\mathcal{T}$. Then $h_t(\mathcal{T}) = s(\mathcal{T})$.

Theorem 2.2.5 shows that the temporal hybridisation number problem is equivalent to finding the minimum cherry-picking sequence. Furthermore, for each cherry-picking sequence, there exists a corresponding network. An example of a cherry-picking sequence is shown in Figure 2.2. In this example, we see that the sequence (c, b, a, d) is a valid cherry-picking sequence, as each leaf is pickable in the sequence.

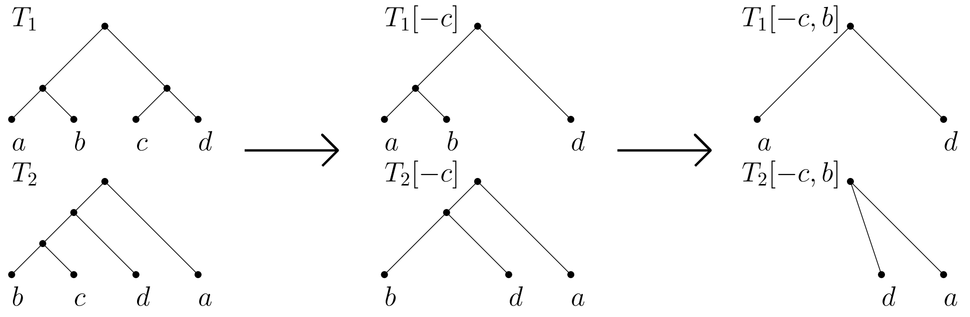


Figure 2.2: Two trees and there corresponding sub trees after first picking c and then b. This shows that (c, b, a, d) is a cherry-picking sequence with $w_{\mathcal{T}}(c, b, a, d) = 2$.

Given a tree set $\mathcal{T}$ and cherry-picking sequence $s = (s_1, s_2, ..., s_n)$. The network corresponding to the cherry-picking sequence is a network displaying $\mathcal{T}$ with a hybridisation number equal to the weight of s. Such a network can be reconstructed by going through the sequence in reverse order. We shall show how to reconstruct a network using induction. Assume we have a network $\mathcal{N}_{\mathcal{X}\setminus\{s_1,s_2,...,s_i\}}$ with a leaf set $\{s_{i+1}, s_{i+2}, ..., s_n\}$, then create node s_i and its parent p_{s_i} with the edge (p_{s_i}, s_i). Then for each leaf x in $N_{\mathcal{T}[-s_1,s_2,...,s_{i-1}]}(s_i)$ split its incoming edge into two edges, whereby creating a node p_x which is the parent of x, and adding the edge (p_x, p_{s_i}). This gives a network $\mathcal{N}_{\mathcal{X}\setminus\{s_1,s_2,...,s_{i-1}\}}$ displaying $\mathcal{T}[-s_1, s_2, ..., s_{i-1}]$. An example is shown in Figure 2.3. Here we reconstruct a network from the tree set of Figure 2.2 and cherry-picking sequence (c, b, a, d).

2.3. Neural Network

Given a dataset $\mathcal{D} \subset \mathbb{R}^n \times \mathbb{R}^m$, which consists of pairs $(\mathbf{x}, \mathbf{y})$, with $\mathbf{x}$ the input vector and $\mathbf{y}$ the output vector, machine learning deals with finding a function $f : \mathbb{R}^n \to \mathbb{R}^m$ such that $f(\mathbf{x}) \approx \mathbf{y}$. Neural networks fall under machine learning and are modelled to replicate brain neurons. This is generally

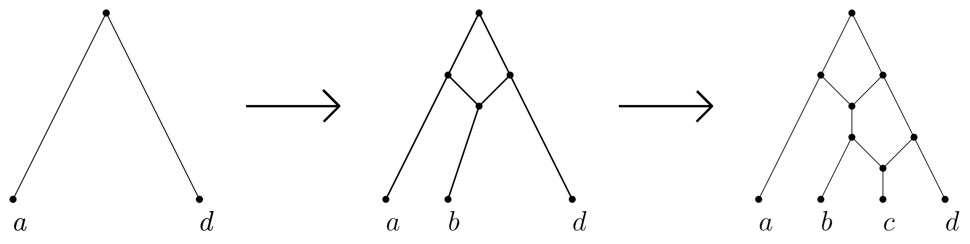


Figure 2.3: The reconstruction of a network from the tree set in Figure 2.2 with cherry-picking sequence (c, b, a, d).

done in the following way. Given an input vector $\mathbf{h}^{(0)} = \mathbf{x}$ and the number of layers l, for each layer $i \in [l]$ we calculate the next vector $\mathbf{h}^{(i)}$ by

$$\mathbf{h}^{(i)} = \sigma^{(i)}(\mathbf{W}_1^{(i)}\mathbf{h}^{(i-1)} + \mathbf{W}_2^{(i)}),$$

with $\sigma^{(i)}$ a non-linear activation function and weights matrix and weighted bias $\mathbf{W}_1^{(i)}$, $\mathbf{W}_2^{(i)}$. The last layer $\mathbf{h}^{(out)} = \mathbf{h}^{(l)}$ the output layer. This is visually shown in Figure 2.4, where each node represents a variable and each column represents a layer. The activation functions $\sigma^{(i)}$ elevate the process from ordinary linear calculations. These functions are chosen beforehand and are non-linear.

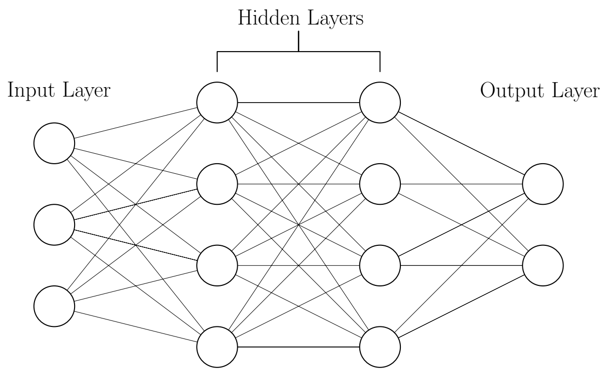


Figure 2.4: A visual representation of the structure of a neural network. Each column represents a layer and has its own channels represented by the circles.

The activation function used in this paper is the rectified linear unit also known as ReLU. This function is defined as

$$\text{ReLU}(x) := \max(0, x).$$

The ReLU is a common choice for the activation function as it causes fewer vanishing gradient problems [18]. However, it has its problems, such as being not differentiable at 0 and the dying ReLU problem [18]. Another function mentioned in this thesis is the LeakyReLU, defined by

$$\text{LeakyReLU}(x, a) := \begin{cases} x, & \text{if } x \geq 0, \\ ax, & \text{if } x < 0. \end{cases} \tag{2.1}$$

with a a constant to control the angle of the negative slope. Additionally, we use the function Softmax : $\mathbb{R}^n \to \mathbb{R}^n$ defined by

$$\text{Softmax}(\mathbf{x})_i := \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}, \quad \text{for } i = 1, 2, \ldots, n. \tag{2.2}$$

2.3.1. Loss Function

Given a neural network, we want to find the values for all weights $\mathbf{W}^{(i)}$, for $i \in [l]$, such that $\mathbf{h}^{(out)} \approx \mathbf{y}$ for all pairs $(\mathbf{x}, \mathbf{y})$ in our dataset. We quantify this approximation by using a loss function.

In this paper we only use classification. This means that the output shows which class the data is an element of. Classification is thus a discretization with the number of classes m as possible answers.

This is done by having $\mathbf{y}$ be zero except for the class of which the pair $(\mathbf{x}, \mathbf{y})$ is part of. This means that $y_i = 0$ for all $i \in [m]$ except for its class j for which we have $y_j = 1$. This discrete vector $\mathbf{y}$ is then compared with the output vector $\mathbf{h}^{(out)} \in \mathbb{R}^m$ using the following function:

$$loss(\mathbf{h}, \mathbf{y}) = -\frac{1}{m} \sum_{i \in [m]} \log \frac{\exp(h_i)}{\sum_{j \in [m]} \exp(h_j)} y_i.$$

In short, this function looks at the value of $\exp(h_i)$ and if $\exp(h_i)$ is larger than $\exp(h_j)$ for all $j \neq i$, then the loss goes to zero. The total loss over a dataset is calculated by averaging each loss.

This, however, can give a skewed view. Take, for example, two datasets with two classes, such that the first dataset has 99% of its data in the first class, and the second dataset has half of its data in the first class. Then NN that classifies all data to the first class, has a substantially lower loss function for the first dataset. This means that total loss is influenced by the distribution of the data.

We shall therefore use the following weighted version:

$$loss(\mathbf{h}, \mathbf{y}) = -\frac{1}{m} \sum_{i \in [m]} w_i \cdot \log \frac{\exp(h_i)}{\sum_{j \in [m]} \exp(h_j)} y_i,$$

with $w_i = \frac{1}{N_i}$ and N_i the number of elements that belong to class i. The total loss over the dataset is thus calculated by:

$$loss_{\text{total}} = \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} loss(f(\mathbf{x}), \mathbf{y}).$$

We can quantify the performance of the model using the loss function. A more practical quantification is the accuracy of the model. The accuracy is the ratio of correct guesses by the model. For this, we use the classification function:

$$cl(\mathbf{x}) = \arg\max_i x_i.$$

An item belongs to the class i if it has the highest value on the i-th position of the vector.

The accuracy of the model can be split into the accuracy for each class i.

$$\text{acc}_i = \frac{\sum_{(\mathbf{x}, \mathbf{y})} y_i \left[cl(f(\mathbf{x})) == i \right]}{\sum_{(\mathbf{x}, \mathbf{y})} y_i == i}.$$

This is the accuracy of the model for the respective class. These values are then used to calculate the average accuracy by

$$\text{acc} = \frac{1}{m} \sum_i \text{acc}_i,$$

where m denotes the number of classes. Calculating the accuracy of the models in this way gives a better representation than the overall accuracy.

2.3.2. Adam

With the loss function defined, we now want to find the set containing all weights $\mathcal{W} = \left\{ \mathbf{W}^{(1)}, \mathbf{W}^{(2)}, ..., \mathbf{W}^{(l)} \right\}$ for which the neural network model $f_{\mathcal{W}}(\mathbf{x})$ has the lowest cost. This leads to the following minimization problem:

$$\min_{\mathcal{W}} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} loss \left(f_{\mathcal{W}}(\mathbf{x}), \mathbf{y} \right).$$

To find the best $\mathcal{W}$ we use a gradient descent method called Adam [23], which gets its name from adaptive moment estimation. Note that a gradient descent method finds local minima. This means that rerunning the optimisation process with a different initialisation could give different results.

Adam incorporates earlier gradients to give a sense of momentum. Like a ball rolling down a hill, the optimiser builds up momentum which helps in leaping out of local minima.

2.3.3. Datasets

The dataset $\mathcal{D}$ in machine learning is a finite subset of the domain of all possible input and output pairs $\mathbf{x} \times \mathbf{y}$. When an NN model is trained on $\mathcal{D}$, it is preferable that it is generally correct over the whole domain and not only on $\mathcal{D}$. However, this is not always the case. To combat this, the dataset is generally split into two disjoint sets, the larger *training set* $\mathcal{D}_{train}$ and *validation set* $\mathcal{D}_{val}$ sometimes also known as the *test set*.

Generally, there can be two possible problems with a dataset. The first is a bad representation of the whole domain. An example of this is a machine learning model that predicts if apples are spoilt based on a picture. If the model is trained on only green apples, then it can not accurately predict all apples. This problem is sometimes hard to observe, especially if both $\mathcal{D}_{train}$ and $\mathcal{D}_{val}$ are from the same biassed dataset or are generated with the same biassed procedure. Note that examining only a certain subdomain can result in better models for the subdomain. Looking at only green apples gives generally higher accurate results for green apples.

The second problem is that of the relative size between the hypothesis class $\mathcal{H} = \{f_{\mathcal{W}} : \mathcal{W}\}$ and the training data. This is because a larger hypothesis class means a larger search domain for the optimization problem, which means better results for the training data, but at the cost of all other data. This is also known as overfitting. An example is using polynomials. If we have n datapoint, then there exists a unique polynomial with (n-1)-degree, that gives perfect results for all datapoints. However, this could result in worse domain accuracy than a 2-degree polynomial. There are thus two ways to solve overfitting, increasing the size of the training data or decreasing the hypothesis class. For neural networks, this means simplifying the model.

2.3.4. K-Fold Cross-Validation

The accuracy of a model depends on different aspects. As aforementioned, the weight optimization can find different local minima depending on its initialization. This means that rerunning the optimization process with a different initialization can result in a better model. However, there is also a bias caused by the split of the dataset into training and validation sets. We compensate for these two stochastic deviations by using k-fold cross-validation [20]. The dataset is divided into k equally sized disjoint datasets, called folds. The optimization process is then executed k times with for time i the i-fold is taken as the validation set and all other folds as the training set.

Using k-fold cross-validation gives k different results. To get the average performance of the model we calculate the average and standard deviation for both loss and accuracy using

$$\bar{x} = \frac{1}{k}\sum_{i=1}^{k} x_i, \qquad s_x = \sqrt{\frac{1}{k-1}\sum_{i=1}^{k}(x_i - \bar{x})^2}, \tag{2.3}$$

with x_i representing the stochastic results for either the loss or accuracy.

2.4. Graph Neural Network (GNN)

In this thesis, we use the graph neural network (GNN) to predict which leaves should be picked and to predict if a set of trees has a temporal solution. We discuss the mathematical concepts of the GNN in this section.

A GNN is a variation of the neural network, which mainly solves two problems. The first is permutation invariance. This means that the permutations of certain input values should give the same results. For example, a picture of a dog is still a picture of a dog if it is upside down or mirrored. The second problem is the incompatibility with different-sized input data. For example, digital pictures of dogs can have different sizes.

The input for a graph neural network consists of a digraph $G = (V, A)$, with V the set of vertices and A the set of arcs, where each vertex $v \in V$ has its own input $\mathbf{x}_v \in \mathbb{R}^n$. For each layer $i \in [l]$ in the GNN, we calculate the vector $\mathbf{x}_v^{(i)}$ for each vertex v by

$$\mathbf{h}_v^{(i)} = \sigma_i\left(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \gamma_i\left(\mathbf{h}^{(i-1)}, \bigoplus_{(u,v)\in E} \phi_i\left(\mathbf{h}_v^{(i-1)}, \mathbf{h}_u^{(i-1)}\right)\right)\right),$$

with σ_i an activation function, γ_i, ϕ_i differentiable functions and where $\bigoplus$ denotes a differentiable, permutation invariant function like sum, mean or max. A Graph Neural Network layer has the same calculations for a node as a neural network layer, however, with an addition which depends on its neighbours. This setup gives the same result for the permutation invariant graphs. Furthermore, different-sized graphs can be used to train the same GNN.

There are different ways to classify a GNN, these are for the nodes, arcs and the graph itself. The values of the output layer $\mathbf{h}_v^{(out)}$ can simply be compared to an output vector $\mathbf{y}_v$ for node classification. For graph classification, we first combine the values of $\mathbf{h}_v^{(out)}$ for each $v \in V$. This is done by

$$\mathbf{h}_G^{(out)} = \bigoplus_{v \in V} \mathbf{h}_v^{(out)},$$

where $\bigoplus$ denotes a differentiable, permutation invariant function like sum, mean or max. This can then be compared to an output vector $\mathbf{y}_G$.

2.4.1. Message-Passing Layers

In this paper, we use different message-passing layers. These layers generally consist of two parts. The first is a weighted part of its previous values. This makes it the same as a neural network if the graph has no edges. The second part consists of data it gains from its neighbours. This has been done in different ways.

The first message-passing layer we introduce is the GraphSAGE operator [17]. This is calculated by

$$\mathbf{h}_i' = \mathbf{W}_1 \mathbf{h}_i + \mathbf{W}_2 \, \mathrm{mean}_{(i,j) \in E} \, \mathbf{h}_j.$$

This is a simple message-passing layer. The information a node receives from its neighbours is averaged, this can be impractical as not all neighbours could be giving important information.

This can be solved by including an attention $\alpha_{i,j}$ term which is itself learnable. This term helps distinguish the neighbours from each other. These attention layers can be improved by using edge features $\mathbf{e}_{ij}$. These features are able to quantify relations between nodes.

The second message-passing layer we use is the graph attention operator [42]. Calculated by

$$\mathbf{h}_i' = \alpha_{i,i} \mathbf{W}_1 \mathbf{h}_i + \sum_{(i,j) \in E} \alpha_{i,j} \mathbf{W}_2 \mathbf{h}_j,$$

$$\alpha_{i,j} = \frac{\exp\left(\mathbf{a}^\top \mathrm{LeakyReLU}\left(\mathbf{W}_3 \mathbf{h}_i + \mathbf{W}_4 \mathbf{h}_j + \mathbf{W}_5 \mathbf{e}_{ij}\right)\right)}{\sum_{(u,v) \in E \cup (u,u)} \exp\left(\mathbf{a}^\top \mathrm{LeakyReLU}\left(\mathbf{W}_3 \mathbf{h}_u + \mathbf{W}_4 \mathbf{h}_v + \mathbf{W}_5 \mathbf{e}_{uv}\right)\right)}.$$

The operator mainly used is the graph transformer operator [36], which is calculated by

$$\mathbf{h}_i' = \mathbf{W}_1 \mathbf{h}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j} \left(\mathbf{W}_2 \mathbf{h}_j + \mathbf{W}_5 \mathbf{e}_{ij}\right),$$

$$\alpha_{i,j} = \mathrm{softmax}\left(\frac{(\mathbf{W}_3 \mathbf{h}_i)^\top (\mathbf{W}_4 \mathbf{h}_j + \mathbf{W}_5 \mathbf{e}_{ij})}{\sqrt{d}}\right)$$

For last, we also introduce the residual gated graph convolution [7], which is calculated by

$$\mathbf{h}_i' = \mathbf{W}_1 \mathbf{h}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j} \cdot \left(\mathbf{W}_2 \mathbf{h}_j + \mathbf{W}_3 \mathbf{e}_{ij}\right),$$

$$\alpha_{i,j} = \mathrm{sigmoid}(\mathbf{W}_4 \mathbf{h}_i + \mathbf{W}_5 \mathbf{h}_j + \mathbf{W}_6 \mathbf{e}_{ij}).$$

2.4.2. Normalisation Layers

Data can come in different kinds of ranges, which influences the backwards propagation. The normalisation of the data equalises these ranges.

In this thesis, we use two types of normalisation layers. The first is the InstanceNorm [9], which is calculated by

$$\mathbf{h}_i' = \frac{\mathbf{h} - \mathrm{E}[\mathbf{h}]}{\sqrt{\mathrm{Var}[\mathbf{h}_i] + \epsilon}},$$

with $\epsilon = 10^{-5}$. This term helps with numerical stability. The mean and standard deviation are calculated per dimension separately for each graph.

The second normalisation we use is the Graph Normilization [9]. This has a similar calculation to the InstanceNorm. However, it has three learnable weights: α, β and γ. This normalisation is calculated by

$$\mathbf{h}_i' = \frac{\mathbf{h}_i - \alpha \cdot \mathrm{E}[\mathbf{h}]}{\sqrt{\mathrm{Var}[\mathbf{h} - \alpha \cdot \mathrm{E}[\mathbf{h}]] + \epsilon}} \cdot \gamma + \beta.$$

3 Exact Solvers and Improvements

The minimum temporal hybridisation number can be solved by finding the cherry-picking sequence with the lowest weight. In this chapter, we commence by presenting two branching algorithms, one for the minimum temporal hybridisation number and one for the temporal network decision problem. Subsequently, we explore potential bounds to expand the branching algorithm into a branch-and-bound approach. Following that, we examine scenarios that do not necessitate branching. In the section after that, we examine substructures which infer no temporal solutions. Afterwards, we delve into two translations of the problem, concluding with an experimental comparison of the algorithms.

3.1. Branching Algorithms

Finding the cherry-picking sequence with the least weight can be done with a basic branching algorithm. This works by dividing the problem into multiple smaller subproblems. In our case, this means picking a different leaf for each subproblem. Such a brute force method has a computational cost bounded by $O\left(n! \text{ poly}(n,t)\right)$. This can be drastically reduced with the next observations.

Observation 3.1.1. Given a phylogenetic tree T, taxa $\mathcal{X}$ and subset $\mathcal{X}' \subset \mathcal{X}$. Let $s = (x_1, ..., x_i)$ be a sequence in which the elements of $\mathcal{X}'$ are picked with $i \leq |\mathcal{X}|$. Then, for a permutation s' of s, we have $T[s] = T[s']$.

From Observation 3.1.1, we derive that generally, the brute force method gets the same subproblem multiple times. From this observation, it follows that we only need to calculate $T|\mathcal{X}'$ for each subset $\mathcal{X}' \in 2^{\mathcal{X}}$ once at most. This reduces the running time to $O\left(2^n \text{ poly}(n,t)\right)$.

A tree set with n leaves has 2^n subproblems. These numerous subproblems can be grouped by the number of elements in the subset. This grouping is useful, as picking a leaf reduces the set of leaves by one. Using this grouping, a visualisation of the improved branching method is shown in Figure 3.1. This figure shows all possible cherry-picking sequences for the tree set in Figure 2.2. Each unique downward path from $\{\}$ to $\{a, b, c, d\}$, corresponds to a unique cherry-picking sequence.

Another way to explain the problem is to observe the problem as a directed hypercube. For which each vertex represents a subset of $\mathcal{X}$ and each arc represents the act of picking a leaf. This incurs some arcs to be non-traversable. The minimum reticulation problem is equal to the least weighted path from $\{\}$ to $\mathcal{X}$. The temporal decision problem also becomes equal to the decision problem if there exists a path from $\{\}$ to $\mathcal{X}$.

For the minimum weight cherry-picking sequence, we introduce the standard branching Algorithm 1. This algorithm works by branching the least-weighted subproblem. Furthermore, the algorithm keeps track of the best route so far. Note that this process is equivalent to Dijkstra's algorithm [31] for the directed hypercube.

The algorithm checks repeatedly if a leaf set is contained in a list. This can be implemented in different ways, with a degree of different time complexity. These complexities can be $O(2^n)$, $O(n)$, or $O(1)$ depending on the data structure used. For example, the $O(n)$ can be obtained by using a binary

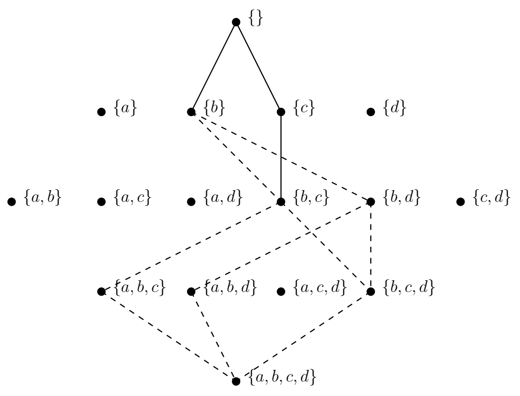


Figure 3.1: For $\mathcal{X} = \{a, b, c, d\}$, each point represents a reduced tree set, which is reduced by their denoted set. The lines represent the branching for the tree set in Figure 2.1, with a solid line representing a weight of 1, and a dashed line a weight of 0.

tree data structure. Furthermore, using a 2^n-sized list to track all possible subsets leads to $O(1)$, but this is at the cost of memory space.

Using the two lower complexities lets us retain $O\left(2^n \text{poly}(n, t)\right)$. However, using a higher complexity $O(2^n)$ leads to $O\left(4^n \text{poly}(n, t)\right)$. Finding the subproblem with the smallest k can also be done in different ways. Note that the value k is an integer and is bounded by $k < n^2$. This allows us to create at most n^2 different 'k'-boxes to put our subproblems in.

A similar algorithm can be used to solve the temporal decision problem. For this problem, the algorithm is done when it finds a cherry-picking sequence. This makes a depth-first search perform better than a breadth-first search. Therefore, we introduce the depth-first-search Algorithm 2 for the temporal decision problem.

This algorithm has the same time complexity issue when looking up if a subproblem has already been explored, which means that it also depends on the data structure used. Note that the last added item is the subproblem with the smallest number of leaves $|\mathcal{X}|$.

Both algorithms discussed in this section have a time complexity of $O\left(2^n \text{poly}(n, t)\right)$. In the subsequent sections of this chapter, we will examine potential improvements by reducing the number of branches and implementing lower bounds. However, these improvements do not reduce the theoretical time complexity.

3.2. Branch and Bound

A general improvement of a branching algorithm is the branch and bound. This method uses a lower and upper bound to prune branches, reducing the running time. Two aspects are important for practical use. It should be close to the true value, and it should be fast to compute. For example, problems with no trivial cherries have a lower bound of 1, as each pickable leaf has a non-zero weight.

In this section, we explore potential lower bounds. We conclude by introducing a branch-and-bound algorithm.

