Universiteit Leiden
The Netherlands

BSc Bioinformatics

Building a phylogeny for the fungal kingdom with ITS data

Casper Carton & Luuk Romeijn

Supervisors:
Fons Verbeek, Rutger Vos, Irene Martorelli & Vincent Merckx

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 8, 2022

Abstract

Having an overarching fungal phylogenetic tree based on ITS data would take away the demand for taxonomic identifications in ITS studies. Such a tree would be a welcome addition to LIACS Mycodiversity Database (MDDB). We propose a two-step approach for the creation of such a tree: i) creating a phylogenetic backbone; ii) expanding this backbone with new sequence data. The backbone is based on data from the UNITE database, using its taxonomic identifications to divide the data into chunks. The algorithm uses k-mer pairwise distancing for the removal of outliers and generating a high-level fungal phylogeny. Expanding the backbone is also based on this alignment-free distance method, using similarity scores to identify subtrees in the backbone that are similar to the to-be-added sequence. Several experiments have been conducted in order to gain insight in the reliability and limitations of the proposed pipeline. Parameter settings are compared and usage recommendations are provided.

Contents

1 Introduction

1.1 Fungal biodiversity

1.1.1 Importance

Fungal biodiversity is an important field of study, as fungi play a big and crucial role in Earth's ecosystems. Fungi are eukaryotic heterotrophs, meaning that each of their cells has a nucleus, and that they take nutrition from their surroundings. Fungi feed on (dead) organic material, which they break down through the production and excretion of extracellular enzymes. Consequently, they classify as decomposers. Fungi are most abundantly present in soil, where they can form long filamentous structures called hyphae (Figure 1, left). By decomposing the organic matter in their environment, they provide nutrients to the vegetation, control soil structure, and regulate the physiological conditions of their surroundings [FHBJ18]. Moreover, some fungi can form mycorrhizal associations. In such a symbiotic relationship, a fungus colonizes a plant's root system (rhizosphere), increasing the plant's nutrient uptake and providing pathogen protection. Despite their essential role in nature, fungi are best known for the sometimes edible fruiting bodies (mushrooms) that only a specific group of fungi (Basidiomycota) forms during sexual reproduction (Figure 1, right).

On top of that, fungi play an important role in the bioindustry as well. Humans have learned to cultivate fungi and use their decomposing and secretory abilities for several applications. For example, fungi are being used for the production of enzymes and antibiotics, and yeast is used in the production of beer and bread [AD03].

It has been widely accepted that fungi comprise one of the kingdoms of life, but their relationships to the other kingdoms and their community size have been a matter of debate over time. Where fungi were initially thought to be related to plants due to a similar morphology, phylogenetic research pointed out that they were instead more closely related to animals [BP93]. As for the kingdom's size, recent estimates lie between 2.2 and 5.1 million species [HLHJ17] [OPJ+05], making it the second-largest kingdom (with animals at first place). However, only 120,000 species have been taxonomically described, meaning that 92% to 98% remain to be identified.

Figure 1: Fungi in nature. Left: underground hyphae network [Source: Wikipedia]. Right: the fruiting body of the famous fly amanita (*Amanita muscaria*) [Source: Hans Veth].

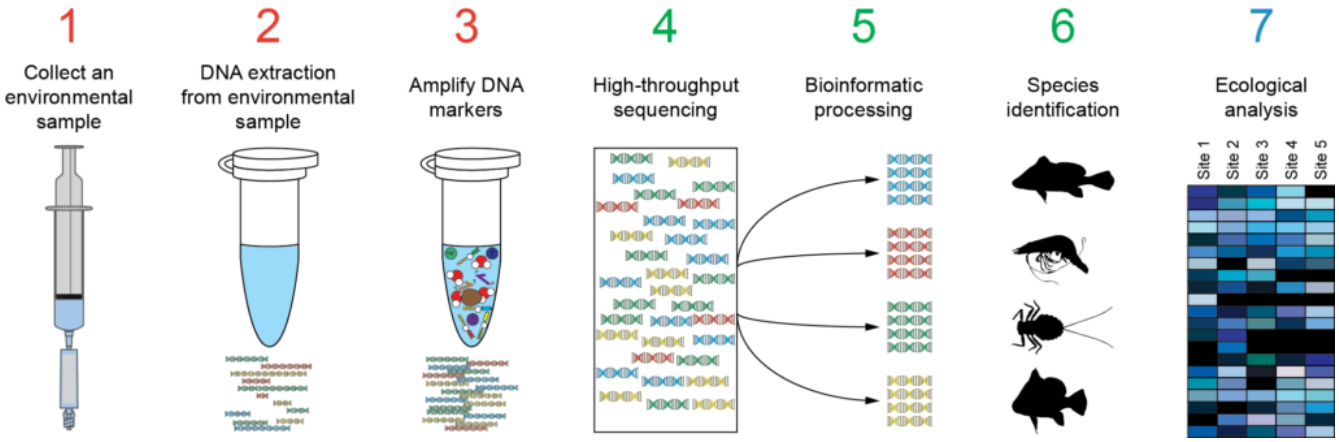

Figure 2: Process of DNA metabarcoding [Source: Nature Metrics].

1.1.2 Metabarcoding

In the past, a soil's fungal composition was described based on morphological characteristics. Nowadays, a technique called DNA metabarcoding is used. Metabarcoding identifies organisms in a sample based on the sequence of a barcode (marker) gene. Barcode genes must have the following two characteristics: i) they must show as few intraspecific and as much interspecific variation as possible; ii) they must contain highly conserved flanking sites [KE08]. The first characteristic is necessary to achieve an almost one-to-one mapping from sequence to species, facilitating the accurate identification of species in a sample. The second requirement is important to be able to extract the barcode gene from a sample (like soil) with PCR primers that match to the flanking sites. The process of DNA metabarcoding is summarized in Figure 2. Standardized pipelines for the preparation of such samples are available [SJV+17], allowing for a less cumbersome and more sensitive species identification compared to traditional morphology-based methods.

For fungi, the optimal barcode gene is considered to be the internal transcribed spacer (ITS) region [SSH+12]. ITS adheres to both of the above-mentioned criteria, and was found to have the highest success rate and the broadest range when compared to other barcode genes. The ITS region is a product of the post-transcriptional processing of a nuclear rRNA gene. As part of the ARISE project, Naturalis will perform fungal biodiversity studies across the Netherlands, generating large quantities of well-annotated ITS data.

1.1.3 ITS databases

The ITS data generated by the many fungal biodiversity studies helps us gain insights about fungi and the environment they live in, but also leads to some challenges. Firstly, the question arises of how similar two sequences need to be in order to make up a species. These clusters of sequences are referred to as operational taxonomic units (OTUs). The UNITE database is a collection of over 1,000,000 public ITS sequences and is regarded as the main reference database for ITS. It attempts to solve the OTU problem by using clustering thresholds, combining sequences with a specific similarity percentage into a so-called species hypothesis (SH) [ANL+10]. The second challenge is that sequences that have not been found earlier still need to be coupled to a taxonomic identification. This problem leads to a high number of unidentified sequences in UNITE (See Section 2.1.2). Lastly,

the heterogenous sampling, processing, and annotation of ITS data makes it challenging to perform large-scale studies, despite the data's availability.

The Mycodiversity Database (MDDB)[1] from Leiden Institute of Advanced Computer Science (LIACS) aims to solve some of the above-mentioned problems as a framework for the analysis of fungal metabarcoding data in a larger context [MHK+20]. Opposed to UNITE, MDDB provides consistent annotation and curation for all its data entries, facilitating large-scale studies. Furthermore, MDDB provides a front-end interface that allows for easy filtering and geographical visualization. As database management system the column-oriented MonetDB is used [IGN+12]. This system allows for quick data retrieval and user-defined Python functions, and is well-suited for the rich annotation (location, pH, biome information) that each data entry contains. MDDB uses Zero-radius OTUs (ZOTUs) instead of species hypotheses with clustering thresholds, meaning that any two sequences that are not exactly the same will be considered a unique taxonomic unit. Nonetheless, each sequence is blasted against UNITE and thus mapped to an SH with a similarity score. This causes each entry to have a taxonomic identification, although many of them are expected to be inaccurate.

1.2 Phylogenetic analysis

Phylogenetic analysis is known as the study of evolutionary relationships between different species, individuals or characteristics of an organism, such as genes or proteins [ZJ12]. It makes use of branching diagrams, called phylogenetic trees, to acquire information about biological diversity and provide insight into the differences that have been created during evolution.

In the past, phylogenetic trees could only be created based on morphological characteristics. But since the existence of DNA metabarcoding, phylogenetic trees can now mostly be based on similarities and differences in genetic characteristics. This makes it easier to distinguish mutations that may have been created over evolution [Bro02]. However, morphological data remains crucial to test and time-scale the molecular trees [LP15].

1.2.1 Phylogenetic trees

The phylogenetic tree, as shown in Figure 3, also known as a phylogeny or evolutionary tree, consists of leaves, nodes and branches. The leaves of the tree, also known as tips, can represent species, individuals or even genes or proteins from either living or extinct lineages. If the leaves represent a formally named group, they are called taxa [SB16]. These taxa are connected to the tree the same way as in nature, through branches. And the branches itself connect to other branches through nodes. But there can be a difference to how many branches are connected through these nodes. When only two outgoing branches are connected to a node, this means that there are two descendants coming from this node. This is called bifurcation, but in case of multifurcation there may be multiple branches connected to a node and there are thus multiple descendants coming from it.

These nodes represent the most recent common ancestor(MRCA) of the entities at the leaves of this sub-tree. This sub-tree is better known as a 'clade', this is a part of the phylogeny and includes an ancestral lineage with all the descendants of that ancestor [Kli09]. This means that taxa with a common ancestor with few nodes in between are more closely related than taxa that are further away from each other in the tree with lots of nodes between them and their common ancestor.

[1] https://mycodiversity.liacs.nl/

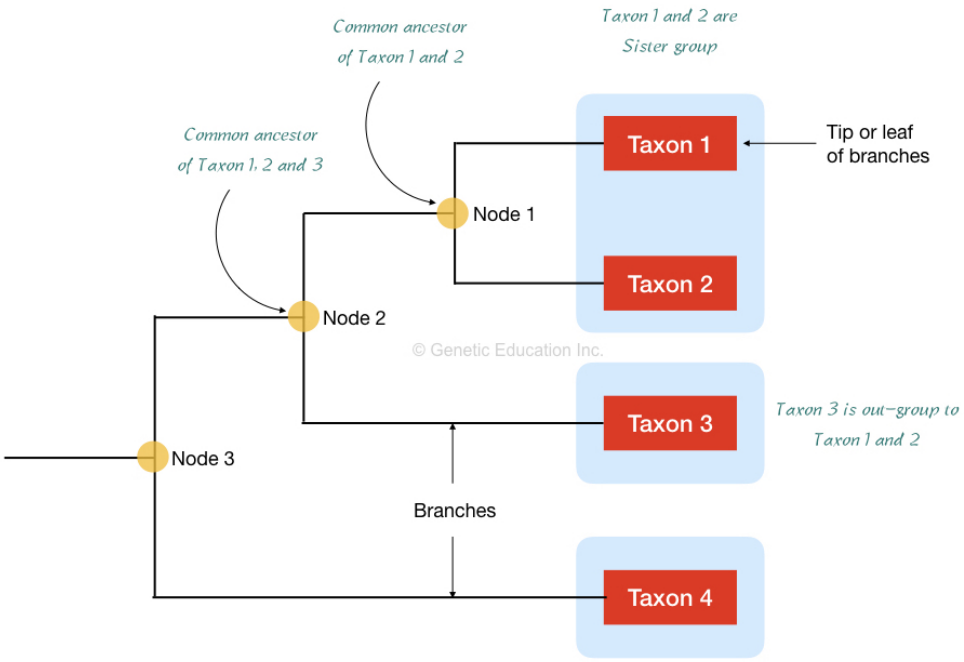

Figure 3: Grapical representation of the phylogenetic tree. [Source: Genetic Education]

1.2.2 Different types of trees

Phylogenetic trees can be visualized differently, as a consequence they also have different meanings. These differences are very important to understand, because they are critical to what can be concluded from the trees. First of all a *rooted tree*, it has a node, called the root, that corresponds with the MRCA of all the taxa at the leaves of the tree. So this is the common ancestor from which all tips have descended and diverged through time [Cho14]. *Unrooted trees* however, only show the relatedness of the taxa, but do not define the evolutionary path [KWMB16].

Secondly, there can be scaled or unscaled phylogenetic trees. In *scaled trees*, the length of the branches are proportional to the amount of evolutionary changes that occurred since the previous node. This means the tree does not only show what taxa are closest related, but also how closely related they are. In a *unscaled tree* however, this is not the case because the length of a branch has nothing to do with the evolutionary divergence [Cho14].

1.3 Research questions

The aim of this research is to create a fungal phylogenetic tree for the ITS data in MDDB. The most important motivation for this is that having an overarching fungal phylogeny would take away the demand for taxonomic identifications in ITS studies. Currently, to be able to identify a fungus in a sample, its barcode must be matched to a taxonomy. While UNITE contains expert-provided taxonomic identifications, many of its sequences are still unidentified. MDDB matches its entries to UNITE SHs through a blast search. However, even if this match is with an identified species, it may still only be with a low similarity score. With a phylogeny, researchers can identify a fungus simply based on the position of its sequence in the tree.

A second motivation is that having a phylogenetic tree adds value to MDDB. Opposed to taxonomic identifications, a phylogeny is based purely on data, which is favorable in some situations. For example, the branch lengths can be used in evolutionary analysis. On top of that, clustering analyses can be enriched with the phylogenetic information, e.g. deriving conclusion about which branches of the tree are over-/underrepresented at specific physiological conditions. Thus, having a phylogeny available will further facilitate fungal biodiversity research with MDDB.

