Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis Computer Science

LCA analysis of metagenomic sequences in Python

Tobias Laß

August 21, 2021

Reviewer

Prof. Dr. Daniel Huson
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Tutor

Timo-Niklas Lucas
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Laß, Tobias:
LCA analysis of metagenomic sequences in Python
Bachelor Thesis Computer Science
Eberhard Karls Universität Tübingen
Thesis period: April 1, 2021 - August 21, 2021

Abstract

The increase in demand for metagenomic analysis inspires the exploration of new methods and implementations in bioinformatics. For this thesis, a new tool called Python Metagenome Analyzer (PYGAN) was developed. PYGAN implements a naive LCA algorithm in Python to analyze metagenomic sequences. The LCA algorithm finds the lowest common ancestor of a set of nodes in a tree structure. Furthermore, the tool surveys new heuristics to project mapped reads in the taxonomy to nodes of a specific rank. The naive or proportional projection is compared to an accession-based projection and a composition of the two approaches. PYGAN draws parallels to MEGAN CE to evaluate the suitability of Python in metagenomic analysis. Python provides a comfortable environment for the development of programs but lags behind in performance. This thesis comes to the conclusion that computationally intensive procedures should be implemented in C/C++ or other efficient languages and connected with Python on a higher level.

Zusammenfassung

Die steigende Nachfrage nach Analyse von metagenomischen Sequenzen inspiriert die Untersuchung neuer Methoden und Implementierungen in der Bioinformatik. Für diese Arbeit wurde ein neues Programm namens Python Metagenome Analyzer (PYGAN) entwickelt. PYGAN implementiert einen naiven LCA-Algorithmus in Python, um metagenomische Sequenzen zu analysieren. Der LCA-Algorithmus findet den niedrigsten gemeinsamen Vorfahren einer Gruppe von Knoten in einer Baumstruktur. Darüber hinaus untersucht das Programm neue Heuristiken, um *reads* in der Taxonomie auf Knoten eines bestimmten Ranges zu projizieren. Die naive oder proportionale Projektion wird mit einer *accession*-basierten Projektion und einer Komposition der beiden Ansätze verglichen. PYGAN orientiert sich an MEGAN CE, um die Eignung von Python für metagenomische Analysen einzuschätzen. Python bietet eine komfortable Umgebung für die Entwicklung von Programmen, ist aber nicht so performant wie alternative Sprachen. Das Ergebnis ist, dass rechenintensive Vorgänge in C/C++ oder anderen effizienten Sprachen implementiert und auf höherer Ebene mit Python verbunden werden sollten.

Acknowledgements

This thesis becomes a reality with the kind support and help of several individuals. I thank Prof. Dr. Huson and Timo-Niklas Lucas for our intriguing and stimulating discussions about LCAs and Python during our frequent online meetings. I extend my deepest gratitude to Tim Süberkrüb and Silvia Tucciarone for their relentless critical feedback and endearing encouragement. For her unconditional and extensive support, I thank Kornelia Laß.

Contents

List of Figures

List of Tables

List of Abbreviations

BLAST	Basic Local Alignment Search Tool
CSV	Comma-Separated Values
CLI	Command Line Interface
CPU	Central Processing Unit
DAA	Diamond Alignment Archive
GTDB	Genome Taxonomy Database
HDD	Hard Disk Drive
LCA	Lowest Common Ancestor
MEGAN	Metagenome Analyzer
MEGAN CE	MEGAN Community Edition
MEGAN LR	MEGAN Long Reads
NCBI	National Center for Biotechnology Information
NCBI-nr	NCBI nonredundant
PYGAN	Python Metagenome Analyzer
SSD	Solid State Drive
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Metagenomics

The goal of metagenomics is to analyze an assemblage of microbes from an environmental sample. By sequencing their DNA, a greater understanding of their composition and operation can be achieved [15]. Environments include extreme cases such as acidic water from mines, as well as soil and aquatic conditions. The microbes in humans are also interesting to study. A human body contains about ten times more prokaryotic cells than of its own eukaryotic cells. The prokaryotes are highly concentrated in the gastrointestinal tract and strongly related to medical conditions. With a better understanding of these correlations from metagenomic research, new therapies for microbial-related diseases could be developed [18].

Some environments feature a high homogeneity of species, which allows for the assembly of almost complete individual genomes. In contrast, other environments show high species diversity that is much more difficult to analyze. This is because clonal cultures may not truthfully reproduce the interactions and individual biological functions within the microbiome [35]. Only a fraction of microbes can be grown in pure laboratory cultures in the first place [11]. Metagenomics offers a solution that bypasses the necessity of pure cultures [17]. While the assembly of complete genomes is difficult, valuable information about the occurrence and distribution of various taxonomic groups can be gained. Since metagenomics considers an entire microbiome instead of an individual microbe, functional connections between community members may also be inspected.

A metagenome is the accumulated gene content of microbes in an environment. One approach to analyze metagenomes utilizes amplicon sequencing of phylogenetic marker genes. Those include 16S or 18s rRNA or - in the case of fungi - internal transcribed spacer genomic regions [19]. This method provides efficient identification of many bacteria and eukaryotes but does not capture viruses [6]. Additional drawbacks are that it only captures the content of targeted genes and causes an amplification bias. Nevertheless, for research with limited computational resources or a more general target, amplicon sequencing is the preferred method [14].

Another approach to analyze metagenomes is whole genome sequencing [21]. One method for this purpose is called shotgun sequencing, which refers to the random sequencing and fragmentation of the metagenome into millions of sequences [33]. These sequences, or reads, capture DNA or RNA of archaea, viruses, bacteria, and eukaryotes without amplification bias. This method allows for a more fine-grained taxonomic profiling of the metagenome, as well as the detection of pathogens and detailed functional analysis. However, this requires a high read count and reference genomes for binning. Shotgun sequencing is the preferred method for analyzing microbiomes and its development encourages an increase in the number of projects and their dimensions in metagenomics.

1.2 LCA analysis

Binning refers to the process of grouping reads into specific bins, which in this case correspond to taxonomy IDs. This kind of taxonomic classification of reads draws parallels to supervised machine learning techniques due to its dependency on established taxonomic groups and reference databases. Hence, the final taxonomic profile is a subset of a known taxonomy and does not allow for additional classification.

There are many different tools for binning reads, but this thesis focuses on and compares to MEGAN CE [14] and its previous versions. The currently latest version recommends DIAMOND [7] as opposed to BLAST [2] for sequence alignment. Both MEGAN [1] and DIAMOND [2] are freely available. Compared to BLASTX, DIAMOND provides an up to 20,000 times faster alignment of DNA sequences against reference databases of protein sequences such as NCBI-nr. The loss of sensitivity is negligible when considering the improvement in performance. Its specialization for high throughput, short reads, and attempt to compute all significant alignments, makes it ideal for metagenomics.

While MEGAN CE combines solutions for taxonomic and functional analysis, this thesis focuses on the analysis of taxonomic compositions. MEGAN CE performs taxonomic binning by mapping each read from the alignment output to a node from a taxonomy database. Such include NCBI [9], GTDB [25] and SILVA [28]. The idea is to find the lowest common ancestor (LCA) of a set of species [13]. The species are obtained from significant alignments for a read.

[1]https://github.com/husonlab/megan-ce
[2]https://github.com/bbuchfink/diamond

They are assumed to contain the source of the sequencing read. The read is then mapped to the corresponding node of the LCA. This naive version of the LCA algorithm provides a simple, but well-scaling option for taxonomic binning even for very large data sets.

Further extensions of the algorithm allow for fine-tuning of the analysis [14]. The quality of the considered alignments can be ensured by preprocessing the alignment output. Hits, whose scores fail to satisfy a min-score threshold or that are too far below the highest score, are filtered out. Furthermore, a minimum support filter may be employed to collect taxons with a low number of mapped reads. Each node, that does not satisfy the minimum support limit, yields its mapped reads to its parent node. This catches false positives and generally condenses the spread of the analysis result. Another possibility is to disregard mappings to specific taxa completely.

The naive LCA algorithm is limited such that it considers each read independently from all other reads. Consequently, it suffers from poor performance as a taxonomic profiling tool. To compensate, MEGAN CE provides a variation called the weighted LCA algorithm that evaluates the significance of reference sequences [14]. That improves the specificity of taxonomic assignments. Additionally, the naive LCA algorithm works under the assumption that, in the context of protein alignment, reads only map to single genes [12]. This holds true for short reads, but longer reads may overlap with multiple genes. The MEGAN LR (long reads) extension alleviates this issue with another variation called interval-union LCA algorithm. It conceptually partitions a read into a set of intervals.

A characteristic all variations of the LCA algorithm share is the binning of reads across all taxonomic ranks. Naturally, reads aligning to more conserved genes are mapped to taxa with higher ranks. In contrast, reads aligning to very specific genes are mapped to taxa of low-level ranks. Another extension of the algorithm, a simple projection algorithm, can be employed to map all of the reads to a single specified taxonomic rank instead [14]. This may be desired for taxonomic profiling. MEGAN CE provides a proportion-based solution to this issue. All reads above the specified rank are projected downwards proportionally to the number of reads mapped to the children of a node. For reads below the specified rank, they are directly mapped to the corresponding ancestor-node of the rank. The limits and an expansion of this approach are also discussed in this thesis.

1.3 Python in Bioinformatics

While MEGAN CE is based on Java and DIAMOND on C/C++, this thesis explores the strengths and limitations of an LCA analysis in Python. The language was published in 1991 by Guido van Rossum and combines multiple programming paradigms [27]. It allows for procedural and object-oriented as well as convenient functional programming styles. This is supported by its very high-level syntax and dynamic type system.

Python's main appeal is the syntax simplicity [4]. It is intuitive, easy to read, and quick to write. Combined with dynamic typing, it removes significant obstacles that inexperienced users would usually encounter. With this high degree of accessibility, it attracts many different scientific communities in need of fast and easy computational solutions. For example, AstraZeneca uses Python for drug development [5] and NASA incorporated Python in the flight software for their Mars drone [22].