The lower bounds in this section generally follow from a subset of leaves. It is, therefore, easier to only observe the relevant part of the cherry-picking sequence.

We call a sequence $s' = (s_{I_1}, s_{I_2}, ..., s_{I_m})$ with $I_1, .., I_m \in [n]$ a subsequence of a cherry-picking sequence $s = (s_1, s_2, ..., s_n)$ if $I_i < I_{i+1}$ for all integers $1 \leq i < m \leq |\mathcal{X}|$. Given a subset $\mathcal{X}' \subseteq \mathcal{X}$, we say that a subsequence *displays* $\mathcal{X}'$ if the subsequence is an ordering of all elements of $\mathcal{X}'$ and only these elements. We shall denote this with $s_{\mathcal{X}'}$.

The first lower bound we examine is the bound resulting from the number of neighbours of a leaf. If there exists a cherry with a high weight, then it is not possible to find a cherry-picking sequence with a smaller weight. This is shown in the following lemma.

Algorithm 1 Branching Algorithm for Minimum Hybridization Number

```
procedure BRANCH($\mathcal{T}$, $\mathcal{X}$)
    $Checked \leftarrow \emptyset$
    $List \leftarrow [(\mathcal{T}, \mathcal{X}, (), 0)]$
    while $List \neq \emptyset$ do
        $(\mathcal{T}, \mathcal{X}, S, k) \leftarrow List.get\_item\_and\_remove\_it$(item with smallest $k$)
        if $\mathcal{X} = \emptyset$ then
            Return $S, k$
        end if
        if $\mathcal{X} \notin Checked$ then
            $Checked \leftarrow Checked \cup \mathcal{X}$
            for $x \in \mathcal{X}$ with $x$ pickable do                              ▷ Branching
                if $X \setminus \{x\}$ in $List$ then
                    $k' \leftarrow List[\mathcal{X} \setminus \{x\}]$
                    if $k' > k + w_{\mathcal{T}}(x)$ then
                        $List[\mathcal{X} \setminus \{x\}] \leftarrow [(\mathcal{T} \setminus \{x\}, \mathcal{X} \setminus \{x\}, S \cup \{x\}, k + w_{\mathcal{T}}(x)))]$
                    end if
                else
                    $List \leftarrow List \cup [(\mathcal{T} \setminus \{x\}, \mathcal{X} \setminus \{x\}, S \cup \{x\}, k + w_{\mathcal{T}}(x))]$
                end if
            end for
        end if
    end while
    Return $\emptyset$
end procedure
```

Lemma 3.2.1. Given a set of phylogenetic trees $\mathcal{T}$ and an arbitrary leaf x. Every cherry-picking sequence s must have a weight of at least $w_{\mathcal{T}}(s) \geq |N_{\mathcal{T}}(x)| - 1$.

Proof. Given a cherry-picking sequence $s = (s_1, s_2, ..., s_n)$. We define the set $\mathcal{X}' = N_{\mathcal{T}}(x) \cup \{x\}$ with $m = |\mathcal{X}'|$ then there exists a subsequence $s_{\mathcal{X}'} = (s_{I_1}, s_{I_2}, ..., s_{I_m})$ of s which displays $\mathcal{X}'$. Let i be the position of x in $s_{\mathcal{X}'}$, i.e., $s_{I_i} = x$, then we know that for s_{I_j} with $j < i$ we have $w_{\mathcal{T}[-s_1,s_2,...,s_{I_j-1}]}(s_{I_j}) \geq 1$. This holds for $j < m-1$ because if we pick x either as last or second last in $s_{\mathcal{X}'}$ then s_{m-1} could be picked as a trivial cherry. Furthermore, we know that x will be picked with $w_{\mathcal{T}[-s_1,s_2,...,s_{I_i-1}]}(x) \geq |N_{\mathcal{T}[-s_1,s_2,...,s_{I_i-1}]}(x)| - 1 \geq |N_{\mathcal{T}}(x)| - i$. Adding $i-1$ times a weight of 1 until we pick x gives us $w_{\mathcal{T}}(s) \geq |N_{\mathcal{T}}(x)| - 1$. □

This lemma shows that there is no solution with a lower weight than the highest-weighted leaf. The weight of a leaf is something the algorithm already calculates for the pickable leaves. Extending this to all leaves is not expected to be time consuming.

Another lower bound comes from the observation that not all cherries can become trivial cherries. Given such a cherry $\{x, y\}$, this means that x or y must be picked with a weight of at least one. If it holds for multiple cherries, then the lower bound becomes the number of such cherries that are leaf-wise independent. We start with some helpful definitions:

Definition 3.2.2 (anti-trivial cherries)**.** Given a set of phylogenetic trees $\mathcal{T}$ and a cherry $\{x, y\}$ in a tree. If there exists no temporal cherry-picking sequence $s = (s_1, ..., s_n)$ and $i \in [n]$ with $\mathcal{T} \setminus \{s_1, ...s_i\}$ having $\{x, y\}$ as a trivial cherry, then we call the cherry an *anti-trivial cherry.*

Definition 3.2.3 (Descendant)**.** Given a phylogenetic tree T, we say that a leaf x is a *descendant* of node v if there exists a directed path from v to x, or $v = x$, i.e., a path of length 0.

Definition 3.2.4. Given a phylogenetic tree $T = (V, E)$, taxa $\mathcal{X}$ and node $v \in V$, we define the set $D_T(v) := \{x \in \mathcal{X} \,|\, x \text{ is a descendant of } v\}$ as the *descendants* of v, we also call them the *leaf descendants*, with in particular $D_T(x) := \{x\}$ for $x \in \mathcal{X}$.

Algorithm 2 Branching Algorithm for The Temporal Decision Problem

```
procedure Pick($\mathcal{T}$, $\mathcal{X}$)
    $Checked \leftarrow \emptyset$
    $List \leftarrow [(\mathcal{T}, \mathcal{X}, ())]$
    while $List \neq \emptyset$ do
        $(\mathcal{T}, \mathcal{X}, S) \leftarrow List.get\_item\_and\_remove\_it$(last added item)    ▷
This is the item with smallest $|\mathcal{X}|$
        if $\mathcal{X} = \emptyset$ then
            Return $True, S$
        end if
        $Checked \leftarrow Checked \cup \mathcal{X}$
        for $x \in \mathcal{X}$ with $x$ pickable do    ▷ Branching
            if $\mathcal{X} \setminus \{x\} \notin Checked$ then
                $List \leftarrow List \cup [(\mathcal{T} \setminus \{x\}, \mathcal{X} \setminus \{x\}, S \cup \{x\})]$
            end if
        end for
    end while
    Return $False$
end procedure
```

Definition 3.2.5. Given a phylogenetic tree T. We say that two nodes p, q are *siblings* if they have the same parent node. Because we work with binary trees, all non-root nodes have one sibling in each tree, which we denote with $p = S_T(q)$.

The definition of anti-trivial cherries is directly linked to the existence of cherry-picking sequences. This implies that if there is no such sequence, then all cherries are anti-trivial cherries. From this follows that determining if a cherry is anti-trivial is NP-hard. However, some cherries are easily checked to be anti-trivial. We shall focus on such cherries. An example of anti-trivial cherries is cherries that are split by the root in another tree.

Lemma 3.2.6. Determining if a cherry is anti-trivial is NP-hard.

Proof. We shall reduce the temporal decision problem to this problem. Given a tree set $\mathcal{T}$. Choose an arbitrary leaf x. Replace leaf x with a cherry $\{x, y\}$ in each tree. This makes $\{x, y\}$ a trivial cherry. We thus have, $\{x, y\}$ is an anti-trivial cherry if and only if there exists no temporal network. □

Observation 3.2.7. Given a set of phylogenetic trees $\mathcal{T}$ with $|\mathcal{X}| > 2$ and a cherry $\{x, y\}$ in a tree. If there exists a tree T with the two children p, q of the root such that either $x \in D_T(p)$ and $y \notin D_T(p)$ or either $x \in D_T(q)$ and $y \notin D_T(q)$, then it is an anti-trivial cherry.

Proof. Without lose of generality, we assume $x \in D_T(p)$ and $y \notin D_T(p)$ and let T' be the tree with $\{x, y\} \in T'$. It should be clear that $\{x, y\}$ can only become a trivial cherry if it becomes a cherry in $T[-s_1, s_2, ..., s_{n-2}]$ for some cherry-picking sequence $s = (s_1, s_2, ..., s_{n-2}, x, y)$. Note that the position of x, y does not matter. And $\{x, y\}$ can only become a trivial cherry when all other leaves are picked. However, when there are only three leaves left, $T'[-s_1, s_2, ..., s_{n-3}]$ has s_{n-2} not in a cherry. We thus have a contradiction, i.e., there exists no such cherry-picking sequence. □

An example of such anti-trivial cherries is shown in Figure 3.2. This type of anti-trivial cherry $\{x, y\}$ can be found with a computational cost of $O(nt)$, with n the number of leaves and t the number of trees, by finding the first common ancestor of x, y and checking if this is the root. These root-split cherries are likely to exist if the root splits the tree into two equivalent-sized parts. However, this is not always the case. We shall therefore introduce the advanced version.

Lemma 3.2.8. Given a set of phylogenetic trees $\mathcal{T}$ and a cherry $\{x, y\}$ in a tree $T' \in \mathcal{T}$ and a different tree T with $\{x, y\} \notin T$ for which node p is the first common ancestor of x, y. If there exists a non-cherry node q in T' with $\{x, y\} \subset D_T(q) \subset D_{T'}(p)$, then $\{x, y\}$ is an anti-trivial cherry.

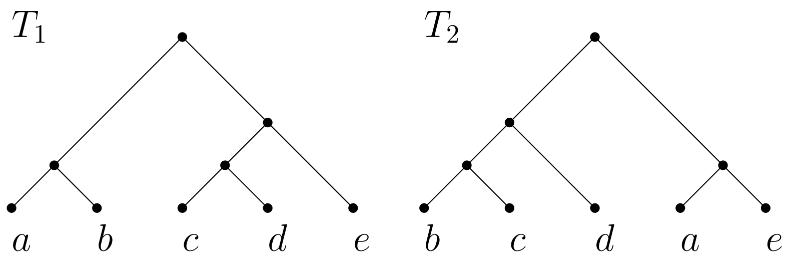


Figure 3.2: Two trees with anti-trivial cherries: $\{a, b\}$,$\{a, e\}$ and $\{b, c\}$. These are cherries in one tree but split by the root in the other tree.

Proof. Proof by contradiction. Without loss of generality, we assume that nodes $a, b \in T$ are the children of node p such that $x \in D_T(a)$ and $y \notin D_T(a)$. It should be clear that $\{x, y\}$ can only become a trivial cherry if it is a cherry in $T[-D_T(p)\backslash\{x, y\}]$, i.e. when all other leaves are picked first. This can only happen if there exists a cherry-picking sequence $s = (s_1, s_2, ..., s_n)$ which contains the subsequence $s_{D_T(p)} = (s_{I_1}, s_{I_2}, ..., s_{I_m}, x, y)$ which is an ordering of the elements of $D_T(p)$. Note that the positions of x and y can be switched, but this does not affect the proof. If we now examine tree T and node q, then we have that $D_T(q)\backslash\{x, y\}$ must contain at least one element. Furthermore, the elements of $D_T(q)\backslash\{x, y\}$ are all elements of $D_T(p)$. We define s_{I_i} as the last leaf picked of the elements $D_T(q)\backslash\{x, y\}$, then s_{I_i} is not in a cherry in $T[-s_1, s_{I_1}, s_{I_2}, ..., s_{I_{i-1}}]$ and can therefore not be picked. This is a contradiction, i.e., there exists no such cherry-picking sequence. □

An example is shown in Figure 3.3. Lemma 3.2.8 includes observation 3.2.7, when node p is the root, then node q can be the parent of the cherry. Note that we only need to check the grandparent of the two leaves in T' to satisfy the conditions of q.

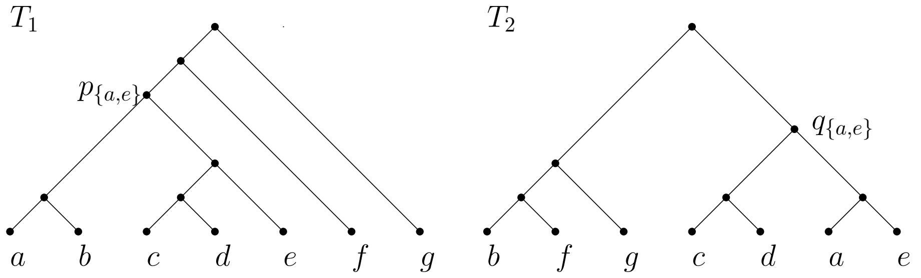


Figure 3.3: Two trees with $\{a, e\}$ as an anti-trivial cherry, with p the first common ancestor, and node q parent node of the cherry. For $\{a, e\}$ to become a trivial cherry in T_1, first the leaves b, c and d need to be picked. However, after picking either c or d, the other can not be picked.

With this, we have a method to find anti-trivial cherries. The number of such disjoint cherries is a lower bound for a tree set, as at least one leaf of an anti-trivial cherry must be picked with a weight of 1.

Theorem 3.2.9. Given a set of phylogenetic trees $\mathcal{T}$ and a disjoint set of anti-trivial cherries C. Then, $h_t(\mathcal{T}) \geq |C|$.

Proof. Given a cherry-picking sequence $s = (s_1, s_2, ..., s_n)$, then for each cherry $\{x_i, y_i\}$ in C with $0 < i \leq |C|$ we define $z_i \in \{x_i, y_i\}$ as the leaf first picked in sequence s. For each z_i we have that $w_s(z_i) \geq 1$, because it is an anti-trivial cherry and thus not a trivial cherry. Using the fact that the cherries in C are disjoint, means that z_i are disjoint. We thus get $h_t(\mathcal{T}) \geq |C|$. □

We have shown that finding some anti-trivial cherries can be done relatively fast. Given a set of anti-trivial cherries, finding the biggest subset of disjoint anti-trivial cherries is equivalent to the maximum cardinality matching problem, which can be solved in polynomial time. The lower bound from disjoint anti-trivial cherries is itself bounded by $\frac{1}{2}|\mathcal{X}| \geq |C|$. This makes it less useful for problems with many reticulations.

Another lower bound can be found by observing leaves which are bound to become a cherry. Before we state this bound, let us first define such leaves.

Definition 3.2.10. Given a set of phylogenetic trees $\mathcal{T}$ and two distinct leaves $x, y \in \mathcal{X}$. We call x, y cherry-bounded, if all possible cherry-picking sequences have $\{x, y\} \in \mathcal{T}[-s_1, s_2, ..., s_i]$ for some $i \in [n]$. Note that if $\{x, y\}$ is a cherry then it is also cherry-bounded.

Lemma 3.2.11. Given a set of phylogenetic trees $\mathcal{T}$ for which a temporal solution exists and two distinct leaves $x, y \in \mathcal{X}$. If there exists a tree $T_i \in \mathcal{T}$ with $y \in D_{T_i}(S_{T_i}(x))$ and there exists a tree $T_j \in \mathcal{T}$ with $x \in D_{T_j}(S_{T_j}(y))$, then x, y are cherry-bounded.

Proof. Given an arbitrary cherry-picking sequence $s = (s_1, s_2, ..., s_n)$ such that $s_m \in \{x, y\}$ is the first of x, y to be picked. If this is x then there exists a tree T_i with $y \in D_{T_i}(S_{T_i}(x))$ and follows that all but one leaf of $D_{T_i}(S_{T_i}(x))$ must be picked before x. This must be y by the assumption that we pick x first. We thus have $\{x, y\} \in T_i[-s_1, s_2, ..., s_{m-1}]$. By symmetry, a similar proof holds for y. □

Definition 3.2.12. Given a leaf y and its ancestor p. We define the depth $depth_p(y)$ as the distance from the ancestor p to y, i.e., the number of edges from p to y.

Lemma 3.2.13. Given a x, y cherry-bounded pair and common ancestors $p_{T_1}, p_{T_2}, ..., p_{T_t}$, one for each tree $T_i \in \mathcal{T}$. Then, $h_t(\mathcal{T})$ can't be lower then the minimum of the following values:

- $\min_{T \in \mathcal{T}}\{depth_{p_T}(y) + depth_{p_T}(x) - 1\}$
- $\max_{T \in \mathcal{T}}\{depth_{p_T}(y) + depth_{p_T}(x) - 2\}$

Proof. Given a cherry-bounded pair $\{x, y\}$ and arbitrary cherry-picking sequence $s = (s_1, s_2, ..., s_n)$ such that $s_m \in \{x, y\}$ is the first of x, y to be picked. Without loss of generality, we assume $s_m = x$. We assume that T is the tree for which x, y becomes the cherry, i.e., $\{x, y\} \in T[-s_1, s_2, ..., s_{m-1}]$. For this to happen, $depth_{p_T}(x) - 1$ leaves needed to be picked with at least one weight to get x to be a direct child of p_T. Note that, a weight of 0 implies that x was in a trivial cherry, and therefore could not be cherry-bounded with y. The same holds for y. This results in the combined weight of $depth_{p_T}(y) + depth_{p_T}(x) - 2$. If we now assume that x is not in a trivial cherry with y then $w_{\mathcal{T}[-s_1,s_2,...,s_{m-1}]}(x) \geq 1$, which results in $depth_{p_T}(y) + depth_{p_T}(x) - 2$. However, if the x is in a trivial cherry with y, then we have a weight of $depth_{p_{T_i}}(y) + depth_{p_{T_i}}(x) - 2$ for all $T_i \in \mathcal{T}$. This results in the two mentioned values. □

The term $\min_{T \in \mathcal{T}}\{depth_{p_T}(y) + depth_{p_T}(x) - 1\}$ is generally smaller for more trees. This means that it can be beneficial to observe only two trees.

Another bound can be derived from the cherry-bounded leaves. Similar to the size of the neighbourhood, the size of different leaves cherry-bounded to x results in a lower bound.

Definition 3.2.14. We call a subset B_x of leaves in $\mathcal{X}$, *x-bounded* if all $y \in B_x$ are cherry-bounded with x. Note that $N_{\mathcal{T}}(x) \subseteq B_x$.

Theorem 3.2.15. Given a leaf x and a x-bounded set B_x. Then, $h_t(\mathcal{T}) \geq |B_x| - 1$.

Proof. Given a cherry-picking sequence $s = (s_1, s_2, ..., s_n)$. We define the set $\mathcal{X}' = B_x \cup \{x\}$ with $m = |\mathcal{X}'|$ then there exists a subsequence $s_{\mathcal{X}'} = (s_{I_1}, s_{I_2}, ..., s_{I_m})$ of s which displays $\mathcal{X}'$. Let I_i be the position of x in $s_{\mathcal{X}'}$, i.e., $s_{I_i} = x$. Then we know that for s_{I_j} with $j < i$ we have $w_{\mathcal{T}[-s_1,s_2,...,s_{I_j-1}]}(s_{I_j}) \geq 1$. This holds for $j < m - 1$ because if we pick x either as last or second last in $s_{\mathcal{X}'}$ then s_{m-1} could be picked as a trivial cherry. Furthermore, we know that x will be picked with $w_{\mathcal{T}[-s_1,s_2,...,s_{I_i-1}]}(x) \geq |N_{\mathcal{T}[-s_1,s_2,...,s_{I_i-1}]}(x)| - 1 \geq |B_{\mathcal{T}}(x)| - i$. Adding $i - 1$ times a weight of 1 until we pick x gives us $w_{\mathcal{T}}(s) \geq |B_{\mathcal{T}}(x)| - 1$. □

This lower bound is a stronger version of the earlier mentioned lower bound $h_t(\mathcal{T}) \geq |N_{\mathcal{T}}(x)| - 1$, as $N_{\mathcal{T}}(x) \subseteq B_x$. This advanced version can use all leaves, not just the cherry. Furthermore, $|B_x|$ can be larger than the number of trees, which makes it even more useful for 2-tree problems.

We have given different lower bounds. Most use a subset $\mathcal{X}'$ of the leaves to show that the subsequence $s_{\mathcal{X}'}$ is picked with a lower bound. Multiple subsequences can be combined into a single lower bound.

Theorem 3.2.16. Given a set of phylogenetic trees $\mathcal{T}$ and two disjoint sets $\mathcal{X}_1, \mathcal{X}_2 \subset \mathcal{X}$ and two lower bounds l_1, l_2. Such that for all cherry-picking sequences s we have $\sum_{s_i \in \mathcal{X}_1} w_{\mathcal{T}[-s_1,s_2,...,s_{i-1}]}(s_i) \geq l_1$ and $\sum_{s_i \in \mathcal{X}_2} w_{\mathcal{T}[-s_1,s_2,...,s_{i-1}]}(s_i) \geq l_2$. Then, $h_t(\mathcal{T}) \geq l_1 + l_2$.

Proof. Given a cherry-picking sequence $s = (s_1, s_2, ..., s_n)$, by definition we have

$$w_{\mathcal{T}}(s) = \sum_{i=1}^{n} w_{\mathcal{T}}[-s_1, s_2, ..., s_{i-1}](s_i) \geq \sum_{s_i \in \mathcal{X}_1} w_{T[-s_1,s_2,...,s_{i-1}]}(s_i) + \sum_{s_i \in \mathcal{X}_2} w_{T[-s_1,s_2,...,s_{i-1}]}(s_i) \geq l_1 + l_2.$$

□

Theorem 3.2.16 can be used with induction to combine multiple lower bounds. The following lower bounds can be combined:

- The lower bound $l = 1$ for an anti-trivial cherry;
- The lower bound $l = |B_x| - 1$ for x-bounded set B_x.

This leads to the following problem. Given a set of lower bounds, each represented by $(l_i, \mathcal{X}_i)$, the bound and the leaf set used for the bound. What is the maximum sum of lower bounds with a disjoint leaf set? Note that a subset $B'_x \subset B_x$ of a x-bounded set is again an x-bounded set. This problem can be translated to a weighted maximum set packing problem, but this problem is NP-hard.

While this does not prove that our problem is NP, we generally expect it to not be easy to solve exact. To break this conundrum, we propose the heuristic approach of repeatedly choosing the highest lower bound.

We have examined different kinds of lower bounds. These bounds can be used in the following basic branch-and-bound Algorithm 3. Note that the algorithm requires an upper bound as input and it does not update this value. In practice, it is a good procedure to also try to find a close upper bound. This could be done by finding a random solution to a subproblem.

The branch-and-bound algorithm has the same computation cost $O\left(2^n \text{poly}(n, t)\right)$ as branching Algorithm 1, under the same condition that it depends on the data structure used. Note that finding a subproblem with the biggest $|\mathcal{X}|$ can be done cheaply by using n lists, each corresponding to the possible size of $|\mathcal{X}'|$.

Finding a lower bound can be relatively costly. These costs can creep up, especially if there is not a lot of pruning, resulting in a slower algorithm. This can be resolved by only calculating the lower bound under the condition that it has a probability of pruning. For example, the maximum lower bound for disjoint anti-trivial cherries is $\frac{1}{2}|\mathcal{X}'|$. Thus, it would be better to not try to find anti-trivial cherries if $\frac{1}{2}|\mathcal{X}| < U_b + k_{\mathcal{X}'}$, with $k_{\mathcal{X}'}$ the smallest weight to the subproblem.

In this section, we have introduced different lower bounds. These are not expected to all be useful. Although, the ones which can be combined have the potential to be beneficial in practice. However, such combined bounds must still be lower than the number of leaves. Thus making it less applicable when the number of reticulations is larger than the number of leaves.

Some definitions, like the cherry-bounded pairs, are poorly defined when there exists no cherry-picking sequence. This does not matter for non-temporal cases, as they do not have an upper bound for which we can compare our lower bound. The definitions could maybe be improved to include non-temporal cases. However, in this thesis, we assume the tree set to have a temporal solution when we talk about these definitions.

3.3. Good Leaves

Given a temporal hybridisation problem, some leaves can be picked with certainty to still give an optimum cherry-picking sequence. An example of these are leaves in trivial cherries [40] or problems with only one pickable leaf. Finding such leaves can help reduce the running time of exact algorithms.

In this section, we introduce some practical methods to determine such leaves. We conclude by discussing how this can be implemented.