Creating this tree cannot be done solely by using an existing tree building algorithm. The root cause for this is that the high computational complexity of tree building algorithms causes them to be extremely slow with large amounts of data. Moreover, since the goal of the phylogeny is to span the entire fungal kingdom, the evolutionary distance between sequences is high. This increases the difficulty of the task.

We propose a two-step approach. Firstly, an ITS backbone tree will be generated based on data from UNITE. UNITE's taxonomy annotations will be used for dividing the data into smaller chunks. Trees will be built for each of these chunks, which then get grafted together to form the backbone. Secondly, this backbone will be expanded with data from MDDB through a phylogenetic placement algorithm. In this paper, the proposed algorithms will be explained and the influence of the involved parameters will be discussed. Specifically, the following research questions will be answered:

> *RQ1: How can a divide-and-conquer approach be applied for the construction of a phylogenetic backbone tree based on UNITE ITS data?*
>
> *RQ2: How can this backbone tree be expanded by efficiently determining the correct position when new data is introduced?*

1.4 Thesis overview

Throughout the project, Luuk Romeijn focussed on answering RQ1, Casper Carton on answering RQ2. Thus, Luuk was responsible for Chapters 3.2 and 4.2, while Casper was responsible for Chapters 3.3 and 4.3.

The used data, software, and hardware will be specified in Chapter 2. An in-depth explanation of the proposed implementation is given in Chapter 3. This section also identifies some parameters. In Chapter 4, we will discuss results of experiments with these parameters, and report on the performance of our algorithm. Finally, a conclusion will be provided in Chapter 5, along with a discussion on the current limitations of our implementation and future work.

This paper is a bachelor thesis for the bioinformatics program at Leiden Institute of Advanced Computer Science (LIACS). Supervision from LIACS was provided by Dr. F.J. Verbeek and I. Martorelli Msc. Supervision from Naturalis was provided by Dr. R. Vos and Dr. V. Merckx served as supervisors.

2 Material & methods

2.1 Data

2.1.1 MDDB sequence data

Sequence data from MDDB was retrieved through the front-end interface. This retrieval functionality was implemented as a preparation part of this project. Currently, this feature (Figure 4) is only accessible at a test version of the website[2]. Two download formats were created, allowing the user to download all data columns in a .csv file and/or to export the ZOTU sequence data in FASTA format. If filters are specified, the download will only contain entries matching the filters. MDDB contains 111,576 ZOTU sequences. In one of the many data curation steps that data submitted to MDDB undergo [MHK+20], it is made sure that each of the sequences contain the same number (250) of characters.

2.1.2 UNITE

The expert-provided taxonomic identification that UNITE contains (opposed to MDDB) for its SHs allows us to make somewhat substantiated decisions on how to divide the data into chunks. To do this, we make the following assumption:

> *Assumption 1: Sequences with the same taxonomic identifications at splitting level will end up in the same clade of a phylogenetic tree.*

The higher the taxonomic rank, the better supported this assumption will be. The splitting level can be a fixed taxonomic rank (e.g. order), but can also be more fluid (Section 3.2.1). Once the chunks have been identified, subtrees are generated for each of them, which then get grafted together into the desired backbone tree.

A UNITE release (version 8.3) containing ITS sequences clustered into species hypotheses (SHs) was downloaded from here [AZP+21]. This release contains three different set of files for different clustering thresholds: 97%, 99%, and a dynamic release containing SHs from a combination of different thresholds (expert-determined, depending on the SH's reliability). To be as conservative as possible, it was decided to use the dataset from the 97% clustering threshold, which contains

[2] https://mycodiversity.liacs.nl/thesis/test

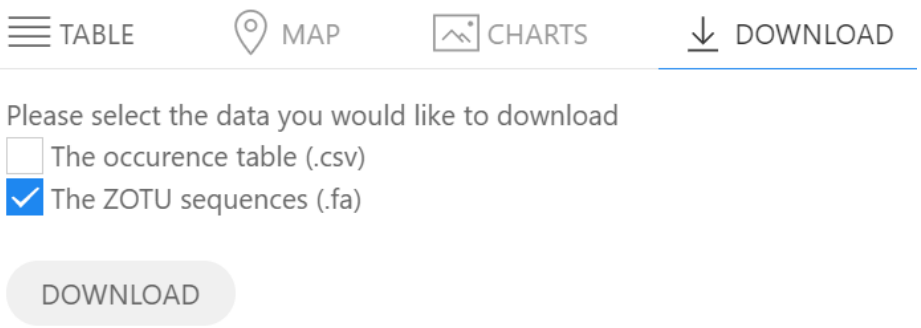

Figure 4: The implemented sequence retrieval feature at MDDB's front-end.

41,898 SHs. For comparison: the 99% threshold dataset contains 60,658 entries, the dynamic dataset contains 58,440. Each SH corresponds to an ITS sequence and a taxonomy.

Figure 5 shows the distribution of taxonomic identifications at phylum level. Note that 1489 sequences have an unidentified phylum. In total, the number of not-fully identified sequences in this release equals 22,364. Fortunately, for dividing the data, a taxonomic identification is only required up to the splitting level. Figure 6 shows the distribution of the length of the sequences within the UNITE dataset.

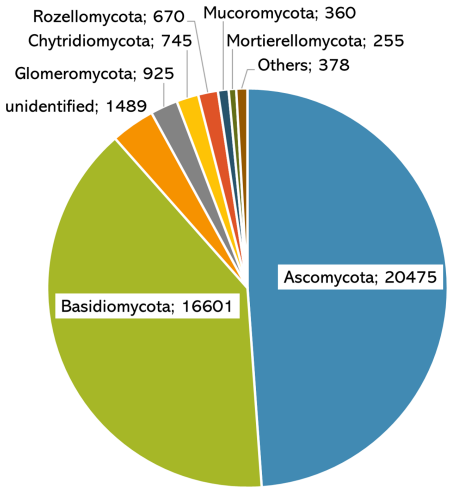

Figure 5: Pie chart of the number of sequences in the UNITE v8.3 release, grouped per phylum.

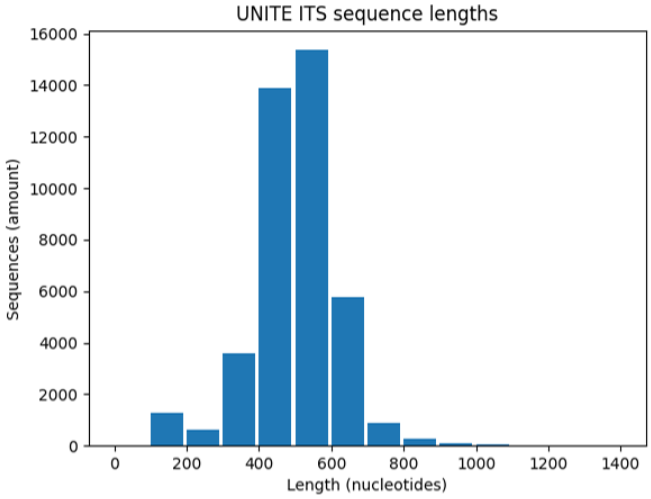

Figure 6: Length histogram of ITS sequences in UNITE v8.3. Dataset with clustering threshold of 97%.

2.2 Software

Python, version 3.8.10, was used for the implementation. This allowed us to make use of multiple useful software packages supported by Python. Such as the Biopython package, for reading and writing from and to fasta files and accessing MAFFT and RAxML, and the alfpy package for alignment-free distance measure.

2.2.1 MAFFT

MAFFT, **M**ultiple **A**lignment using **F**ast **F**ourier **T**ransform, is a tool used for multiple sequence alignment of amino acid or nucleotide sequences. It tries to align the sequences in such a way that as many characters in each sequence are matched without creating too much gaps within the sequences [SSM22] (see Figure 7).

MAFFT runs on the command-line and therefore needs some input parameters from the user. The most important parameter is an input file in fasta format with the sequences that needs to be aligned. On top of that, there are a lot more parameters that can be specified, such as what algorithm is used for all pairwise alignment or parameters that influence the scoring system of the aligned sequences. For the pairwise alignment it is possible to use the smith-waterman (l-INS-i) algorithm

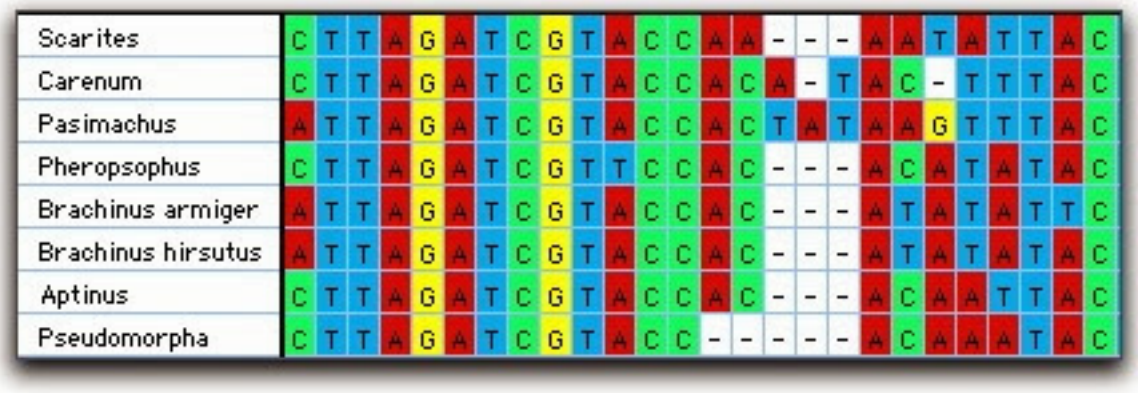

Figure 7: An example of multiple sequence alignment where for some sequences a gap is introduced to improve the alignment. [Source: MCQ Biology]

[SW81], which performs a local alignment, or the needleman-wunsch (FFT-NS-2) algorithm [NW70], which performs a more global alignment. Version 7.490 was used in our implementation.

For our Python pipeline, the *MafftCommandline* application from the Biopython package was used to access MAFFT from within the Python code. For this, it also needs the path to the MAFFT executable file as a parameter. Besides the executable and the input file, we specified the output order to be based on the alignment instead of the input order, because this seems to be of importance when using these results. We also set the pairwise alignment algorithm to be the local, smith-waterman, algorithm.

2.2.2 RAxML

The most important feature to create and expand a phylogenetic tree is to actually build a tree. To do this, we use RAxML (**R**andomized **A**xelerated **M**aximum **L**ikelihood) version 8.2.4, which is a maximum likelihood tree search algorithm that returns trees with good likelihood scores [Sta14]. It needs a file with aligned sequences as input and returns multiple files, where in our case the *RAxML_bestTree* file is the most important, which contains the tree with the highest likelihood score.

Just like MAFFT, RAxML runs on the command line and requires some input parameters. First of all, it needs the input file of fasta format with the aligned sequences. Secondly, the model that is used for nucleotide or amino acid substitution. In case of working with nucleotides, this can be either GTRCAT or GTRGAMMA and in case of amino acids, either PROTCAT or PROTGAMMA. Thirdly and lastly, a name under which the output files can be found. Besides these three required parameters, there are more that can be specified. Constraints can be added to the tree to which the program must adhere. This way it is possible to tell the program that certain sequences belong together and it should not change these relationships. It is also possible to define a sequence from the input as an outgroup. This will be the first branch in the tree and is therefor of importance in rooting the tree correctly.

In our Python pipeline, the *RaxmlCommandline* application from the Biopython package was used to access RAxML from within the Python code. This application needs the path to the RAxML executable as extra parameter for this. We made use of the GTRCAT model, since we are

working with nucleotides and our supervisor recommended this model over GTRGAMMA. We also made use of the constraints and outgroups parameter, but these are dependent on the input file.

2.2.3 Alfpy

A key component of the proposed algorithm is an alignment-free distance measure, for which we use Alfpy [ZVAK17]. In [ZGB+19], several alignment-free distance measures are compared. In this study, Alfpy is well-performing. On top of that, it is directly implemented in Python, which gives us the opportunity to make small modifications and ensures a seamless integration in our own Python pipeline.

Two distance measures from the Alfpy Python module [ZVAK17] were investigated. The w-metric is a simple distance measure originally meant for proteins, and calculates a pairwise similarity score based on differences in single-character frequencies in combination with a substitution matrix [VGOA04]. The other distance measure is based on k-mer patterns. First, it generates a list of all possible alphabet (nucleotide) combinations with length k. The occurrence counts of these k-mers become a vector representation of the sequence data. Alfpy allows several distance functions (e.g. Euclidian). In our implementation, $k = 6$ in combination with Google distance is used, which performed well in [ZGB+19]. A downside to the k-mer distance method is that it is computationally intensive - especially for higher values of k - and thus much slower to calculate than the w-metric.

The implementation makes use of a matrix containing all the pairwise distances for sequences in UNITE. To save the time of recalculating the entire distance matrix ($\approx 42,000 \times 42,000$) for every run of the algorithm, the matrix is calculated once and exported to a binary file. We calculated distance matrices for the w-metric and 6-mer Google distance, which completed in approximately 7 hours and 36 hours, respectively. Loading the matrix takes about 25 seconds. For reproducability purposes, our program can also exported/load matrices that were stored as a flattened .csv file, which takes into account symmetry along the diagonal to reduce storage. The .csv versions of the calculated distance matrices can be found here [10.5281/zenodo.6799940].

2.3 Hardware

We were provided access to MDDB's server by LIACS, which runs on an Intel(R) Xeon(R) x5355 CPU (8 cores) with 32GB memory. Most experiments were performed on this server. An important note is that the implementation was developed while keeping in mind a possible future migration to LIACS Life Science Cluster (LLSC). A divide-and-conquer approach is well suitable for distributed computing, which LLSC could offer.