A characteristic that Python shares with Java is its platform independence [27]. Any system that has an interpreter for Python installed, can run Python code written on any other system. Combined with the fact, that it is an interpreted language and not compiled, it maximizes Python's portability. Code snippets can be shared as plain text and seamlessly executed on other systems. This scripting approach allows for rapid prototyping and testing in an interactive shell [3].

Apart from easily shareable scripts, Python allows for effortless distribution of full modules [4]. The Python Package Index (PyPi) is an extensive, publicly available repository of packages. Users may deploy their packages to `https://pypi.org/`. The Python Package Manager *pip* is a standard Python module, that is executed as a program. It assists in the deployment, installation, and management of packages. This makes Python suitable for collaboration.

The drawback that comes with Python's flexibility is that it tends to show worse performance than compiled languages [24]. However, Python provides interfaces to seamlessly integrate more efficient code written in languages including C, C++, and Fortran. That way, a developer retains the comfort of writing an interpreted script while also having access to highly performant functionalities. Consequently, Python is ideal as a *runner* language that effortlessly ties together foreign components.

There is a great interest in powerful scientific libraries to alleviate Python's lack of native science functionalities. A library called NumPy [23] is the backbone of Python's scientific computing capabilities. It forms the basis for the development of more sophisticated scientific libraries such as SciPy [34] or bioinformatics libraries including Biopython [8]. The bioinformatics community relies on collaboration as well as high performance. With Python's flexibility and supplemented scientific computing capabilities, it provides a suitable environment for the development of bioinformatical applications.

Chapter 2

LCA analysis tool PYGAN

2.1 Motivation

MEGAN already provides means for analysis and exploration of sequencing data. PYGAN is a tool created for this thesis. It aims to explore the suitability and benefits of such analysis in Python. For the scope of this thesis, an LCA analysis is implemented along with the necessary structures to perform one. The program is capable of accepting alignment and taxonomy data as well as generating output in plain text. Furthermore, PYGAN may also be used as a foundation to expand into further analysis and interactive exploration. In addition, PYGAN implements a new heuristic for downward-projection as discussed later.

2.2 Architecture

PYGAN is realized as a scriptable Python module. As it only depends on the Python standard library, no further requirements apart from Python 3.6 or higher are needed. After installation it allows the user to perform an LCA analysis on reads aligned with DIAMOND by calling a single function in its top-level module *lca_analysis.py*. The function *run* performs a single LCA analysis, but the user has the option to script their own analysis. For that purpose, a set of functions is available in *lca_analysis.py* that can be combined to perform a fully customizable analysis. This enables the reuse of partial results and reduces the runtime of subsequent analyses substantially. Full instructions on how to script the PYGAN module can be found in the doc strings of the project [1].

[1] `https://github.com/Tobi208/pygan/`

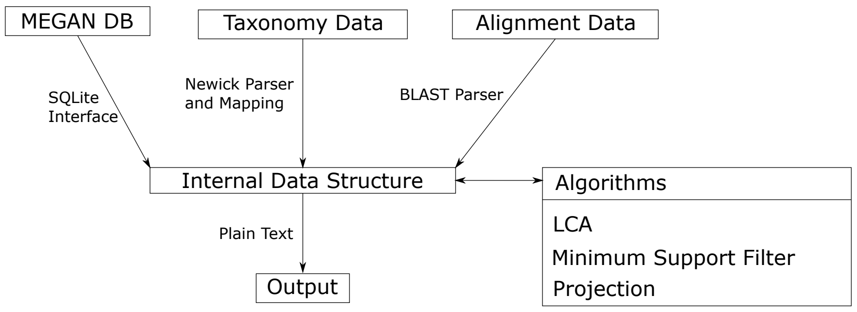


Figure 2.1: PYGAN code architecture

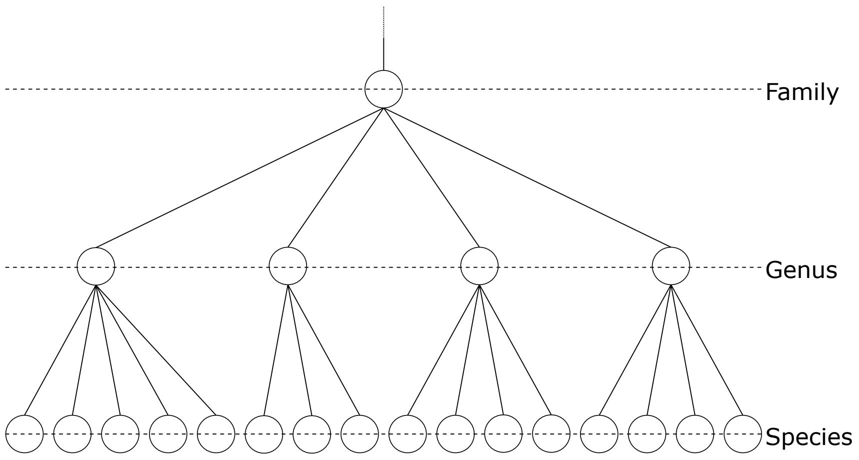
The tool is composed of multiple distinct components that interact with other components or external files. The set of files that needs to be installed with PYGAN consists of the *Megan Map Database*, a desired type of taxonomy tree, and a taxonomy mapping. The taxonomy tree and mapping are parsed to the internal data structure, whereas the database provides means to interpret the user input. A set of algorithms can be applied to the input data to populate the data structure. The output is purely generated from the data structure which contains the results. This compartmentalization of the functionalities allows for potential extensions of the tool or possibly to split it into multiple modules.

2.3 Data Structure

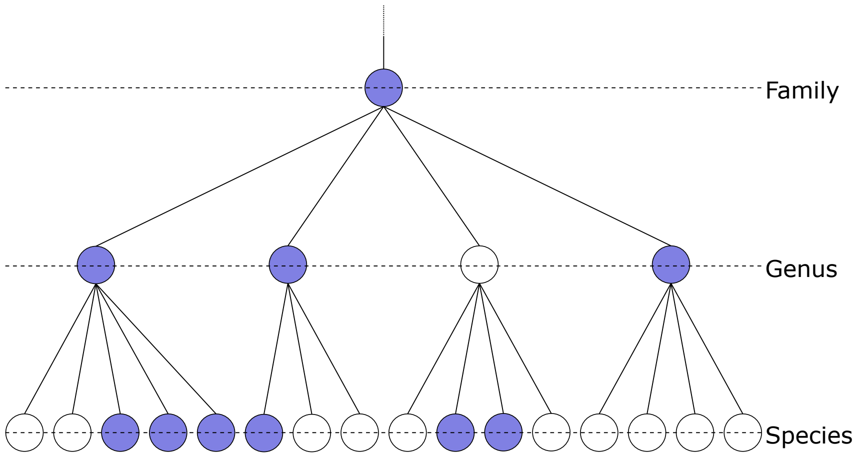
PYGAN runs on common phylogenetic trees. Each of these trees is implemented as a hash map of its nodes and a pointer to the root node. Each phylogenetic node contains information about its ID, scientific name, scientific rank, and pointers to its parent node and an arbitrary amount of child nodes. The nodes can be equipped with this data before any analysis has taken place. It is purely parsed from the taxonomy tree and taxonomy mapping. After the analysis, nodes will also contain information either about the number of reads or a list of read IDs mapped to them. The results in the phylogenetic nodes can then be written into a plain text file.

Phylogenetic trees may contain a large number of nodes, but their depth is limited by the number of taxonomic ranks. This property suggests a high number of children per node. PYGAN generally traverses the tree depth-first, which suits the shallow structure.

Figure 2.2a illustrates an example of a part of a phylogenetic tree. The tree extends further upwards and possibly downwards, but each node shares the same characteristics. The levels of the tree are labeled by taxonomic ranks. Figure 2.2b shows a taxonomy populated with alignment data.



(a) Example branch of a phylogenetic tree. Apart from its structure, each node contains information about its ID and scientific name.



(b) Example branch of a populated (blue circles) phylogenetic tree. The populated subtree forms the alignment taxonomy.

Figure 2.2: Example branches of an unpopulated and a populated phylogenetic tree. Legend available in figure 2.3.

⚪ Taxon without reads mapped	Number of mapped reads
🔵 Taxon with reads mapped	Accumulated sum of reads
🔴 Taxon of target rank without reads mapped	Total accumulated sum of reads
🟢 Taxon of target rank with reads mapped	Number of accession per hit
⚫ Taxon of rank 'unspecified'	Number of accessions per cluster
◌ Marked taxon	
🟡 Taxons corresponding to accessions	

Figure 2.3: Universal legend for figures in this thesis.

2.4 Input

2.4.1 Alignment Data

For the purpose of this thesis, a set of *daa* files containing results of DIAMOND runs was parsed into a readable format using DIAMOND. This was achieved with the command **./diamond.exe view -a file.daa -f 6 qseqid sseqid bitscore -o out.txt**. It extracts the *query sequence ID*, the *subject sequence ID* and the *bit-score* from the DIAMOND output. In this case, the query sequence ID denotes the *read ID* and the subject sequence ID to *accession*. Each accession is associated with a read ID and is rated with a bit-score. An accession suggests a taxonomic origin of a read. The bit-score describes the quality of that suggestion.

The phylogenetic tree is populated with information from the alignment data. Reads are mapped to nodes; nodes containing reads form the alignment taxonomy within the tree. Figure 2.4 illustrates the concept of mapping reads. The accessions are the keys to determine which node a read should be mapped to.