Algorithm 3 Branch and Bound Algorithm for Minimum Hybridization Number

```
procedure BB($\mathcal{T}$, $\mathcal{X}$)
    $solution \leftarrow (nan, \emptyset)$
    $S \leftarrow ()$
    $k \leftarrow 0$
    $List \leftarrow [(\mathcal{T}, \mathcal{X}, S, k)]$
    while $List \neq \emptyset$ do
        $(\mathcal{T}, X, S, k) \leftarrow List.get\_item\_and\_remove\_it$(item with biggest $|\mathcal{X}|$)
        if $X = \emptyset$ then
            update $solution$ with $(k, S)$
        end if
        if $U_b \leq L_b(\mathcal{T}, S, k) + k$ then                                ▷ Prune
            continue
        else
            for $x \in \mathcal{X}$ with $x$ pickable do                            ▷ Branching
                if $X \setminus \{x\}$ in $List$ then
                    $k' \leftarrow List[\mathcal{X} \setminus \{x\}]$
                    if $k' > k + w_{\mathcal{T}}(x)$ then
                        $List[\mathcal{X} \setminus \{x\}] \leftarrow [(\mathcal{T} \setminus \{x\}, \mathcal{X} \setminus \{x\}, S \cup \{x\}, k + w_{\mathcal{T}}(x)))]$
                    end if
                else
                    $List \leftarrow List \cup [(\mathcal{T} \setminus \{x\}, \mathcal{X} \setminus \{x\}, S \cup \{x\}, k + w_{\mathcal{T}}(x))]$
                end if
            end for
        end if
    end while
    Return $solution$
end procedure
```

Some leaves can be determined to lead to an optimal solution when picked. This helps reduce the number of branches in the aforementioned algorithms in this chapter. To begin, it is essential to establish a precise definition of these leaves.

Definition 3.3.1. Given a solvable temporal hybridisation problem. We say that $x \in \mathcal{X}$ is a *good* leaf if there exists a cherry-picking sequence with $s' = (x, s_2, ..., s_n)$ which picks x first and is optimal, i.e., $h_t(\mathcal{T}) = w(s')$. We call a leaf *bad* if it is not a good leaf.

Good leaves are an important part of finding a fast solver. Observing such leaves early reduces the number of branches and, in turn, reduces the computational cost. Note that there are no good leaves if there is no solution. However, in practice, this means that we do not know if a leaf is a good leaf until a cherry-picking sequence is found. Therefore, we sometimes use the term 'good leaf' for leaves that can be picked and lead to an optimum cherry-picking sequence if one exists. In other words, we also use the term for problems which have no solution.

Some tree sets contain a subproblem within itself, which can be picked independently of any other leaves. These subproblems must then contain a good leaf.

Given a set of phylogenetic trees, we label two nodes in different trees the same if they have the same leaf descendants. Per this definition, the root and each leaf are the same nodes in different trees. If the same non-leaf node p exists in every tree of a tree set, then we call it a *cluster* and define $D(p)$ as the set of leaf descendants of the cluster.

Lemma 3.3.2. Given a set of phylogenetic trees $\mathcal{T}$ and cluster at node v. Let $s = (s_1, s_2, ..., s_n)$ be a cherry-picking sequence for $\mathcal{T}$ and let i be the smallest integer value for which $s_i \in D(v)$. Then, $s' = (s_i, s_1, s_2, ..., s_{i-1}, s_{i+1}, ..., s_n)$ is a cherry-picking sequence for $\mathcal{T}$ and $w(s) = w(s')$.

Proof. Take i such that it is the smallest integer value for which $s_i \in D(p)$. In other words, s_i is the first to be picked of the elements in $D(p)$. Because $|D(p)| > 1$ we have that $N_{\mathcal{T}}(s_i)$ consists of elements s_j with $j > i$. In other words $s_1, s_2, ..., s_{i-1} \notin N_{\mathcal{T}}(s_i)$. This means that s_i can indeed be picked first and it does not influence the weight of $s_1, s_2, ..., s_{i-1}$, i.e., $w_{\mathcal{T}}(s_i) = w_{\mathcal{T}[-s_1,s_2,...,s_{i-1}]}(s_i)$ and $w_{\mathcal{T}[-s_i,s_1,s_2,...,s_{m-1}]}(s_m) = w_{\mathcal{T}[-s_1,s_2,...,s_{m-1}]}(s_m)$ for $1 \leq m < i$. □

Observation 3.3.3. Given a set of phylogenetic trees $\mathcal{T}$ and two subsets of the taxa $\mathcal{X}_1, \mathcal{X}_2 \subset \mathcal{X}$ such that $\mathcal{T}|\mathcal{X}_1$ and $\mathcal{T}|\mathcal{X}_2$ contains the cluster p in all trees in both tree sets. Then, the two tree sets generated from p are the same.

Proof. The tree set generated form p is $\mathcal{T}|D(p)$. Note that we must have $p \subseteq \mathcal{X}_1$ and $p \subseteq \mathcal{X}_2$, as otherwise p would not be a cluster in $\mathcal{T}|\mathcal{X}_1$ or $\mathcal{T}|\mathcal{X}_2$. □

From the lemma, it follows that a cluster can be safely picked before any other leaf, and therefore contains at least one good leaf. Note that the cluster must be picked following an optimal solution of the cluster itself.

Observation 3.3.3 follows that for a given hybridisation problem we only need to calculate the best solution of each unique cluster once. It is thus practical to keep track of solved clusters.

Another type of good leaves are leaves whose neighbours are blocked by themselves. We shall call these *low-hanging* leaves.

Definition 3.3.4. Given two leaves x, y, and a tree T, y is called *blocked* by x if $\{x, y\} \notin T$ and $x \in D_T(S_T(y))$. For a tree set $\mathcal{T}$, we say that y is called *blocked* by x, if there exists a $T \in \mathcal{T}$ for which y is called blocked by x. From the definition follows, that being blocked means that y can not be picked until either x is picked or x, y has become a cherry in T.

Lemma 3.3.5. If x is a pickable leaf, and every neighbour is blocked by x, then x is a good leaf.

Proof. Given a cherry-picking sequence $s = (s_1, s_2, ..., s_n)$, we first show that x must be picked before its neighbours. Assume $y \in N_{\mathcal{T}}(x)$ is the first neighbour picked and this is done before x is picked, then we know that the subsequence $s_{D_T(S_T(y))}$ which displays $D_T(S_T(y))$ must have x at the last position. And y can only be picked after all of $D_T(S_T(y))\backslash\{x\}$ is picked. However, this means that the $N_T(x)$ is picked before y, which is a contradiction. We therefore know that x must be picked before its neighbours. Take $s_i = x$ we get that $s' = (x, s_1, s_2, ..., s_{i-1}, s_{i+1}, ..., s_n)$ is a valid cherry-picking sequence as s_j are not neighbours of x for $j < i$ and $w_{\mathcal{T}}(s) = w_{\mathcal{T}}(s')$ as again s_j are not neighbours of x for $j < i$. Meaning $w_s(s_i) = w_{s'}(s_i)$ for $i \in [n]$. □

In this section, we introduced good leaves. From the definition of good leaves, it follows that if a problem has a good leaf, then it can be picked without the loss of an optimal solution. For our branching algorithms, this means that if we find a good leaf, then it is sufficient to only branch into this one leaf. Furthermore, if we have a subset of the pickable leaves that we know must contain a good leaf, then we only need to branch into the leaves in this subset.

For leaves in trivial cherries and low-hanging leaves, this means we only need to branch into one of these. If there is a cluster, then we only need to branch for the leaves in the cluster. For multiple clusters, it is better to branch over the smallest cluster.

Good leaves can be implemented in Algorithms 1,2, and 3 in the following way. Replace the branching step 'for $x \in$ pickable' with '$x \in \mathcal{G}$', where $\mathcal{G}$ is a set of pickable leaves which contains a good leaf. For the branch and bound, it could also help to save computation time by taking the size of $\mathcal{G}$ into account for the decision to calculate the lower bound. For example, when $|\mathcal{G}| = 1$.

3.4. Non-Temporal Cases

To find if a set of trees can be displayed on a temporal network, we introduced Algorithm 2. While this algorithm is fast for most temporal cases, it still has a worst-case running time of $O(2^n \text{poly}(n, t))$. However, some cases can be shown to be non-temporal by their structure.

In this section, we introduce substructures for which the tree set is non-temporal. This could help reduce the running time for the aforementioned problems.

To look for substructures in the set of trees, we do not always need to take all trees into account. It is sometimes enough to look at a subset of the tree set.

Lemma 3.4.1. Given a set of phylogenetic trees $\mathcal{T}$ and subset $\mathcal{T}' \subset \mathcal{T}$. If there exists no temporal cherry-picking sequence for $\mathcal{T}'$, then there also exists no temporal cherry-picking sequence for $\mathcal{T}$.

Proof. Given a cherry-picking sequence $s = (s_1, s_2, ..., s_n)$, the sequence needs to be valid for each tree. Meaning, for each tree T we must have that s_i is in a cherry in the tree $T[-s_1, s_2, ..., s_{i-1}]$ for $i \in [n]$. This means that if there exists no cherry-picking sequence for a tree set, then adding trees to the tree set means that there still does not exist a cherry-picking sequence. □

Some tree sets do not have any pickable leaves, making them non-temporal. The smallest version of such a tree set consists of 3 leaves and 3 trees, with a different cherry in each tree. Therefore, each of these leaves is unpickable.

The smallest non-temporal problem containing two trees consists of 4 leaves, with only one cherry per tree, which are disjoint with each other.

Some tree sets will always be reduced to one of these problems. For both non-temporal cases discussed, we introduced a substructure for which a tree set always reduces to the respective non-temporal case.

Lemma 3.4.2. Given a set of phylogenetic trees $\mathcal{T}$ and three distinct leaves $x, y, z \in \mathcal{X}$. if all of the following are true:

- x is blocked by y and z;
- y is blocked by x and z;
- z is blocked by x and y,

then there exists no temporal cherry-picking sequence.

Proof. From x being blocked, we have that y or z needs to be picked before x. We also have that x or z needs to be picked before y. From these two statements, we can conclude that z must be picked before x and y. However, this contradicts the statement that z is blocked by x and y. Meaning there can not exist a cherry-picking sequence. □

Lemma 3.4.3. Given a set of phylogenetic trees $\mathcal{T}$ and four distinct leaves $a, b, x, y \in \mathcal{X}$. If the following is true:

- a is blocked by x and y;
- b is blocked by x and y;
- x is blocked by a and b;
- y is blocked by a and b,

then there exists no temporal cherry-picking sequence.

Proof. From a and b being blocked, we have that x or y needs to be picked before a and b. We also have that a or b needs to be picked before x and y. From these two statements, we can conclude that there can not exist a cherry-picking sequence. □

The two lemma show that some cases can be determined to be non-temporal without the need to pick a leaf.

We have shown different situations for which we can conclude that there is no cherry-picking sequence. These could be implemented into Algorithms 1 and 2 by checking if such a substructure exists and if it does, prune the subproblem. However, checking for such substructures for all leaves is $O(3^n)$ and $O(4^n)$ depending on the lemma. This makes it too costly to check for each subproblem.

3.5. Cherry-Growing Sequence

The cherry-picking sequence is generally seen as a sequence in which leaves should be picked. However, there is no reason why the sequence should be in the order of picking and not the other way around.

In this section, we introduce the reverse search for cherry-picking algorithms.

Cherry picking is an intuitive method as the trees become smaller for every leaf picked. At the start of this chapter, we translated the problem to a shortest path problem. This implies that it can also be solved in reverse. For this, we start with the following definition.

Definition 3.5.1. Let $(x_1, x_2, ..., x_n)$ be an ordering of the elements in $\mathcal{X}$. We say that this sequence is a *cherry-growing sequence* for a set of phylogenetic trees $\mathcal{T}$ precisely if each x_i with $i \in \{2, 3, ..., n\}$ labels a leaf of cherry in each tree that is contained in $\mathcal{T}[-x_n, x_{n-1}, ..., x_{i+1}]$. Furthermore, we define $\mathcal{T}[-\{x_1, x_2, ..., x_i\}] \cup x_j := \mathcal{T}[-\{x_1, x_2, ..., x_i\} \setminus \{x_j\}]$ as growing x_j.

The cherry-growing sequence is by definition the reverse order of a cherry-picking sequence. Proof for cherry-picking sequences thus also holds for cherry-growing sequences.

With this, we introduce a variant to the branching algorithm. The cherry-growing algorithms. These work by finding the path from $\mathcal{T}[-\mathcal{X}]$ to $\mathcal{T}$. This slightly changes the discussed branching algorithms, from checking pickable leaves to checking growable leaves, as shown in Algorithm 4.

Given a tree set $\mathcal{T}$ and a subproblem $\mathcal{T}[-\{x_1, x_2, ..., x_i\}]$. The leaves in $\{-x_1, x_2, ..., x_i\}$ could possibly be grown, wherefore we need to check $\mathcal{T}[-\{x_1, x_2, ..., x_i\} \setminus \{x_j\}]$ for each $j \in [i]$. However, there is a faster method.

The faster method follows from another method to construct $\mathcal{T}[-\{x_1, x_2, ..., x_i\}]$ from $\mathcal{T}$.

Observation 3.5.2. Given a tree set $\mathcal{T}$ and cherry-picking sequence s. For $i \in [n]$, $\mathcal{T}[-s_1, s_2, ..., s_i]$ can be constructed by starting from $\mathcal{T}$ and removing nodes and relabelling.

Proof. Generally picking a leaf is defined as removing the node and contracting its parent. However, when there exists a cherry-picking sequence and we pick s_i in sequence the following can also be done. First label each node in each tree by their leaf descendants. Then, remove both leaves of each cherry of s_i and adjust each label by removing s_i from it. □

Note that the only relabelling that is done is relabelling by taking a subset of the label.

Theorem 3.5.3. Given a leaf-descendant labelled tree set $\mathcal{T}$ and subset $\mathcal{X}' \subset \mathcal{X}$. If $\mathcal{T}[-\mathcal{X}']$ can not be constructed from $\mathcal{T}$ by removing nodes and relabelling by taking a subset of the previous label then there exists no cherry-picking sequence s with $\bigcup_{i=1}^{|\mathcal{X}'|} s_i$.

Proof. Proof by contra position of Observation 3.5.2. □

When growing leaves we can thus keep track of the relabelling from $\mathcal{T}$ to $\mathcal{T}[-\mathcal{X}']$. Note that a leaf $x \in \mathcal{X}'$ can only be grown from a leaf $y \notin \mathcal{X}'$. Growing is equivalent to reverse picking. It is thus done by adding the x and y as children of the old y and updating the descendant labelling.

3.6. Combined-Tree-Set Graph

Our problem starts with a set of trees for which we want to find the minimum reticulation network. This tree set could contain redundant information. For example, it could contain the same tree multiple times. In this section, we introduce a graph which can be constructed from a tree set, which retains all important information.

The set of trees can have redundant information. This holds especially for problems with many trees and a few reticulations. We solve this by combining all trees into a graph without the loss of crucial information.

Algorithm 4 Cherry Growing - Temporal Decision Problem

```
procedure GROW($\mathcal{T}$, $\mathcal{X}$)
    $Checked \leftarrow \emptyset$
    $List \leftarrow [(\emptyset, \mathcal{X}, ())]$
    while $List \neq \emptyset$ do
        $(\mathcal{T}, \mathcal{X}, S) \leftarrow List.get\_item\_and\_remove\_it$(last added)    ▷
This is the item with smallest $|\mathcal{X}|$
        if $\mathcal{X} = \emptyset$ then
            Return $True, S$
        end if
        $Checked \leftarrow Checked \cup \mathcal{X}$
        for $x \in \mathcal{X}$ with $x$ growable($\mathcal{T}_0, \mathcal{T}$) do    ▷ Branching
            if $\mathcal{X} \setminus \{x\} \notin Checked$ then
                $List \leftarrow List \cup [(\mathcal{T} \cup x, \mathcal{X} \setminus \{x\}, \{x\} \cup S)]$
            end if
        end for
    end while
    Return $False$
end procedure
```

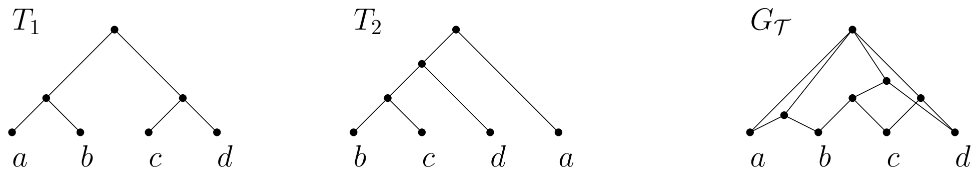


Figure 3.4: An example of two trees and its combined-tree-set graph.

Definition 3.6.1. Given a tree set $\mathcal{T}$ containing t trees with $T_i = (V_i, E_i) \in \mathcal{T}$ for $i \in [t]$. Label each node v of each tree by their leaf descendants $D(v)$ and we call two nodes in different trees the same node if they have the same label. Then, combine all trees into a graph $G_{\mathcal{T}} = (V_{\mathcal{T}}, E_{\mathcal{T}})$ with $V_{\mathcal{T}} = \bigcup_{i=1}^{t} V_i$ and $E_{\mathcal{T}} = \bigcup_{i=1}^{t} E_i$. We call this graph the *combined-tree-set graph.*

An example of the combined-tree-set graph is shown in Figure 3.4. For the combined-tree-set graph, we define $w_{G_{\mathcal{T}}}(x)$ as the number of different parents of x minus 1. Furthermore, we define $G_{\mathcal{T}}[-x]$ as removing x from $G_{\mathcal{T}}$ and all labels after which all nodes with the same label are contracted.

Theorem 3.6.2. Given a tree set $\mathcal{T}$ and its combined-tree-set graph $G_{\mathcal{T}}$. A sequence s is only a cherry-picking sequence for $\mathcal{T}$ if and only if it is a cherry-picking sequence for $G_{\mathcal{T}}$. Furthermore, a sequence has the same weight for both $\mathcal{T}$ and $G_{\mathcal{T}}$.

Proof. We shall show that $G_{\mathcal{T}}[-s_1, s_2, ...s_{i-1}] = G_{\mathcal{T}[-s_1,s_2,...s_{i-1}]}$ and that a pickable leaf has the same weight. The proof then follows from induction.
From the construction of $G_{\mathcal{T}}$ it follows that x is pickable for $G_{\mathcal{T}}$ if and only if x is pickable for $\mathcal{T}$. Let us now assume x is indeed pickable. We shall show that $G_{\mathcal{T}}[-x] = G_{\mathcal{T}[-x]}$. For leaf-descendants labelled $T \in \mathcal{T}$ we can construct $T[-x]$ by removing x from T and each label. After which all nodes with the same label are contracted. This process is the same for each node with the same label. This results in $G_{\mathcal{T}}[-x] = G_{\mathcal{T}[-x]}$.
Same weight: For x pickable, from the construction of $G_{\mathcal{T}}$ follows that $y \in N_{\mathcal{T}}(x)$ if and only if $\{x, y\} \in G_{\mathcal{T}}$. From this follow $w_{G_{\mathcal{T}}}(x) = w_{\mathcal{T}}(x)$. □

Reducing the tree set into its combined-tree-set graph and updating the leaf-descendant of all ancestors takes some computational time. However, it saves time when identifying clusters. Furthermore, it helps to reduce redundant information.

3.7. Experimental Results

In this chapter, we have introduced different possible methods and improvements to solve the minimum temporal hybridisation problem and the temporal network decision problem. In this section, we examine the difference in practice. The section is split into the respective problems.

3.7.1. Hybridisation Problem

For the hybridisation problem, we started by introducing the basic branching Algorithm 1. After that, we introduced a branch-and-bound algorithm. In this subsection, we examine the performance of these algorithms and the application of good leaves and different bounds.

To test the algorithms, we generated 1600 tree sets for each leaf size $n \in \{14, 17, 20, 21, 22, 23, 24, 25\}$. These are generated in two ways. Half of the instances are generated using a network-based approach and the other half are generated by generating random trees. These methods are discussed in Chapter 5. The data consists uniformly of 2 to 5 trees. Furthermore, each tree set starts with at least multiple leaves to pick, and they do not contain trivial cherries.

We have selected these types of test cases due, as these are the ones we solve to obtain the datasets used by the graph neural network.

We shall first examine the branching algorithm. This algorithm can potentially be improved by reducing the number of branches, which can be done by finding good leaves. A good leaf reduces the number of branches to one. In the previous chapter, we identified three types of good leaves. These are the trivial cherry, the cluster, and the low-hanging leaf.

We shall compare the following five branching algorithms:

- Grow, which uses cherry growing;
- Branch, which uses cherry picking;
- Branch$_{tc}$, which uses cherry picking and checks for trivial cherries;
- Branch$_{lh}$, which uses cherry picking and checks for trivial leaves and low-hanging leaves. ;
- Branch$_{cl}$, which uses cherry picking and checks for clusters.

The four cherry-picking algorithms use the combined-tree-set graph $G_{\mathcal{T}}$.

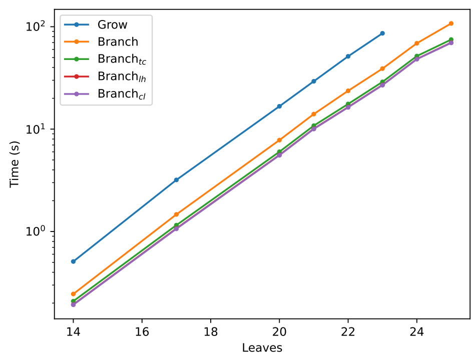
The time cost of the branching algorithms is shown in Figure 3.5a. Here we observe that the average time of each algorithm has a similar relationship to the number of leaves. The growing algorithms perform the worst. This is expected to be caused by two things. The first is an increased time cost of finding growable leaves. Furthermore, the implementation copies the initial tree set $\mathcal{T}$ into every subproblem, which could be improved.

For the cherry-picking algorithms, all three implementations of good leaves outperform the standard branching algorithm. Furthermore, we see that the search for clusters and low-hanging leaves results in a faster algorithm. However, not significant compared to the exponential time growth for larger problems. Observing the data of 23 to 25 leaves, the exponential growth seems to be around 1.6^n.

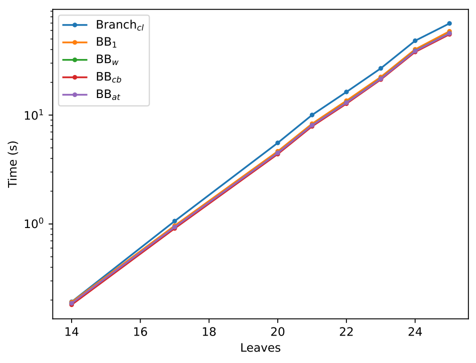
For the branch-and-bound algorithms, we compared 4 different lower bounds. The first BB$_1$ is the bound of 1 when there are no trivial leaves. The second BB$_w$ is the disjoint weights of pickable leaves. The third BB$_{cb}$ is the disjoint weights of cherry-bounded leaves. And the last BB$_{at}$ is the lower bound resulting from disjoint anti-trivial cherries. Finding the disjoint sets is done by repeatedly picking the largest, and at random when needed. Each algorithm is given an upper bound of $k + 2$ with k as the optimal reticulation. The algorithm also tries to find an upper bound, by picking leaves randomly. The algorithms do this when a better subproblem is found for each list of different subproblem sizes of $|\mathcal{X}|$. Furthermore, each branch and bound uses the combined-tree-set graph. This does give a computational advantage to finding clusters.

For the condition that looks for a lower bound, we use $3(U_b - k) < |\mathcal{X}| + 5$. This expression was chosen by observing that all three lower bounds generally are lower than $\frac{|\mathcal{X}|}{3}$. The '+5'-term is chosen to give a preference to smaller problems, which are computationally less costly.

The running times of these algorithms were found to be relatively the same. The fastest algorithm in order of speed is the BB$_{cb}$ with a shared second place between BB$_w$ and BB$_{at}$. The slowest was



(a) Branching algorithms.



(b) Branch-and-bound algorithms.

Figure 3.5: The average running time of different algorithms as a function of number of leaves. (a) Five branching algorithms, with the subscript denoting which good leaves are looked for. The BRANCH$_{lh}$ line hides behind BRANCH$_{cl}$. Both are slightly faster than BRANCH$_{tc}$. (b) Best performing branching algorithm compared to the branch-and-bound algorithm, with the subscript denoting the type of bound used. The results for the four branch-and-bound algorithms are relatively the same, making their lines indistinguishable. In order of fastest: BB$_{cb}$, BB$_w$, BB$_{at}$, BB$_1$.

found to be BB$_1$. However, the difference is minimal as shown in Figure 3.5b. Here we observe a similar exponential relationship between time and the number of leaves as for the branching algorithms. Observing the highest data points, we again found an exponential increase of around 1.6.

We have given the algorithms a relatively low upper bound to better observe the effect of the different lower bounds. The resulting difference is negligible. To check if this was because of the condition to check the lower bound. We also compared a few cases with the condition $2(U_b - k) < |\mathcal{X}|$, which gave a worse result, but again negligible.

The data used consists of two different types of trees. Separating these types did not result in any significant difference in relative performance. The same holds when the data is separated by the number of trees.

We can conclude that the lower bounds used are currently not effective for practical use in the branch and bound for our test cases. Finding good leaves resulted in a more noticeable but still small difference. This is expected to result from the fact that we branch into a network. When a node is pruned in a tree, all its subproblems are pruned with it. However, this does not hold for pruning a node in a network. The same holds for reducing the number of branches. That said, a far greater branch reduction could eventually lead to the lower bounds having a greater impact.