3 Implementation

3.1 Overview

A high-level overview of the implemented pipeline is shown in Figure 8. The pipeline is split up into two parts: backbone creation and backbone expansion. In this section, we will explain each algorithm (sub)component shown in Figure 8 in its own subsection. We first provide an overview of both of the two parts. The algorithms have been implemented in Python, the code of which is

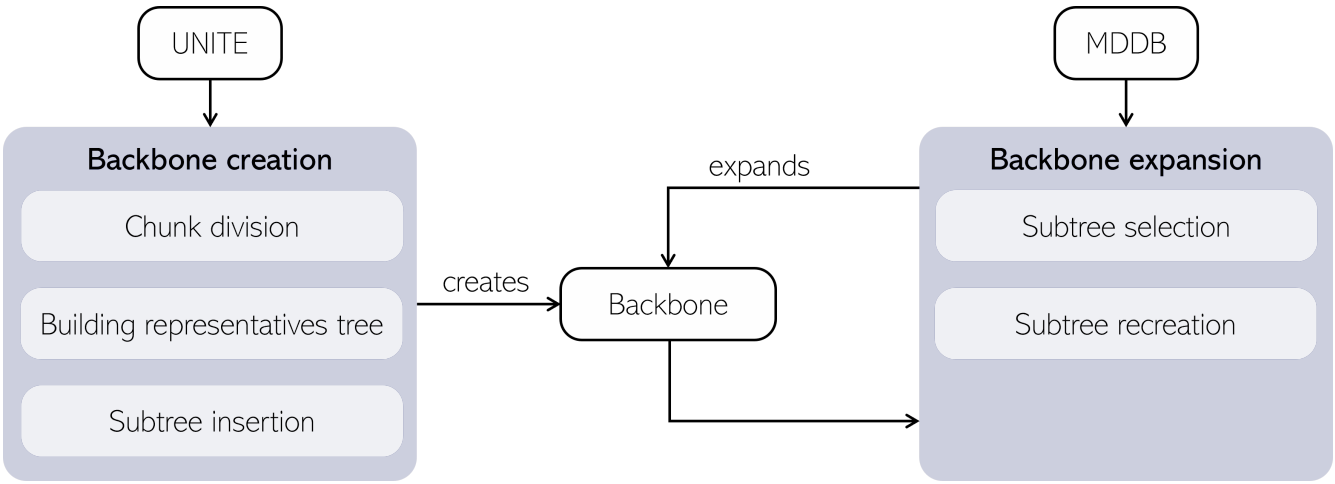

Figure 8: An overview of the pipeline implementation. A backbone phylogeny is created using UNITE data. This backbone gets expanded using data from MDDB.

available on Github[3].

Figure 9 is a graphical representation of the backbone creation algorithm. First, UNITE's data is divided into chunks based on taxonomic rank. These chunks go through some filtering steps. Then, a high-level tree is built by using sequences that serve as examplars for their chunks, we call these the chunk representatives. The representatives tree is initially used to identify an outgroup for each chunk, after which the chunks are ready to undergo alignment and tree building using MAFFT and RAxML respectively. Finally, the chunk representatives are replaced by the actual subtrees, resulting in the UNITE backbone phylogeny.

Figure 11 is a graphical represenation of the backbone expansion algorithm. First, the closest representative and then closest overall sequences are calculated by using 6-mer in combination with Google distance. These closest sequences are used to determine the most recent common ancestor and thus the subtree the new sequence belongs. Once sorted on depth and outgroup determined, the subtrees can be recreated from deepest to closest to the root. Replacing the old subtrees with the new updated subtree, makes the backbone expanded with the new sequence(s).

3.2 Backbone creation

The backbone creation part of the algorithm has several parameters that can be set to own preference. This subsection introduces the algorithm components and parameters one by one. Finally, we will provide an overview of the parameters and give instructions on how to run the pipeline.

3.2.1 Chunk division

Dividing the data into chunks is a crucial part of the implementation, as a good division is necessary to generate a reliable backbone within a reasonable amount of time. In the implementation,

[3] https://github.com/luukromeijn/MDDB-phylogeny

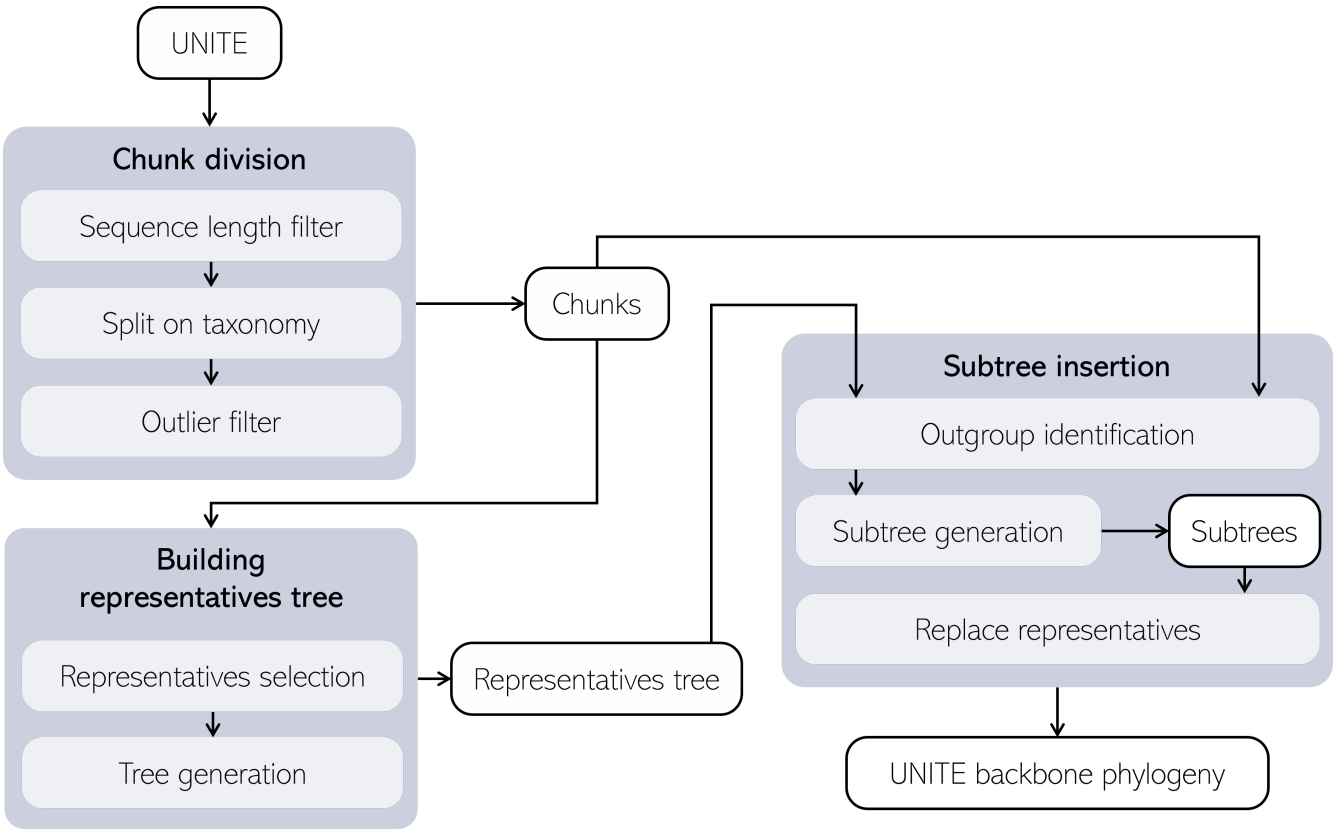

Figure 9: A graphical representation of the backbone creation algorithm.

`Chunks.py` contains most of the classes and methods involved in this process.

The `UniteData` class in `Chunks.py` was created to read and hold the data corresponding to a UNITE release. It parses the taxonomy file, creating a nested dictionary object. Each nesting level corresponds to a taxonomy rank, with at its core a list of SHs corresponding to that taxonomy. The dictionary has been converted to a file in JSON format and is stored here [File].

After having imported the UNITE data, the sequences can be grouped into chunks based on their taxonomy. This generates instances of the `Chunk` class. We define splitting level $1 \leq d \leq 6, d \in \mathbb{N}$ as a taxonomy rank, in which $d = 1$ corresponds to splitting at phylum level, $d = 6$ corresponds to splitting at species level, and all values in between correspond to the remaining taxonomy levels in order. All chunks will be split at least at depth d_{min}. If the number of sequences in the resulting chunk exceeds $size_{max}$, we continue to split further until d_{max}. By default, $size_{max} = \infty$. Setting $d_{min} = d_{max}$ results in a simple split at depth d_{min}. The recursive function is described in pseudocode in Algorithm 1.

3.2.2 Data filtering

Four data filtering steps occur within the algorithm. When `UniteData` loads in the data of a UNITE release, it discards sequences with a length that deviates more than a length tolerance factor l from the average length of the sequences in the data. This was done for the sake of increasing the accuracy of the distance measure that will be used. This distance measure is based on k-mer frequencies and is therefore inaccurate for sequences with a divergent length. To get an insight into

Algorithm 1 Splitting UNITE data based on taxonomy.

Input: Nested tree t with SH entries as leaves, depths $d_{min}, d_{max} \in \mathbb{N}$ with $1 \leq d_{min} \leq d_{max} \leq 6$, and maximal chunk size $size_{max} \in \mathbb{R}$

```
1: function SPLIT(t, d_min, d_max, size_max)
2:     chunks ← ∅
3:     for child node n in t do
4:         if rank(n) = 'unidentified' then
5:             continue
6:         end if
7:         chunk ← RETRIEVE_SHS(n)
8:         if d_min > 0 then
9:             chunks ← chunks + SPLIT(t[r], d_min − 1, d_max − 1, size_max)
10:        else
11:            if |chunks| > size_max && d_max > 0 then
12:                chunks ← chunks + SPLIT(t[r], d_min − 1, d_max − 1, size_max)
13:            else
14:                chunks ← chunks + chunk
15:            end if
16:        end if
17:    end for
18:    return chunks
19: end function
```

the lengths of the sequences, refer to Figure 6.

A second filtering step occurs within Algorithm 1 (line 4-5). Sequences for which the taxonomy at splitting level is unidentified are discarded. Note that these sequences still have a high potential to be accurately positioned in the backbone tree using the backbone expansion algorithm.

After being divided into chunks, outlier detection is applied as a filter step. Given a strictness level $o \in \mathbb{R}^+$, we define a sequence i to be an outlier if and only if $\text{avg}(d(i,j)) < o \times \text{std}(d(k,j)) + \text{avg}(d(k,j))$ for all sequences j, k in the matrix. Outliers are discarded from the chunks.

Finally, some chunks will contain less than 3 sequences. These chunks are discarded, since no proper tree can be generated from them. Each batch of discarded sequences is exported in a separate FASTA file. In doing so, we can distinguish those sequences that may still be accurately placed in the backbone (e.g. unidentified sequences) from those that will not (too short/long sequences).

3.2.3 Representatives tree

A high-level tree containing two representatives for each chunk is used to graft the subtrees together. The idea behind this is that these two examplar sequences should comprise a fork-like clade in the tree. The root of this fork can then be replaced by the corresponding subtree. However, the high evolutionary distance between the sequences makes generating such a high-level tree challenging. We propose and compare two different distance-based methods for selecting the representatives. The `PairwiseDistance` class in `DistanceData.py` takes care of this, while the `RepresentativesTree`

class in `Supertree.py` contains methods for initializing the alignment and tree generation.

The first method $a = 0$ is based on which sequences best represent their chunks in terms of distance, i.e. have the lowest average distance to the other sequences in the chunk. For a sequence $i \in chunk$, the average distance $d_{avg}(i, chunk)$ is described in Equation 1. The representatives are the 2 sequences i for which this value is the lowest.

$$d_{avg}(i, chunk) = \frac{\sum_{j \in chunk} d(i, j)}{|chunk|} \tag{1}$$

The alternative representative selection method $a = 1$ selects sequences i and j that have the highest distance to each other, compared to all other pairwise distances within the chunk. This is described in Equation 2.

$$\arg\max_{i,j}\{d(i, j)\} \tag{2}$$

An advantage of this method is that since these two have the highest evolutionary distance within the chunk, the branch length from the root of the high-level tree to the root of the representatives fork will be shorter than with the first method. This shorter branch length is a more accurate representation of the evolutionary distance to the chunk root. A disadvantage of this method is the extra level of divergence in the alignment now that the representatives per chunk are more unlike each other. This results in a higher number of non-matching forks, making more constraints necessary.

Both representative selection methods are imperfect and need constraints to enforce all the examplar sequences to form forks. It was confirmed (See: Section 4) that method 1 causes most representatives to stick together and thus only a limited number of constraints is necessary. Yet by default, the algorithm was implemented to always constrain all examplar forks (using the `-g` grouping constraint in RAxML). This was done to prevent a constrain on fork x to cause the separation of a fork y. Keeping in mind that most of the forks match up anyways, this was regarded a safe decision. Our algorithm can also perform a *full constraint*: forcing all taxonomic ranks above the chunk splitting level to stick together. While this introduces even more constraints to the tree generation algorithm, we already assumed that the taxonomy up to the splitting level of the chunks was correct. On top of that, it had been confirmed that higher taxonomic levels like phyla grouped together well and mismatches were rare exceptions.

Parsing through the representatives tree allows for easy outgroup identification. An alternative approach to this is using the distance matrix. An outgroup then equals a sequence $i \notin chunk$ for which the average distance to the ingroup is lower than for all sequences $j \notin chunk$. This approach was found to cause less accurate trees, but is still used in the implementation as a backup for when finding an outgroup in the representatives tree fails (when the subtree is an outgroup itself).