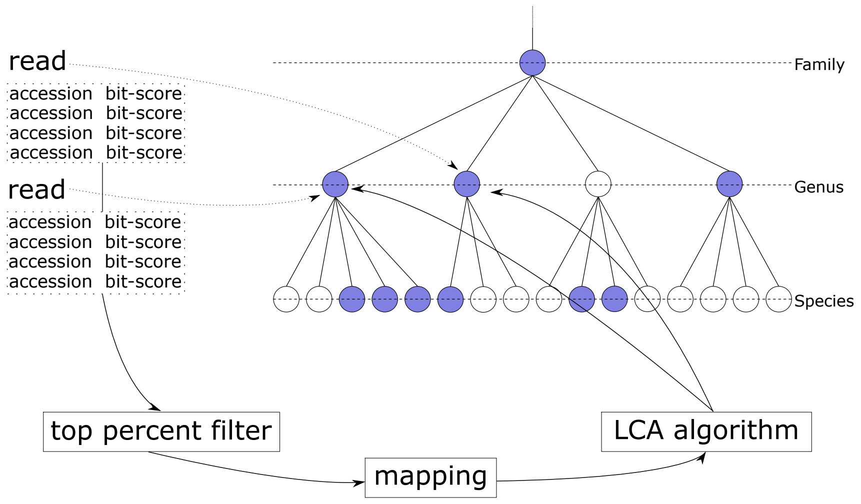


Figure 2.4: The taxonomy is populated with alignment data. The goal is to assign reads to taxons. Accessions provide the necessary information for the read-to-node mapping. The filtered and mapped accessions are used by the LCA algorithm. Legend available in figure 2.3.

To parse the data, the tool requires it to be in a tabulated format as well as a mapping. It should describe which column corresponds to which type of information. This allows the user to input individually generated data sets as long as a mapping is provided. While redundant information does not impact the scalability of the parsing, it does diminish the performance. This is because a larger file would have to be loaded into the memory.

To reduce the false-positive rate, the user can specify a *top score percentage* as a parameter. It is used to filter low-scoring alignments. If a top score percentage of 0.1 was passed, the accession with the highest score would be considered 100%. Every accession with a score lower than 90% would be discarded. The user can script the tool to either apply the top score percentage filter before mapping the accessions or afterward. Since not all accessions can be mapped using the database, the filter may yield different results depending on the time of application.

The result of the alignment data parsing consists of two ordered lists. One list contains the read IDs that will be relevant for the analysis. The second list contains collections of accessions that correspond to a read ID from the first list. While this may cause potential issues if the user interferes with the order of the lists, this structure improves the parsing performance by a factor of 1.8.

2.4.2 MEGAN Database

For the purpose of this thesis, PYGAN uses the *megan-map-Jan2021*. It is an SQLite database containing a mapping of NCBI-nr accessions to taxonomic and functional classes including NCBI and GTDB. Depending on the user's input, the database is used to map accessions from the alignment data to NCBI or GTDB taxonomy ID as shown in figure 2.4. The responsible tool component takes a collection of accessions as keys and looks up their corresponding taxonomy IDs in the database. The result can be obtained as a mapping of keys to values or as a corresponding collection of values.

In the alignment input data for this thesis, the number of accessions that have to be looked up in total can reach up to 10 million. This requires careful consideration concerning the performance of the tool. PYGAN uses the standard Python library as an interface to the database. In order to optimize the lookups, it is to be considered that the database access performance is highly hardware-dependent. Storing the database on an SSD compared to an HDD improves the performance by a factor of 15. Additionally using 64-bit Python compared to a 32-bit Python further improves the performance by a factor of 1.1. The lookups can be optimized on a software level too. Requesting differently sized chunks of accessions at once in an SQL-query instead of individually or collectively further improves the performance. This is shown in table 2.1. The performance does not scale predictably with the chunk size and slightly varies from run to run. To accommodate the user's hardware specifications, the chunk size can be specified as a parameter. For subsequent runs, the resulting collection of taxonomy IDs can be serialized and stored as a binary file. This means a complete lookup only has to be performed once for each data file.

Accessions per query	Duration of mapping in s
50000	99.92
25000	77.3
10000	62.44
5000	70.18

Table 2.1: Total duration of accession to taxonomy ID mapping depending on the query chunk size. All time measurements for this thesis were performed with an Intel i5-2500 3.30GHz CPU, 16 GB DDR3 RAM, an SSD, on 64-bit Windows with 64-bit Python 3.9.

2.4.3 Taxonomy Data

The tool requires an empty taxonomy tree to fill with taxonomy data from the DIAMOND output. This project can use either the NCBI taxonomy tree or the GTDB taxonomy tree that is also recommended for MEGAN6. The tree is available in a *Newick* format [16] without distances and all nodes named, e.g. *(A,B,(C,D)E)F*. It uses taxonomy IDs as node names. While there exist public Python modules for parsing Newick files, the specification of the exact format allows for a much more efficient custom implementation. For this use case, the custom implementation is more than 20 times faster than the *python-newick 1.3.0* library.

Only a single traversal of the string is performed during the parsing. With the sole character options being brackets, commas, or digits, each iteration calls a switch case. A comma denotes the creation of a node on the same level as the previous node - a sibling. An opening bracket denotes the creation of a node with children, while a closing bracket indicates the completion of that nodes' children. Digits are collected to create the labels of the nodes. Throughout the iterations, pointers to the current node that is being built and its parent node are maintained. That means the tree can already be traversed while it is being constructed with a minimal requirement of pointers. Thus the algorithm scales linearly with the number of characters in the Newick-string. The different cases are illustrated in figure 2.5.

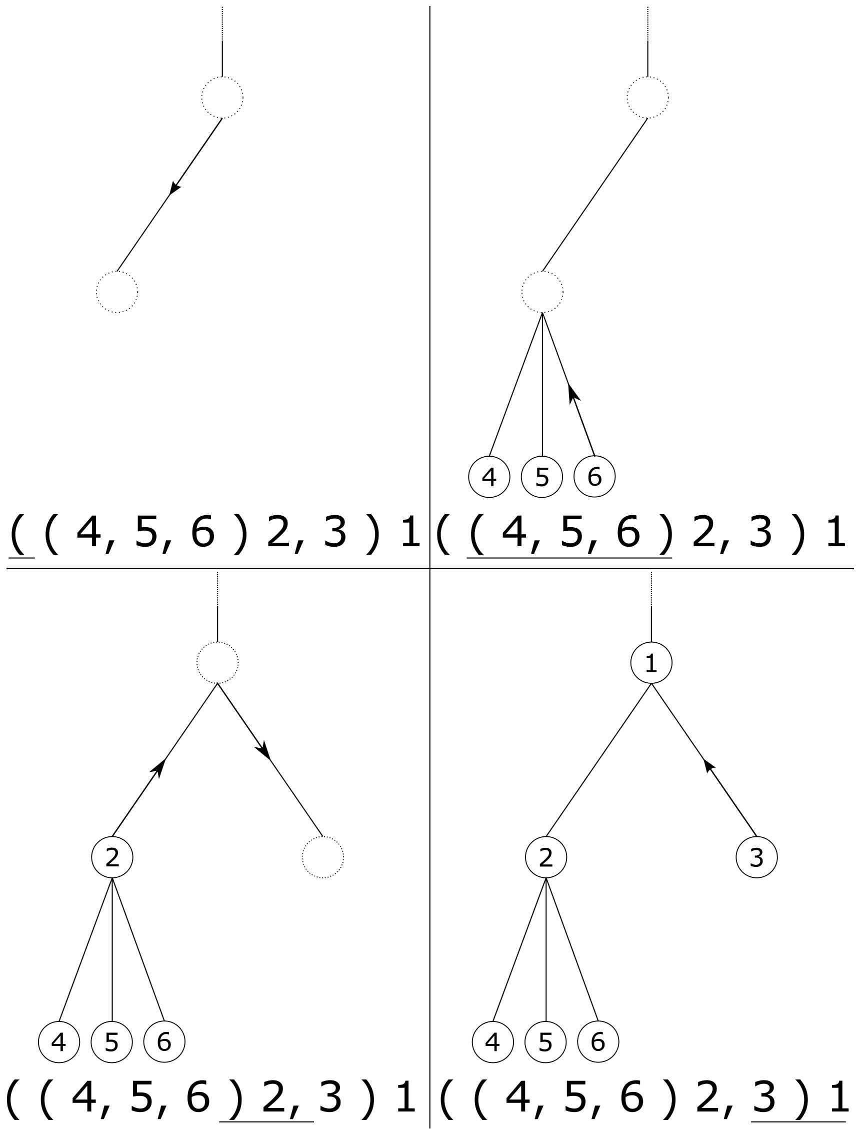


Figure 2.5: Parsing of a Newick-string to an internal tree structure. Opening brackets create a new child, closing brackets return to the parent, commas create siblings. Dotted nodes are not yet completed.

The nodes maintain the tree structure by pointers to their parents and children. However, on a top-level the tree is stored as a pointer to its root and a hash map of its nodes. The nodes' taxonomy IDs are used as keys for the hash map, allowing for node lookups in constant time. That greatly increases the performance of the analysis as well as the mapping of scientific names and ranks.

Additionally to the taxonomy tree, a mapping of taxonomy IDs to scientific names and ranks needs to be provided with the tool. The lines of the file are traversed once. Each entry is looked up in the hash map of nodes. If it exists, the node is populated with information about its name and rank. The mapping file does not specify the name of the rank, but a corresponding ID. The mapping of rank IDs to ranks is hard-coded in the tool since it is specialized for MEGAN6 formats. Since the node and rank ID lookups are in constant time, the parsing scales linearly with the number of entries in the mapping file.