The tree sets used in the results had no trivial leaves, as they were prepicked. Giving a slightly skewed view that picking trivial leaves is not as important.

We also would have liked to compare the running time with the FPT algorithms but the algorithm was unable to solve all tree sets with 14 leaves in a feasible time. This is understandable as it contained problems with $k > 20$.

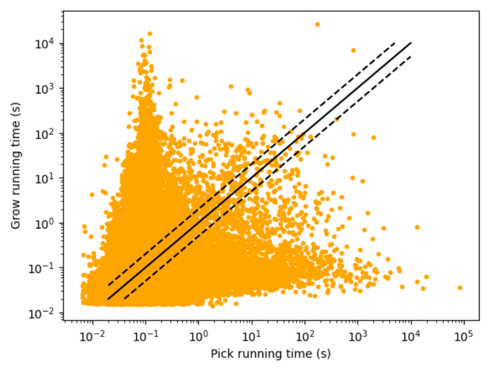
3.7.2. Temporal Decision Problem

The Temporal Decision Problem is easier than the hybridisation problem, as we only need to find a cherry-picking sequence. To do this, we introduced two depth-first search Algorithm 2 by picking leaves and Algorithm 4 by growing leaves. We shall call these PICK and GROW respectively.

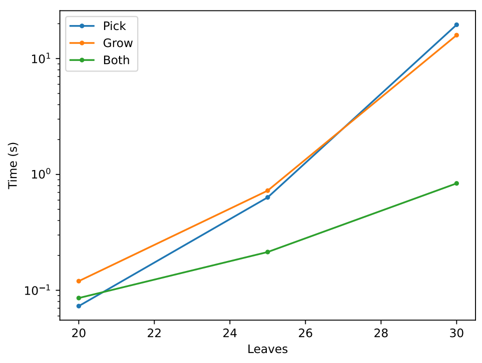
For each leaf size $n \in \{20, 25, 30\}$ we generate 16000 tree sets. These are generated by a network-based approach. The method for generating is discussed in Chapter 6. The data consists uniformly of 2 to 5 trees.

The running time for both algorithms is shown in Figure 3.6a with each dot representing a problem. Observe that the distribution is not positively correlated and some problems are substantially faster for one than the other. Furthermore, most points are outside the $\frac{1}{2}x \leq y \leq 2x$ area. This means that running both algorithms in parallel results in a faster algorithm. For problems with 30 leaves, this resulted in it being around 29 times faster. The average running time of these three algorithms is shown in Figure 3.6b.

The running time can differ drastically for two similar problems, as only a growing sequence or picking sequence needs to be found. The running time depends partly on the problem being temporal or not. The ratio between these two depends on the number of leaves. Therefore, we examine only the median of the 0.1 worst running times to calculate the exponential relationship between running time and leaves. From this, we find an exponential growth of around 1.5^n for Pick and Grow, and 1.2^n for their combination. Note that these are rough estimations.



(a)



(b)

Figure 3.6: (a) The running time distribution for both Grow and Pick. The three black lines are $2y = x$, $y = x$ and $y = 2x$. (b) The average running time for the Pick and Grow algorithms and their parallel combination Both, as a function of number of leaves.

Both Pick and Grow are depth-first search algorithms. It should be clear that these are faster than their breadth-first variant as the depth of a solution is $|\mathcal{X}|$ when there is a solution and equally fast if there is no solution.

In conclusion, running cherry-picking and cherry-growing algorithms in parallel greatly reduces the running times compared to running each algorithm individually.

The cause could be that they perform well for the other's worst-case problems. This is expected to be at least partly the cause, as non-temporal problems have disjoined search spaces for Pick and Grow. However, it could also be caused by temporal problems with being a depth-first search. Possibly picking a wrong leaf early in the algorithm.

4

Evaluating Graphs for GNN

In this thesis, we are interested in the performance of machine learning in the study of constructing phylogenetic networks out of a set of trees. The primary focus of our thesis revolves around the use of machine learning to predict if a leaf is a good leaf, which leads to the optimal solution.

There are different machine-learning models. In this thesis, we use the Graph Neural Network (GNN). This is an adaptation of neural networks, which utilises graphs within its input. The method was selected because the problem inherently involves graphs. Furthermore, the model allows for different-sized graphs. This means that a single model can be trained on a range of leaf set sizes. Additionally, the model is permutation invariance.

With GNN selected, the next step is to translate a set of trees into a single graph. In this chapter, we compare the performance of two different models, each with its own input graph. The first uses the combined-tree-set graph $G_{\mathcal{T}}$ and the second uses a complete graph of all leaves.

In this chapter, we first introduce the DAGNN. This adaptation partly solves a message-passing problem which occurs when we want to use the combined-tree-set graph $G_{\mathcal{T}}$ as an input graph for the GNN. We then compare the two models by examining their ability to find substructures like low-hanging leaves.

4.1. Directed Acyclic Graph Neural Network (DAGNN)

Before we can use a GNN for predictions, it needs an input graph. This graph should preferably be unique for different problems. One option is to use the combined-tree-set graph $G_{\mathcal{T}}$ introduced in Chapter 3. This graph contains all relevant information of the tree set and has less redundant information. However, this comes with a problem.

General graph neural networks are not good at communicating information between distant nodes [1], as message-passing layers generally only pass information over one edge. This means that for two nodes to communicate, there needs to be at least as many message-passing layers as their path length. Looking at phylogenetic trees, these path lengths vary for each tree. This means that we need a GNN which is able to adapt to different tree depths if we want communication between all nodes.

In this subsection, we reintroduce an adaptation to a general message-passing layer which solves this issue for directed acyclic graphs (DAGs). This is then compared with the recursive GNN and the general GNN.

Generally a GNN passes information over only one edge for each layer. This could be improved if there is a clear order in which the nodes of the GNN should be updated. Such an ordering exists for DAGs.

Observation 4.1.1. *For a DAG $G = (V, E)$ with $N = |V|$ there exists an ordering of all nodes $o = (v_1, v_2, ..., v_N)$ with each unique $v_i \in V$ for $i \in [N]$, such that for each arc $(v_j, v_k) \in E$ we have $j < k$.*

For a clear example, we look at a simple message-passing layer. This calculated the values $\mathbf{h}_v$ for node v using

$$\mathbf{h}_v^{(t+1)} = f(\mathbf{h}_v^{(t)}, \sum_{(u,v)\in E} \mathbf{h}_u^{(t)}),$$

with the layer number as a superscript. With the ordering of the observations, this can be adjusted to

$$\mathbf{h}_v^{(t+1)} = f(\mathbf{h}_v^{(t)}, \sum_{(u,v)\in E} \mathbf{h}_u^{(t+1)}).$$

We shall call these types of layers DAG-layers and GNN, which uses these, DAGNN. Note that, DAGNNs are computationally slower because not all nodes can be updated in parallel. In pursuit of solving our message-passing problem, we rediscovered the DAGNN [38, 4].

The above method seems easy to implement. However, in PyTorch Geometric [15], the layers expect both input $\mathbf{h}_v^{(t)}$ and $\mathbf{h}_u^{(t+1)}$ to be of the same size. This is resolved by concatenating each $\mathbf{h}_v^{(t)}$ with $\mathbf{h}_v^{(t+1)}$ with $\mathbf{h}_v^{(t+1)}$ being a zero vector if it has not yet been calculated.

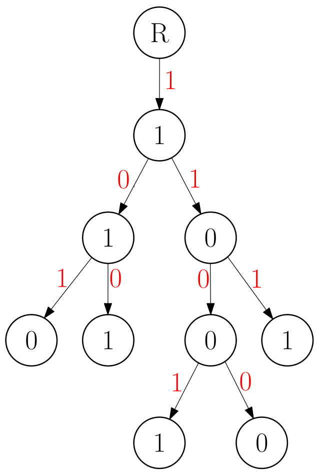


Figure 4.1: A rooted directed tree, with weights indicated by 0 or 1 beside the arcs. The parity of each path is shown inside the nodes.

Test Case To compare the DAGNN with the standard GNN, we introduce the following problem. Given a rooted directed tree, with weights $0, 1$ for the arcs, what is the parity of the path beginning from the root? In other words, given a node v and its path p_n represented by a sequence of arcs, what is the value of $\sum_{e\in p_v} w(e) \pmod 2$? An example is given in Figure 4.1.

To simplify the problem we integrate the weight of the arc into the head node of the arc. With this, the root can be removed and the weight can be implemented as node features. Furthermore, we shall only use binary trees.

Models We shall compare three different models all based on the GraphSAGE layer. The first is the Standard GNN.

$$\mathbf{h}_i^{(t+1)} = \text{ReLu}(\mathbf{W}_1^{(t)}\mathbf{h}_i^{(t)} + \mathbf{W}_2^{(t)}\mathbf{h}_j^{(t)} + \mathbf{W}^{(t)}\mathbf{x}_i), \quad \text{for } 0 \le t < 8$$

$$\mathbf{h}_i^{(out)} = \mathbf{W}_1^{(9)}\mathbf{h}_i^{(9)} + \mathbf{W}_2^{(9)}.$$

For the second model we use a Recursive variant. Here the same weights are the same for the first 9 layers.

$$\mathbf{h}_i^{(t+1)} = \text{ReLu}(\mathbf{W}_1\mathbf{h}_i^{(t)} + \mathbf{W}_2\mathbf{h}_j^{(t)} + \mathbf{W}_3\mathbf{x}_i), \quad \text{for } 0 \le t \le 8$$

$$\mathbf{h}_i^{(out)} = \mathbf{W}_1^{(9)}\mathbf{h}_i^{(9)} + \mathbf{W}_2^{(9)}.$$

The third method is the aforementioned DAGNN,

$$\mathbf{h}_i^{(1)} = \text{ReLu}(\mathbf{W}_1\mathbf{x}_i^{(0)} + \mathbf{W}_2\mathbf{h}_j^{(1)} + \mathbf{W}_3\mathbf{x}_j),$$

$$\mathbf{h}_i^{(out)} = \mathbf{W}_1^{(1)}\mathbf{x} + \mathbf{W}_2^{(1)}.$$

Each node v is given the weight of the incoming arc as feature $\mathbf{x_i}$. We start with layer $\mathbf{h}_i^{(0)} = \mathbf{x_i}$. Each hidden layer uses 10 channels.

Results To compare the performance of the models on our test cases, we generate training data and will compare their performance on the test set, which will consist of cases with larger depths. This is done so that we can assess which models can learn communication over undefined distances.

For the training data, we generated 400 trees consisting of 400 trees, each with a depth of 7, 19 nodes and random weights. The models were tested on 100 trees with a depth of 9 and 29 leaves.

To get a better understanding of their performance distribution, we ran each model 20 times with different initialisations. Each model is trained using 500 epochs using the standard Adam optimizer with a learning rate of 0.005.

The experimental results in this thesis were performed on the Delft Blue high-performance computer. The nodes have 48 cores consisting of two Intel XEON E5-6248R 24C 3.0GHz CPUs, with a total of 192 GB of memory. The code used in this thesis is found on GitHub [8]. The programming language is Python. The GNN is implemented using the PyTorch Geometric library.

For the result, we display the three quartiles of the test accuracy in Figure 4.2. The figure shows that the STANDARD and RECURSIVE models were not able to classify the test data with a higher depth correctly. While STANDARD did learn faster for the first 100 epochs.

Table 4.1 shows the average of the 10 best training runs for each model. The accuracy is given for training data and the test set. For the test set, we observed the accuracy for all nodes and the accuracy for nodes with at most a depth of 7, which the models are trained on.

From the table, we see that all three methods can classify the nodes correctly for both the training data and the testing data with the same depth. However, only the DAG model was able to extrapolate what it has learned to nodes with larger depths. This was expected from the STANDARD method, as it can not learn for higher depth than its training data. The RECURSIVE model does have the potential to solve problems with higher depth. However, this only resulted in a small increase in the test accuracy compared to the STANDARD model. The DAG model can solve problems with larger depth than it was trained on.

Model	Train Acc	Test Acc	Test Acc$_{\text{depth}\leq 7}$
GENERAL	1.000	0.912	1.000
RECURSIVE	1.000	0.928	1.000
DAG	1.000	1.000	1.000

Table 4.1: The test and train accuracy for the different models trained on problems with a maximum depth of 7. The second set of test accuracy is the accuracy for nodes with a depth of 7 or lower. All models can learn the problem for the depth they are trained on. However, only the DAG is able to solve higher depths correctly.

In this section, we examined the DAGNN and its performance in long-range communication. We have seen that is able to consistently send information down the graph even for untrained depths. Furthermore, the number of layers does not change depending on depth.

RECURSIVE model used in this section is similar to the known recurrent GNN. These generally keep recalculating $\mathbf{h}^{(t+1)}$ until it converges. These recurrent GNNs fall outside the scope of this study but could be interesting for future research.

4.2. Using GNN to Find Substructures

In Chapter 3, we showed that some sets of trees can be determined to be non-temporal and some leaves can be determined to be good leaves by looking for some substructure. In this section, we examine the performance of two types of GNN to find such substructures.

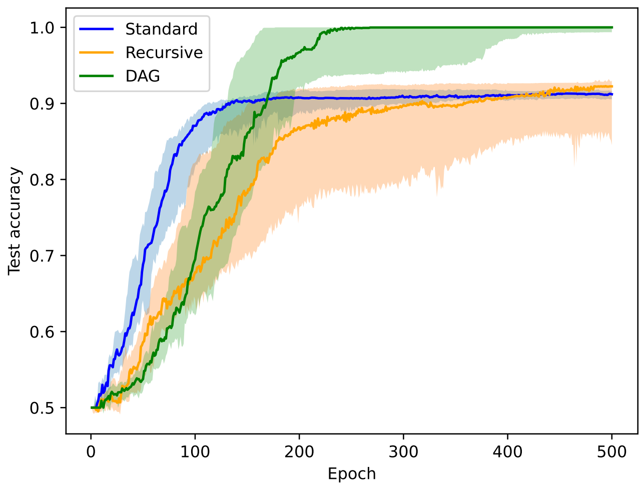


Figure 4.2: The test accuracy for the three different models. Each model was tested 20 times with different initialisations. The area shows the performance from the lower quartile to the upper quartile, and the median is represented by the solid lines. The DAG is the only model able to reach 100% accuracy but takes more epochs to learn than the outperforming Standard.

In this thesis, we aim to utilise GNN for predicting if leaves are good leaves and for predicting if a tree set is temporal. For this, we implement two types of input graphs. The first is the combined-tree-set graph and for the second we implement a Clique consisting of all leaves. To test the ability of these implementations, we shall examine their capability to find substructures relevant to the main goal of this thesis.

Test Cases To determine the ability to find substructures in tree sets, we introduce 3 practical problems. The first is to look for low-hanging leaves, which are good leaves. For the other two problems, we shall look at substructures which result in the tree set being non-temporal.

For the first problem, we want to determine if a leaf x is pickable and all its neighbours are blocked by x. These leaves are called low-hanging leaves. We shall denote the problem and dataset by LH. For this problem, we generate training data consisting of 1000 tree sets with 3 trees and 20 leaves. In this dataset, there are 508 low-hanging leaves out of the 20000 leaves in total. For the testing, we generated 200 tree sets with 3 trees and 30 leaves.

For the second problem, we are looking for the following substructure:

- x is blocked by y and z in some tree;
- y is blocked by x and z in some tree;
- z is blocked by x and y in some tree.

This is visually shown in Figure 4.3a. Such a substructure means that the tree set is non-temporal. This was proven in Chapter 3. We shall call this problem NT3. For this problem, we generate 6000 tree sets each with 3 trees and 10 leaves. Such that half of these contain the structure and half do not. Furthermore, we use 4000 tree sets for the training data and the remaining 2000 for the test data.

For the last problem, we shall look for the following:

- a is blocked by x and y in some tree;
- b is blocked by x and y in some tree;
- x is blocked by a and b in some tree;
- y is blocked by a and b in some tree.

This is visually shown in Figure 4.3b. This was also shown to be non-temporal. We shall call this problem NT4. For this problem, we generate 6000 tree sets each with 2 trees and 10 leaves. Such that half of these contain the structure and half do not. Furthermore, we use 4000 tree sets for the training data and the remaining 2000 for the test data.

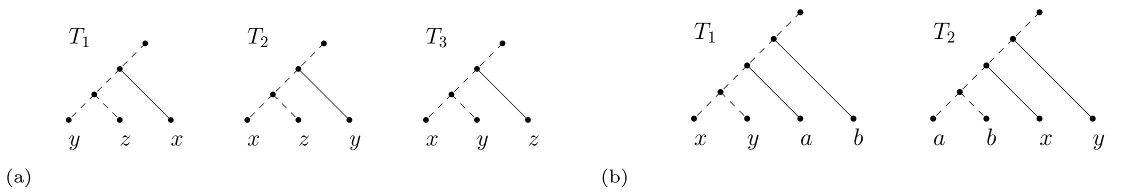


Figure 4.3: Two examples of non-temporal structures. Solid lines represent arcs and dashed lines represent paths.

Models For the aforementioned problems, we compare two different GNN models. The first uses the combined-tree-set graph as input and implements the DAGNN discussed in the previous section. The second model uses a complete graph of the leaves and implements edge features to quantify their relationship.

DAGNN The first model we call the DAG. This model uses the combined-tree-set graph as input. This is done by labelling each node v by its leaf-descendants $D(v)$ and then combining all trees such that all nodes with the same labelling are merged into one node. An example is shown in Figure 4.4. Note that there is no relevant data lost while transforming the tree set in this way.

For the model, we use the DAGNN discussed in the previous chapter. This adaptation to the GNN is able to transfer information over long distances using a single layer. In the model, we will use 6 hidden layers. The uneven layer will transfer information from the root to the leaves. While even layers, send information from the leaves to the root.

For the model, we use the following calculations:

$$\mathbf{h}_i^{(0)} = \mathbf{x}_i^{(0)}$$

$$\mathbf{h}_i^{(t)} = \text{ReLu}(\text{Layer}(\mathbf{h}_i^{(t-1)}, \bigcup_{(j,i)\in E} \mathbf{h}_j^{(t)}, \mathbf{h}_j^{(t-1)})), \qquad \text{for } t \in \{1,3,5\},$$

$$\mathbf{h}_i^{(t)} = \text{ReLu}(\text{Layer}(\mathbf{h}_i^{(t-1)}, \bigcup_{(i,j)\in E} \mathbf{h}_j^{(t)}, \mathbf{h}_j^{(t-1)})), \qquad \text{for } t \in \{2,4,6\}.$$

For the leaf classification in LH we only classify nodes i which are leaves, for which the output becomes:

$$\mathbf{h}_i^{(out)} = \mathbf{W}_1 \mathbf{h}_i^{(6)} + \mathbf{W}_2.$$

For the graph classification in NT3 and NT4 we first pool the data of all leaves into a single vector. This is done using max-pool, which takes the maximum value of all nodes for each index.

$$\mathbf{h}^{(out)} = \mathbf{W}_1 \,\text{max-pool}_{i\in\mathcal{V}}(\mathbf{h}_i^{(6)}) + \mathbf{W}_2.$$

For the layer, we use three different types: TransformerConv, GATv2Conv and SAGEConv.

Clique The second type of input graph we introduce is the complete graph of the leaves. This is a graph in which every node is connected to every other node. This means that this input graph does not have to send the information over multiple edges. The standard GNN can thus be applied.

We shall call this model Clique. For the model, we use the following calculations:

$$\mathbf{h}_i^{(0)} = \mathbf{x}_i^{(0)}$$

$$\mathbf{h}_i^{(t)} = \text{ReLu}(\text{Layer}(\mathbf{h}_i^{(t-1)}, \bigcup_{\{i,j\}\in E} \Big(\mathbf{h}_j^{(t-1)}, \mathbf{e}_{i,j}^{(t-1)}\Big))), \qquad \text{for } 1 \leq t \leq 6.$$

For the leaf classification in LH we only classify nodes i which are leaves, for which the output becomes:

$$\mathbf{h}_i^{(out)} = \mathbf{W}_1 \mathbf{h}_i^{(6)} + \mathbf{W}_2.$$

For the graph classification in NT3 andNT4 we first max-pool the data of all leaves.

$$\mathbf{h}^{(out)} = \mathbf{W}_1 \,\text{max-pool}_{i\in\mathcal{V}}(\mathbf{h}_i^{(6)}) + \mathbf{W}_2.$$

For the Layer we use two different types, TransformerConv and GATv2Conv. Note that SAGEConv does not support edge features.

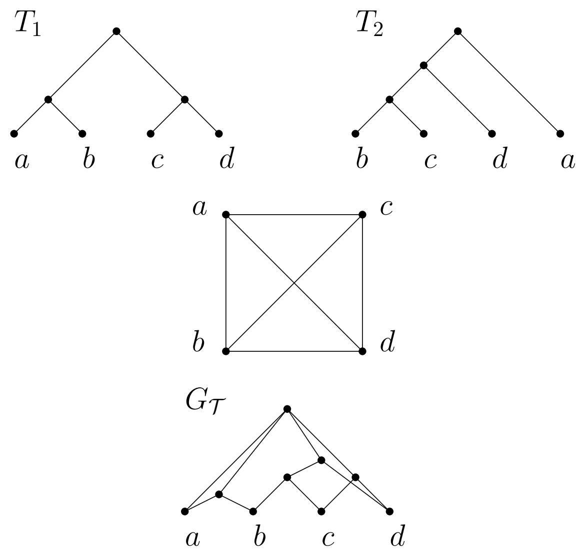


Figure 4.4: An example of a tree set and its two different input graphs used for the GNN. The top shows the two trees in the tree set. The middle graph shows the CLIQUE of the leaf set. The bottom shows the combined-tree-set graph.

Results We want to find the performance of the two models in finding substructures, which correlates to our main goal of using GNN in heuristics. This is done to firstly compare the models for practical use and, secondly, examine their capabilities of solving substructures. Furthermore, to understand if the models are feature-dependent, we examine their performance with minimal features compared to many features.

For each model, we ran the model with 300 epochs using a standard Adam optimizer with a learning rate of 0.005. Each model ran 24 times, with different initial conditions, to observe the distribution of the results.

We first examine the performance of the models with minimal features. The features are shown in Table 4.3 and 4.3. For the DAG, we use the first four features. The CLIQUE uses features 1, 7, 8 and 9.

The best 6 results are shown in Table 4.2. For the LH problem, we observe that the CLIQUE model outperforms the DAGNN and even has 100% accuracy with the TransformerConv layer. The DAGNN models also have a high accuracy of above 0.99%. For the NT3 problem. We observe that each DAG

		LH	NT3	NT4
DAG	Sage	0.9909 ± 0.0005	0.872 ± 0.004	0.927 ± 0.007
	Gat	0.9936 ± 0.0015	$\mathbf{0.877 \pm 0.002}$	0.934 ± 0.008
	Trans	0.9927 ± 0.0016	0.875 ± 0.004	0.928 ± 0.008
CLIQUE	GAT	0.99982 ± 0.00003	0.764 ± 0.004	0.915 ± 0.003
	Trans	$\mathbf{1.00000 \pm 0.00000}$	0.825 ± 0.001	$\mathbf{0.985 \pm 0.002}$

Table 4.2: The avarage test accuracy and standard deviation for the best 6 out of 24 results for each model and different layers for each problem. The best-performing model for each problem is highlighted in boldface.

outperforms the CLIQUE. Looking back at the data, we find that this is partly because the input features are not enough to deduce if a tree set is NT3 or not. In the NT4, we observe that again the CLIQUE Transformer layer results in the highest accuracy.

For previous results, we examined the models trained on a few features. To compare if more features increase the accuracy, we rerun the models with all features in Table 4.3 and 4.4.

We call the problems with all these features NT3$_{EF}$ and NT4$_{EF}$. The models for these problems are ran the same way as their minimalistic feature counterparts. The results are compared in Table 4.5.

From the table, we observe that adding extra features for NT3 generally results in higher accuracy. The CLIQUE model improves by around 4% and 8%. However, for the NT4 problem, we observe a

Num	Feature Name	Description
1	Indegree	The indegree of the node.
2	Outdegree	The outdegree of the node.
3	isLeaf	0 or 1 depending if the node is a leaf.
4	isPickable	0 or 1 depending if the node is a pickable leaf.
5	isRoot	0 or 1 depending if the node is the root.
6	NrTrees	The number of trees which contain the node.
7	NrDescendants	The number of leaf descendants of the node.
8 & 9	distanceToRoot$_{min,max}$	The minimum and maximum distance to the root.
10	NrTrivial	The number of trivial leaves after picking all descendants.

Table 4.3: The features used by the DAGNN. Only feature 1 to 4 are used for LH, NT3 and NT4.