3.2.4 Backbone generation

The representatives tree and all chunks are aligned using MAFFT with default settings. A parameter was added to optionally enforce the use of algorithm l-INS-i, which has been shown to be more accurate but slower than the default FFT-NS-2 algorithm [NWT06]. Trees are generated from

Symbol	Code reference	Range	Description
l	`length_tolerance`	$\mathbb{R}$	Allowed factor of deviation compared to the average sequence length in UNITE.
d_{min}	`min_split_depth`	$[1, 6]$	The data is at least split up at this taxonomy rank.
d_{max}	`max_split_depth`	$[1, 6]$	The data will not be splitted beyond this taxonomy rank.
$size_{max}$	`max_chunk_size`	$\mathbb{R}$	Size threshold for which the algorithm will split chunks further (if allowed by d_{max}).
o	`outlier_strictness`	$\mathbb{R}^+$	Number of standard deviations that a sequence's average distance to its chunk must be lower than the average pairwise distance in the entire dataset.
a	`representatives_alg`	$\{0, 1\}$	Determines which function is used to determine the chunk representatives (Section 3.2.3).
Full constraint	`full_constraint`	$\{0, 1\}$	Constrains all taxonomic ranks in the representatives tree if set to true. Only constrains the representatives to fork if false.
l-INS-i	`localpair`	$\{0, 1\}$	If true, forces the use of the l-INS-i MAFFT algorithm instead of the default FFT-NS-2.

Table 1: Parameters of the backbone creation part of the pipeline. The order in the table corresponds to the required input order when calling `run.py`.

these alignments by RAxML. For the representatives tree, the `-g` grouping constraint is used to constrain the representatives to form forks. The GTRCAT substitution model is used. An outgroup parameter is provided to the algorithm with the sequence that was identified as outgroup. The resulting representatives tree is rooted with midpoint rooting, picking the node between the two most distant sequences in the tree as root.

The entire backbone creation algorithm can be run by calling the `run.py` script, which will fetch the UNITE data from the data folder and create an instance of the `Backbone` class from `Supertree.py`. The program makes use of several previously-mentioned parameters, which are listed in Table 1. The values must be provided in the command line after `run.py` in the order of the table. This was done since hard-coded values for these variables would be based on largely unsubstantiated decisions and would require a future user to look deep into the code. On top of that, it allows for easy experimentation.

Calling `run.py` will automatically create a result folder, named after the used parameter values. The structure of this folder is shown in Figure 10. It is also possible to call the involved classes and methods outside of `run.py`, but all these classes assume the presence of a directory that is structured as shown in Figure 10.

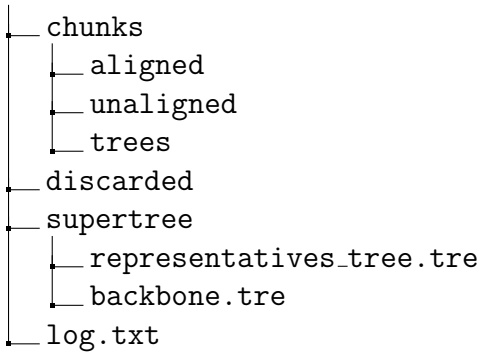

Figure 10: Structure of result directory. All subdirectories and the most important files are displayed.

3.3 Backbone expansion

As mentioned in section 1.1.1, still a lot of fungal species are yet to be discovered and identified. This means that the generated backbone needs to be able to expand when new species get discovered and identified. But, rebuilding the entire tree with all currently known species is very inefficient. Therefore, we introduce a method which only rebuilds parts of the tree and places those back in the tree. A graphical representation of this algorithm is shown in Figure 11.

Before being able to determine what part of the tree needs to be rebuild with the new data, some initialization steps need to be done. All necessary fasta files need to be read into separate so called seqrecord variables from alfpy, which will make it easier to handle and compare the data. This includes the file with all sequences in the backbone tree, all chunk files, the representatives file and the file with the new sequences. On top of the fasta files, the backbone tree also needs to be read into a variable. After this, we can start with the subtree selection part of the algorithm.

3.3.1 Subtree selection

Determining where the newly introduced data belongs in the backbone tree goes in steps per one new sequence. Assuming the sequences do not exist in the tree yest, for each sequence, we start with calculating the distance to the sequences that are in the tree. This method is described in pseudocode in Algorithm 2. Since working with a tree of around 16.000 sequences, it takes too much time and memory to calculate the distances to all sequences. Therefore, we start with calculating the distances to only the representatives of all chunks. Based on a range of smallest distances, we select the representatives that will be used in the next step. When a fixed amount is used, it might be the case that a sequence that is almost as close as the last selected one, gets ignored. To preserve this, the upper-bound of this range is determined by the smallest distance found, *closest*, plus a percentage, $margin\_rep$, of this smallest distance. In case of the closest representative being at $closest = 0.5$ and the percentage at $margin\_rep = 0.1$, we select the representatives with a distance score between 0.5&0.55 for the next step.

The next step is to determine a more specific region where the new sequence belongs in the tree. To do this, we need to calculate the distances to more sequences than just the representatives. Therefor, we need to determine what chunks the representatives we selected in the previous step belong to. These chunks are saved with a three digit code for simple distinction. The sequence id's

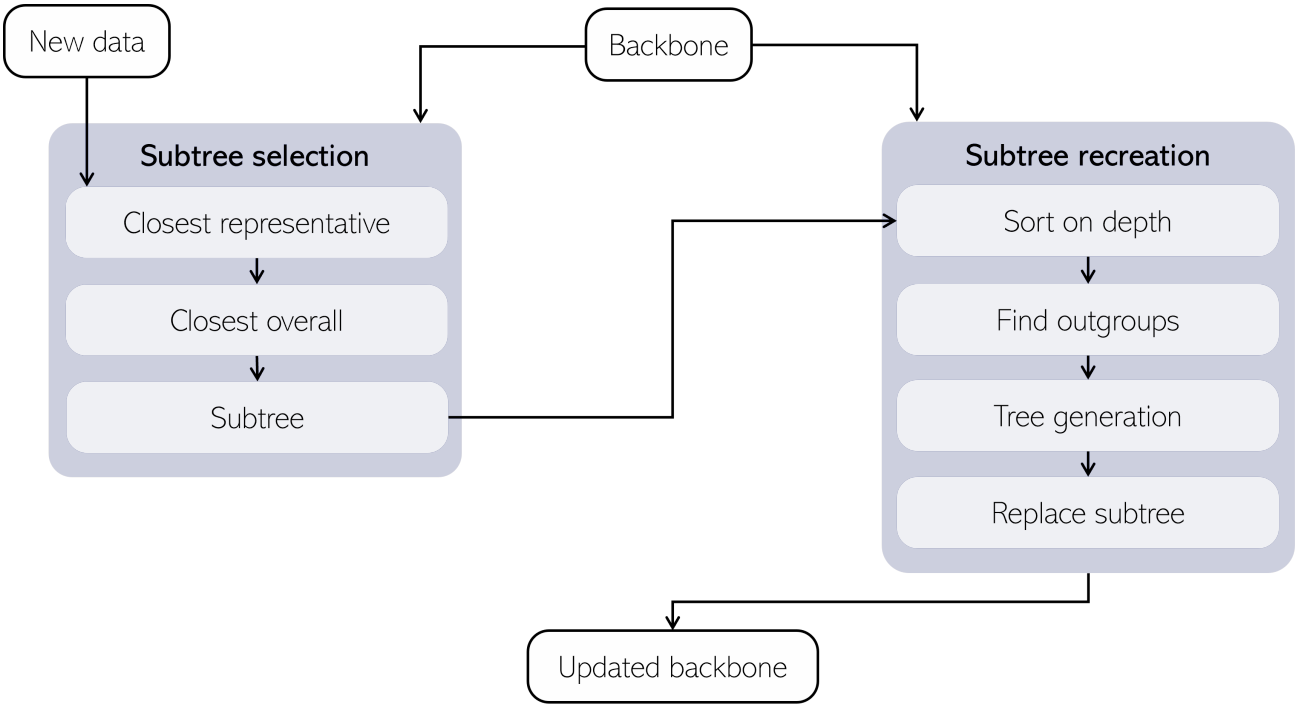

Figure 11: A graphical representation of the backbone creation algorithm.

Algorithm 2 Calculating distance between new and old sequences

Input: seqrecords variables *new_sequences* and *known_sequences* and the *margin* percentage

```
for sequence new_seq in new_sequences do
    if new_seq not in known_sequences then
        distances ← ∅
        for sequence known_seq in known_sequences do
            distance ← PAIRWISE_DISTANCE(new_seq, known_seq)
            distances ← distances + distance
        end for
        best_distances ← ∅
        closest ← MIN(best_distances)
        sorted_distances ← SORT(distances)
        for sorted in sorted_distances do
            if sorted < closest * (1 + margin) then
                best_distances ← best_distances + sorted
            end if
        end for
    end if
end for
```

from the representatives also contain these codes, the same as their corresponding chunk. By using a dictionary when reading the chunks into the seqrecord variables, it is now simple to select the chunks that belong to the selected closest representatives. For all sequences in these few selected chunks, we again calculate the distance to the new sequence and select the range of closest sequences. This range is now determined with the *closest* distances plus the percentage $margin\_chunks$, which can be different than $margin\_rep$. We will experiment with these two percentages in Section 4.3.1.

The next step is to really select the subtree where the new sequence should belong according to the calculated distances. To do this, we select the most recent common ancestor (mrca) of all the closest sequences from the previous step. There exists a function from the Biopython package to select this mrca from the backbone tree for us. This function does not only return the mrca, but includes all information on the clade underneath this mrca. Which means that this is the subtree where the new sequence will be added to.

What step next to take, depends on if you want to enter the new sequences into the tree one by one or per groups as much as possible. If adding one by one, the next step would be recreating the selected subtree with the new sequence and then repeat all previous steps for all new sequences. When adding per groups, we now add the mrca new sequence id combination to a dictionary and repeat all the selection steps for all new sequences. This way, afterwards multiple new sequences could be stored under the same mrca key in the dictionary. Next, we can continue with recreating the subtrees under the mrca's with it's, possibly multiple, new sequences.

3.3.2 Subtree recreation

We start with the dictionary containing the mrca's and the new sequences that belong to that subtree. Unfortunately, it is not possible to just start recreating all subtrees. Because if unintentionally a subtree with a common ancestor closely to the root of the backbone gets recreated first, it changes a big part of the tree. But this means that the program can not find the common ancestors that were inside this subtree anymore. To prevent this from happening, we sort the common ancestors based on their depth in the tree.

Now it is possible to start recreating the subtrees one by one, going from the deepest to the one closest to the root. Starting with determining the outgroup by going back to the ancestor of the mrca and select two sequences from the subtree's neighbouring clade. Then create a constraint tree from the subtree plus it's outgroup, for later on as input for RAxML. Next, create the input for MAFFT, by writing the id and sequence of all leaves/childs from the subtree plus outgroup plus the new sequences to a fasta file. This and the following steps only happen, if the amount of sequences that would have to written to the fasta file is below a certain threshold, in our case 500 sequences. Otherwise the alignment can become less successful and rebuilding the tree will take too much time. Assuming the size of the subtree is small enough to continue, the aligned fasta output file can then be used as input for RAxML, which will build the subtree with the new sequences added. Lastly, the old subtree in the backbone will be replaced with the newly build tree. After having done this for all selected mrca's/subtrees, the backbone is expanded with the newly introduced sequences.

3.4 Tree quality measure

A tree quality heuristic is necessary to enable a structured comparison between the different parameter settings. For this, we (again) make use of UNITE's taxonomic identifications. Given a

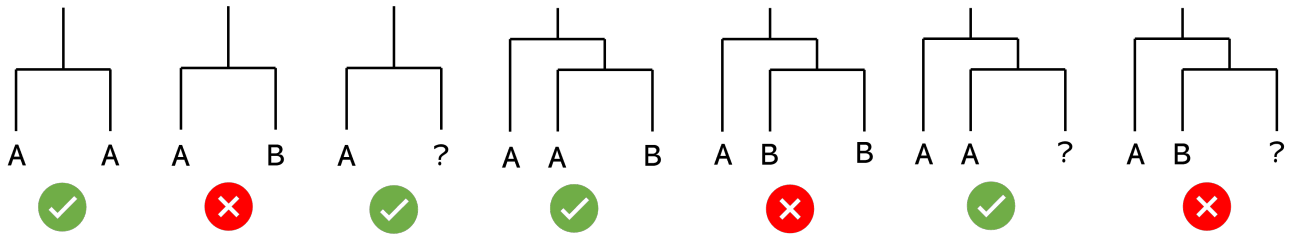

Figure 12: How the tree quality heuristic decides whether terminal leaves are matching (v) or non-matching (x) in different scenarios. A and B indicate two different taxonomic identities. ? indicates an unidentified identity.

tree consisting only of UNITE data, we can count the number of non-matching terminals/leaves. The fewer there are, the better we consider a tree to be. Figure 12 shows the possible scenarios that can occur while determining the number of non-matching terminals and whether our implementation regards these subtrees as matching or non-matching. Note the inclusion of possible unidentified sequences (in the figure represented with '?') and terminals whose sister nodes are preterminal. In any other scenarios that the algorithm finds, the count of non-matching leaves is not increased. The number of unique leaves is also reported. This has been implemented in the `Supertree.py`. Tree quality assessment can easily be performed using `inspect_tree.py`. The value of the proposed heuristic can be calculated at different resolutions, corresponding to each taxonomic rank.

We must point out some sidenotes to the quality measure. Firstly, a lower score is not necessarily better. This is due to possible faulty taxonomic identifications, or an uneven number of sequences comprising a taxonomic rank (e.g. if a single sequence is given a certain taxon, it will always be part of a non-matching terminal clade). Another sidenote here is that the heuristic does not capture the coherence of non-terminal leaves. For example, while the leaves of a tree may match together perfectly, taxons can still be highly scrambled on higher levels. Thus, the number of non-matching forks merely provides an indication of the quality of the tree, and is not a all-encompassing quality measure. Visual inspection remains a necessity.

4 Experiments & results

Several experiments were performed, generating different versions of the backbone tree. In this section these experiments will be explained and their results will be analysed.