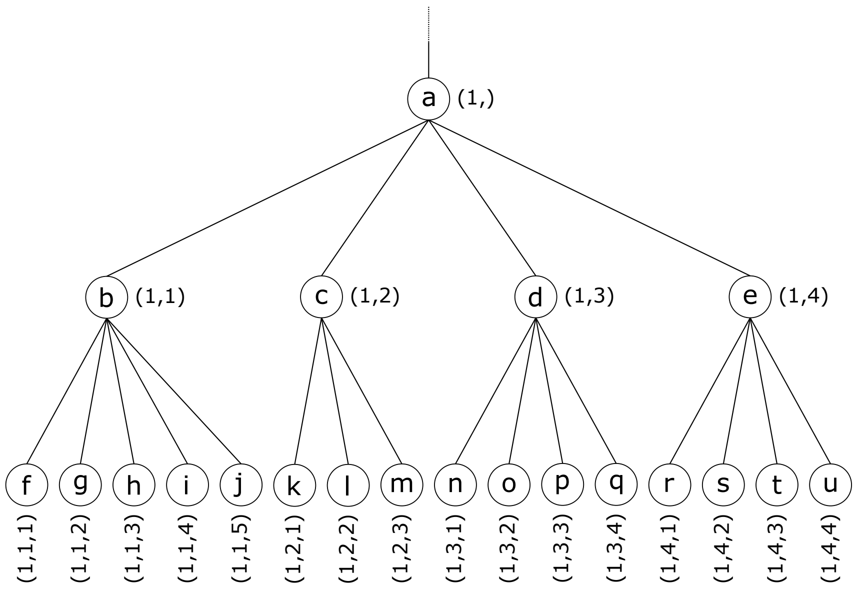
The result of the taxonomy data parsing is a phylogenetic tree as shown in figure 2.2a. So far, its nodes only contain information about the tree structure and their taxonomy IDs, names, and ranks. This empty tree may be serialized and stored as a binary file. However, the deserialization of the binary file to the internal data structure does not perform better than parsing the taxonomy data. Regardless, if any algorithms have been applied to the phylogenetic tree, it may be desirable to store the tree containing the results in a binary file. Storing a tree also allows the user to output the results in different formats without performing the same analysis again.

2.5 Algorithms

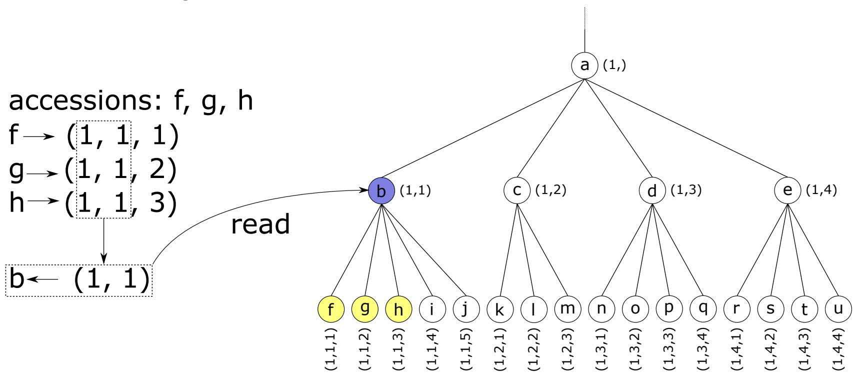
After constructing the empty phylogenetic tree and parsing and mapping the alignment data, the *LCA analysis* can be performed to populate the tree with taxonomy data (figure 2.4). It is not only possible to map reads to nodes in the tree with the Lowest Common Ancestor (LCA) algorithm, but also postprocessing the tree with a *minimum support filter* and *projections* to refine and customize the results.

2.5.1 LCA

The LCA algorithm is performed to map a read to a node in the taxonomy tree by its corresponding accessions [1]. PYGAN implements a naive LCA algorithm, which requires preprocessing of the tree. First, a mapping of node addresses to taxonomy IDs and vice versa has to be computed. This is achieved by traversing the tree depth-first. The collection of children in each node is ordered, which allows for a deterministic enumeration of them. In each recursive call, the accumulated path to the current node and its ID are entered in both mappings. For the next call for every child, the accumulated path is extended by the index of the child node. E.g. a node of depth 5 may have the address $(7, 3, 9, 0, 8)$ and one of its child nodes the address $(7, 3, 9, 0, 8, 1)$. The maximum length of any address equals the height of the tree. This phase is shown in figure 2.6a.



(a) Computation of the addresses of each node in the phylogenetic tree. Return a mapping of nodes to addresses and of addresses to nodes.

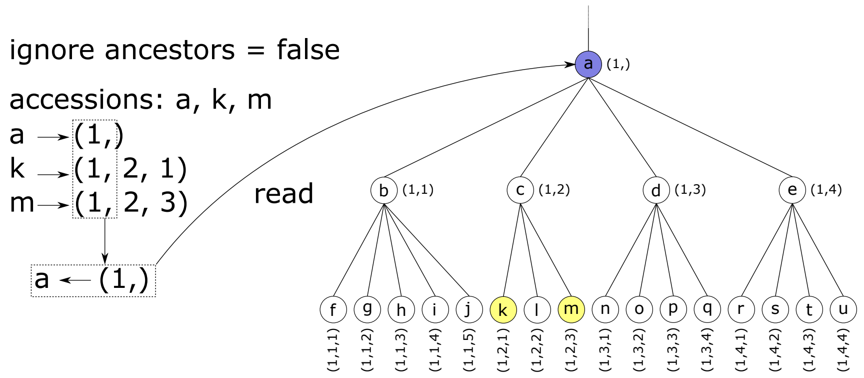


(b) Find the longest common prefix of a set of addresses. Map the corresponding read to a node of the phylogenetic tree.

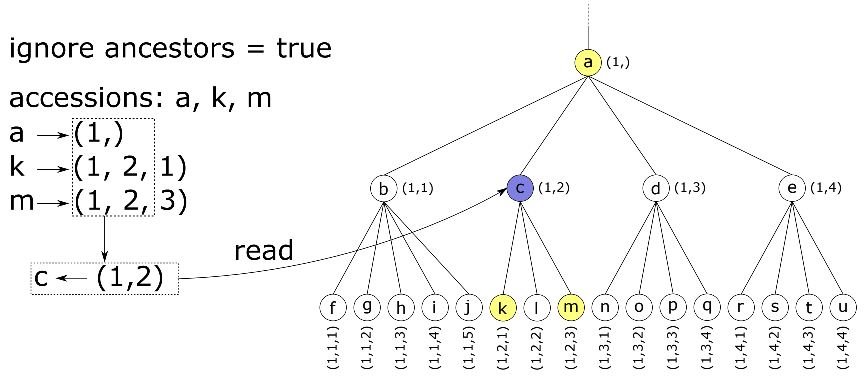
Figure 2.6: Preprocessing and mapping of the LCA algorithm. Legend available in figure 2.3.

After preprocessing the bidirectional mapping of addresses to IDs, the LCA algorithm does not require the tree structure any longer. A collection of taxonomy IDs corresponding to a single read can be passed to the algorithm and it will compute the lowest common ancestor with only the mappings. The concept is illustrated in figure 2.6b. The address of the lowest common ancestor equals the longest common prefix of all query addresses. The tool iterates over the lists containing the query addresses until the numbers at the current index are not identical anymore. This means that the paths of the corresponding query nodes start diverging here. The lowest common ancestor can not be on a diverging path and thus its address is up to the last matching number in the paths. This address is then returned.

The algorithm also has a flag for whether or not to ignore ancestors. If the user specifies to consider ancestors, the shortest address is used as a reference for the other addresses. In that case, the longest common prefix can only have a maximum length of the shortest address. Once the end of the shortest address is reached and no paths have diverged yet, it is forcibly returned as the result (figure 2.7a). This means that if one query is the ancestor of another node, the ancestor is returned as the lowest common ancestor. If on the other hand the user specifies to ignore ancestors, the ancestor node will never be returned as the result. In that case, the longest address is used as a reference for the other addresses. Once numbers at indices beyond their length are being compared, the comparisons automatically evaluate to true (figure 2.7b).



(a) Consider ancestors as possible nodes to map a read to. Use the shortest address as reference.



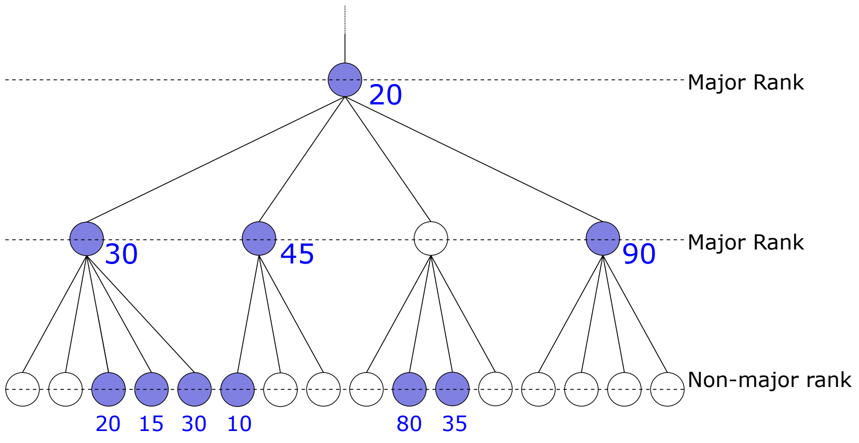
(b) Disregard ancestors as possible nodes to map a read to. Use the longest address as reference.

Figure 2.7: Different behavior of considering or disregarding ancestors are possible results of the LCA application. Legend available in figure 2.3.

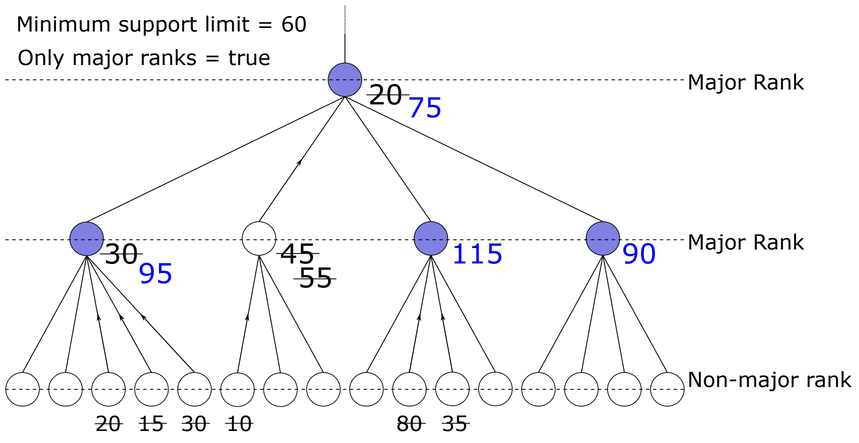
The LCA algorithm is performed on every collection of taxonomy IDs from accessions corresponding to a read. Each read is then mapped to its computed node in the taxonomy tree. After the LCA analysis, the phylogenetic tree contains the entire taxonomy from the alignment data.