	Node Features	
Num	Feature Name	Description
1	isPickable	0 or 1 depending if the node is a pickable leaf.
2	Weight	The number of different Neighboors.
3 & 4	disToRoot$_{min,max}$	The minimum and maximum distance to the root.
5 & 6	blockedBy$_{min,max}$	The min and max values of the number of leaves to be picked before this leaf can be picked, over all trees.
	Edge Features	
Num	Feature Name	Description
7	isCherr	0 or 1 depending if the two nodes are in a cherry.
8 & 9	Block	0 or 1 depending if the two nodes block one another.
10 & 11	NrPickCherry	The number of leaves to be picked before the two leaves are in a cherry
12 - 15	disToPar$_{min,max}$	The min and max distance to the first common ancestor, for both leaves
16	blocked	Number of trees in which either leaves block eachother
17 & 18	disPP$_{min,max}$	The minimum and maximum size of the symmetric differences between first common ancestors.

Table 4.4: The features used by the Clique. The results of LH, NT3 and NT4 only use feature 1 and 7 to 9.

decline in accuracy. There is no loss in input data. This means that the loss in accuracy comes from the statistical noise from uncorrelated features.

From this chapter, we conclude that both the DAGNN and Clique models can be used to find substructures that are relevant in finding good leaves and determining if a problem is temporal or not. However, it also shows that adding extra features can reduce accuracy.

		NT3	$NT3_{EF}$	NT4	$NT4_{EF}$
DAG	Sage	0.872 ± 0.004	0.875 ± 0.004	0.927 ± 0.007	0.925 ± 0.003
	Gat	0.877 ± 0.002	0.875 ± 0.006	0.934 ± 0.008	0.931 ± 0.003
	Trans	0.875 ± 0.004	$\mathbf{0.884 \pm 0.006}$	0.928 ± 0.008	0.933 ± 0.003
Clique	GAT	0.764 ± 0.004	0.847 ± 0.004	0.915 ± 0.003	0.903 ± 0.011
	Trans	0.825 ± 0.001	0.861 ± 0.006	$\mathbf{0.985 \pm 0.002}$	0.953 ± 0.006

Table 4.5: The average test accuracy and standard deviation for the best 6 out of 24 results for each model and different layers for each problem. The best-performing model for each problem is highlighted in boldface.

5
Predicting Good Leaves with GNN

The definition of good and bad leaves is given in Chapter 3. Picking a good leaf leads to an optimal solution while picking a bad leaf does not. In this chapter, we examine the use of graph neural networks in predicting whether a leaf is good or bad.

The practical use of these GNNs is to be used in heuristics. However, classifying leaves into either good or bad leaves ignores the magnitude of bad leaves. Picking a different good leaf does not matter. The worst-case scenario is that the problem becomes harder, but the optimal solution can still be obtained. This is not the case for bad leaves. The worst-case scenario for picking a bad leaf is that the problem becomes non-temporal. Even if the problems keep being temporal, the new optimal solution can differ depending on the leaf picked.

This could potentially be solved by quantifying these different kinds of bad leaves. However, such implementation must solve the following problems. How to quantify bad leaves which lead to non-temporal problems. The second problem is caused by the number of trees. Tree sets with more trees tend to have a larger range of weight gain. This can result in bias. The problem is also harder, making the models less accurate. We thus implement the simple classification of two classes.

This chapter starts by introducing two different methods for generating the tree sets. The first follows a phylogenetic model which makes it more practical, while the second method is a more random approach.

After explaining how the datasets are generated, the models for the GNNs are introduced. These are adaptations of the DAG and Clique used in the previous chapter.

After this, we examine the experimental results of the models. This chapter concludes by examining the cost and performance of the features used.

5.1. Generating Tree Sets

Generating a set of trees can be done in different ways. The method chosen will undoubtedly influence the performance of the GNN. Furthermore, the GNN is always biased to the training data, and thus similar generated tree sets. This could be used to our advantage, as the practical application is for evolutionary studies.

In this section, we introduce two methods to generate data. The first follows an evolutionary model. The second method generates harder cases.

In this thesis, we use two methods to generate data. The first method generates a binary phylogenetic network, which it then splits into a tree set. This method simulates evolutionary networks [33, 21]. Generating this network works as follows. First, create a cherry, then repeatedly add leaves using either one of the two following events. A speciation event, which replaces a leaf with a cherry. Or a temporal gene transfer event, which needs two leaves and adds a combined leaf of the two.

- A speciation event: pick a random leaf l and add two new leaves l_1 and l_2 with the arcs (l, l_1) and (l, l_2).

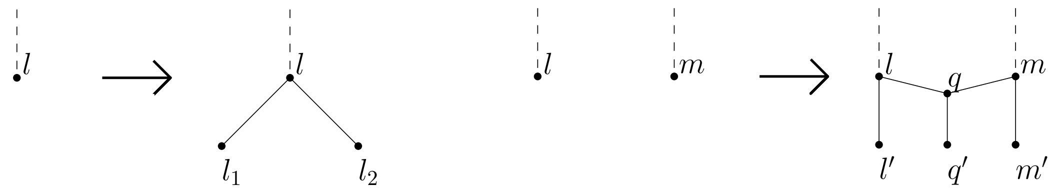
- The temporal gene transfer event: Pick two random leaves l, m and add three new leaves l', m' and q' and a new node q with the following arcs (l, l'), (m, m'), (q, q'), (l, q) and (m, q).

The leaves for these events are chosen by a uniform distribution. Furthermore, we generate networks with a predetermined number of leaves n and the reticulation number k. This means that we need $n-2$ events.

The distribution between the two events is chosen such that for each event, there is a $k/(n-2)$ probability that it is a temporal gene transfer event and such that there are precisely k temporal events. Such a network is then split into a set of trees.

Each tree is constructed by removing one of the edges of each reticulation, which is chosen with equal probability. Each reticulation node is contracted. The last tree of the tree set is constructed differently. This is constructed by the same method except the tree gets all reticulation edges which were not used in any of the other trees. Otherwise, the phylogenetic network was created with redundant reticulations.

This method generates a set of trees which has the upper bound k for its temporal reticulation number. Note that we start with a binary temporal phylogenetic network. This means that $k < n-2$ and there must be a temporal solution for the tree set.



(a) Speciation event.

(b) Temporal gene transfer event.

Figure 5.1: The two possible events in temporal simulation of phylogenetics.

The second method we use to generate a tree set is by generating each tree with only speciation events. This is done by picking a random leaf l and adding a new leaf such that the two leaves are in a cherry. If we name the leaves by the order they were added, starting with a cherry, then the tree has the cherry-picking sequence $(n, n-1, .., 2, 1)$. This holds for each tree generated and thus for the whole tree set.

This gives a set of trees which has a temporal solution. However, this does not give a bound on the number of reticulations.

5.2. Models

In this section we introduce the exact formulation of the models. Furthermore, we also compare the results between small adjustments.

The experimental results in this chapter were performed on the Delft Blue high-performance computer. The nodes have 48 cores consisting of two Intel XEON E5-6248R 24C 3.0GHz CPUs, with a total of 192 GB of memory. The code used in this thesis is found on GitHub [8]. The programming language is Python. The GNN is implemented using the PyTorch Geometric library.

For finding the weights for the model, we use the standard Adam optimisation step, starting with a learning rate of 0.01 which is reduced by half every 50 epochs and the process is repeated for 200 epochs. The models are run using 5-fold cross-validation.

To determine between good and bad leaves, we compare two types of models. The DAG and the CLIQUE. The DAG uses the combined-tree-set graph as input, while the CLIQUE uses a complete graph of all leaves. This is further discussed in the previous chapter.

Both models are adjusted compared to the previous chapter. For the DAG, we first normalised the features using InstanceNorm. For the DAG we use the following:

$$\mathbf{h}_i^{(0)} = \text{ReLU}(\mathbf{W}_1^{(0)} \,\text{InstanceNorm}(\mathbf{x}_i^{(0)}) + \mathbf{W}_2^{(0)}),$$

$$\mathbf{h}_i^{(t-0.5)} = \text{ReLU}(\text{TransformerConv}(\mathbf{h}_i^{(t-1)}, \bigcup_{(j,i)\in A} \left\{\mathbf{h}_j^{(t-0.5)}, \mathbf{h}_j^{(t-1)}\right\})), \qquad \text{for } t \in [3],$$

$$\mathbf{h}_i^{(t)} = \text{ReLU}(\text{TransformerConv}(\mathbf{h}_i^{(t-0.5)}, \bigcup_{(i,j)\in A} \left\{\mathbf{h}_j^{(t)}, \mathbf{h}_j^{(t-0.5)}\right\})), \quad \text{for } t \in [3].$$

For the leaf classification, we only classify nodes i which are pickable leaves. For which the output becomes:

$$\mathbf{h}_i^{(out)} = \mathbf{W}_1^{(3)}\mathbf{h}_i^{(3)} + \mathbf{W}_2^{(3)}.$$

We divide the model in three parts. The initializing layer consists of a normalization layer and a basic 1-layer NN, after which we call the 3 message passing layers the hidden layers, ending with the output layer. For each layer, excluding the output layer, we use 20 channels. In other words, the size of $\mathbf{h}_i$ is 20.

For the Clique model we use a similar structure:

$$\mathbf{h}_i^{(0)} = \text{ReLU}\left(\mathbf{W}_1^{(0)}\,\text{GraphNorm}(\mathbf{x}_i) + \mathbf{W}_2^{(0)}\right)$$
$$\mathbf{h}_{i,j}^{(0)} = \text{ReLU}\left(\mathbf{W}_3^{(0)} \cdot \text{GraphNorm}(\mathbf{e}_{i,j}^{(0)}) + \mathbf{W}_4^{(0)}\right)$$
$$\mathbf{h}_i^{(t)} = \text{ReLU}(\text{TransformerConv}(\mathbf{h}_i^{(t-1)}, \bigcup \mathbf{h}_j^{(t-1)}, \mathbf{h}_{i,j}^{(t-1)})), \quad \text{for } t \in [3]$$
$$\mathbf{h}_{i,j}^{(t)} = \text{ReLU}(\mathbf{W}_1^{(t)}\mathbf{h}_{i,j}^{(t-1)} + \mathbf{W}_2^{(t)}\mathbf{h}_i^{1} + \mathbf{W}_3^{(t)}\mathbf{h}_j^{1} + \mathbf{W}_4^{(t)}), \quad \text{for } t \in [2]$$
$$\mathbf{h}_i^{(out)} = \mathbf{W}_1^{(3)}\mathbf{h}_i^{(3)} + \mathbf{W}_2^{(3)}.$$

Normalisation In practice, normalisation helps the optimiser find a better local minimum. The best normalisation to use depends on both the model and the dataset. The normalisations chosen for both the DAG and Clique are the best types chosen from the different tested normalisations shown in Table 5.1. Here each model is the same as mentioned except for the normalisation used. The best is defined by having the highest test loss. Note that most of these losses range within 1%. This means that we can not conclude which is the best if we take statistical errors into account. Furthermore, the difference between train and test was not taken into account.

		train		test	
	type of normalisation	accuracy	loss	accuracy	loss
Clique	No normalization	0.773 ± 0.003	0.465 ± 0.004	0.770 ± 0.003	0.471 ± 0.003
	InstanceNorm	0.780 ± 0.001	0.448 ± 0.001	0.774 ± 0.002	0.462 ± 0.004
	GraphNorm	0.781 ± 0.003	0.448 ± 0.004	$\mathbf{0.776 \pm 0.003}$	$\mathbf{0.459 \pm 0.005}$
	LayerNorm	0.765 ± 0.002	0.474 ± 0.003	0.763 ± 0.002	0.480 ± 0.002
DAG	No normalization	0.733 ± 0.002	0.518 ± 0.003	0.729 ± 0.005	0.530 ± 0.007
	InstanceNorm	0.741 ± 0.004	0.508 ± 0.005	$\mathbf{0.731 \pm 0.004}$	$\mathbf{0.528 \pm 0.007}$
	GraphNorm	0.737 ± 0.004	0.513 ± 0.006	0.729 ± 0.004	0.531 ± 0.006

Table 5.1: The average accuracy and loss with standard deviation over the training and testing data of different normilisatons used in DAG and Clique. The reason of choice is highlighted in bold.

Layer type For both models, we also tested different types of layers. The results are shown in Table 5.2. For the Clique, we observed that the Trans layer is the best with a 3% increase in test accuracy compared to the rest. For the DAG, we observe similar ranges of accuracy.

Number of Hidden Layer The number of hidden layers was, for both models, chosen to be 3. Remember that increasing the number of layers can give better results. However, by increasing the search space, it could also be more likely to get stuck at a local minimum. The results for different numbers of layers are shown in Table 5.3

Number of Hidden Channels The size of the hidden channels also influences the performance of the model. Table 5.4 shows the results for using different numbers of hidden channels.

		train		test	
	Layer type	accuracy	loss	accuracy	loss
CLIQUE	GATCq	0.750 ± 0.007	0.501 ± 0.010	0.743 ± 0.007	0.512 ± 0.011
	ResGatedCq	0.741 ± 0.007	0.513 ± 0.010	0.740 ± 0.007	0.515 ± 0.011
	Trans	0.781 ± 0.002	0.448 ± 0.005	$\mathbf{0.776 \pm 0.003}$	$\mathbf{0.460 \pm 0.005}$
DAG	GatDAG	0.717 ± 0.004	0.539 ± 0.006	0.715 ± 0.004	0.546 ± 0.005
	SageDAG	0.732 ± 0.004	0.518 ± 0.005	0.728 ± 0.004	0.529 ± 0.005
	Trans	0.741 ± 0.004	0.508 ± 0.005	$\mathbf{0.731 \pm 0.004}$	$\mathbf{0.528 \pm 0.007}$

Table 5.2: The average accuracy and loss with standard deviation over the training and testing data of different layer types used in DAG and CLIQUE. The best test losses are highlighted in bold.

		train		test	
	nr of layer	accuracy	loss	accuracy	loss
Clique	1	0.729 ± 0.001	0.529 ± 0.001	0.727 ± 0.003	0.531 ± 0.003
	2	0.767 ± 0.002	0.473 ± 0.004	0.764 ± 0.002	0.479 ± 0.003
	3	0.781 ± 0.003	0.449 ± 0.005	0.777 ± 0.004	0.460 ± 0.006
	4	0.778 ± 0.004	0.453 ± 0.005	0.774 ± 0.003	0.464 ± 0.006
	5	0.786 ± 0.001	0.441 ± 0.001	0.780 ± 0.002	0.454 ± 0.005
	6	0.786 ± 0.001	0.440 ± 0.001	0.781 ± 0.002	0.451 ± 0.004
DAG	1	0.684 ± 0.001	0.572 ± 0.002	0.681 ± 0.004	0.579 ± 0.004
	2	0.727 ± 0.002	0.526 ± 0.004	0.724 ± 0.004	0.535 ± 0.003
	3	0.741 ± 0.005	0.509 ± 0.006	0.732 ± 0.005	0.528 ± 0.008
	4	0.741 ± 0.004	0.505 ± 0.007	0.731 ± 0.003	0.529 ± 0.002
	5	0.745 ± 0.013	0.500 ± 0.023	0.735 ± 0.010	0.520 ± 0.016
	6	0.728 ± 0.001	0.527 ± 0.001	0.725 ± 0.003	0.537 ± 0.005

Table 5.3: The average accuracy and loss with standard deviation over the training and testing data of different numbers of hidden layers used in DAG and CLIQUE.

Combination The models can also be combined. We shall introduce two types of combinations. The first, COMB, consists of both models in parallel. However, $\mathbf{h}^{(t)}_{DAG,i}$, and $\mathbf{h}^{(t)}_{Clique,i}$ are both updated by using a linear combination of the two for $0 \leq t \leq 3$. The following calculations are done after each layer:

$$\mathbf{h}^{(t)}_{DAG,i} = \mathbf{W}^{(t)}_a \mathbf{h}^{(t)}_{DAG,i} + \mathbf{W}^{(t)}_b \mathbf{h}^{(t)}_{Clique,i},$$

$$\mathbf{h}^{(t)}_{Clique,i} = \mathbf{W}^{(t)}_c \mathbf{h}^{(t)}_{DAG,i} + \mathbf{W}^{(t)}_d \mathbf{h}^{(t)}_{Clique,i},$$

for each node i, which is a leaf.

The second combination we introduce is called COMB2. This model is simpler as it runs the DAG and CLIQUE in parallel, but it only takes a learnable linear combination for the output layer.

$$\mathbf{h}^{(out)}_i = \mathbf{W}^{(3)}_a \mathbf{h}^{(3)}_{DAG,i} + \mathbf{W}^{(t)}_b \mathbf{h}^{(3)}_{Clique,i}.$$

The results are shown in Table 5.5. From this, we observe that COMB outperforms COMB2. We shall abandon COMB2 as it is the inferior version of COMB with a similar running time.

The COMB seems to perform similarly for 14 and 20 hidden channels when observing the test accuracy. However, the difference between train and test accuracy implies that the model with 20 hidden channels has started to overfit. Therefore, we shall use the COMB with 14 hidden channels in Chapter 7.

		train		test	
	hidden channels	accuracy	loss	accuracy	loss
Clique	10	0.756 ± 0.006	0.491 ± 0.005	0.753 ± 0.005	0.495 ± 0.006
	14	0.773 ± 0.002	0.464 ± 0.001	0.768 ± 0.002	0.472 ± 0.004
	20	0.781 ± 0.003	0.449 ± 0.005	0.777 ± 0.004	0.460 ± 0.006
	28	0.788 ± 0.001	0.438 ± 0.002	0.780 ± 0.001	0.454 ± 0.002
	40	0.783 ± 0.006	0.445 ± 0.010	0.775 ± 0.006	0.462 ± 0.011
DAG	10	0.718 ± 0.004	0.540 ± 0.007	0.719 ± 0.004	0.542 ± 0.005
	14	0.726 ± 0.012	0.524 ± 0.016	0.724 ± 0.008	0.532 ± 0.012
	20	0.741 ± 0.005	0.509 ± 0.006	0.732 ± 0.005	0.528 ± 0.008
	28	0.744 ± 0.003	0.501 ± 0.006	0.733 ± 0.004	0.525 ± 0.005
	40	0.743 ± 0.006	0.503 ± 0.009	0.730 ± 0.003	0.529 ± 0.005

Table 5.4: The average accuracy and loss with standard deviation over the training and testing data of different numbers of the hidden channels used in DAG and Clique.

		train		test	
	hidden channels	accuracy	loss	accuracy	loss
Comb	10	0.789 ± 0.002	0.438 ± 0.003	0.783 ± 0.002	0.451 ± 0.003
	14	0.803 ± 0.001	0.413 ± 0.002	0.791 ± 0.002	0.438 ± 0.004
	20	0.811 ± 0.010	0.397 ± 0.014	0.792 ± 0.006	0.440 ± 0.011
Comb2	10	0.780 ± 0.001	0.454 ± 0.001	0.776 ± 0.003	0.462 ± 0.005
	14	0.785 ± 0.003	0.446 ± 0.004	0.779 ± 0.001	0.458 ± 0.003
	20	0.793 ± 0.005	0.430 ± 0.009	0.779 ± 0.002	0.458 ± 0.005

Table 5.5: The average accuracy and loss with standard deviation over the training and testing data of different number of the hidden channels used in Comb and Comb2.

5.3. Experimental Results

In the previous section, we defined the three models: DAG, Clique, and Comb. These models were tested on two different datasets, which were generated differently. The first, RN, is generated with the use of a phylogenetic model. While the RTS uses a random approach.

The results are shown in Table 5.6. From the table, observe that the RTS data is harder to learn for

	RTS	RN
Model	accuracy	accuracy
DAG	0.732 ± 0.005	0.941 ± 0.005
Clique	0.777 ± 0.004	0.968 ± 0.001
Comb	0.791 ± 0.002	0.974 ± 0.001

Table 5.6: The average test accuracy with standard deviation over two different types of data for the models: DAG, Clique and Comb.

all models. Resulting in accuracy below 80%. However, the practical generated data RN, has a much higher accuracy above 94%.

The models seem to perform relatively consistently as the Comb outperforms the Clique, which in turn outperforms the DAG.

5.4. Feature Examination

The GNN uses the features to predict if a leaf is a good leaf or a bad leaf. However, it is generally unknown why the calculations lead to a prediction. This also means that it is hard to determine the usefulness of a feature. In this section, we examine the effect of removing a feature on the accuracy of the GNN and the computation cost for a prediction.

To compare the features, we first group them. These groupings are generally determined by the calculation needed to calculate the features. In Table 5.7 and 5.8 we average results with standard deviation and Figure 5.2 shows the box plot of the test loss. For the DAG, we find that removing one of the sets of features does not change the test accuracy by more than 1%.

For the Clique, we find that features 8 & 9 and 10 & 11 are both important, as they improve test accuracy by 1%.

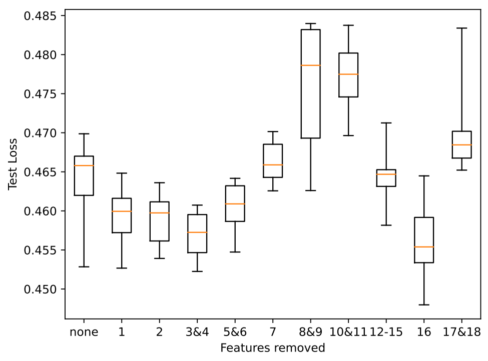
DAG	train		test	
Features removed	accuracy	loss	accuracy	loss
none	0.741 ± 0.005	0.509 ± 0.006	0.732 ± 0.005	0.528 ± 0.008
1&2	0.736 ± 0.005	0.515 ± 0.005	0.729 ± 0.005	0.530 ± 0.007
3-5	0.730 ± 0.009	0.521 ± 0.008	0.724 ± 0.009	0.535 ± 0.010
6	0.741 ± 0.003	0.506 ± 0.005	0.732 ± 0.003	0.525 ± 0.005
7	0.742 ± 0.008	0.506 ± 0.011	0.732 ± 0.007	0.526 ± 0.010
8&9	0.735 ± 0.007	0.518 ± 0.006	0.729 ± 0.005	0.530 ± 0.005
10	0.741 ± 0.007	0.509 ± 0.010	0.731 ± 0.007	0.528 ± 0.010

Table 5.7: The average accuracy and loss with standard deviation over the training and testing data for different features removed for the DAG. Removing features seem to not influence the accuracy in a significant way.

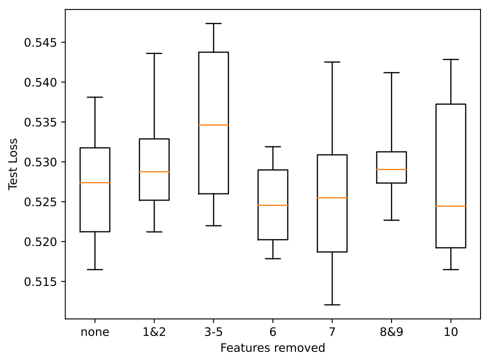
Clique	train		test	
Features removed	accuracy	loss	accuracy	loss
none	0.781 ± 0.003	0.449 ± 0.005	0.777 ± 0.004	0.460 ± 0.006
1	0.782 ± 0.002	0.448 ± 0.003	0.776 ± 0.002	0.460 ± 0.004
2	0.781 ± 0.002	0.449 ± 0.003	0.776 ± 0.002	0.459 ± 0.003
3&4	0.782 ± 0.002	0.447 ± 0.003	0.778 ± 0.002	0.457 ± 0.003
5&6	0.780 ± 0.002	0.451 ± 0.003	0.776 ± 0.002	0.461 ± 0.003
7	0.778 ± 0.002	0.456 ± 0.003	0.772 ± 0.001	0.466 ± 0.003
8&9	0.770 ± 0.005	0.468 ± 0.008	0.766 ± 0.005	0.476 ± 0.008
10&11	0.771 ± 0.002	0.466 ± 0.004	0.765 ± 0.004	0.477 ± 0.005
12-15	0.780 ± 0.003	0.452 ± 0.004	0.774 ± 0.002	0.464 ± 0.004
16	0.783 ± 0.003	0.446 ± 0.006	0.778 ± 0.003	0.456 ± 0.005
17&18	0.775 ± 0.003	0.460 ± 0.005	0.770 ± 0.004	0.470 ± 0.006

Table 5.8: The average accuracy and loss with standard deviation over the training and testing data for different features removed for the DAG. Removing features 8&9, 10&11 and 17&18 results in a decrease of accuracy.

The group of features comes with its own computational cost. This is the time it takes to generate the value for each node or edge of the model. These costs are shown in Figure 5.3a and 5.3b. They are compared to the computational cost of the model. For the Clique, we find that feature 6 could reduce the time by a factor of 2. For the DAG, the computation cost consists mostly of calculating feature 6.