4.1 Distance measures

To get an insight into the performance of both distance measures, an evaluation method was implemented. Based on the assumption that sequences within chunks should have a lower distance to each other than to the entire dataset, the evaluation method calculates the score as follows:

$$d_{avg}(seqs) = \frac{\sum_{i \in seqs} \sum_{j \in seqs} d(i,j)}{|seqs| \times |seqs|} \tag{3}$$

$$evaluate(d, chunks) = \frac{\sum_{chunk \in chunks} (d_{avg}(chunk) - d_{avg}(chunks))}{|chunks|} \tag{4}$$

As Equation 4 shows, the resulting score describes the average difference between distances within the chunks and the average distance within the entire matrix. This score is expessed in standard deviations and thus allows for comparing different metrics. The lower the score, the better the distance measure. An overview of the scores for the two different distance measures and different splitting levels can be found in Figure 13 and 14.

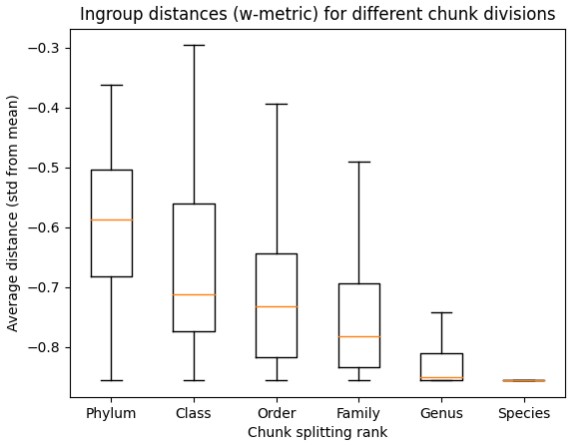

Figure 13: Average difference between w-metric distances within the chunks and the average w-metric distance within the entire matrix, for different chunk splitting levels, expressed in standard deviations.

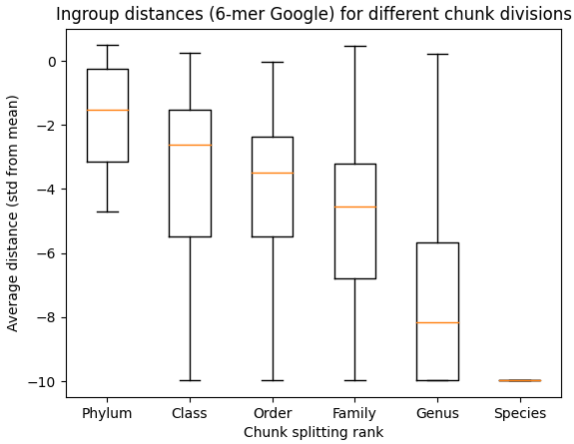

Figure 14: Average difference between k-mer Google distances within the chunks and the average k-mer Google distance within the entire matrix, for different chunk splitting levels, expressed in standard deviations.

The figures show that, as expected, the 6-mer Google distance measure has a much higher change in distances than w-metric does. The average 6-mer distance within chunks that are split on order is more than 3 standard deviations lower than the average distance within the entire matrix. This not only shows that the 6-mer Google distance is capable of discriminating similar sequences from non-similar sequences, but also supports our assumption of being able to make a responsible splitting decision by using the data's taxonomy.

The performance difference between these two distance measure was also noticed outside of this experiments. As mentioned in Section 3.2.3, pairwise distances can be used for outgroup selection. We observed (by looking at alignments) that using 6-mer Google distance would lead to the selection of outgroups that were more similar to the ingroup than when using w-metric.

4.2 Backbone creation

The experiments for the backbone creation part of the pipeline were generated using the `run.py` script with different parameter settings. These settings and their corresponding results are listed in Table 8. Furthermore, Table 9 gives a short description corresponding to the entries in Table 8, along with a link to the backbone tree files on Github. We will use $l = 0.2, d_{min} = 3, d_{max} = 4, size_{max} = 1500, o = 2.0$ as a baseline. Note that this baseline is conservative (strict filters), reducing computation time. This allowed us to generate 11 trees for experimentation.

Settings	# chunks		Chunk size							
			pre-filter				post-filter			
	pre	post	avg	std	min	max	avg	std	min	max
$d_{min} = d_{max} = 3$	247	180	109.5	349.9	1	3880	78.7	232.3	3	2241
$d_{min} = d_{max} = 4$	696	492	34.8	114.8	1	2251	38.4	124.1	3	2241
$d_{min} = 3, d_{max} = 4,$ $\text{size}_{max} = 1500$	288	215	93.0	235.7	1	2251	74.8	208.3	3	2241

Table 2: The influence of different chunk division settings on the number and size of chunks. Before and after applying data filters (length filter, outlier removal, small chunk filter). Row 1 corresponds to splitting on order, the second to splitting on family, the third to a hybrid approach (extra split only for big chunks). Further settings: $l = 0.2$, $s = 2.0$, others: 0.

4.2.1 Chunk division

To investigate the effect of which splitting level to choose (see: Assumption 1 and Section 3.2.1), we compare: i) splitting on order ($d_{min} = d_{max} = 3$); ii) splitting on family ($d_{min} = d_{max} = 3$); and iii) splitting on order but making an extra split on family for those chunks of which the size exceeds $size_{max} = 1500$. In Table 8, these correspond to row 2, 3, and 1, respectively. Table 2 lists even more statistics, describing the effect of the different parameter settings on the number of chunks and their size. From here on, we will refer to the appraoch with extra split based on $size_{max}$ as the *hybrid* setting.

The average size of the chunks decreases when we split on family instead of order. This decrease is also observed with the hybrid setting, but here the decrease is slightly less big. The standard deviation of the chunk sizes for the hybrid split makes a more notable decrease. This is expected, as big chunks of sequences have been splitted up further into their families. Note that the big order-level chunk with size 3880 is splitted up in the other two settings. Still, there is a chunk of 2251 sequences present, which is a value higher than $size_{max} = 2251$. This is because the *Aspergillaceae* family consists of 2251 sequences. As this is family-level, the chunk will not be split up further.

While the average chunk size decreases, the total number of sequences increases when splitting on family (seen in Table 8) instead of order. This is because outlier removal happens after chunk division and is based on average distance to the ingroup. Order-level chunks contain sequences that are more different to each other than those in family-level chunks. This is shown in Figure 14. Consequently, more of them will be discarded. While having more sequences, splitting on family leads to a 42 minutes quicker backbone generation than splitting on family, due to the fact that chunk sizes are lower.

Remarkably, Table 8 shows that splitting on family leads to discarding three phyla. This is caused by the fact that family-level chunks of these phyla are too small to make up a tree. One of the phyla that gets discarded is *GS*01, which is a rather small but literature-supported [TBP+17] phylum. Whether discarding these phyla is problematic depends on how accurately they can be added in the backbone expansion part of the algorithm.

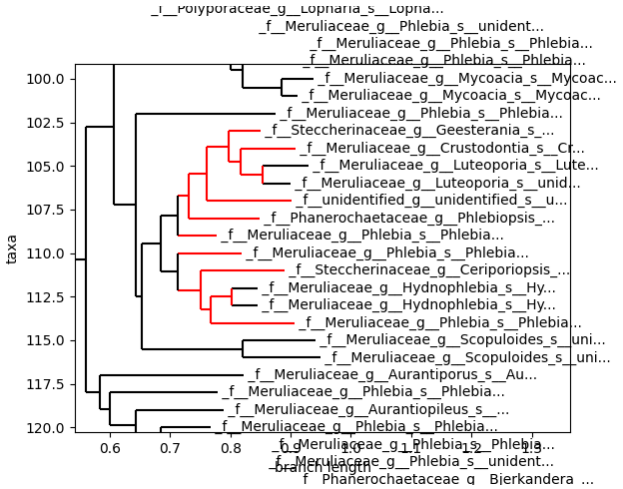

Figure 15: Messy family relationships occur when splitting on order ($d_{min} = d_{max} = 3$), as the *Steccherinaceae* family pops up at different places in the tree. SHs have been replaced with taxonomies. The red branches provide an indication of where to look.

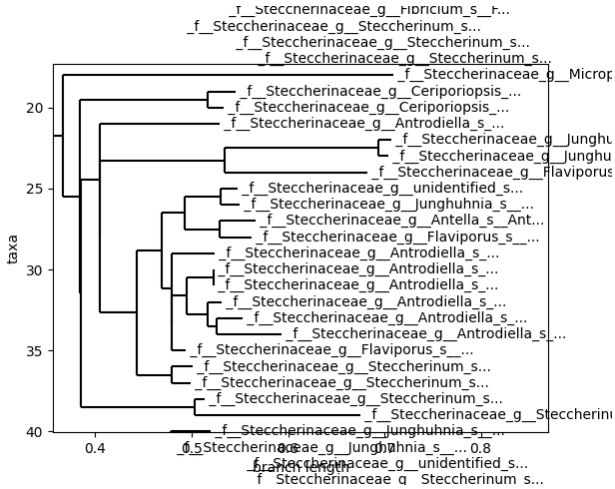

Figure 16: A part of the *Steccherinaceae* subtree, formed when splitting on family or using the hybrid approach with $size_{max} = 1500$. Family relationships are constrained, but some inconsistencies in genuses can still be observed (e.g. in *Flaviporus*)

Splitting on order leads to quite a high number of non-matching leaves, even at family level (434). This is a serious problem, as taxonomists will likely consider the phylogenetic tree to be incorrect. Figure 15 illustrates this problem, as the *Steccherinaceae*, *Phanerochaetaceae*, and *Meruliaceae* families are somewhat scrambled. Upon inspection of the backbone, such messy family relationships seemed to occur most often in big subtrees. Splitting on family can solve this problem, e.g. Figure 16 shows the *Steccherinaceae* family. Nonetheless, a close observer can still identify some oddly placed genuses (e.g. *Flaviporus*).

Non-matching leaves also occur - in much higher quantities - at species level. Splitting on family leads to the highest number (4723), but also leads to the highest number of unique species in the tree (8895, compared to 5876 when splitting at order level). Genuses seem to be a bit better preserved when splitting on family.

Thus, it has been shown that making an extra split on family slightly improves the backbone's quality. Some of the order-level chunks seem to be too big and diverging, which is why an extra split is recommended. This can also be achieved with the hybrid approach. The other side of the coin here is that dividing chunks based on family is a bigger assumption than dividing them based on order.

4.2.2 Data filtering

Outlier removal is necessary in the backbone creation algorithm to make Assumption 1 more robust, as not all taxonomic identifications in UNITE are correct. Parameter o controls the strictness of outlier removal, filtering out each sequence with a diverging average distance to the rest of the chunk (see: Section 3.2.2). Length tolerance factor l controls discarding based on a diverging length. In this section, the effect of these parameters will be investigated by looking at the chunk of the

l	o	# discarded caused by	
		length	outliers
1000	0.0	0	2511
1000	2.0	0	21598
0.2	0.5	11984	1658
0.2	1.0	11984	3467
0.2	2.0	11984	10627

Table 3: Number of sequences that get discarded and the reason for their discarding. Using $d_{min} = 3, d_{max} = 4, size_{max} = 1500$.

Tremellales order.

Figure 17 shows what happens when filtering is not applied at all: an alignment with many gaps. Some of the sequences seem to not fit in. These sequences are targeted to be removed from the alignment with outlier removal. Since $o \geq 0$, sequences that have a higher distance to their ingroup than the average distance in the matrix are always removed, regardless of o. Thus, there is always some level of outlier removal present in any parameter setting of the algorithm.

The necessity of length filtering is also reflected in the results. Without applying a length filter ($l = 1000$), 21598 sequences are considered to be outliers (for $o = 2.0$) and are thus discarded. In the case of the *Tremellales* chunk, this causes it to be discarded entirely, which is an undesired situation. The *Tremellales* chunk is not just an exception. As Table 8 (row 8) shows, the number of unique families, genuses, and species is lower without the use of a length filter. This indicates - as expected - that outlier removal without a length filter causes it to be inaccurate. When we use the same value for o, but apply a length filter $l = 0.2$, *Tremellales* is not entirely discarded, only outliers have been removed. This improved the quality of the alignment, as seen in Figure 20. Moreover, now that the outlier removal step has become more accuate, such a high value for o may not be necessary anymore.

Figures 18, 19, 20, it is shown what happens when an increasing amount of filtering is applied. As o increases, the number of gaps decreases and the sequences seem to align better. However, the fact that the alignment improved is not noticeable in the number of non-matching leaves relative to the number of unique ranks, shown in Table 8. A small but noticeable improvement when using $o = 2.0$ compared to $o = 0.0$ is seen in the tree of the *Tremellales* chunk (Figures 21 and 20).

So while length filtering has been proven to be a necessity, setting the value of o too high may lead to discarding many sequences without improving much upon the tree's quality. Table 8 shows that the number of unqiue species is higher for lower values of o. Thus, determining the ideal parameter setting is about balancing the number of included species with the number of introduced inconsistencies in the backbone.

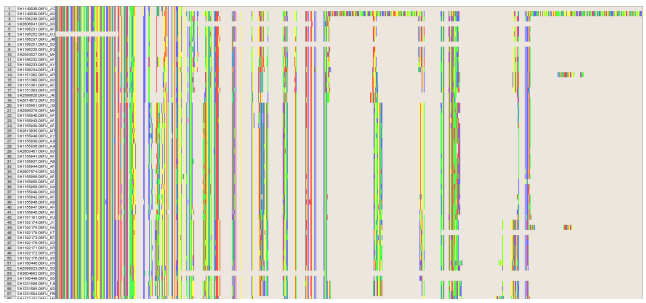

Figure 17: Alignment of *Tremellales* chunk, no filtering applied.