2.5.2 Minimum Support Filter

To reduce the spread of the LCA result, a minimum support filter can be applied. The user can specify a minimum support limit that denotes how many reads a node must have at least mapped to it. This allows the user to filter out insignificant mappings. E.g. if a minimum support limit of 60 was specified, a node with less than 60 reads mapped to it will have its mapped reads passed to its parent node. An example input is constructed in figure 2.8a. The algorithm for the minimum support filter is illustrated in figure 2.8b.



(a) Example input for a minimum support filter. Reads are mapped to nodes. Nodes are part of a major or minor rank.



(b) Apply the minimum support filter with a limit of 60 and the flag to push to major ranks.

Figure 2.8: Example of an application of the minimum support filter. Legend available in figure 2.3.

This is implemented by a depth-first traversal of the phylogenetic tree. The recursive call is performed before pushing the reads upwards. This ensures that the filter is applied bottom-up and that if any reads are being pushed upwards, all children of the current node have no reads mapped to them anymore. The propagation of reads throughout the tree is further discussed in the chapter 3.

The minimum support filter further considers the specification of whether reads are only allowed to be mapped to the major ranks. These consist of *kingdom*, *phylum*, *class*, *order*, *family*, *genus* and *species* [20]. This option is available, because PYGAN supports the additional ranks *domain*, *varietas*, *species group* and *subspecies*. These may not always be desired in the results. In this case, any node that is not of a major rank will have its mapped reads pushed upwards indiscriminately during the traversal.

2.6 Output

To obtain a readable and usable result from the analysis, the information in the taxonomy tree can be formatted and written to a plain text file. The internal tree structure can be prepared for printing in multiple ways.

The user can specify to prefix rank abbreviations to the scientific names of the nodes. E.g. if the representative node of *Campylobacter* was to have its rank prefixed, it would be printed as *g__Campylobacter* with a 'g' for genus.

Furthermore, the user can specify to display the entire path in the phylogenetic tree down to a node. The path is prefixed to its scientific name. E.g. the path to the representative node of *Thermoanaerobaculia* would be displayed as */NCBI/cellular organisms/Bacteria/Acidobacteria/Thermoanaerobaculia* if taken from the NCBI taxonomy.

The last option specifies how to print the reads mapped to a node. It can be displayed as an accumulated number, e.g. 200 total reads mapped to a node. Another option is to print the list of read IDs as a comma-separated list. The former option allows for better readability while the latter option allows for the result to be used in more detailed further computations.

After preprocessing the tree with these options, its nodes are parsed to a string and written to a specified output file in plain text. The tree can also be output multiple times in different formats after being preprocessed once.

Chapter 3

Projection

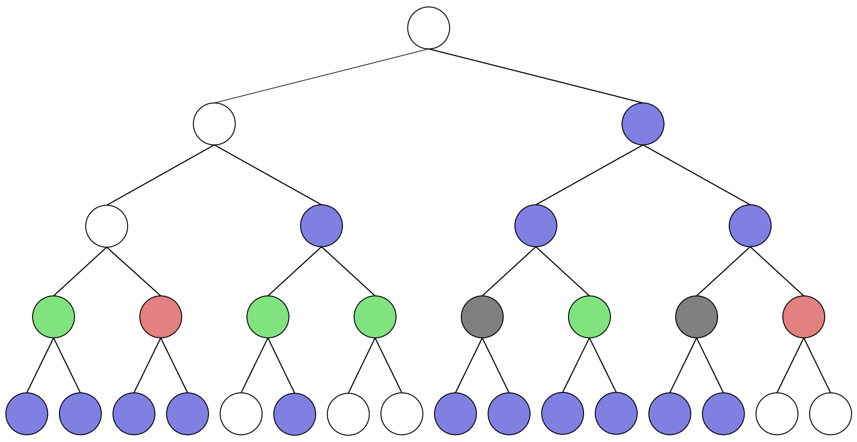
3.1 Motivation

It was discussed in chapter 2 that the user may only be interested in mapping reads to major ranks. The solution was to push reads from non-major ranks upwards until they hit a node of a major rank. This approach is intuitive because no information is lost and none is inferred. There is no ambiguity about the hierarchy since each node in the phylogenetic tree has exactly one parent node. E.g. a set of reads mapped to a node with the rank of genus are undoubtedly included within the reads mapped to the children of a single family-ranked node. If the genus node was not desired to be included in the result, its mapped reads would be moved upwards to its parent node of rank family.

However, the user may specify to project all reads to a certain rank. Depending on the specified rank's position in the hierarchy, a more sophisticated approach may be required. The approach to *upwards*-projection does not need to be changed, but a decision has to be made on how to handle *downwards*-projection. E.g if the user wanted to project all reads to nodes of rank genus, how would a family node project its mapped reads down to its children?

3.2 Proportional Downwards-Projection

One way to solve this issue is with *proportional* downwards-projection. The PYGAN implementation of this approach consists of two phases: bottom-up and top-down, each realized as a single depth-first traversal. As part of the bottom-up phase, a regular upwards-projection is applied to all nodes below the target rank as seen in figure 3.1. For this, a function is called recursively on the nodes starting from the root. If the recursion passes a node of the target rank, a flag is set for further calls. With the flag set, nodes pass their reads upwards to their ancestor of the target rank. This ensures that all reads are contained within nodes of the target rank or above. If a node below the target rank does not have a node of the target rank as an ancestor - due to incomplete NCBI or GTDB data - its reads are not projected upwards. These branches are considered failures.


(a) Possible result of an application of the LCA algorithm. Starting point of a projection algorithm.

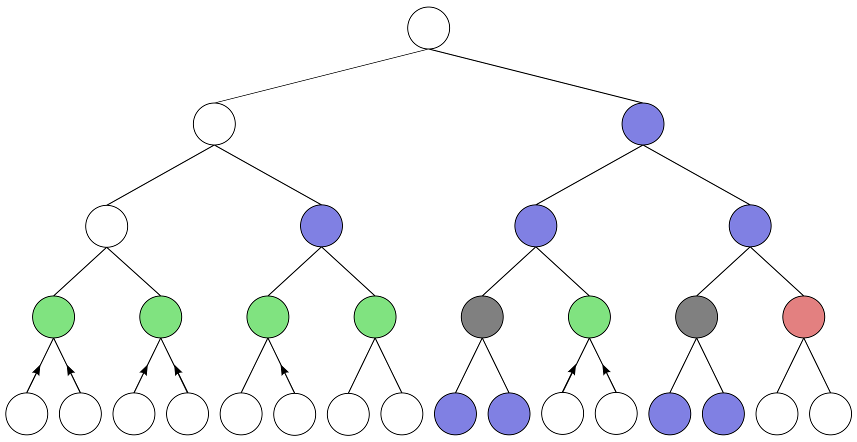
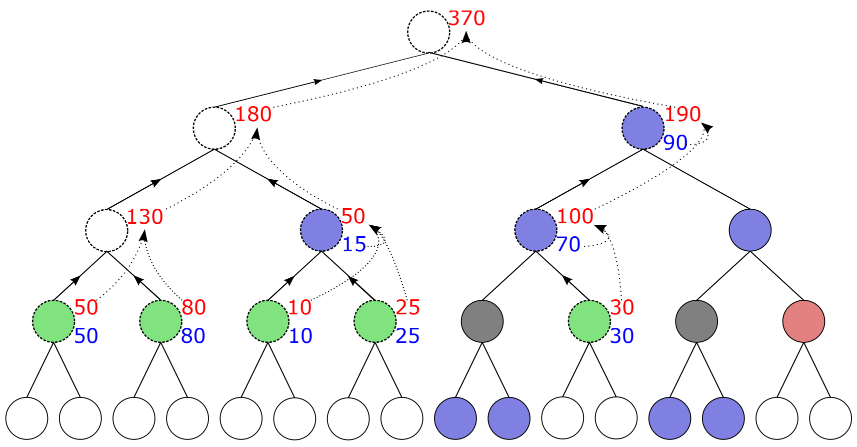

(b) Universal part of any bottom-up projection. Reads below the target rank are passed upwards.

Figure 3.1: Possible input for a projection algorithm. Illustrates basic upwards-passing of reads below a target rank. Legend available in figure 2.3.

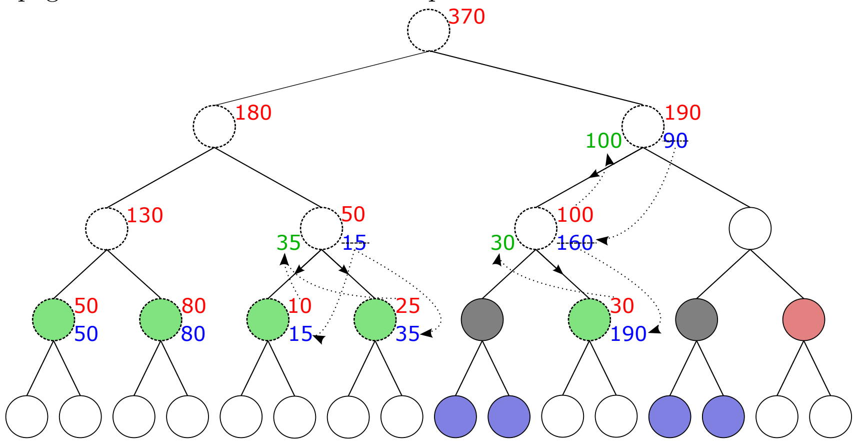
After the recursive calls on nodes below the target rank are completed, the bottom-up phase changes. Nodes of the target rank count the number of reads now mapped to them. They are marked as successful and can be used in the second phase. Then they propagate their marked status and number of mapped reads to their parent nodes. In return, the parent nodes add their own number of mapped reads to the accumulated sums. Their sums and markings are then further propagated upwards. This process is illustrated in figure 3.2a

At the end of this bottom-up process, each node above the target rank contains an accumulated sum of reads below. This is necessary for the proportional top-down projection. Only the number of accumulated reads is propagated and not the read IDs themselves. This measure prevents lateral projection of reads.