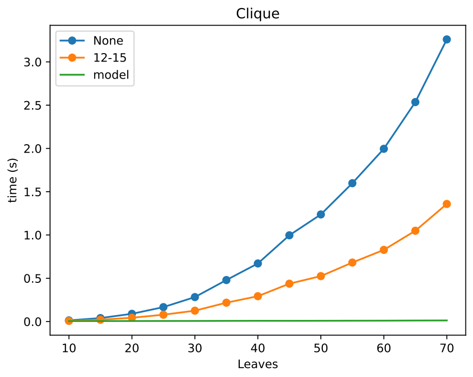


(a) Clique

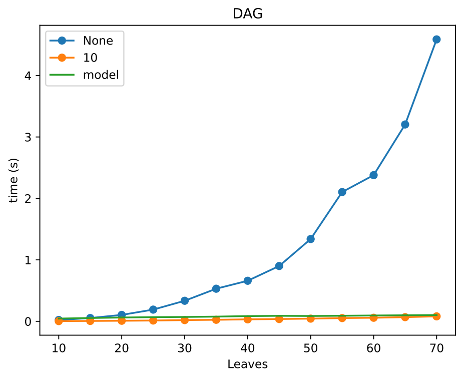


(b) DAG

Figure 5.2: The boxplot of the test loss. Comparing the effect of removing the denoted features. (a) For the Clique, we observe that removing 8&9, 10&11 and 17&18 results in a significant increase in loss. (b) For the DAG, we find no significant difference.



(a) Clique



(b) DAG

Figure 5.3: The running time of generating the features compared to the running time with the model for examples with 2 trees. The label denotes which features are left out. For the Clique, removing features 12 to 15 reduces the time by half. For the DAG, feature 10 takes substantially the longest time to generate.

6
Temporal Decision Problem with GNN

In the previous chapter, we examined the performance of GNN in predicting good leaves. This led to satisfactory results. For the second application of GNN, we shall examine their use in predicting if a tree set is temporal.

In this chapter, we first explain the method used for generating the dataset. Afterwards, we discuss the GNN used in this chapter. We conclude with experimental results.

6.1. Obtaining Data

We again generate a binary phylogenetic network, from which we generate a tree set. This is done similarly to generating the temporal network except by replacing the temporal gene transfer with lateral gene transfer. Thus generating such a network works as follows. First, create a cherry. Then add leaf by either one of the two following events. A speciation event, which replaces the leaf with a cherry. Or a lateral gene transfer event, which needs two leaves and adds a combined leaf of the two.

- A speciation event: Choose a random leaf l and add two new leaves l_1 and l_2 with the arcs (l, l_1) and (l, l_2).
- The lateral gene transfer event: Choose two random leaves l, m such that m is not the only tree child of its parent. Then add two new leaves l' and m' and the arcs (l, l'), (m, m') and (l, m).

The network and the resulting tree set are generated in the same way as in Chapter 5.

The dataset used consists of tree sets generated with 35 leaves and with $k = 25$. The data consists of 80000 tree sets. Each quarter of these sets contains several trees in the range of 2 to 5. Each of these quarters has a 4:1 ratio of temporal solution to non-temporal.

6.2. Models

We use the same models as in the previous section for the DAG, Clique, and Comb. However, the output is turned into graph classification. This is done by replacing the output vector with:

$$\mathbf{h}^{(out)} = \mathbf{W}_1 \,\text{max-pool}_i(\mathbf{h}_i^{(3)}) + \mathbf{W}_2.$$

For finding the weights for the model, we use the standard Adam optimisation step, starting with a learning rate of 0.01 which is reduced by half every 50 epochs and the process is repeated for 200 epochs.

6.3. Experimental Results

The experimental results in this chapter were performed on the Delft Blue high-performance computer. The nodes have 48 cores consisting of two Intel XEON E5-6248R 24C 3.0GHz CPUs, with a total of

192 GB of memory. The code used in this thesis is found on GitHub [8]. The programming language is Python. The GNN is implemented using the PyTorch Geometric library.

The three models are run using 5-fold cross-validation. Each model was tested with different values for the number of hidden channels ch and the number of hidden layers l. The results are shown in Figure 6.1. Here we see that the best COMB model performs similarly to the best CLIQUE model both outperforming the DAG.

			train		test	
Model	hc	l	accuracy	loss	accuracy	loss
CLIQUE	20	1	0.781 ± 0.002	0.448 ± 0.004	0.767 ± 0.006	0.468 ± 0.007
	20	2	0.809 ± 0.001	0.400 ± 0.002	0.787 ± 0.006	0.433 ± 0.008
	20	3	0.810 ± 0.011	0.398 ± 0.020	0.786 ± 0.006	0.433 ± 0.010
	20	4	0.810 ± 0.003	0.392 ± 0.005	0.796 ± 0.004	0.421 ± 0.005
	20	6	0.806 ± 0.007	0.395 ± 0.013	0.788 ± 0.003	0.434 ± 0.006
	10	3	0.786 ± 0.005	0.432 ± 0.004	0.779 ± 0.004	0.445 ± 0.007
	14	3	0.805 ± 0.004	0.402 ± 0.008	0.791 ± 0.005	0.426 ± 0.006
	20	3	0.810 ± 0.011	0.398 ± 0.020	0.786 ± 0.006	0.433 ± 0.010
	28	3	0.831 ± 0.005	0.357 ± 0.008	0.799 ± 0.006	0.417 ± 0.010
DAG	20	1	0.786 ± 0.002	0.441 ± 0.002	0.770 ± 0.004	0.469 ± 0.003
	20	2	0.814 ± 0.005	0.397 ± 0.010	0.783 ± 0.003	0.442 ± 0.006
	20	3	0.801 ± 0.009	0.420 ± 0.015	0.765 ± 0.009	0.474 ± 0.011
	20	4	0.802 ± 0.006	0.417 ± 0.014	0.769 ± 0.005	0.470 ± 0.006
	10	3	0.780 ± 0.006	0.456 ± 0.006	0.768 ± 0.007	0.473 ± 0.012
	14	3	0.800 ± 0.005	0.425 ± 0.009	0.775 ± 0.006	0.462 ± 0.008
	20	3	0.801 ± 0.009	0.420 ± 0.015	0.765 ± 0.009	0.474 ± 0.011
	28	3	0.835 ± 0.026	0.367 ± 0.042	0.781 ± 0.005	0.454 ± 0.009
COMB	10	3	0.828 ± 0.002	0.364 ± 0.004	0.798 ± 0.002	0.420 ± 0.004
	14	3	0.853 ± 0.005	0.322 ± 0.008	0.803 ± 0.012	0.415 ± 0.013

Table 6.1: The average accuracy and loss with standard deviation, with hc the number of hidden channels and l the number of layers.

From the DAG we observe that two layers seem to give the best results. We can also observe that increasing the number of hidden channels increases the difference between the train accuracy and test accuracy. This means that for these models a slight overfitting occurs.

The best-performing model has an accuracy of around 80% for the cases similar to the dataset which has 35 leaves. This is in practice unhelpful. Each of the 80000 tree sets was solved correctly using the algorithm in the previous chapter. This algorithm was able to solve most cases within minutes.

A far better heuristic is to use the exact algorithm to solve the problem correctly combined with a stopping time. Let the algorithm run till the designated time and then assume the problem is non-temporal.

Even if we do not consider the time it takes to solve 80000 cases of the dataset and the time it takes to optimize a model, the 80% accuracy is too low for practical use. Furthermore, the accuracy can not be improved unless the model is changed. The time-based heuristic seems therefore better.

7 GNN Heuristic

In a previous chapter, we examined GNNs trained to find good leaves, i.e., leaves that lead to an optimal solution. In this chapter, we examine heuristics which use such GNN and compare them with their random variant.

The chapter starts with the introduction of two types of algorithms. The first type follows the predictor and tries to return a cherry-picking sequence. The second type of heuristics we introduce is an adaptation to the Monte Carlo tree search.

After this, we examine the effect of different weighted loss functions. This is done by changing the weights in the loss function. In Chapter 5, these were chosen to make the results comparable, but this is not necessary in practice.

We conclude this chapter with experimental results. The results are divided into the two types of data used in this thesis.

7.1. Heuristic Algorithms

Chapter 5 resulted in different models for the graph neural network (GNN) to predict if a leaf is good to pick or not, i.e., could lead to the optimal solution when picked. In this section, we introduce two types of implementations of a predictor. Furthermore, we discuss the running time of the GNNs.

With a predictor, in our case the GNNs, repeatedly picking leaves can lead to a cherry-picking sequence. However, it can also get stuck in a non-temporal subproblem.

We shall introduce two different heuristics. The first repeatedly picks leaves using a predictor. This is a relatively fast method and returns either a cherry-picking sequence or returns that it did not find a temporal solution. We call this cherry-picking heuristic CPS with the subscript denoting the predictor used. Note that, we could also pick leaves a random. For this, we use the same algorithm, but call the predictor random.

This algorithm is shown in Algorithm 5. The algorithm repeatedly picks leaves. If the problem gets stuck on a non-temporal subproblem then it returns ∞ for the reticulation. The algorithm picks leaves by first checking if there exists a trivial cherry and picking these leaves. This is done because trivial cherries are good leaves, as discussed in Chapter 3. The other types of good leaves could also be implemented.

If no trivial cherries are found, the algorithm uses the predictor to find a set $\mathcal{P}$ of good leaves. Note that, if we find no predicted good leaves, then $\mathcal{P}$ is equal to the set of pickable leaves. Similarly, for the random predictor, the set $\mathcal{P}$ is equal to the set of pickable leaves. Note that the exact solver is also used as a predictor in small cases when the exact solver is faster than the GNN.

Using this setup, we know that the $\mathcal{P}$ is non-empty. From $\mathcal{P}$ we then pick a leaf at random. This is generally done by picking a leaf with a uniform distribution. Picking the leaves randomly makes the heuristic nondeterministic. The algorithms can thus be repeated to hopefully find a better lower-weighted cherry-picking sequence.

An adaptation is to incorporate the results of the predictor into the distribution. The GNN predicts for leaf v by calculating the predicted value p_v, which is between 0 and 1. With this, we calculate the relative probability $p_l + \alpha$, with α a constant. By adjusting α, we can change the relative probability. For $\alpha > 0$, this results in occasionally picking predicted bad leaves. While $1 < \alpha < 0$ tends to pick leaves with the highest p_l. Note that we work with relative probability. This means that the probability is calculated by $\frac{\max(0, p_l+\alpha)}{\sum_i \max(0, p_i+\alpha)}$. Furthermore, if we have $\sum_i \max(0, p_i + \alpha) = 0$ then we fall back to the uniform distribution of pickable leaves.

The third option studied in this thesis is picking the highest predicted good leaf. This is the leaf with the highest p_i. We call this method *greedy*. This makes the heuristic deterministic but could help in performance.

There are many different options for picking a good leaf. The options chosen in this thesis and the reasons behind them are discussed in the next section. They are also chosen to be used as a baseline and could be improved by analysing larger practical problems.

Algorithm 5 Cherry-Picking Sequence Heuristic (CPS)

```
procedure CPS($\mathcal{T}$, $\mathcal{X}$)
    $S \leftarrow \emptyset$
    $r \leftarrow 0$
    while $\mathcal{X} \neq \emptyset$ do
        if $\{x \mid x \in \mathcal{X}$ with $x$ pickable$\} = \emptyset$ then
            Return $\infty, S$
        end if
        if $\mathcal{X}$ contains trivial cherry $x$ then
            $\mathcal{X} \leftarrow \mathcal{X} \setminus \{x\}$
            $S \leftarrow S \cup \{x\}$
            $\mathcal{T} \leftarrow \mathcal{T} \setminus \{x\}$
        else
            $\mathcal{P} \leftarrow$ Predictor($\mathcal{T}$)                ▷ Predicted good leaves
            $x \leftarrow$ random_choice($\mathcal{P}$)
            $\mathcal{X} \leftarrow \mathcal{X} \setminus \{x\}$
            $S \leftarrow S \cup \{x\}$
            $r \leftarrow r + w_{\mathcal{T}}(x)$
            $\mathcal{T} \leftarrow \mathcal{T} \setminus \{x\}$
        end if
        Return $r, S$
    end while
end procedure
```

The second heuristic we shall introduce is the network search NS described in Algorithm 6. This improves the CPS heuristic by keeping track of previous data. The network search is an adaptation of the Monte Carlo tree search. However, the search space is not a tree but a network, as represented in Figure 3.1. We adapted the Monte Carlo tree search to search a network. This is done by keeping track of the best path for each subproblem. All subproblems visited in each loop are tracked using the list *update*, which is then updated with all relevant subproblems.

The network search is an improvement in a number of ways. The first improvement comes from calculating the $\mathcal{P}$ only once for each subproblem $\mathcal{X}'$. For each cherry-picking sequence, we do not need to recalculate $\mathcal{P}$ if we visited the subproblem before. This is especially useful for the larger subproblems, as these are generally visited the most.

Another improvement is the knowledge of a subproblem being fully searched. It is possible to keep track of all checked subproblems and remove them from future searches if they are fully searched. This means each new search is unique. This partly helps the heuristic to not get stuck in a non-temporal region.

The third improvement is the application of a greedy picker. In the next section, we shall observe that picking the highest predicted good leaf gives better results than randomly choosing a leaf. This comes at the cost of being unrepeatable. However, the NS can keep track when it is the first visit of a subproblem. This means it can pick greedy the first time it visits a new subproblem and choose a random one after subsequent visit.

Using NS also comes with a cost. This is the extra memory needed to keep track of the network search. And the extra time it takes to check if a subproblem has already been calculated or not. Furthermore, updating the best-known weights for each subproblem can be quite costly.

Algorithm 6 Network Search Heuristic (NS)

```
procedure NS($\mathcal{T}$, $\mathcal{X}$)
    $Checked \leftarrow G(\emptyset, \emptyset)$
    Data[$\mathcal{X}$] = ($\mathcal{P}_{\mathcal{X}}$, $Pickable_{\mathcal{X}}, \infty, True$)
    $Checked.add\_node(\mathcal{X})$
    $Update \leftarrow \emptyset$
    while $StopTime > RunTime$ do
        $\mathcal{T}' \leftarrow \mathcal{T}$
        $\mathcal{X}' \leftarrow \mathcal{X}$
        ($\mathcal{P}_{\mathcal{X}'}$, $Pickable_{\mathcal{X}'}, r_{\mathcal{X}'}, FirstTime$) = Data[$\mathcal{X}'$]
        while $Pickable_{\mathcal{X}'} \neq \emptyset$ AND $|\mathcal{X}'| > 0$ do
            if $FirstTime$ then
                $x \leftarrow greedy(\mathcal{P}_{\mathcal{X}'})$
                Data[$\mathcal{X}'$] $\leftarrow (-, -, -, False)$
            else
                $x \leftarrow$ random_choice($\mathcal{P}_{\mathcal{X}'}$)
            end if
            $Checked.add\_arc(\mathcal{X}, \mathcal{X}')$ with $weight = w_{\mathcal{T}'}(x)$
            $\mathcal{T}' \leftarrow \mathcal{T}' \setminus \{x\}$
            $\mathcal{X}' \leftarrow \mathcal{X}' \setminus \{x\}$
            $Update \leftarrow Update \cup \mathcal{X}'$
            if $\mathcal{X}' \notin$ Data then
                Data[$\mathcal{X}'$] = ($\mathcal{P}_{\mathcal{X}'}$, $Pickable_{\mathcal{X}'}, \infty, True$)
            end if
        end while
        while $Update \neq \emptyset$ do
            $child \leftarrow Update.get\_item\_and\_remove\_it$(last added item)
            for $parent\ of\ child \in Checked$ do
                if $r_{parent} > r_{child} + w(parent, child)$ then
                    $r_{parent} = r_{child} + w(parent, child)$
                    $Update \leftarrow Update \cup parent$
                end if
                if $Pickable_{child} = \emptyset$ then
                    $Pickable_{parent} \leftarrow Pickable_{parent} \setminus (parent \setminus child)$
                    $\mathcal{P}_{parent} \leftarrow \mathcal{P}_{parent} \setminus (parent \setminus child)$
                    $Update \leftarrow Update \cup parent$
                end if
            end for
        end while
    end while
    Return $r_{\mathcal{X}}$
end procedure
```

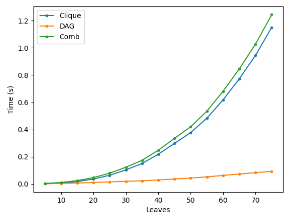
The running time of the GNN consists of two parts. The first part is to translate the tree set to the GNN and calculate the features. This time depends primarily on the features themselves, as shown in a previous chapter, not all features are worth their time. In this section, we examine the running time of the GNNs. We thus use the reduced features, discussed in Chapter 5.

The time it takes to generate the features and the graph differs for each model. For the Clique, the worst time complexity is $O(n^2)$, which is caused by the edge features. For the DAG, however, this is only $O(n)$. The second part consists of calculating the prediction $\mathbf{h}^{(out)}$ from the features. This is more costly for the DAG than for the Clique.

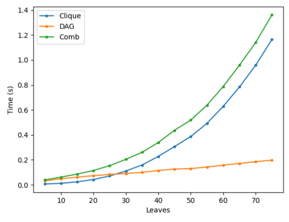
The time to generate the features is shown in Figure 7.1a. These running times are for tree sets with two trees. The figure shows the non-linear increase for the Clique. The figure also shows the time for Comb, which is the sum of the other two.

The total time to get a prediction for two trees is shown in Figure 7.1b. Here we observe that the DAG takes more time to predict for cases with less than 27 leaves. And that the Clique takes more time when there are more than 27 leaves.

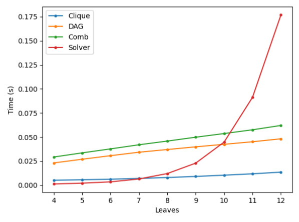
Using a predictor should not take more time than an exact algorithm. The running time for a prediction of a model and the running time of an exact solver is shown in Figure 7.1c for problems with two trees. The figure shows that the exact solver is faster for 7 leaves or fewer. The DAG and Comb take a lot more time, resulting in the exact solver being practically as fast for 10 leaves.



(a)



(b)



(c)

Figure 7.1: Three figures showing the time it takes depending on the number of leaves for the models to get a prediction for tree sets consisting of two trees. (a) The runtime of translating the set of trees into the graph used by the model. This includes calculating the features. (b) The total time cost to get a prediction for the three different models. The Clique becomes slower than the DAG around 27 leaves. (c) The total time cost compared with the running time of the exact solver.

7.2. Adjusting the GNN for the Heuristic

For the experiments in previous chapters, we used the weighted loss function, as discussed in the preliminaries. These weights were chosen to cancel the preference of the model caused by an unequal ratio of good and bad leaves. These weights can be adjusted to obtain a preference for either good or bad leaves.

Changing the weights of the loss function could lead to better results of the heuristic. In this section, we examine the effect changing these weights has. We also examine the performance of using a greedy picker, which picks the highest predicted good leaf.

The weights of the loss function determine if the GNN tends to have false positives or false negatives. Increasing the weight of good leaves means the GNN is punished harder for wrongly labelling good leaves. The reverse also holds. We can thus influence the type of error of the GNN. These types have a different effect on the heuristics.

For example, given a tree set $\mathcal{T}$ and its output vector $\mathbf{y} = [1, 0, 0, 1, 0]$ which labels the pickable leaves if they are good or bad with a 1 or 0 respectively, then a higher good-weighted GNN_g will give $\text{GNN}_g(\mathcal{T}) = [1, 0, 1, 1, 0]$, while a higher bad-weighted GNN_b will give $\text{GNN}_b(\mathcal{T}) = [0, 0, 0, 1, 0]$. In this example, the GNN_g has a 2 in 3 chance to pick a good leaf, while GNN_b has a 1 in 1 chance. Picking randomly has a 2 in 5 chance. In this example, we see that increasing the bad-leaf weight looks more promising if there are more than two good leaves. However, this could also lead to all 0 results, which could fall back to a 2 in 5 correct result.

If we pick a leaf randomly out of the good-labeled leaves, then using a GNN to determine which leaf to pick can result in different types of errors. The first is the all-0 result. This issue could be resolved through three potential methods: randomly selecting a leaf, employing an alternative weighted GNN, or opting for the leaf v with the highest value p_v.

The second type of error is having bad leaves labelled good and at least one good leaf labelled good. The error could again be reduced by picking the leaf v with the highest value p_v. However, this could also lead to worse results if a bad leaf is predicted to be the best leaf to pick. Another solution could be by repeating the heuristic.

		RTS			RN		
		accuracy			accuracy		
	weight ratio	bad	good	mean	bad	good	mean
Clique	2^{-2}:1	0.429	0.956	0.692	0.930	0.985	0.957
	2^{-1} : 1	0.604	0.891	0.748	0.957	0.977	0.967
	1 : 1	0.775	0.777	0.776	0.973	0.965	0.969
	$2^{0.5}$: 1	0.839	0.776	0.772	0.980	0.958	0.969
	2^{1} : 1	0.889	0.629	0.760	0.982	0.947	0.964
	$2^{1.5}$: 1	0.930	0.536	0.733	0.987	0.938	0.962
	2^{2} : 1	0.956	0.443	0.700	0.989	0.928	0.959
DAG	2^{-1} : 1	0.482	0.899	0.691	0.925	0.968	0.947
	1 : 1	0.726	0.737	0.731	0.951	0.947	0.949
	2 : 1	0.881	0.512	0.697	0.966	0.912	0.939

Table 7.1: The test accuracy for both good and bad leaves and their mean for different weighted loss functions. By multiplying the weight of bad leaves with the given weight factor. By adjusting this factor, one can increase the accuracy of one type by sacrificing the other.

The third type of incorrectness is having bad leaves labelled good and no good leaf labelled good. This type is the most troublesome as not following the GNN is the only good option. This means that even repeating the heuristic would not give the correct result.

In this thesis, we generalise the leaves into two classes. They are either bad or good leaves. While this is not a problem for good leaves, there is a practical difference between bad leaves. In the worst case, picking a bad leaf could make the tree set non-temporal. Even if the tree set retains being temporal, the new optimal solution can differ. GNNs trained using this concept have the potential to be better predictors in the use of heuristics.

Different Accuracies for GNN Before we examine the heuristics, we will first examine the effect of changing the weights in the optimisation process of the GNN. The weights are first chosen such that they cancel out the bias caused by the distribution of good and bad leaves. After that, we multiply one of these weights with a given factor.

We shall use the term ratio for this factor, as it is equivalent to having an unweighted loss function and a class distribution in the dataset of the given ratio.

The same method as in Chapter 4 is chosen to train the models. The GNNs are trained on the same data as in this chapter. Furthermore, we have not yet removed features for this data. This does not change the results in any significant way.

Table 7.1 shows the accuracy for both the bad and good leaves and their average. The average accuracy is highest for the 1:1 ratio, as expected. For both RTS and RN datasets, changing the ratio can greatly increase the accuracy of either bad or good leaves at the cost of the other. This cost is generally higher for lower 1:1-ratio accuracy.

Single Algorithm Runs To test which weight ratio is the best, we examined 2000 tree sets ranging from 2 to 5 trees. These are generated similarly to the datasets in Chapter 4. We again call these RTS and RN. The cherry-picking heuristics are tested 24 times for each tree set. The GNNs are trained using the respective dataset.

We shall use the CPS in Algorithm 5 to compare the result between different ratios of weights used in the loss function. In short, the algorithm picks a leaf in a trivial cherry when possible. Otherwise, it picks a random pickable leaf preferred by the GNN. The heuristics are evaluated by looking at when the first bad leaf is picked. This has the advantage of not having to deal with the value of non-temporal solutions. However, this does not incorporate the magnitude of bad leaves.

Figure 7.2 shows the ratio of bad sequences given the number of leaves picked for different weighted Clique. This is compared with a randomized version, which would pick a random pickable leaf if there are no trivial cherries. From the figure, we observe that using any GNN beats randomly picking a leaf, as expected. Furthermore, the models that favour the accuracy of good leaves perform the worst.

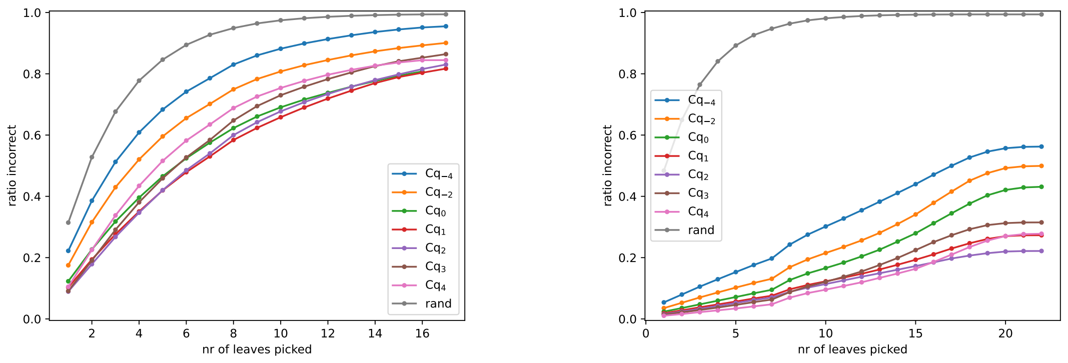
For the RTS dataset, the Clique models with a $2^{0.5} : 1$ and $2 : 1$ ratio seem to give the best result. However, increasing the ratio further results in worse results. For the RN dataset, we observe different results. The higher the accuracy for bad weight, the better the heuristic seems to perform, especially when we only let it pick a few leaves.