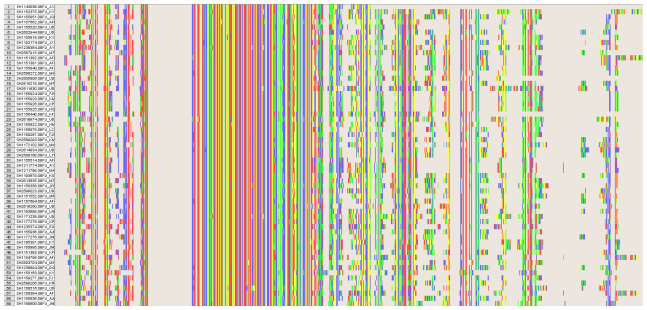

Figure 18: Alignment of *Tremellales* chunk, $l = 1000, o = 0.0$.

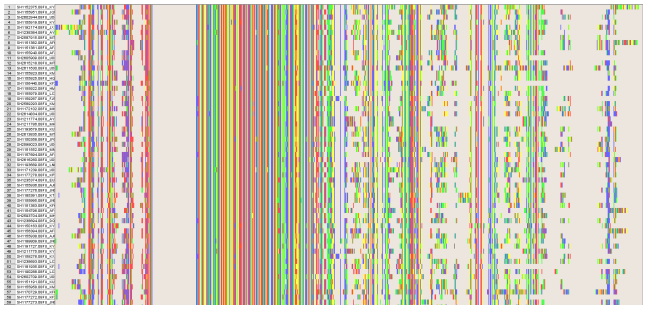

Figure 19: Alignment of *Tremellales* chunk, $l = 0.2, o = 0.5$.

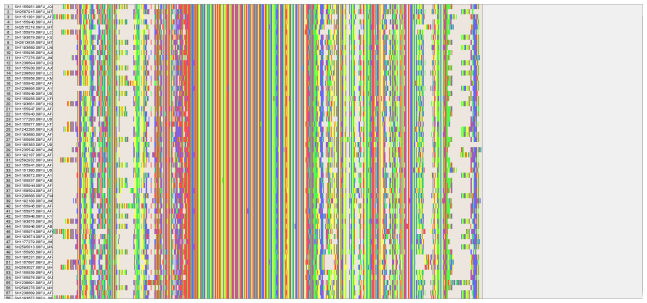

Figure 20: Alignment of *Tremellales* chunk, $l = 0.2, o = 2.0$.

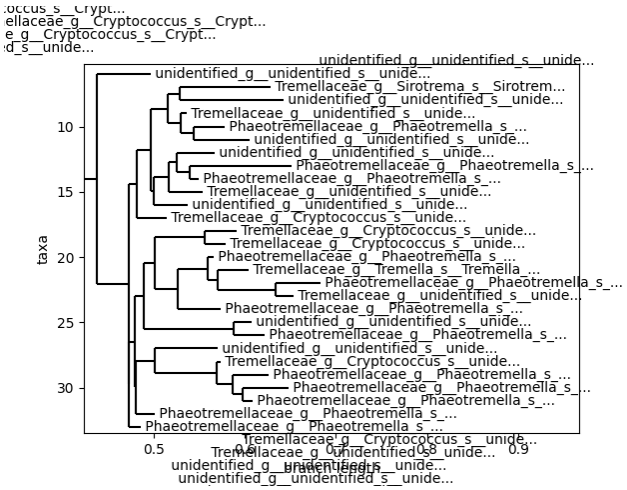

Figure 21: *Tremellales* chunk subtree (zoomed in), $l = 1000, o = 0.0$. The *Tremellaceae* and *Phaotremellaceae* families have been mixed up.

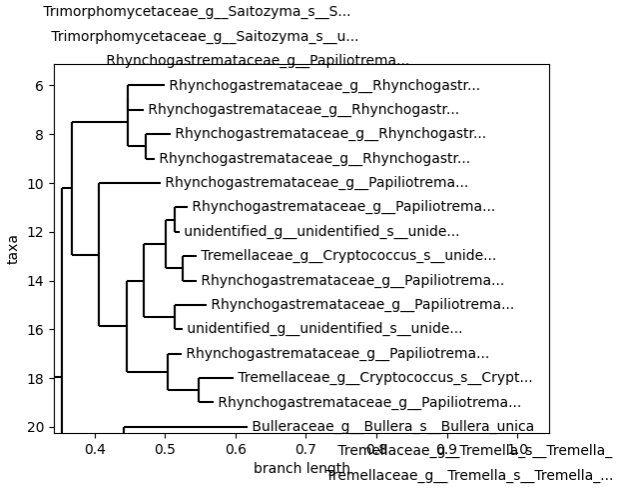

Figure 22: *Tremellales* chunk subtree (zoomed in), $l = 0.2, o = 2.0$. Families are better preserved but inconsistencies still occur.

a	Constraint	Distance to forks (avg)	Non-matching forks			
			Phylum	Class	Order	Family
0	None	0.81	0	1	4	13
	Forks	0.82	0	0	0	7
	Full	0.93	0	0	0	7
1	None	0.68	6	30	107	135
	Forks	0.65	0	0	0	41
	Full	0.82	0	0	0	41

Table 4: Comparison of two representative selection methods with different types of constraints. Using $l = 0.2$, $d_{min} = 3$, $d_{max} = 4$, $size_{max} = 1500$, and $o = 2.0$.

4.2.3 Representatives tree

Two representative selection algorithms were proposed in Section 3.2.3, either selecting the two most average sequences of a chunk ($a = 0$) or the two most distant sequences within a chunk ($a = 1$). For comparing these two methods, we do not need to generate entire backbone trees. Instead, we compare the representatives trees and determine the following values:

1. The number of non-matching forks. We determine these mismatches separately for each taxonomic level.

2. The average distance to the fork-clade.

The experiment was conducted using $l = 0.2$, $d_{min} = 3$, $d_{max} = 4$, $size_{max} = 1500$, and $o = 2.0$. In the implementation, representatives are constrained to stick together and form a fork. Here, we will also investigate to what extent this constraint is necessary and justifiable by looking at completely unconstrained trees. On top of that, the full constraint parameter will be investigated, which not only forces representatives to stick together, but also their higher taxonomic ranks.

Aligning ITS sequences that span the entire fungal kingdom is a challenging task, which is reflected in the alignment in Figure 23. The sequences in this alignment were chosen by representatives selection method $a = 0$. The alignment is full of gaps and mismatches are common. Still, rows of two sequences are observed, meaning that despite the seemingly messy alignment, the representatives align well with each other. This is also reflected in the tree: Table 4 shows that without any constraint, only a couple of mismatches occur when using $a = 0$. Coherence of higher taxonomic ranks cannot be concluded based on Table 4's data alone, but upon visual inspection of the tree it was confirmed that these ranks are well conserved (with some exceptions). An example of incorrect fork formation is shown in Figure 24.

Introducing the fork constraint completely removes non-matching forks at phylum, class, and order level. As mentioned above, Table 4's data was generated using the hybrid chunk division approach, only splitting on family for those chunks that would have been bigger than $size_{max} = 1500$. This means that only the chunks that underwent this extra split will be constrained at family level. This explains why there are still 7 non-matching forks even when constrained.

For $a = 1$, the number of non-matching forks at family level lies even higher. This is caused

by the alternative representatives selection method, which chooses the sequences most distant to each other. These two sequences are very likely to be part of different taxonomic families, explaining the increase.

Without any constraints, there is a high number of non-matching forks with this approach. Apparently, the tree generation algorithm is not able to produce a close-to-truth tree now that the representatives differ much from each other. However, the average distance to the fork clades is smaller compared to $a = 0$. As explained in 3.2.3, a problem with using $a = 0$ is too long branches. $a = 1$ was designed to mitigate this problem and the results indicate that this works. However, the high number of non-matching forks (when unconstrained) in $a = 1$ is a major downside to this alternative approach, as introducing constraints may therefore not be justifiable.

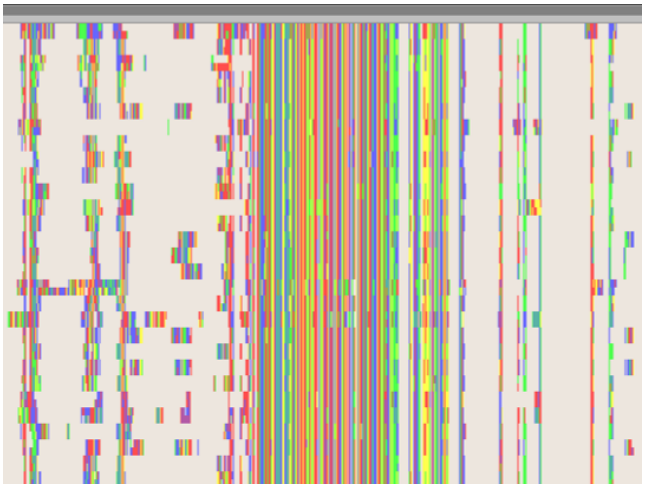

Figure 23: The alignment for the representatives tree ($a = 0$) has a high number of gaps and mismatches. Still, the representatives align together well and form rows of 2.

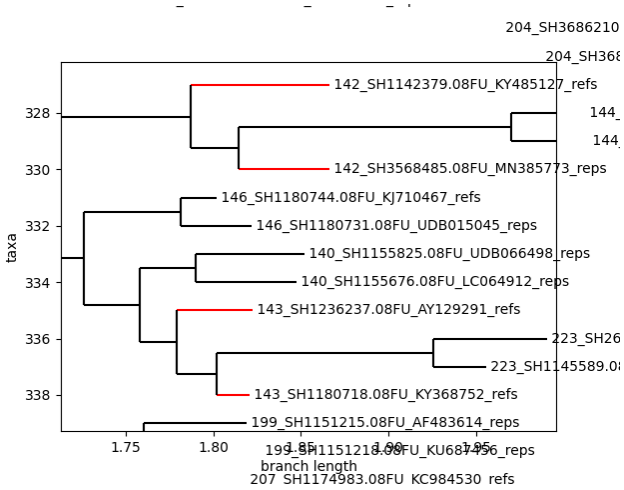

Figure 24: Not all forks in the representatives tree form well. The first three numbers in the leaves of this tree indicate the corresponding chunk. Representatives of chunk 142 and 143 do not fork.

4.2.4 Constraints

One of the parameters in the algorithm is the full constraint. This constraint may be desired, because high-level inconsistencies occasionally occur in a not fully constrained representatives tree. For example, Figure 25 shows that two *Basidiomycota* and *Ascomycota* sequences comprise a clade. While these inconsistencies are not picked up based on non-matching forks - after all their forks are fine - they are very wrong and would introduce enormous mistakes to our tree.

Figure 26 shows a constrained representatives tree, indicating the two biggest phyla. All taxonomic ranks up to the splitting level are preserved. A downside to this approach is the assumption that all these identifications are correct.

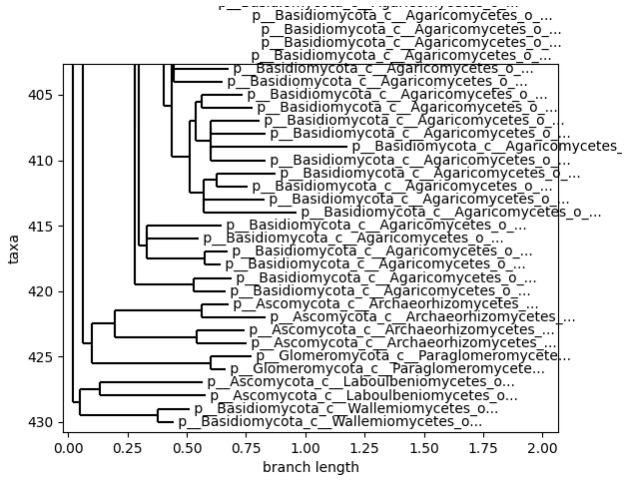

Figure 25: Inconsistencies occur in a not fully constrained representatives tree. In the bottom part of this figure, several phyla are scrambled.

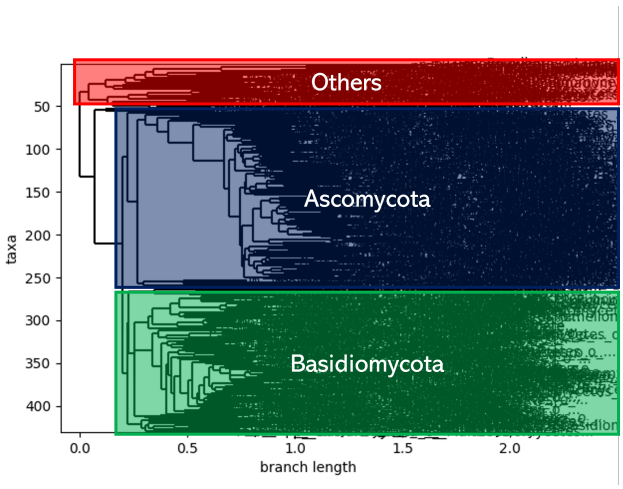

Figure 26: The phyla are preserved in a constrained representatives tree. The biggest phyla are *Ascomycota* and *Basidiomycota*.

4.2.5 l-INS-i

Table 8 shows that using the l-INS-i MAFFT algorithm leads to a slightly longer runtime but a lower number of non-matching leaves. This greater accuracy (and longer runtime) of l-INS-i is mentioned in literature as well [NWT06].

Inconsistencies on taxonomy levels like families and genuses make our tree less reliable - especially when these taxonomic relationships have been proven in multi-gene phylogenetic studies. For example, when asking taxonomist Jorinde Nuytinck at Naturalis to inspect the subtree corresponding to the *Russulales* order, she pointed out that within the genus clade of *Lactifluus*, some *Russula* and *Lactarius* sequences were present (Figure 27). Using l-INS-i instead of the default FFT-NS-2 enabled us to eliminate some of these problems (see Figure 28), but not all of them.