The top-down part is realized as a second depth-first traversal starting at the root. The current node of the recursion attempts to split its reads up and pass them to its viable children. For this, all marked children are collected and the sum of their accumulated sums computed. For each child, its portion of the nodes' reads is determined as seen in figure 3.2b.



(a) Second phase of the proportional upwards-projection. Mark nodes and propagate accumulated sum of reads upwards.



(b) Proportional downwards-projection or top-down phase. Distribute nodes to children as calculated by equation 3.1.

Figure 3.2: Visualization of the critical phase of the proportional upwards-projection and the proportional downwards-projection. Legend available in figure 2.3.

Let $r : Nodes \mapsto \mathbb{N}_0$ be a function that returns the number of reads mapped to a node. Let $acc\_sum : Nodes \mapsto \mathbb{N}_0$ be a function that returns a node's accumulated sum of reads. Let $cs : Node \mapsto Nodes^x \mid x \in \mathbb{N}$ be a function that returns the children of a node. Let $total\_acc\_sum : Nodes^x \mapsto \mathbb{N}_0 \mid x \in \mathbb{N}$ be a function that return the total accumulated sum of reads of a tuple of nodes. Then $p : Nodes \times Nodes \ \mapsto \mathbb{N}_0$ is the function that describes how many reads are passed from a node to one of its child nodes as defined in equation 3.1. The number is proportional to the ratio of the accumulated sum of the child node to the total accumulated sum of all children.

$$p(n, c) = \lceil acc\_sum(c) \cdot total\_acc\_sum(cs(n))^{-1} \cdot r(n) \rceil \tag{3.1}$$

Here is the example calculation from figure 3.2b. A node has 15 mapped reads, a child node has an accumulated sum of 10 and the total accumulated sum of the children is 35. $\lceil 10 \cdot 35^{-1} \cdot 35 \rceil = 5$ reads would be projected downwards to the child. This is computed for every viable child of the current node. Due to float to integer rounding the last child may receive fewer reads than it deserves. Before the next recursion, all reads of the current node will have been passed to its marked children. If the branch is a failure, i.e. there is no descendant of the target rank, the recursion will never be called on the branch. That means unmarked nodes keep their mapped reads. This is also the case for marked nodes that have no reads or the total accumulated sum of its children is zero.

When the top-down phase is over, the proportional downwards-projection is concluded. Every proportionally projectable read is mapped to a node of the target rank and all other reads are mapped to the same nodes as before.

3.3 Accession-Based Downwards-Projection

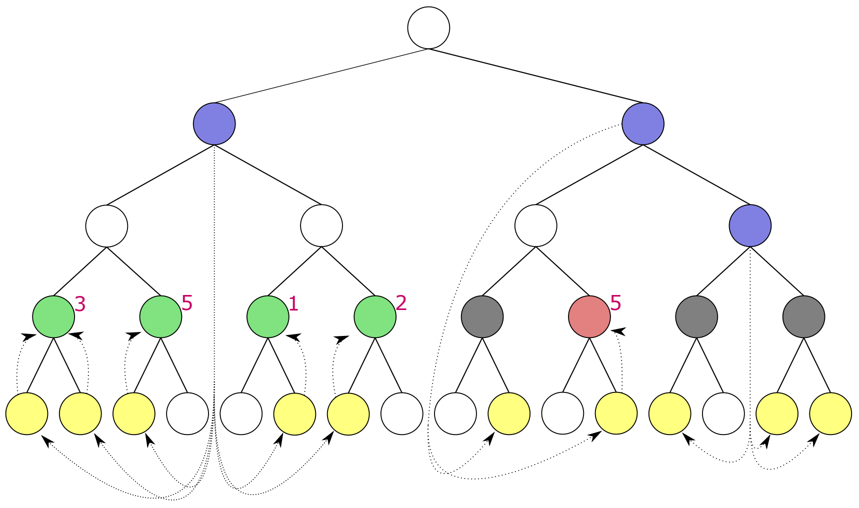
The problem that arises with the previously discussed approach stems from gaps in the taxonomy generated from the alignment data. The LCA algorithm may not map any reads to a node of the target rank or below. Furthermore, a node above the target rank may not have any reads mapped to any of its descendants of the target rank or below. The problem is that there is no information about the proportions it should project its reads downwards to the target rank. During the bottom-up part of each projection algorithm, a projection attempt is considered a failure if a node below the target rank has no ancestor of the target rank. This happens due to incomplete mapping data from which nodes are not necessarily assigned a rank. If a node above the target rank has no descendants of the target rank due to incomplete data, it is considered a failure too and will not have its reads moved. However, if there exist descendants of the target rank and only information about the proportions is missing, a new approach may solve the issue.

PYGAN offers a new solution to downwards-projection. The idea is to re-evaluate the mappings of the LCA with consideration of the target rank. The algorithm is structurally similar to the proportional approach. It is divided into two phases: bottom-up and top-down. The bottom-up phase collects reads from nodes below the target rank by traversing the tree depth-first. If the bottom-up function is called on a node of the target rank, the node passes itself as a projection target to the recursive calls for lower nodes. If the function is called with a projection target, the current node pushes its mapped reads directly to the target. This way reads are only pushed once instead of recursively propagated upwards to the target. The process is conceptually the same as illustrated in figure 3.1.

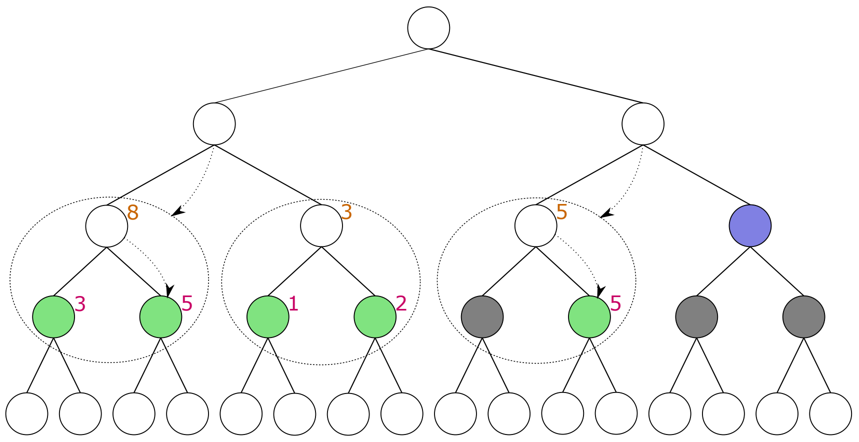
The steps of the bottom-up phase of the accession-based approach are a subset of the steps for the bottom-up phase of the proportional approach. It is only for nodes above the target rank for which the accession-based approach provides a different solution. It takes an assumption about a read that is mapped to a node by the LCA algorithm into account. The mapping implies that the corresponding accessions of the read map to descendants of that node. If at least of one those descendants is below the target rank, the accession-based approach can project that read downwards.

The top-down function is called recursively, depth-first starting at the root. A call on a node of the target rank terminates the recursion. For a node above the target rank, an attempt is made for each of its reads to project the read downwards. First, the taxons that correspond to the accessions of the read are acquired. Then it is counted how many taxons are descendants of nodes of the target rank. These are called *hits* and are seen in figure 3.3a. If no taxons have an ancestor of the target rank, there are no hits and the read can not be projected downwards. It is left in place instead.

Successful hits are then clustered to a specified degree. E.g. if the clustering degree was 1 and the target rank genus, hits would be clustered up to the rank of family. The cluster with the highest sum of taxons mapped to its corresponding hits is chosen as the fittest. The hit of the fittest cluster with the highest number of taxons mapped to it is chosen as the fittest. Lastly, the read is pushed downwards to the fittest hit. This process is illustrated in figure 3.3b. The procedure is repeated for every read mapped to a node. The clustering avoids mapping to false-positive outliers.



(a) First phase of the accession-based downwards-projection. Unravel the accessions corresponding to a read and relate their taxons to a node of the target rank. Nodes of the target rank found that way are considered hits (here all green nodes). The red node is not part of the alignment taxonomy yet, but is found as a hit too.



(b) Second phase of the accession-based downwards-projection. Cluster hits together to a certain degree. Here the clustering degree is 1. That means nodes 1 rank above the target rank collect hits below them.

Figure 3.3: Find suitable hits for reads and cluster them during the accession-based downwards-projection. The best hit of the best cluster receives the read. The previously red node is now green, i.e. it was added to the alignment taxonomy. Legend available in figure 2.3.

After all recursive calls of the top-down function are completed, the accession-based downwards-projection is concluded. Each read that can be related to a node of the target by its accessions is projected. Reads that can not be related are mapped to the same nodes as before the projection.

3.4 Comparison

The two approaches produce different results. The results were obtained with identical settings apart from the projection method used. In the remaining chapter *failed* describes reads or nodes that could not be projected *downwards*. It is to be noted that a minimum support filter should be applied after the projection and that nodes from the target rank should be exempted from the filter. That would condense the non-projectable reads into fewer nodes while the result of the projection would remain untouched.

The proportional algorithm can project 99.33% of reads downwards. 92.38% of reads are successfully pushed downwards by the accession-based algorithm. A later discussed composition of the two approaches projects 99.33% of reads downwards successfully (table 3.2). The proportional, accession-based and compositional algorithms pushed down reads of 89.75%, 83.97% and 91.91% of nodes respectively (table 3.1).

Projection Method	Nodes ↑TR	Nodes →TR	Nodes ↓TR
None	472	160	1931
Proportional	52	524	358
Accession	85	1546	358
Composition	41	1546	358
None	453	119	1779
Proportional	44	455	297
Accession	70	1263	297
Composition	36	1263	297
None	328	67	938
Proportional	33	251	131
Accession	48	699	131
Composition	25	699	131

Table 3.1: Results of the projection approaches. Lists number of nodes above, on or below the target rank. The analysis was done with a top percent score of 10%, the target rank genus and a clustering degree of 1. The different projection methods were performed on the data sets *Alice01-1mio-Jan-2021*, *Alice03-1mio-Jan-2021*, and *Alice06-1mio-Jan-2021*. $\uparrow TR$ means above the target rank, $\rightarrow TR$ on the target rank and $\downarrow TR$ below the target rank.