For the RN dataset, observe that the higher accuracy of the GNNs results in picking fewer bad leaves. While randomly picking leaves is as bad as for the RTS dataset. Furthermore, we observe that a higher weight for bad leaves improves performance.

The performance of the DAG is shown in Figure 7.3. This figure shows a similar relationship when using the DAG models. Favouring bad leaves gives better results. The DAG performs worse than the Clique, as expected when comparing the accuracies.

A small deviation of the heuristic is not to pick a random good leaf, but the leaf which the GNN determined to be the best good leaf. Such a greedy application outperforms its non-greedy variant, as shown in Figure 7.3. The greedy heuristics tend to perform similarly, especially when used on RTS. This is expected to result from the effect changing the weights of the loss function has on the results. In theory, the GNN is not able to learn anything new when these weights are adjusted. The model will only give preference to the corresponding class. In other words, the predicted best good leaf is likely to be a good leaf independently of the weights chosen for the loss function.

For the RN, we observe that there generally is a big increase in incorrect sequences for all but the random heuristics from 7 to 8 leaves. This is especially apparent for one of the greedy algorithms in Figure 7.3. This is expected to be caused by the fact that the GNN was not trained on smaller-sized problems. This can be solved by using multiple GNNs for the heuristic, each trained on different leaf set sizes.

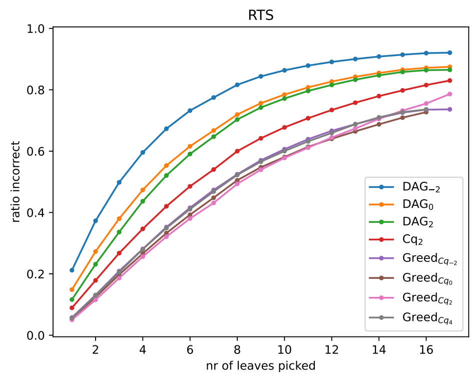


(a) RTS (b) RN

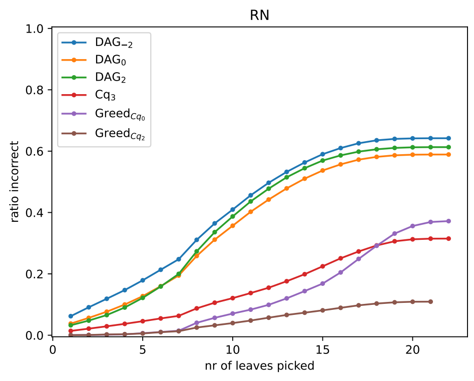
Figure 7.2: The number of incorrect solutions given the number of leaves picked. Each figure represents its corresponding dataset. The Clique_i is used with the subscript i denoting the ratio of weights used with $\sqrt{2}^i : 1$. (a) For the RTS we find that for $i = 1, 2$ tends to perform the best. (b) For the NTS we find that larger i tends to perform better.

Multiple Algorithm Runs We have examined the performance for a single output of the CPS heuristic. But a more practical use is to repeat the heuristic multiple times. Therefore, we also examine the performance of the heuristic repeated 24 times. Figure 7.4 shows these results. The ratio of incorrect sequences now depends on the 24 different runs. The figure takes the best sequence out of these 24 into account.

The results for RTS show that the previously worst models now perform the best for the first leaves picked. This of course comes from the fact that these models tend to have more good leaves, resulting in a higher chance of at least picking a good leaf in the first few tries. Even the model with the random predictor can pick two good leaves in a row in 24 tries. The results of the models converge at 6 leaves



(a) RTS

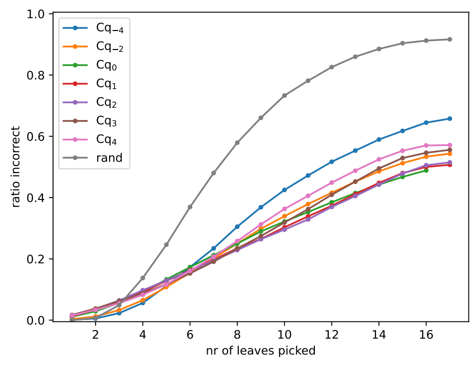


(b) RN

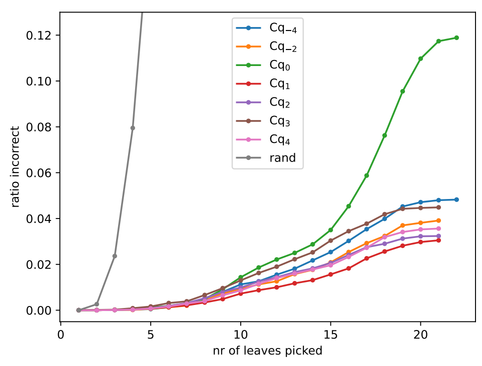
Figure 7.3: The number of incorrect solutions given the number of leaves picked. Each figure represents its corresponding dataset. The CLIQUE$_i$ and DAG^i are used with the subscript i denoting the ratio of weights used with $\sqrt{2}^i : 1$. The figure also shows the result for the greedy variant, which picks the best predicted good leaf, with the subscript denoting the model used. (a) For RTS, we find that DAG is outperformed by the CLIQUE. Furthermore, the greedy models outperform the CLIQUE (b) For RTS, we find that DAG is outperformed by the CLIQUE. Furthermore, the greedy models mostly outperform the CLIQUE.

picked. For picking more than 7 leaves we find that again the models with a $2^{0.5} : 1$ and a $2 : 1$ ratio generally outperform the rest.

For the RN dataset, we find that repeating the heuristic 24 times generally finds the optimal result for more than 95% of the problems. We previously found that higher bad-weighted models performed better. However, repeating the heuristic gives a more randomized result. This probably results from a relatively high error caused by statistical differences in the performance of the models. In other words, the ratio incorrect is so low that statistical biases can become dominant.



(a) RTS



(b) RN

Figure 7.4: The number of incorrect solution given the number of leaves picked. The heuristics are repeated 24 times for each case, after which we picked the best-performing results of these 24. Each figure represents its corresponding dataset. The CLIQUE$_i$ is used with the subscript i denoting the ratio of weights used with $\sqrt{2}^i : 1$. (a) For the RTS we find that for $i = 1, 2$ tends to perform the best. (b) For the NTS we find that most models were able to find the optimal solution for more than 95% of the cases.

We examined the practical effect of changing the weights of the loss function. We observed that moderately favouring the accuracy of bad leaves gives generally the best results. We found that picking the best predicted good leaves is generally better. However, this makes it deterministic. This makes it unrepeatable and worse than repeating the standard algorithms. The DAG models, which give worse accuracy, also perform worse in the practical example. Furthermore, there was an increase in incorrect results when the number of leaves becomes smaller than it was trained on.

7.3. Experimental Results

Heuristics are useful for finding suboptimal solutions for large problems. However, this could cause a problem, as the GNN is not trained on these problems. In this section, we shall examine the performance of the two heuristics defined in this chapter for large problems with 50 and 100 leaves. We shall also examine the performance of the GNN for tree sets with 10, 20 and 50 trees.

In this chapter, we observed that the heuristic performs best when the GNN prefers bad leaves. The GNN used in this section therefore trained with a factor 2 increase of the bad-leaf weight of the loss function. The GNN was also observed to have a decrease in performance for problems with a smaller number of leaves than the training set. We expect the same to hold for cases with a larger number of leaves. To resolve this, we use multiple GNNs. Each trained on a different range of leaves.

For this, we generate for both RTS and NT, four datasets with corresponding validation sets. Each of these datasets consists of tree sets generated by their respective method such that there are no trivial cherries and more than one leaf can be picked. The datasets differ in the number of leaves, ranging from 6-10, 11-15, 16-20 and 16-25. The three lower range datasets consist of 40000 tree sets consisting of different numbers of leaves in their range and of tree sets of the size from 2 to 5 trees. The corresponding validation set of size 20000 is constructed with a similar method. However, for the dataset of the range 6-10, there are not many unique problems, especially with two trees. These scarce problems are all put in the training dataset and none in the validation set. Both the training set and the validation set have a skewed distribution of the number of leaves caused by a lack of uniqueness.

The dataset with a range of 16-25 mainly consists of problems with 16 to 20 leaves. However, 1/7 of the data consists of 21 to 25 leaves. Note that these problems are exponentially longer to solve, resulting in far less available data. For the validation set for the range 16-25, we used 9000 tree sets, of which 2/3 have 16 to 20 leaves and the remaining part has 21 to 25 leaves. Note that the 21-to-25-leaf data was injected to hopefully result in a GNN with high accuracy for larger problems.

For each dataset, we trained the Clique, DAG and Comb with again the use of standard Adam optimizer with a learning rate of $0.01\sqrt{2}$ which is reduced by a factor $\sqrt{2}$ every 75 epochs. For the loss function, we use the weighted cross entropy loss such that the bad leaves are weighted twice as high and such that the ratio of good and bad leaves does not influence the loss. The optimization for the models is done for 450 epochs using 5-fold cross-validation which is stopped early if the the test loss is not improved in the last 60 epochs.

After the optimization of the models, we pick the best of the 5 folds using the validation sets. The loss is shown in Table 7.2. From these values, we observe that the DAG model performs the worst and the Comb performance is slightly better than the Clique.

For both CPS and NS, which use GNN as their predictor, we use the exact solver if the subproblem has 7 or fewer leaves. For subproblems with leaves between 8 and 20, we use the corresponding model which is trained on those leaves. For subproblems with $|\mathcal{X}'| > 20$, we use the model trained on 16-25 leaves. For their random variant, we keep picking randomly.

	RTS				RN			
	validation loss				validation loss			
model	6-10	11-15	16-20	16-25	6-10	11-15	16-20	16-25
Clique	0.287	0.381	0.412	0.418	0.133	0.152	0.144	0.139
DAG	0.289	0.416	0.477	0.470	0.166	0.228	0.195	0.204
Comb	0.234	0.360	0.403	0.399	0.109	0.143	0.140	0.134

Table 7.2: The validation loss for the models chosen for the heuristic. For the four different datasets of the different leaf ranges for both the RTS and RN tree sets.

To compare the heuristics we shall examine the average relative error. Because we can not find the solution of problems with $n > 25$ leaves in a feasible time, we determine the relative error for a tree set $\mathcal{T}$ as $r_{heur}(\mathcal{T}) - r_{best}(\mathcal{T})$ with r_{heur} the minimum reticulation found by using the specified heuristic and r_{best} the minimum value of all heuristics.

There is however a problem with the average relative error. This comes from the fact that there is a feasible chance that the heuristic is unable to find a temporal solution. We therefore examine two values to assess the heuristics. These are the ratio of non-temporal solutions and the average relative error for temporal solutions. However, this means that the average relative error can be inaccurate if relatively many solutions are non-temporal.

The performance for the heuristics is compared using the same running time. For the CPS heuristics, this means that the algorithms are repeated within this time. This gives a sequence of different results. Every permutation of the sequence is equally likely and gives us a smoother result. However, this is unfeasible to calculate for large sequences. Therefore we shall use a similar method, by first determining the probability of each outcome. After which we sample from this distribution. Given a sequence of outcomes, we first order possible outcomes from best to worst $(R_1, R_2, ...)$ and calculate their corresponding probability $(p_{R_1}, _{R_2}, ...)$. The probability that a new sequence of size N sampled using the probability result in R_1 as best is given by $P(R_{min} = R_1) = 1 - (p_{R_1} - 1)^N$. The probability for $P(R_{min} = R_i)$ can then be calculated using induction.

7.3.1. Random-Tree Based

In this thesis we examined two types of datasets. In this section, we examine the results of the heuristics trained on random tree sets, RTS. These problems are generated to be temporal. For these problems, the GNN had an accuracy below 80% to predict good and bad leaves.

The heuristics are tested on problems with different numbers of leaves, ranging from 20 to 50. For each leaf size, we test 960 problems which consists of tree sets containing 2 to 5 trees. The heuristics are tested using a 1-minute time frame. The results are shown in Figure 7.5. The figure shows that the NS outperforms their corresponding CPS heuristic, as it finds more feasible solutions and solutions with lower reticulations. Examining the random predictor, we observe that it outperforms the others for the lower number of leaves. Furthermore, both have a high non-temporal ratio, which results from being a faster algorithm. The DAG predictor seems to generally be able to find more feasible solutions. Note that the algorithm is slower than the Clique for smaller problems, but the DAG still outperforms it.

We also examined the effect of the running time on the algorithms for the problems with 50 leaves. These results are shown in Figure 7.6. Here the heuristics are tested with a 5-minute time frame. Similar observations can be made. The NS outperforms their counter CPS. Furthermore, we observe a steeper decline for the NS.

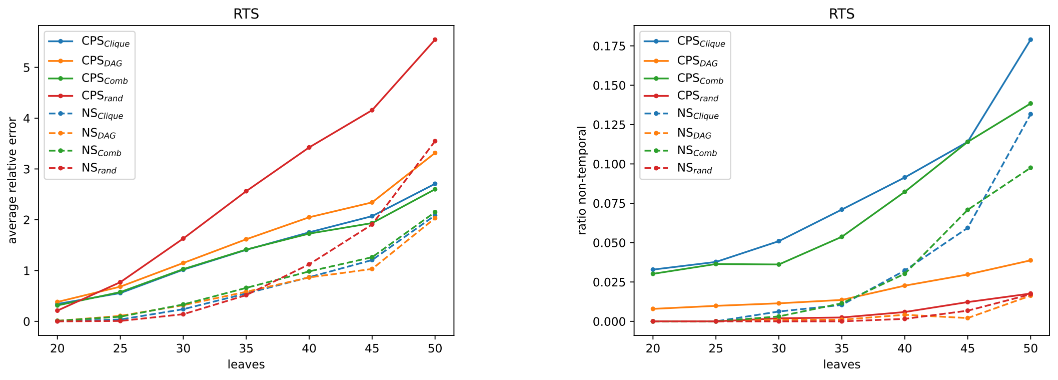


Figure 7.5: The average relative error and non-temporal ratio for different problem sizes and for both the CPS and NS heuristics with the subscript denoting the predictor used. The heuristics are run for 1 minute. The NS outperforms their corresponding CPS giving better results and finding more feasible results. The NS$_{\mathrm{DAG}}$ heuristic performs generally the best.

Large Leaf Sets We also examined problems with 100 leaves. For which we run the heuristics for 30 minutes. The results are shown in Figure 7.7. For the randomized versions, we observe that the NS takes some time to outperform its CPS counterpart. This is expected to come from the extra computational cost for keeping track of the network search, which takes some time to outperform by

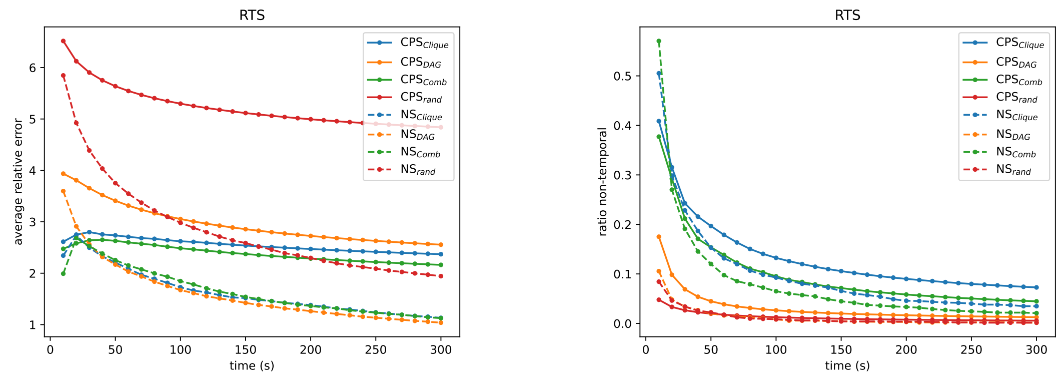


Figure 7.6: The average relative error and non-temporal ratio for different running time and for both the CPS and NS heuristics with the subscript denoting the predictor used. The results are for problems with 50 leaves. The NS outperforms their corresponding CPS giving better results and finding more feasible results. The $\mathrm{NS}_{\mathrm{DAG}}$ heuristic performed the best.

finding redundancies. This is less relevant for the GNN predictors, as the predicted set $\mathcal{P}$ of good leaves is significantly smaller. For the GNN predictors, we find again similar relations. The NS again outperforms their CPS counterpart. The heuristics using the DAG GNN find the most feasible solutions.

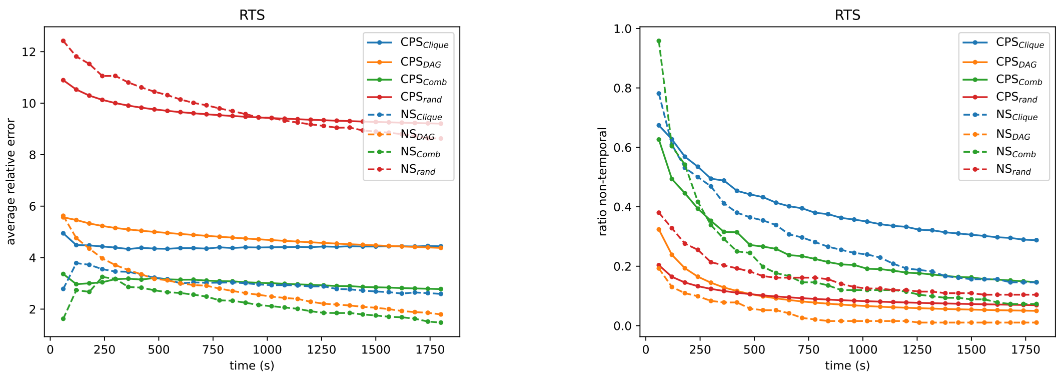


Figure 7.7: The average relative error and non-temporal ratio for different running time and for both the CPS and NS heuristics with the subscript denoting the predictor used. The results are for problems with 100 leaves. The NS outperforms their corresponding CPS giving better results and finding more feasible results. The $\mathrm{NS}_{\mathrm{DAG}}$ heuristic performed the best.

Non-Uniform Picking The results of the heuristics observed so far used a uniform probability when picking one of the good leaves. Another option is to use a predicted value p_i for leaf i for this probability. This method has been discussed at the start of this section, which included a variable α for adjusting p_i. Note that this includes all leaves not only good leaves.

The performance is compared for $\alpha \in \{-0.4, -0.2, 0, 0.2\}$. Positive α will cause all leaves to be able to be picked. While negative α greatly favours the best good leaves. We shall test these adaptations on problems with 50 leaves and for the CPS algorithm.

The results are shown in Figure 7.8. This shows that larger values of α give worse results. Furthermore, picking good leaves with a uniform distribution performs better, except for $\alpha = -0.4$. However, these did result in slightly more non-temporal solutions. Only for $\alpha = 0$ did we find slightly more temporal solutions. Note that if $\alpha \leq -0.5$, the heuristic would only pick from predicted good leaves.

From this section, we can conclude that the GNN heuristics perform generally better than the randomised versions. Except in finding temporal solutions. Furthermore, the GNN-based heuristics are able to perform well even for far larger problems on which the GNNs were not trained.

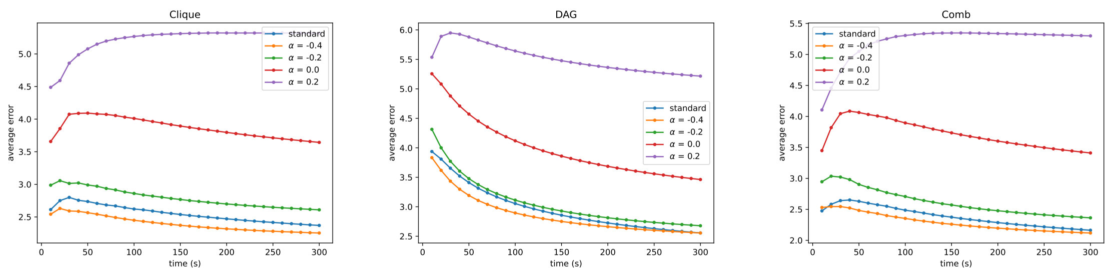


Figure 7.8: The performance of the CPS heuristic for problems with 50 leaves depending on time. Comparing the effect of the standard, picking good leaves with uniform probability, with picking leaf i depending on the predicted value p_i with different addition biases α. The tree figures each represent their own GNN model. The standard approach is better than the rest, except for the $\alpha = -0.4$.

Additionally, introducing a small probability of picking predicted bad leaves resulted in a worse performance. Even when the predictor had relatively low accuracy, following the predictor is the best choice. Furthermore, using the predicted value in the probability could improve performance, but only when good leaves are picked.

7.3.2. Network Based

The second type of problems we discussed in this thesis are problems which are constructed by first generating a network, RN. These problems are closer to practical problems found in phylogenetics.

The heuristics are again tested on problems with different numbers of leaves, ranging from 20 to 50. For each leaf size, we test 960 problems which consists of tree sets containing 2 to 5 trees. The heuristics are tested using a 1-minute time frame. The results are shown in Figure 7.9. We again examined 50 leaf problems but with a 5-minute time frame. This is shown in Figure 7.10. Both figures show that all heuristics are generally able to find a temporal solution, except for a few cases.

The figures show that the NS outperforms their corresponding CPS heuristic, as it finds more feasible solutions and solutions with lower reticulations. Looking at the random predictor we observe that it outperforms the others for the lower number of leaves. Furthermore, both have a high non-temporal ratio, which results from being a faster algorithm. The Clique predictor seems to generally be able to find the best solutions.

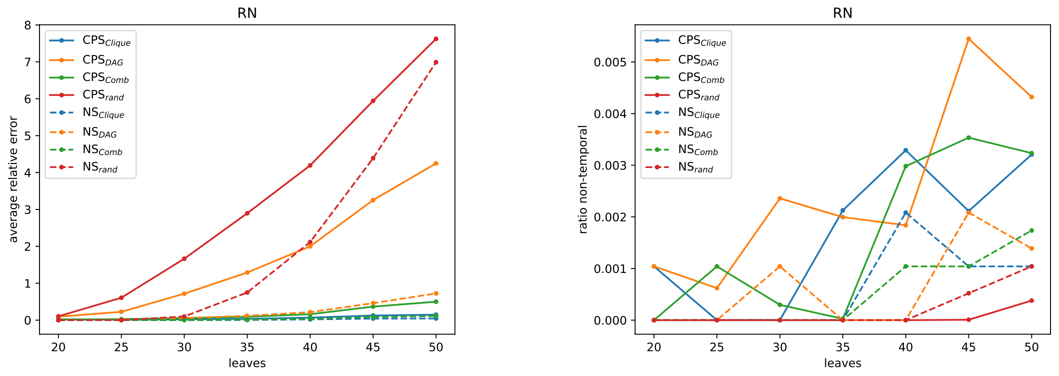


Figure 7.9: The average relative error and non-temporal ratio for different problem sizes and for both the CPS and NS heuristics with the subscript denoting the predictor used. The heuristics are run for 1 minute. The NS outperforms their corresponding CPS giving better results and finding more feasible results. The NS_{DAG} heuristic performs generally the best.

Large Leaf Sets We also examined the results of the heuristics for problems with 100 leaves for which we ran the algorithms for 30 minutes. The results are shown in Figure 7.11. From this, we observe that for the randomized predictor, the NS seem to not be able to outperform its CPS counterpart.

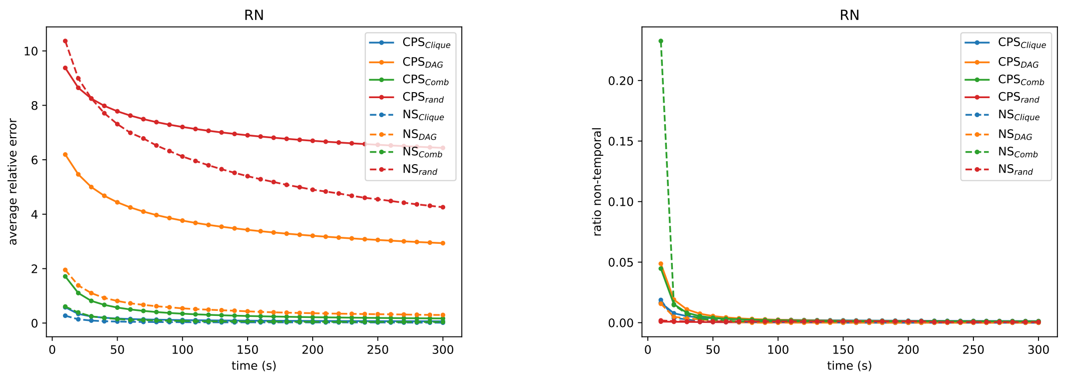


Figure 7.10: The average relative error and non-temporal ratio for different running time and for both the CPS and NS heuristics with the subscript denoting the predictor used. The results are for problems with 50 leaves. The NS outperforms their corresponding CPS giving better results and finding more feasible results. The NS_{DAG} heuristic performed the best.