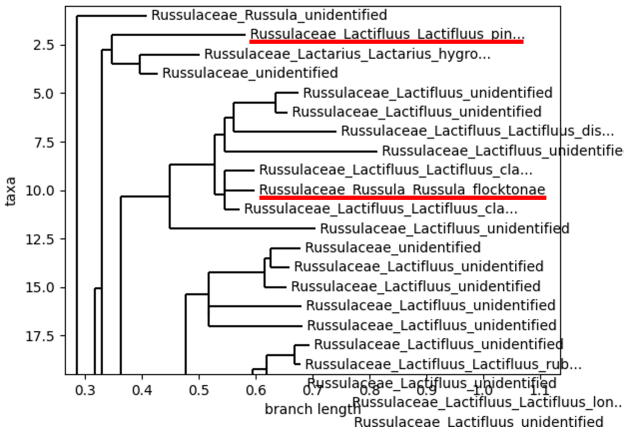

Figure 27: In the *Russulales* subtree, some *Russula* and *Lactarius* SHs were present within the clade of *Lactifluus*.

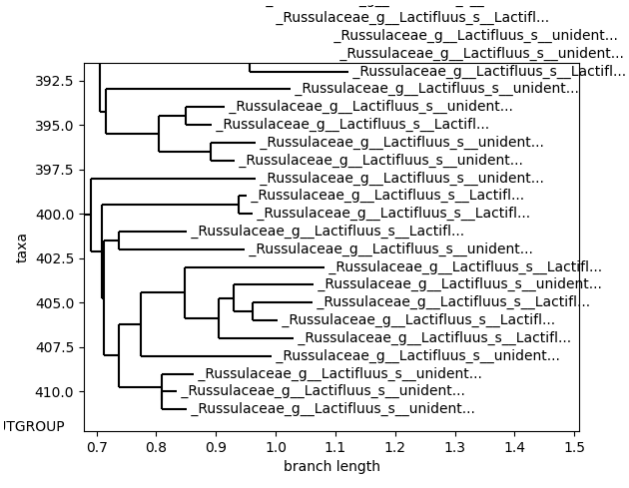

Figure 28: Using MAFFT algorithm l-INS-i resulted in a slightly better genus consistency.

margin_rep %	margin_chunks %	childs_amount	amount_in_range	distance_range
1	1	22,08	1,44	0,426 – 0,427
1	2,5	189,24	2,47	0,426 – 0,430
1	5	504,71	6,58	0,426 – 0,437
1	10	580,05	28,92	0,426 – 0,455
2,5	1	23,33	1,43	0,425 – 0,426
2,5	2,5	238,26	2,44	0,425 – 0,430
2,5	5	419,34	6,3	0,425 – 0,437
2,5	10	483,18	30,5	0,425 – 0,455
5	1	23,74	1,43	0,425 – 0,426
5	2,5	434,92	2,52	0,425 – 0,429
5	5	427,74	7,93	0,425 – 0,437
5	10	497,47	59,09	0,425 – 0,454
10	1	362,28	1,44	0,424 – 0,425
10	2,5	489,29	2,53	0,424 – 0,428
10	5	453,69	8,8	0,424 – 0,435
10	10	509,97	133,37	0,424 – 0,453

Table 5: Comparison of different combinations of *margin_rep* and *margin_chunks* percentages on the *distant*-discarded sequences. Where both the *childs_amount* and *amount_in_best_distance* increase with an increasing percentage, while the distance range remains almost the same.

4.3 Backbone expansion

4.3.1 Percentage margin

The two *margin* percentages used during the distance calculation in the subtree selection are of great influence of the subtree's size. To determine what works best, we experimented with all combinations of four percentages, 1, 2.5, 5 and 10. For each run, we calculate the average amount of sequences, from the chunks, that are within the distance range, calculate the average amount of childs inside the selected subtree and the average smallest distance. With these three values, we tried to find a balance between enough exploration to find the closest sequences in the chunks and not letting the subtree get to large. This experiment is run with the first 100 sequences from the *distant*-discarded sequences. The results are shown in Table 5.

What stands out first, are the increasing values of *childs_amount* and *amount_in_best_distance* when increasing the *margin_chunks* percentage. The fact that the *amount_in_best_distance* value gets higher makes sense, since the selection range gets wider. But a higher *amount_in_best_distance* value does not have to mean a higher *childs_amount* value, since the extra selected sequences could still be in the same subtree. However, the worst found sequence within the range, does get worse with a higher *margin_chunks* percentage. Which means that more sequences get selected that are further away and therefore the subtree increases in size.

The second thing that is important to mention, is the impact of the *margin_rep* percentage. With an increasing percentage, the smallest and worst found distance within the range seems to improve/decrease slightly. This would mean that exploring more chunks, thus more sequences,

Percentage combinations	Non-matching forks			
	Phylum	Class	Order	Family
2,5% - 1%	2	2	4	443
2,5% - 2,5%	0	0	0	438
5% - 1%	2	2	3	442

Table 6: Comparison of three *margin* percentage combinations

would improve finding the closest sequences. We however also see, that for the higher *margin_rep* percentages, the *childs_amount* value already starts increasing at a lower *margin_chunks* percentage. This would mean, that more often sequences from different chunks get selected and therefore results in larger subtrees.

We believe the best combination is either with the 2,5 or 5 percent *margin_rep*. We ignored the 1% because this does not explore enough chunks to really determine the best subtree. The 10% is ignored, because this results in too large subtrees from which the size exceeds the threshold to rebuild the trees. This leaves us with 2,5 and 5 percent, from both we also ignore the 5 and 10 percent *margin_chunks*, because the subtrees also become too large. This is also the case for the 2,5% *margin_chunks* in combination with 5% *margin_rep*. This leaves us with 2,5% in combination with either 1 or 2,5% or 5% with 1%. We decided to execute the tree quality check on these three combinations to determine which would probably work the best, see Table 6. From this we can conclude that the 2,5% - 2,5% combination works the best and will thus be used in further experiments.

4.3.2 Per group vs one-by-one

Runtime difference plus maybe tree quality measure

4.3.3 Add discarded sequences

Distant, smallchunks, short_or_long and unidentified

4.3.4 Add MDDB sequences

The last experiment we executed, was trying to add the data from MDDB to the generated backbone. Unfortunately this took significantly more time and also more skipped subtrees, because they were too large, than when adding the discarded sequences. So to check why there was this big difference, we did the same experiment as with the varying margin percentages in section 4.3.1, but now with the MDDB data, see Table 7.

There are two things that really stand out, the *childs_amount* and the *smallest_distance*. First of all, the higher *amount_in_best_distance* and *smallest_distance* values compared to Table 5. The higher smallest distance value means a wider range of closest sequences and therefore a higher *amount_in_best_distance* value. Secondly, the huge difference between the *childs_amount* with the *distant* data and the MDDB data. The value being higher than the threshold of 500 for rebuilding would explain why more subtrees are skipped. It also means that still large subtrees need to be rebuild, which explains why it takes more time to add these MDDB sequences to the backbone.

margin_rep %	margin_chunks %	childs_amount	amount_in_best_distance	smallest_distance
1	1	1321,05	3,44	0,71
2,5	1	1729,16	5,27	0,71
5	1	-	-	-
10	1	-	-	-

Table 7: Comparison of different combinations of *margin_rep* and *margin_chunks* percentages on the *distant*-discarded sequences. Where - means that there was a memory error when computing the distances, because of too many calculations.

The higher *smallest_distance* value, can indirectly also explain why adding these MDDB sequences to the backbone is not such a success. Because, it is weird that there are no sequences in the entire backbone of around 16.000 species that are relatively close to the MDDB data. The reason for this, is a problem we have already discussed in the previous section. The 6-mer in combination with Google distance calculation, has trouble with different lengths of sequences. Now are the MDDB sequences curated at a fixed length of 250 nucleotides long and as shown in Figure 6, the UNITE sequences are most of the time much longer. Which thus results in the problem that sequences seem very far away from each other, while in fact they might be very similar.

5 Conclusion

5.1 Research questions

During this research, our main goal was to create a phylogenetic tree for the fungal kingdom based on MDDB ITS data. To realise this, we split our approach into two main steps: the creation of a backbone phylogeny and the expansion of the backbone. During the development process, we experimented with lots of built backbones and new sequences that have been added. Of which some experiments have been shown in Section 4. After all these experiments , we have come to the conclusion that our approach is definitely possible, but is also still far from working optimal and perfect. In this section, we present our conclusions and discuss our results.

RQ1: How can a divide-and-conquer approach be applied for the construction of a phylogenetic backbone tree based on UNITE ITS data?

One of the most important aspects of the proposed divide-and-conquer approach is splitting the data up into chunks at a certain taxonomy splitting level, generating subtrees for each of those chunks, and then grafting those subtrees together. This approach only holds under the assumption that sequences with the same taxonomic identfications at splitting level should end up in the same clade of a phylogenetic tree.

The assumption was made more robust through outlier removal, which allowed us to successfully remove diverging sequences from the chunks that had been created. It has been shown that in order to do this accurately, a length filter with tolerance $l = 0.2$ must also be applied. To prevent the discarding of too many sequences while still guaranteeing a reasonable quality, we recommend using $o = 1.0$ as outlier strictness. The desired value for o may differ per user, as some are likely to

prefer a smaller but high quality tree, while others might prefer a tree that offers a broader range of unique taxa.

In our experiments, it was shown that splitting on family (opposed to order) improves the quality of the produced tree. However, this increases the chances of Assumption 1 being violated, due to faulty taxonomic identifications. We therefore recommend a hybrid split ($d_{min} = 3$, $d_{max} = 4$, and $size_{max} = 1500$), such that only big order-level chunks will be split up further. In case a user would prefer a full family split ($d_{min} = d_{max} = 4$), we recommend to set $o = 2.0$ for outlier removal.

A high-level representatives tree is generated in order to graft the subtrees together. Two alternative representative selection methods were compared. While the use of $a = 0$ likely causes inaccurate branch lengths, the number of necessary constraints is limited. This is in our view favourable. In some scenarios, one could need a scaled tree with accurate branch lengths, for which $a = 1$ can be used. Our experiments point out that using $a = 1$ leads to shorter branches, but demands for many more constraints.

Furthermore, we strongly encourage the use of the l-INS-i and *full constraint* parameters. Using l-INS-i has been confirmed to lead to better results but comes at the cost of extra computation time. The necessity of constraining all taxonomic ranks until splitting level was proven as well.

Based on the above conclusions, the algorithm is recommended to be run as follows:

`python src/run.py 0.2 3 4 1500 1.0 0 1 1` [TODO: do this experiment]

The resulting backbone is likely to contain inaccuracies, as the entire tree was built using ITS sequences only. Despite that, the number of non-matching sequences compared to the total sequence count is evidence that the tree is consistent to some degree.

RQ2: How can this backbone tree be expanded by efficiently determining the correct position when new data is introduced?

The most important aspect in expanding the backbone is selecting smaller subtrees where the new data belongs instead of rebuilding the entire backbone. Once the subtrees are selected for all new sequences, they can be expanded, possibly by multiple sequences when they belong in the same subtree.

When selecting the smaller subtrees, an alignment-free 6-mer in combination with Google distance calculation method is used to determine what sequences are closest to the new data. This works faster than a method that does make use of an alignment. However, this method also does not work optimally. When working with two sequences that are of very different length, they are seen as very distant to each other, while in fact they can be very similar on the part of the smaller sequence. This means that in certain cases the wrong subtree might be selected and thus the sequence is added in the wrong place to the subtree. We thought of three possible solutions to this problem. The best solution would be to use a method that can handle the difference in length, because then there would not have to be made any changes to the data. The second option would be to curate the UNITE data to the same fixed length as the MDDB data. But this means losing big parts of lots of sequences which could be of importance to the placement. Our last suggestion would be to make use of the MDDB data before curation, so at their original lengths.

The distances are first calculated to the representatives and next to the sequences in the closest chunks, from which then the subtrees are determined. To determine how many chunks have to be

explored, a range was used which was capped at the closest distance plus a *margin_rep* percentage. This way you prevent a cut off point where lots of sequences are close to each other, what will happen when using a fixed amount of closest sequences. Such a range was also used when determining the closest sequences from the selected chunks, only now with the *margin_chunks* percentage. The best combination of these two percentages was both at $2,5\%$. The smallest and average, within the range, found distances however, did improve when using higher percentages for both ranges. But we decided not to use these percentages, because they also meant much larger subtrees and would therefore take much more time.

Once the subtrees were determined, the expanding could really start. Unfortunately, some subtrees still became way too large to rebuild for only one new sequence, so we set a threshold at a maximal size of 500 leaves. This means that not even all new data gets added to the backbone by our algorithm. These sequences have to be added by hand afterwards or the algorithm should be given extra information on the subtrees these sequences belong in.

The distance calculation is split into the two separate parts, because otherwise a memory error would occur. This happens when too much calculations have to be done at once, but when split into smaller portions it's fine. Unfortunately, when entering large files with new data the memory error occurs anyway. This happens when trying to add more than 200 sequences in one go. So to solve this, the calculation method should be called multiple times with smaller portions. This is something that can be interesting to try if new data indeed comes in larger groups.

5.2 Discussion

A proper and formal tree assessment method for big phylogenetic trees was lacking throughout the development process. Luckily, we could use UNITE's taxonomic identifications for this. While counting non-matching leaves allowed us to get some insight into which settings work well and which do not, this quality measure is far from ideal, as explained in Section 3.4. Moreover, visually inspecting big trees is a time-consuming task. This is partially caused by the fact that tree visualization tools have difficulty with handling the big tree files. One alternative way for tree assessment would be calculating support values for branches of the (sub)trees, which RAxML does through bootstrapping. However, this would be an extremely time-consuming task for trees of our size. We were able to deduce whether our implementations worked or not, but a better tree quality measure could have made the development process more goal oriented and efficient.