The proportional projection retains a much lower number of nodes with any reads mapped to them than the accession-based approach. That is because the accession-based projection adds more nodes to the alignment taxonomy. Here, reads can be mapped to taxons that were previously not part of the alignment taxonomy. However, the accession-based approach has to be passed the alignment data to make this possible. In contrast, the proportion-based approach

Projection Method	Reads ↑TR	Reads →TR	Reads ↓TR
None	415634	217617	133580
Proportional	3252	739957	23622
Accession	42206	701003	23622
Composition	3240	739969	23622
None	447360	209445	130836
Proportional	5062	760370	22209
Accession	40560	724872	22209
Composition	5050	760382	22209
None	407984	387004	61427
Proportional	394	852816	3205
Accession	9300	843910	3205
Composition	386	852824	3205

Table 3.2: Results of the projection approaches. Lists number of reads above, on or below the target rank. Continuation of table 3.1.

only requires the taxonomy and only projects reads onto nodes included in the known taxonomy.

The percentage of nodes retained that contain non-projectable nodes is similar in both approaches. This suggests a common characteristic for failed nodes of both approaches. A node that does not have any non-empty descendants of the target rank is failed by the proportional approach. A node whose reads can not be related to any descendants of the target rank by their accessions is failed by the accession-based approach. The projections require a node to have descendants mapped as the target rank. Both kinds of failure can occur if the taxonomy mapping data is incomplete. If a branch has taxons of the target rank mapped as unspecified, no projection can be performed.

In contrast to the failed nodes, the percentage of non-projectable reads differs by 6.95 percent points (table 3.2). The accession-based approach has a lower rate of success than the proportion-based method. This is because only a single descendant with the target rank needs to contain reads for the proportional projection to push down all reads of a node. In contrast, the successful projection of a single read in the accession-based approach does not relate to the success or failure of the projection of other reads. In the alignment data for this project most accessions map to taxons of the rank species. That means for any target rank of species or above, the downwards-projection should produce suitable hits. In reality, this is complicated by incomplete mapping of ranks. This property of the accession-based projection makes it more vulnerable to incomplete mapping data than the proportional approach.

The approaches differ conceptually and both raise questions for their validity. The proportion-based projection chooses reads to push downwards arbitrarily. The context of the reads is not considered. The same analysis with data identical except for its order could produce different results. As a consequence, only the number of reads mapped to nodes should be considered in the results since that is deterministic. On the other hand, the accession-based approach considers the context of the reads. In essence, the projection unravels a result of the LCA algorithm and partly imitates the algorithm. By restricting the maximum rank of the result, a lot of information is disregarded. Thereby, the quality of the result is lower. However, this makes the projection of reads deterministic and more truthful than proportional projection.

3.5 Composition

Since both algorithms only change the mapping of reads in the phylogenetic tree, their application on the tree can be composed. Applying the same algorithm multiple times does not change the result, but applying one algorithm to the result of the other's changes the final result significantly. The proportional projection is vulnerable to gaps in the alignment taxonomy, whereas the accession-based projection is more vulnerable to incomplete taxonomy mapping. By applying the algorithms successively, they partly cover each other's weaknesses.

As previously observed, the accession-based approach increases the number of nodes in the alignment taxonomy. By nature of the algorithm, only new nodes of the target rank are added. Therefore, a taxonomy previously modified by the accession-based approach provides a greater number of targets for the proportional projection. Consequently, the proportional algorithm distributes reads more evenly. While this process remains non-deterministic due to the application of the proportional projection, it produces a more fine-grained result. On the other hand, reads that are failed by the accession-based projection are caught by the proportional algorithm.

Table 3.1 illustrates these observations. The compositional approach produces an alignment taxonomy with fewer nodes than the accession-based approach. During each accession-based phase, the same new nodes are added to the alignment taxonomy. However, during the proportional phase of the compositional approach other nodes are removed from the alignment taxonomy. For this data, the number of failed nodes is approximately halved.

A node is successful in the proportional projection if any read can be projected. In contrast, a node is only successful in the accession-based projection if all of its reads can be projected. Applying the proportional algorithm after the accession-based algorithm imposes the more lenient condition for success on the full process. The difference in nodes above the target rank shows that many nodes failed by the accession-based approach are salvaged by a succeeding proportional run.

On the other hand, a purely proportional run has 1.27 more failed nodes than a composed run (table 3.1). This shows that the accession-based projection can add previously inaccessible branches to the taxonomy for the proportional projection. Therefore, both approaches' success-rate benefit from a compositional application.

The effect on the composition becomes more apparent when observing the number of failed reads instead of nodes. Applying a succeeding proportional run reduces the total percentage of failed reads from 7.62% to 0.67% (table 3.2). This further illustrates the accession-based approach's vulnerability to incomplete mapping data. The composition imposes the proportional projection's robustness against incomplete mapping data on the entire process. In contrast, the decrease of failed reads from the proportional to the compositional approach is minimal. Thus the improvement of failed nodes does not translate to a comparable improvement of failed reads.

Performing a proportional projection before an accession-based projection does not improve the results. The nodes that are failed by proportional projection are also failed by accession-based projection unless every failed read can be related to a node of the target rank. The minimal difference in failed reads of the proportional and compositional approach shows how improbable that is.

3.6 Conclusion

Accession-based projection is the method of choice when higher sensitivity is desired. Proportional projection should be used when a denser result and greater robustness against incomplete mapping data are beneficial. Composition of the two approaches provides a compromise that increases sensitivity and robustness against incomplete mapping data. If necessary, the minimum support filter can be employed to condense the result. PYGAN satisfies different needs by providing access to both approaches and their composition.

Chapter 4

Comparison to MEGAN CE

4.1 Concept

MEGAN CE [14] is a stand-alone program written in Java. It provides a variety of taxonomic and functional analyses of metagenomic data. The results of analyses can be interactively explored through a graphical interface. The program is highly flexible in input data, analysis configurations, and output generation. It covers a high amount of use cases. An example of an alignment taxonomy can be seen in figure 4.1. The tree is fully interactive.

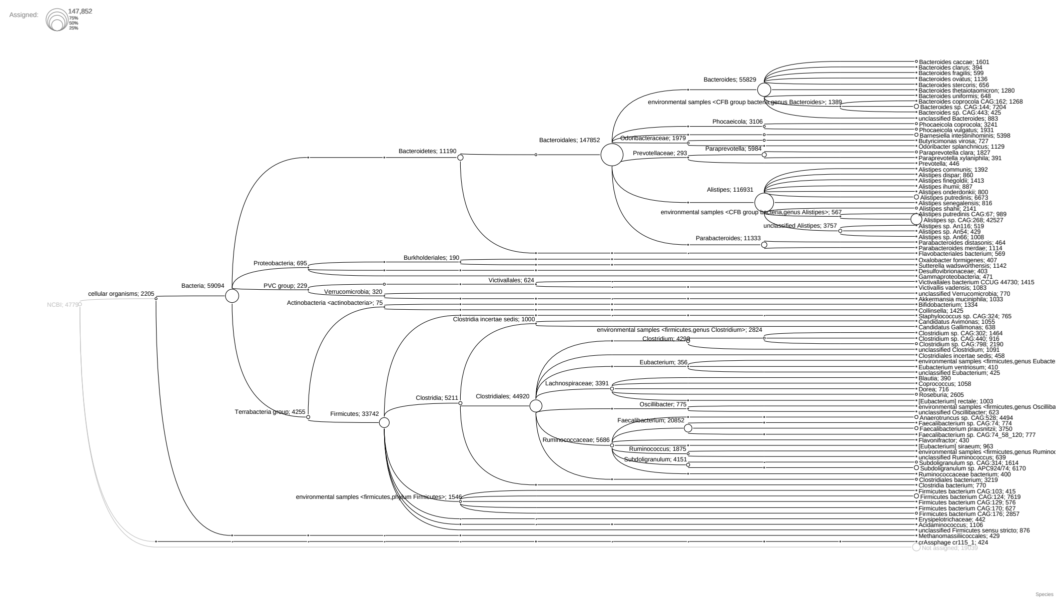


Figure 4.1: The MEGAN command used generate the taxonomy is available in the appendix A.1. The view displays all nodes of the taxonomy uncollapsed.

PYGAN follows a different paradigm. In contrast to MEGAN, it only provides a solution to a singular use case - an LCA analysis. Therefore, PYGAN's implementation is contained to the minimum amount of code necessary to run an LCA analysis. PYGAN does not have a graphical interface nor a command line interface. It has to be scripted. MEGAN offers parameters for the naive LCA analysis that PYGAN does not: *Min Score*, *Max Expected*, *Min Percent Identity*, and *Min Read Length*. PYGAN does not refine the alignment data apart from using a top score filter.

4.2 LCA analysis

An LCA analysis each was performed on the data set *Alice01-1mio-Jan-2021.daa*. The parameters for the MEGAN run are shown in appendix A.2 and the parameters for the PYGAN run in appendix A.3. To offset the lack of additional parameter options in PYGAN, the corresponding parameters in the MEGAN analysis were left empty. The results of the analyses were exported to a tabulated plain text file containing the nodes of the alignment taxonomies and their number of mapped reads. The output was analyzed with a script as seen in appendix A.4.

Taxon	Rank	PYGAN	MEGAN	Factor
Alistipes putredinis	Species	1465	6673	21%
Alistipes shahii	Species	490	2141	22%
Barnesiella intestinihominis	Species	1250	5398	23%
cellular organisms	-	5026	2205	227%
Bacteroidetes/Chlorobi group	-	466	196	237%
PVC group	-	555	229	242%
NCBI	-	35314	4779	738%

Table 4.1: Part of the results of LCA analyses performed according to appendix A.3, A.2, A.4. Shows the number of reads mapped to a taxon by PYGAN/MEGAN and the factor of PYGAN to MEGAN.