Furthermore, we observe that again all heuristics are generally able to find feasible solutions and the Clique GNN seems to be the best predictor.

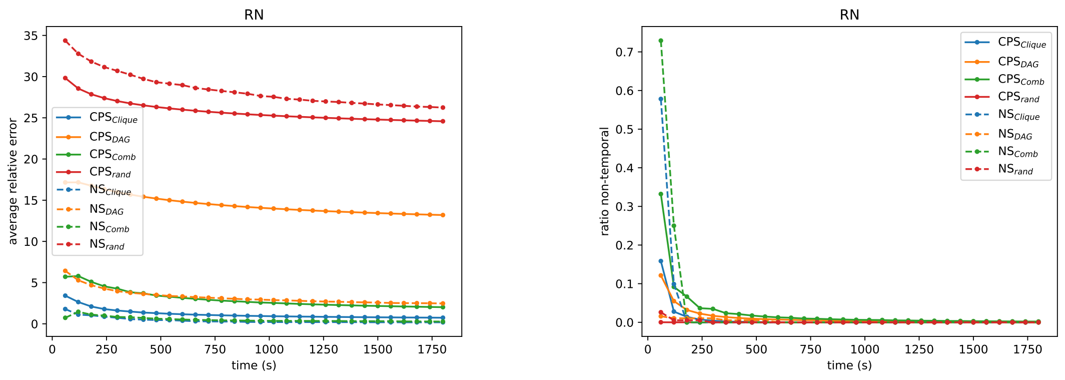


Figure 7.11: The average relative error and non-temporal ratio for different running time and for both the CPS and NS heuristics with the subscript denoting the predictor used. The results are for problems with 100 leaves. The NS outperforms their corresponding CPS giving better results and finding more feasible results. The NS_{DAG} heuristic performed the best.

Large Tree Sets In the practical use of phylogenetics the problems can have more than 5 trees. We thus also examine the performance of the heuristics for problems with 10, 20 and 50 trees. We do this for problems with 50 leaves. The results are shown in Figure 7.12, 7.13 and 7.14. Here we observe that the Clique still outperforms the others for 10 and 20 trees. However, it slightly falls off for problems with 50 trees. Note that NS_{RANDOM} seems to improve the most. This is expected as an increased number of trees generally decreases the search space.

7.3.3. Concluding Remarks

In this chapter, we have examined two types of heuristics for finding a low reticulation number problem. We found that using the GNN as a predictor gives good results and is generally able to outperform the randomized variant for both types of heuristics. Finding random cherry-picking sequences was outperformed by the network search. This was because it can reduce redundant calculations to calculate the predictions and it is able to implement choosing the best predictor for the first visit.

We also observed that the results are generally the best for problems found in phylogenetics. For most of these problems, we found similar results close to zero for their average relative error. This indicates that the relative error is close to the exact error.

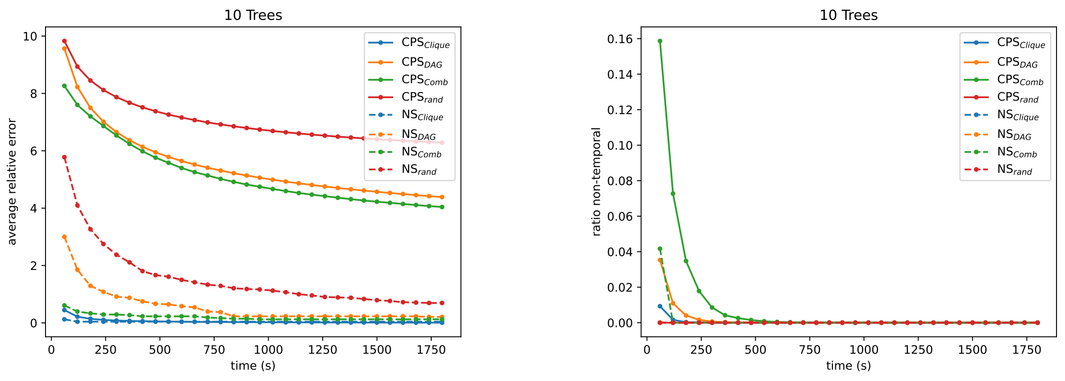


Figure 7.12: The average relative error and non-temporal ratio for different running time and for both the CPS and NS heuristics with the subscript denoting the predictor used. The results are for problems with 50 leaves and 10 trees. The NS outperforms their corresponding CPS giving better results and finding more feasible results. The NS$_{\text{DAG}}$ heuristic performed the best.

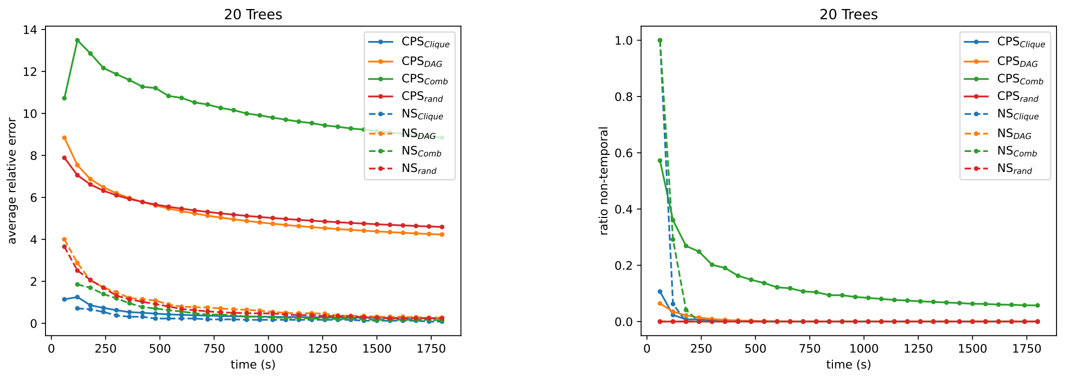


Figure 7.13: The average relative error and non-temporal ratio for different running time and for both the CPS and NS heuristics with the subscript denoting the predictor used. The results are for problems with 50 leaves and 20 trees. The NS outperforms their corresponding CPS giving better results and finding more feasible results. The NS$_{\text{DAG}}$ heuristic performed the best.

The GNN were thus able to learn an underlying correlation, as it is able to give good results for problems with 50 and 100 leaves. While it was only trained on problems of 25 leaves or less. The performance of each GNN depends on the ability to find good leaves for untrained data. However, this is not how we chose our models. This means that problems which performed worse on the validation set perform better in practice. Thus, the performances of the GNN heuristics could change depending on many variables discussed in this previous chapter.

This detachment between validation and practical use could partly be resolved by using Reinforcement learning. The GNN are then not trained at distinguishing good and bad leaves but trained to perform well in the application of the heuristic itself. This has the potential to get even better results, furthermore, it is able to learn larger problems as reinforced learning does not need exact solutions.

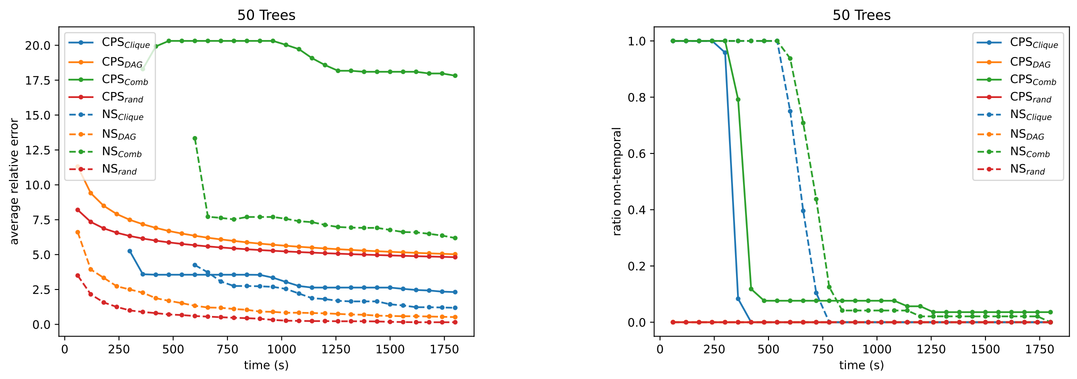


Figure 7.14: The average relative error and non-temporal ratio for different running time and for both the CPS and NS heuristics with the subscript denoting the predictor used. The results are for problems with 50 leaves and 50 trees. The NS outperforms their corresponding CPS giving better results and finding more feasible results. The NS$_{\mathrm{DAG}}$ heuristic performed the best.

8 Conclusion

Finding the minimum temporal hybridisation number is an NP-hard problem. We introduced a branching algorithm with a complexity of $O(2^n \text{poly}(n,t))$ improving the $O(5^k \cdot l \cdot t)$ FPT algorithm for problems with $k > 0.44n$. In practice, we found an exponential growth of 1.6^n.

We also introduced a branch-and-bound algorithm and lower bounds, which are derived from so-called anti-trivial leaves and cherry-bounded leaves. However, in practice, this only resulted in a slightly faster algorithm, as it resulted in the same exponential growth.

Furthermore, we reduced redundancies in the tree set by translating the tree set to the combined-tree-set graph. We also introduced cherry-growing sequences, which are the reverse of cherry-growing sequences.

For the temporal decision problem, we introduced a similar $O(2^n \text{poly}(n,t))$ algorithm. This has an interesting relationship with its cherry-growing counterpart. In general, their worst cases are easily solved by their counterparts. Consequently, running both algorithms in parallel greatly outperforms both individual algorithms. The time complexity of either individual algorithm was found to have exponential growth of 1.5^n, while the parallel version had 1.2^n.

To distinguish if a leaf is good or bad to pick, we introduced different graph-based models. The DAG uses the aforementioned combined-tree-set graph and the Clique uses a complete graph of all leaves. For the DAG model, with far-away leaves, we use a Directed Acyclic Graph Neural Network (DAGNN). However, this resulted in a significantly slower algorithm for $n < 35$ compared to the Clique. We also introduced Comb, which is a combination of the other two.

These models were tested on two types of data. For randomly generated trees, we found that it was hard for the models to learn. Resulting in accuracy below 80%. However, the more practical generated data RN, which follows an evolutionary model, had a much higher accuracy of above 94%. Furthermore, the Comb consistently outperformed the Clique, which in turn outperformed the DAG.

We also examined the performance of the models to determine if an instance has a temporal solution. The data for this was also generated for practical cases. However, this only resulted in an accuracy of around 80%, which made it impractical.

Before we applied the GNN within a heuristic, we first examined the practical effect of changing the relative weights of the loss function used to generate the model. This results in an increase or decrease in the accuracy of good leaves while doing the reverse for bad leaves. From this, we observed that it is more practical to increase the accuracy of the bad leaves at the cost of the accuracy of the good leaves.

We also observed that picking the best-predicted good leaves determined by the GNN outperforms randomly picking a good leaf. However, picking the best-predicted good leaf is deterministic, and therefore its performance cannot be improved by repeating the algorithm.

The GNN-based picking heuristics were compared to a random picking heuristic. Furthermore, we introduced network search heuristics which either used the GNN as a predictor or picked leaves at random.

The GNNs were trained on problems ranging from 15 to 25 leaves. The GNN-based heuristics were able to outperform their random variant. Especially when they were tested on problems with 50 and 100 leaves. For all the data observed, we found that all network searches outperformed the picking heuristics.

9 Discussion

In Chapter 3, we found branching algorithms which solved the minimum temporal hybridisation number with a running time of $O(2^n \text{poly}(n,t))$. We also examined branch-and-bound algorithms, but they were only slightly faster. The running time is significantly faster than the $O(5^k \cdot n \cdot t)$ FPT algorithm for large k. In this thesis, we examined problems with relatively high k. The Random Tree Set (RTS) dataset had k values larger than the number of leaves n. The other dataset we used had an upper bound of $k \leq \frac{2}{3}n$. This is relatively high, as other papers tend to use $k \leq \frac{1}{3}n$. For such cases, it could be that the aforementioned FPT algorithm outperforms our branch-and-bound algorithm.

The bounds discussed in this thesis were found to be not useful in practice. The first reason comes from the way we branch. There are generally multiple paths leading to a subproblem. Thus, pruning does not prune all its sequential subproblems, as there can exist another path. Secondly, all the bounds discussed had an upper limit of n. The branch and bound could thus have significantly faster performance for low k instances, but this needs to be tested. It would be interesting to try to find lower bounds that can be greater than n. This could potentially greatly reduce the running time.

The second improvement was the application of good leaves. The number of branches could be reduced to a small set which contains a good leaf. This again reduced the running time, but not by a significant amount, except for at the beginning of the algorithms. The inadequate improvement is again expected to come from the multiple paths leading to a subproblem. However, similar to the lower bounds, the implementation could lead to significantly faster performance for low k problems, but this needs to be tested. Further research into good and bad leaves could help reduce branching and thereby reduce the running time. In this thesis, we only found how to distinguish good leaves and did not find any application of bad leaves. However, finding bad leaves could also help in giving a better understanding of the problem itself.

The lower bounds and the good leaves discussed in this thesis could also be applicable to other algorithms. The running time of the FPT algorithm could benefit from implementing these concepts.

For the temporal decision problem, we devised an algorithm with a running time of $O(2^n \text{poly}(n,t))$, as it also searches the same space. The algorithm is a depth-first approach. This makes the algorithms faster in practice than the breadth-first variant. There are no available algorithms that are specialised for this problem for comparison. The practical use of the algorithm is to run it before the algorithms for finding the minimum temporal hybridisation number, as it is faster and gives an upper bound or shows that no solution exists.

We also introduce a cherry-growing algorithm. This was noteworthy slower than its cherry-picking variant for the minimum temporal hybridisation number. However, it is as fast as a cherry-picking algorithm for the temporal decision problem. Furthermore, running both in parallel and stopping when one of the runs is completed was found to have a substantially smaller exponential increase in running time in practice. This increase could be explained by two things. The first is that the worst case for a picking algorithm is not the worst case for the growing algorithm and vice versa. Secondly, the depth-first aspect could be the reason, as picking the wrong leaf could lead to a tree set with no temporal solution.

The lower bounds discussed in this thesis do not simply translate to lower bounds for the growing algorithm. The concept of good leaves also does not translate to cherry growing. For example, trivial cherries are always good to pick, but not always good to grow. It is interesting to look at these two concepts for cherry growing in future research.

We also introduced substructures, that cause the tree set to be non-temporal. While finding such substructures could be useful for some instances, it is generally less useful in practice. Finding the substructures we introduced in this thesis had a running time of $O(n^3)$ and $O(n^4)$. This makes them quite slow for repeated use. However, limited use could help in reducing running time. Further research could produce more of these substructures. This could help us understand the problem itself better.

The graphs used in the graph neural networks are discussed in Chapter 4. We first examine the problem of communication between distant nodes in graph neural networks. We partially solve this problem by reintroducing the DAGNN. We found that DAGNN outperforms the general NN and recursive variant in repeated communication. However, it is only applicable to DAG as the name implies.

The two types of input graphs are the combined-tree-set graph and the complete graph of the leaves. The running time for both GNNs is polynomial dependent on n. It is thus unexpected that the GNN could solve the NP problems exactly. However, finding substructures can be done in polynomial time. Nevertheless, the models were unable to find 100% accuracy for two of the three substructures examined. Therefore, we expect the models to mainly have learned statistical correlations instead of logical deductions.

In Chapter 6, we examined the performance of the GNN in predicting good leaves to pick. For this concept, we introduced two data types. The first results from simulating an evolutionary model. These resulted in accuracies between 94.1% and 97.4% for cases with 25 leaves and 2 to 5 trees. GNNs are thus a great way to predict between good and bad leaves for these kinds of cases. Note that the data was simulated with a reticulation number of $k = 15$.

We introduced three types of models, the Clique, DAG and their combination Comb. Comb gave the highest accuracy, followed by Clique and lastly DAG. These models were tested with different deviations. Note that the performance of our models depends on multiple parameters and choices. These influence the performance. Neural networks is a highly researched subject. There are thus many improvements to the models in this thesis to be made. For example, the use of dropouts and post-training. A second type of improvement is the graph of the GNN. Other graphs could be used, or small adaptations to the ones in this paper. Furthermore, in this thesis, we did not try to find the best features. Looking for other features could thus increase performance.

The performance of GNN to predict if a set of trees is temporal or not was discussed in Chapter 7. Here the models were trained on data generated using evolutionary models which produce tree-child networks. This resulted in an accuracy between 78.6% and 80.3% for the different models. This training data contained problems with 35 leaves and had 2 to 5 trees.

The accuracy is too low to be used in practice. This is because the exact solver is, in most cases, done within a few seconds, as it either finds a cherry-picking sequence or quickly determines to be non-temporal in these cases. Running the exact solver with a predetermined runtime is thus expected to be a better heuristic for the temporal decision problem.

For future research, we could examine the use of GNN in predicting leaves that make the problem non-temporal when picked. This could be helpful to the heuristics in this thesis as it was not always able to find a temporal solution.

The GNNs used were trained with a preference for bad leaves. This means the GNNs tend to predict more bad leaves, which results in a higher accuracy for bad leaves at the cost of the accuracy for good leaves. This was observed to have better performance. However, this could change for cases with more leaves or more trees than we trained on.

There are no published heuristics for temporal hybridisation number, this makes it hard to compare. We hope to have created a strong baseline. The use of heuristics is to at least find networks with a small reticulation number. This can also be used for an upper bound. Both heuristics could be improved by looking for good leaves. In this thesis we only implemented picking trivial cherries, but we could also look for low-hanging leaves and clusters. The second type of heuristics could also be improved in other ways. There has been a lot of research in Tree Search algorithms. For example, there are Monte Carlo tree search algorithms, which pick the most promising moves. This could increase the performance of our random variant.

Another option is to apply the ideas from this thesis to tree-child networks. We expect that a similar exact solver can be made with a running time of $O(2^{(nt)}\text{poly}(n,t))$ or even faster. Furthermore, we expect that also the GNN and heuristics can be extended to the tree-child variant.

Bibliography

[1] Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. *arXiv preprint arXiv:2006.05205*, 2020.

[2] Maria-Florina Balcan. Data-driven algorithm design. *arXiv preprint arXiv:2011.07177*, 2020.

[3] Giulia Bernardini, Leo van Iersel, Esther Julien, and Leen Stougie. Constructing phylogenetic networks via cherry picking and machine learning. *Algorithms Mol Biol*, 18(13), 2022.

[4] Monica Bianchini, Marco Gori, and Franco Scarselli. Processing directed acyclic graphs with recursive neural networks. *IEEE Transactions on Neural Networks*, 12(6):1464–1470, 2001.

[5] Magnus Bordewich and Charles Semple. Computing the minimum number of hybridization events for a consistent evolutionary history. *Discrete Applied Mathematics*, 155(8):914–928, 2007.

[6] Sander Borst, Leo van Iersel, Mark Jones, and Steven Kelk. New FPT algorithms for finding the temporal hybridization number for sets of phylogenetic trees. *Algorithmica*, 84(7):2050–2087, 2022.

[7] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[8] Theo Broeders. Temporal Networks and GNN, 2024. URL: `https://github.com/fonemillion/Phylogenetics.git`.

[9] Tianle Cai, Shengjie Luo, Keyulu Xu, Di He, Tie-Yan Liu, and Liwei Wang. Graphnorm: a principled approach to accelerating graph neural network training, 2021. arXiv: `2009.03294 [cs.LG]`.

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

[11] Yangrui Chen, Jiaxuan You, Jun He, Yuan Lin, Yanghua Peng, Chuan Wu, and Yibo Zhu. SP-GNN: learning structure and position information from graphs. *Neural Networks*, 161:505–514, 2023.

[12] Janosch Döcker and Simone Linz. On the existence of a cherry-picking sequence. *Theoretical Computer Science*, 714:36–50, 2018.

[13] Janosch Döcker, Leo van Iersel, Steven Kelk, and Simone Linz. Deciding the existence of a cherry-picking sequence is hard on two trees. *Discrete Applied Mathematics*, 260:131–143, 2019.

[14] Vijay Prakash Dwivedi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Graph neural networks with learnable structural and positional representations. *arXiv preprint arXiv:2110.07875*, 2021.

[15] Matthias Fey and Jan E. Lenssen. PyTorch Geometric: graph neural networks library. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019. URL: `https://github.com/pyg-team/pytorch_geometric.git`.

[16] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32, 2019.

[17] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[18] Zheng Hu, Jiaojiao Zhang, and Yun Ge. Handling vanishing gradient problem using artificial derivative. *IEEE Access*, 9:22371–22377, 2021.

[19] Peter J Humphries, Simone Linz, and Charles Semple. Cherry picking: a characterization of the temporal hybridization number for a set of phylogenies. *Bulletin of mathematical biology*, 75(10):1879–1890, 2013.

[20] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, et al. *An introduction to statistical learning*, volume 112. Springer, 2013.

[21] Remie Janssen and Pengyu Liu. Comparing the topology of phylogenetic network generators. *Journal of bioinformatics and computational biology*, 19(06):2140012, 2021.

[22] Kun Jing and Jungang Xu. A survey on neural network language models. *arXiv preprint arXiv:1906.03591*, 2019.

[23] Diederik P Kingma and Jimmy Ba. Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[24] Yassin Kortli, Maher Jridi, Ayman Al Falou, and Mohamed Atri. Face recognition systems: a survey. *Sensors*, 20(2):342, 2020.

[25] Simone Linz and Charles Semple. Attaching leaves and picking cherries to characterise the hybridisation number for a set of phylogenies. *Advances in Applied Mathematics*, 105:102–129, 2019.

[26] Rune B. Lyngsø, Yun S. Song, and Jotun Hein. Minimum recombination histories by branch and bound. In Rita Casadio and Gene Myers, editors, *Algorithms in Bioinformatics*, pages 239–250. Springer Berlin Heidelberg, 2005.

[27] Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Communications of the ACM*, 65(7):33–35, 2022.

[28] Luay Nakhleh, Tandy Warnow, and C Randal Linder. Reconstructing reticulate evolution in species: theory and practice. In *Proceedings of the eighth annual international conference on Research in Computational Molecular Biology*, pages 337–346, 2004.

[29] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[30] On cherry-picking and network containment. *Theoretical Computer Science*, 856:121–150, 2021.

[31] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[32] Hervé Philippe and Christophe J Douady. Horizontal gene transfer and phylogenetics. *Current opinion in microbiology*, 6(5):498–505, 2003.

[33] Joan Carles Pons, Celine Scornavacca, and Gabriel Cardona. Generation of level-kk LGT networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 17(1):158–164, 2020.

[34] Marcelo Prates, Pedro HC Avelar, Henrique Lemos, Luis C Lamb, and Moshe Y Vardi. Learning to solve NP-complete problems: a graph neural network for decision tsp. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33 of number 01, pages 4731–4738, 2019.

[35] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[36] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. Masked label prediction: unified message passing model for semi-supervised classification. *arXiv preprint arXiv:2009.03509*, 2020.

[37] Roy D Sleator. Phylogenetics. *Archives of microbiology*, 193:235–239, 2011.

[38] Veronika Thost and Jie Chen. Directed acyclic graph neural networks. *arXiv preprint arXiv:2101.07965*, 2021.

[39] Leo van Iersel, Steven Kelk, and Celine Scornavacca. Kernelizations for the hybridization number problem on multiple nonbinary trees. *Journal of Computer and System Sciences*, 82(6):1075–1089, 2016.

[40] Leo van Iersel, Remie Janssen, Mark Jones, Yukihiro Murakami, and Norbert Zeh. A practical fixed-parameter algorithm for constructing tree-child networks from multiple binary trees. *Algorithmica*, 84(4):917–960, 2022.

[41] Leo van Iersel, Steven Kelk, Nela Lekić, Chris Whidden, and Norbert Zeh. Hybridization number on three rooted binary trees is EPT. *SIAM Journal on Discrete Mathematics*, 30(3):1607–1631, 2016.

[42] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[43] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[44] Louxin Zhang, Niloufar Abhari, Caroline Colijn, and Yufeng Wu. A fast and scalable method for inferring phylogenetic networks from trees by aligning lineage taxon strings. *Genome Research*, 33(7):1053–1060, 2023.

[45] Qi Zhao, Ze Ye, Chao Chen, and Yusu Wang. Persistence enhanced graph neural network. In *International Conference on Artificial Intelligence and Statistics*, pages 2896–2906. PMLR, 2020.