A second point of discussion is the rooting of the tree. It was briefly mentioned in Section 3.2.4 that midpoint rooting of the representatives tree was applied. This worked in practice, but is not the most substantiated decision. Preferably, the tree would have been rooted by a non-fungi ITS sequence. UNITE contains such sequences as well, e.g. for animals. Since animals are the closest relatives of fungi, outgroup rooting was tried for three different animal ITS sequences. Unfortunately, none of these experiments were successful. An explanation for this is that the full spectrum of possible ITS sequences could already be present in the backbone, such that new sequences will always be placed somewhere nested in the tree, instead of being natural outgroups.

Perhaps one of the most notable results is that when using representatives selection method $a = 1$, not many constraints are needed to form a taxonomy-preserving phylogenetic tree. This is surprising given the high evolutionary distance between the sequences and the seemingly messy

alignment (Figure 23). Perhaps one of the lessons to learn here is that while some alignments may appear to be of low quality (i.e. containing many gaps and mismatches), they still may contain close-to-truth evolutionary information. Based on this, it is important not to tune parameter settings by solely looking at alignments. For example, the alignment shown in Figure 20 (with $o = 2.0$) looks much better than the one in Figure 19 (with $o = 0.5$), but this should not necessarily mean that $o = 2.0$ is optimal.

Throughout the research, we have seen varying performance levels of the 6-mer Google measure. On the one hand, Section 4 indicated that the distance measure was well capable of discriminating sequences within chunks from those outside. Also, its appliance in outlier removal became an important aspect of our implementation. On the other hand, its performance in the backbone expansion part was not always optimal, mainly because of its high sensitivity to differences in sequence lengths. As mentioned before, this is especially problematic for sequences from MDDB.

Another noteworthy mention is the discovery of identical sequences for different SHs (with slightly different taxonomic identifications) in UNITE. This hints at barcode sharing: two species sharing the same sequence for a barcode gene. Evolutionary speaking, this is possible. However, it presents an interesting challenge to the backbone expansion algorithm. What if two sequences from totally different parts of the tree are the same? This could lead to regenerating very big parts of the backbone when new similar sequences get added (the current implementation would skip such trees). Further investigating our backbone tree could lead to interesting insights about barcode sharing, either identifying a potential challenge, or assuring that such events rarely occured with ITS.

Finally, our produced ITS backbone trees should not be regarded as a replacement for multi-gene phylogenetic analyses. Reasons for this are that the backbone is not statistically supported, branch lengths may not be completely accurate, and inconsistencies have been shown to occur. Still, given the popularity of using ITS in biodiversity studies, the backbone can be an enrichment to MDDB and biological research in the future.

5.3 Future work

As mentioned in the discussion, there are still some flaws/uncertainties about our proposed method. But also interesting and useful additions to our method, which are interesting for future works. First of all, we mentioned the troubles of different lengths between the UNITE and MDDB data, where the MDDB data is curated to a fixed length of 250. It would be interesting to see the uncurated MDDB data, to see if these lengths come closer to the UNITE data and therefore might be better placed in our backbone.

Secondly, we mentioned the absence of a method that can control the quality of a tree. During this research we were not able to control if the trees build were correct or did not make sense at all. A method that can control the quality of a build tree, would help in deciding which settings work best when adding new data.

Lastly, it would be very useful to integrate our method into MDDB so the phylogenetic tree can be shown on the mycodiversity website. When doing this, another visualisation method is needed, because all leaf ID's overlap, which makes the tree unreadable unless zooming in. The leaf ID's also need to be named by one and the same method, because right now the data from UNITE makes

use of SH codes and MDDB of OTU codes. When all leaves are named by the same method, it will by easier to do an analysis on the tree. On top of that, when integrating to MDDB it would be very useful if the tree can expand automatically when new data is updated to the database. With this implementation, the newly added data is not saved to the fasta files, so when adding even newer data, these can not compare against the latest sequences. This would be necessary when integrating into MDDB. Lastly, the runtime of our algorithm leaves room for a lot of improvement, especially the recreation of the selected subtrees. We suggest the use of parallel computation on LLSC, which could recreate all subtrees at the same time instead of sequential.

References

[AD03] Jose L. Adrio and Arnold L. Demain. Fungal biotechnology. *International Microbiology*, 6(3):191–199, Sep 2003.

[ANL+10] Kessy Abarenkov, R. Henrik Nilsson, Karl-Henrik Larsson, Ian J. Alexander, Ursula Eberhardt, Susanne Erland, Klaus Høiland, et al. The UNITE database for molecular identification of fungi – recent updates and future perspectives. *New Phytologist*, 186(2):281–285, March 2010.

[AZP+21] Kessy Abarenkov, Allan Zirk, Timo Piirmann, Raivo Pöhönen, Filipp Ivanov, R. Henrik Nilsson, and Urmas Kõljalg. Unite qiime release for fungi, 2021.

[BP93] S L Baldauf and J D Palmer. Animals and fungi are each other's closest relatives: congruent evidence from multiple proteins. *Proceedings of the National Academy of Sciences*, 90(24):11558–11562, 1993.

[Bro02] Terence Austin Brown. *Genomes*. Wiley-Liss, 2002.

[Cho14] Supratim Choudhuri. Chapter 9 - phylogenetic analysis**the opinions expressed in this chapter are the author's own and they do not necessarily reflect the opinions of the fda, the dhhs, or the federal government. In Supratim Choudhuri, editor, *Bioinformatics for Beginners*, pages 209–218. Academic Press, Oxford, 2014.

[FHBJ18] Magdalena Frac, Silja E. Hannula, Marta Bełka, and Małgorzata Jedryczka. Fungal biodiversity and their role in soil health. *Frontiers in Microbiology*, 9, 2018.

[HLHJ17] David L. Hawksworth, Robert Lücking, Joseph Heitman, and Timothy Y. James. Fungal diversity revisited: 2.2 to 3.8 million species. *Microbiology Spectrum*, 5(4):5.4.10, 2017.

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[KE08] W John Kress and David L Erickson. DNA barcodes: genes, genomics, and bioinformatics. *Proc Natl Acad Sci U S A*, 105(8):2761–2762, February 2008.

[Kli09] Richard M. Kliman. Reading a phylogenetic tree: The meaning of monophyletic groups. 2009.

[KWMB16] T. Kinene, J. Wainaina, S. Maina, and L.M. Boykin. Rooting trees, methods for. *Encyclopedia of Evolutionary Biology*, page 489–493, Apr 2016.

[LP15] Michael S.Y. Lee and Alessandro Palci. Morphological phylogenetics in the genomic age. *Current Biology*, 25(19):R922–R929, 2015.

[MHK+20] Irene Martorelli, Leon S. Helwerda, Jesse Kerkvliet, Sofia I. F. Gomes, Jorinde Nuytinck, Chivany R. A. van der Werff, Guus J. Ramackers, Alexander P. Gultyaev, Vincent S. F. T. Merckx, and Fons J. Verbeek. Fungal metabarcoding data integration framework for the mycodiversity database (mddb). *Journal of Integrative Bioinformatics*, 17(1):20190046, 2020.

[NW70] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[NWT06] Paulo AS Nuin, Zhouzhi Wang, and Elisabeth RM Tillier. The accuracy of several multiple sequence alignment programs for proteins. *BMC Bioinformatics*, 7(1):471, Oct 2006.

[OPJ+05] Heath E. O'Brien, Jeri Lynn Parrent, Jason A. Jackson, Jean-Marc Moncalvo, and Rytas Vilgalys. Fungal community analysis by large-scale sequencing of environmental samples. *Applied and Environmental Microbiology*, 71(9):5544–5550, 2005.

[SB16] A.D. Scott and D.A. Baum. Phylogenetic tree. In Richard M. Kliman, editor, *Encyclopedia of Evolutionary Biology*, pages 270–276. Academic Press, Oxford, 2016.

[SJV+17] Anne Schöler, Samuel Jacquiod, Gisle Vestergaard, Stefanie Schulz, and Michael Schloter. Analysis of soil microbial communities based on amplicon sequencing of marker genes. *Biology and Fertility of Soils*, 53(5):485–489, Jul 2017.

[SSH+12] Conrad L. Schoch, Keith A. Seifert, Sabine Huhndorf, Vincent Robert, John L. Spouge, C. André Levesque, Wen Chen, Elena Bolchacova, Kerstin Voigt, Pedro W. Crous, Andrew N. Miller, Michael J. Wingfield, et al. Nuclear ribosomal internal transcribed spacer (its) region as a universal dna barcode marker for fungi. *Proceedings of the National Academy of Sciences*, 109(16):6241–6246, 2012.

[SSM22] Mohammad Yaseen Sofi, Afshana Shafi, and Khalid Z. Masoodi. Chapter 6 - multiple sequence alignment. In Mohammad Yaseen Sofi, Afshana Shafi, and Khalid Z. Masoodi, editors, *Bioinformatics for Everyone*, pages 47–53. Academic Press, 2022.

[Sta14] Alexandros Stamatakis. RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312–1313, 01 2014.

[SW81] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[TBP+17] Leho Tedersoo, Mohammad Bahram, Rasmus Puusepp, R. Henrik Nilsson, and Timothy Y. James. Novel soil-inhabiting clades fill gaps in the fungal tree of life. *Microbiome*, 5(1):42, Apr 2017.

[VGOA04] Susana Vinga, Rodrigo Gouveia-Oliveira, and Jonas S. Almeida. Comparative evaluation of word composition distances for the recognition of SCOP relationships. *Bioinformatics*, 20(2):206–215, 01 2004.

[ZGB+19] Andrzej Zielezinski, Hani Z. Girgis, Guillaume Bernard, Chris-Andre Leimeister, Kujin Tang, Thomas Dencker, Anna Katharina Lau, Sophie Röhling, Jae Jin Choi, Michael S. Waterman, Matteo Comin, Sung-Hou Kim, Susana Vinga, Jonas S. Almeida, Cheong Xin Chan, Benjamin T. James, Fengzhu Sun, Burkhard Morgenstern, and Wojciech M. Karlowski. Benchmarking of alignment-free sequence comparison methods. *Genome Biology*, 20(1):144, Jul 2019.

[ZJ12] Nadine Ziemert and Paul R. Jensen. Chapter eight - phylogenetic approaches to natural product structure prediction. In David A. Hopwood, editor, *Natural Product Biosynthesis by Microorganisms and Plants, Part C*, volume 517 of *Methods in Enzymology*, pages 161–182. Academic Press, 2012.

[ZVAK17] Andrzej Zielezinski, Susana Vinga, Jonas Almeida, and Wojciech M. Karlowski. Alignment-free sequence comparison: benefits, applications, and tools. *Genome Biology*, 18(1):186, Oct 2017.

A Appendix

	l	d_{min}	d_{max}	$size_{max}$	s	a	Full constraint	l-INS-i	# sequences	size		runtime (h)	non-matching / unique					
										avg	std		Phylum	Class	Order	Family	Genus	Species
1	0.2	3	4	1500	2.0	0	0	0	16074	74.8	208.3	5.02	0/16	0/50	0/180	434/434	1225/1966	3641/6834
2	0.2	3	3	0	2.0	0	0	0	14617	78.7	232.3	4.32	0/16	0/50	0/180	451/407	1152/1828	3154/5876
3	0.2	4	4	0	2.0	0	0	0	18913	38.4	124.1	4.82	0/13	0/50	0/165	0/492	1452/2489	4723/8895
4	0.2	3	4	1500	2.0	1	0	0	16074	74.8	208.3	5.42	0/16	0/50	0/180	451/434	1231/1966	3688/6834
5	0.2	3	4	1500	2.0	0	1	0	16074	74.8	208.3	5.49	0/16	0/50	0/180	436/434	1231/1966	3635/6834
6	0.2	3	4	1500	2.0	1	1	0	16074	74.8	208.3	4.84	0/16	0/50	0/180	446/434	1225/1966	3686/6834
7	0.2	3	4	1500	2.0	0	0	1	16074	74.8	208.3	5.58	0/16	0/50	0/180	404/434	1192/1966	3612/6834
8	1000	3	4	1500	2.0	0	0	0	15141	70.1	206.1	5.79	0/15	0/50	0/178	234/369	946/1602	3542/6460
9	0.2	3	4	1500	0.5	0	0	0	25049	107.5	247.9	10.7	0/16	0/57	0/197	772/633	1966/3051	5466/10608
10	0.2	3	4	1500	1.0	0	0	0	23237	101.5	244.0	8.20	0/16	0/54	0/193	727/584	1852/2841	5170/9938
11	1000	3	4	1500	0.0	0	0	0	34231	136.4	297.6	19.5	0/18	0/67	0/211	908/696	2450/3536	7186/13998

Table 8: Result summary of the generated UNITE backbones, with different parameter settings. Please refer to Table 9 for a short description of and file link to the trees.

	Description	Folder name (tree link)
1	Strict & hybrid split (baseline)	l0.2_s3_4_1500_o2.0_a0
2	Split on order	l0.2_s3_3_0_o2.0_a0
3	Split on family	l0.2_s4_4_0_o2.0_a0
4	Alternative repr. method	l0.2_s3_4_1500_o0.2_a1
5	Constrain high-level taxa	l0.2_s3_4_1500_o2.0_a0_constr
6	4 & 5	l0.2_s3_4_1500_o2.0_a1_constr
7	l-INS-i MAFFT algorithm	l0.2_s3_4_1500_o2.0_a0_localpair
8	No length filter	l1000.0_s3_4_1500_o2.0_a0
9	Low strictness ($o = 0.5$)	l0.2_s3_4_1500_o0.5_a0
10	Medium strictness ($o = 1.0$)	l0.2_s3_4_1500_o1.0_a0
11	No filters	l1000.0_s3_4_1500_o0.0_a0

Table 9: Each of the generated UNITE backbones. Indices correspond to Table 8. Data can be found on https://github.com/luukromeijn/MDDB-phylogeny/tree/main/results/thesis%20results/ under the given directory names. Click on the folder names for direct access to the backbone file.