PYGAN produced an alignment taxonomy with 126 nodes, whereas MEGAN produced an alignment taxonomy with 201 nodes. Every node of the PYGAN taxonomy was included in the MEGAN taxonomy. 32562 out of 766831 (4.25%) reads in the MEGAN taxonomy were mapped to taxons not included in the PYGAN taxonomy. The majority of nodes in the PYGAN taxonomy have 70% to 130% as many reads as the corresponding nodes in the MEGAN taxonomy. Notable differences are listed in table 4.1. Such include Alistipes putredinis (21%), Alistipes shahii (22%), and Barnesiella intestinihominis (23%). These taxons all have the low-level rank of species. The other side of the spectrum contains PVC group (242%), Bacteroidetes/Chlorobi group (237%), and cellular organisms (227%). These are taxons of upper ranks. In this analysis, PYGAN mapped reads to higher ranking nodes more often than

MEGAN. While the difference is of greater magnitude in lower ranks, the number of reads mapped to the root node (NCBI) shows the greatest difference. PYGAN mapped 738% more reads to the root node than MEGAN.

4.3 Projection

The MEGAN projection was performed with the same settings as in tables 3.1, 3.2. Adjusted to MEGAN's inflation of nodes of PYGAN, the result is similar to PYGAN's proportional projection as seen in table 4.2. The difference is that MEGAN has only one failed node below the target rank and zero failed reads above the target rank.

	↑TR	→TR	↓TR
Nodes	499	553	1
Reads	0	701047	19039

Table 4.2: Continuation of table 3.1. Applied to the first data set *Alice01-1mio-Jan-2021*.

4.4 Discussion

The implementation of the LCA algorithm in PYGAN is identical to the implementation in MEGAN. One reason why the is such a difference in the number of nodes and mapped reads is the different minimum support filter implementation. MEGAN stops the upwards-propagation of reads as soon as a node satisfies the limit. On the other hand, PYGAN keeps applying the filter to ancestors of satisfactory nodes. PYGAN's implementation of the minimum support filter could be tweaked towards MEGAN's implementation in the future if so desired.

However, the reads are still not mapped equally when the minimum support filter is turned off. The disparity barely changes. Another explanation could be a difference in the processing of the alignment data. The trend that PYGAN maps to nodes of higher ranks more often than MEGAN suggests a difference in the quality of alignment data ultimately used as an input. When a more widely distributed set of accessions is used, the LCA will naturally be of higher rank. Furthermore, the issue could arise from differences in rules of which reads or nodes to display in the taxonomy.

The result of the projection in table 4.2 suggests different handling of problematic nodes and reads. There is a single node regarded as below the target rank that contains every read failed by the MEGAN projection. In contrast, PYGAN does not move failed reads to a collective failure node but rather does not move failed reads at all. Considering that difference, PYGAN's proportional projection implements MEGAN's naive projection faithfully. The same discussion about projection methods in chapter 3 applies to MEGAN.

Due to the conceptual differences of MEGAN and PYGAN, it is insensible to compare performances. Even though only the same functionality was tested, MEGAN comes with a massive overhead. A meaningful comparison of performance to other programming languages can only be drawn when a comparable set of instructions is surveyed. PYGAN completes a full LCA analysis in 110 s on average with hardware detailed in table 2.1. About 16 s are needed to parse the taxonomy data. Roughly the same time is needed to perform the algorithms. At least 80 s out of the total 110 s are spent loading data from the hard drive.

4.5 Conclusion

MEGAN impresses with a large variety of functionalities and customization compared to PYGAN. The graphical interface provides flexibility that can not be achieved by a scriptable module. On the other hand, MEGAN's graphical interface is not relevant when the tool is integrated into a process. Executing MEGAN as a CLI levels the playing field for PYGAN. In that use case, PYGAN provides a comparable experience to MEGAN for LCA analysis.

The implementation in Python allows for the sharing of analysis scripts, easy access over PyPi, and comfortable expansion of the code. However, the same comfort could be achieved by implementing an interface to the MEGAN CLI in Python. It would alleviate the issues with technical re-implementation details. Another option would be to implement MEGAN in C/C++ and add a layer of Python on top of that. The seamless communication between C/C++ and Python would omit the need to communicate over a CLI. Furthermore, this would ease the bottleneck PYGAN has with loading data from the hard drive.

Chapter 5

Outlook

5.1 Improvements on PYGAN

PYGAN has multiple points to be improved upon. It would be beneficial to increase its modularity. Since PYGAN is compartmentalized, some of its components are fit to be independent modules themselves. Splitting off the phylogenetic tree data structure and turning it into a standardized library would greatly increase its accessibility to other tools. With a standardized interface, a collaboration between different groups would become seamless, because it obviates the need to parse data from various formats. Such an independent library could then be extended to contain more than just taxonomy information, i.e. support functional profiling approaches including KEGG, eggNOG, and SEED to include more aspects of MEGAN [14]. It could remain a scriptable module and or be potentially extended to a CLI application or to support visual desktop applications.

Another independent module could be created to provide taxonomy trees and mappings and information from the MEGAN Map database in a single package. The NCBI and GTDB taxonomy trees as well as mappings could be included in the MEGAN Map database by adding a new table for each type. Table 5.1 illustrates how each taxonomy entry could include its ID, the ID of its parent node, scientific name, and rank. The NCBI table for this would contain approximately 2.3 million entries, but generating the internal tree structure and populating it with information would no longer require parsing two separate plain text files. With a modified database, only a single file would have to be provided for the program, which in return would increase its usability. The interface to the database could be provided by a separate module to make the database more accessible to other tools.

Additionally to means of functional analysis, PYGAN could improve on its naive implementation of the LCA algorithm. More sophisticated variations include the weighted LCA algorithm that improves the specificity of taxonomic

Taxonomy ID	Parental ID	Scientific Name	Scientific Rank
645	642	Aeromonas salmonicida	Species
...	...	...	...
243007	243006	Aiolopus thalassinus	Species
...	...	...	...
2115845	61319	Brachyalestes	Genus

Table 5.1: Example table that could be added to the MEGAN Map database for NCBI and GTDB taxonomy data.

assignments. Another variation of the LCA algorithm could be implemented to process long reads more truthfully since the naive approach is most suitable for short reads only.

Currently, PYGAN only accepts alignment data in a tabulated format, which limits its usability. Alignment formats further include *pairwise*, *query-anchored*, *XML* and *CSV* [2]. To increase its usability, this tool could be extended to support more alignment output formats by implementing more parsers.

While PYGAN is not intended to support any visualization in its current design, it would benefit from further output customization. It could be expanded with an option to sort the output entries by their number of mapped reads. Additionally, an option to group entries by their ranks or an option to group entries by their paths in a depth-first or breadth-first approach could be implemented. The user may also desire to specify which format the results should be listed in. Choices could include tabular and comma-separated.

The current code itself is a subject of future optimization. Multiprocessing would theoretically be a great way to drastically improve the performance because many steps are independent of each other. However, the need to pass large amounts of data between processes currently makes it impractical and does not save much time. Instead, it should be considered to make use of Python's seamless C/C++ integrability. This way, computing-intensive steps could be implemented in more performant languages, while all of the usability comforts Python provides are retained.

To increase PYGAN's accessibility it should be considered to expand it to be more than a scriptable module. An additional CLI would allow users to use it as a stand-alone program as Python allows the execution of modules. This would benefit users with little Python experience because they would not need to write any code to use PYGAN. Furthermore, it would provide a robust interface for other tools or integration into a pipeline, as it could be executed by any program with access to the system's command line.

5.2 Future of Python in Bioinformatics

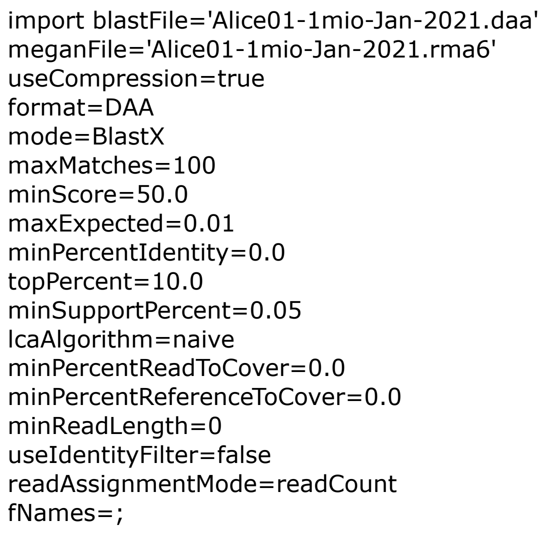
Scientific computing is a highly competitive environment for programming languages. Python has to improve to maintain its relevance in the field. The glaring performance issues in comparison to compiled languages are a significant factor [10]. Guido van Rossum, the founder of Python, revealed in May 2021 that the Python Software Foundation aims to increase the performance of Python by a factor of 5 [32]. While more efficient Python interpreters exist, CPython is most commonly used [29]. The improvements specifically target CPython. Funding for the development has been secured and improvements are expected to be released in the following years.

The integration of C, C++, and Fortran libraries is one strategy used to mend Python's performance issues. As an alternative to accelerating parts of libraries in a C-based language, more modern alternatives could be employed. One such possibility is Rust, a systems programming language that has seen its first stable release in 2015 [30]. Rust offers a novel approach to systems programming by providing low-level performance characteristics, as usually seen in languages of C and C++, as well as providing memory safety and high-level abstractions [31].

The integration of other programming languages in Python programs is an attempt to bridge the dilemma of accessibility versus performance. The elegance of Python's solution is debatable because employing code in a different language undermines the accessibility Python aims to provide in the first place. A better approach would be to explore Python-native solutions to the dilemma. Julia is a programming language that competes with the same merits as Python [26]. However, it comes significantly closer in performance to C than Python. Therefore, it omits the need to work around the accessibility versus performance dilemma. The size of Python's community secures the language's longevity, but the rise in popularity of competitors like Julia could impact the preferences of computational scientists.

Appendix A

Further Tables and Figures



```
import blastFile='Alice01-1mio-Jan-2021.daa'
meganFile='Alice01-1mio-Jan-2021.rma6'
useCompression=true
format=DAA
mode=BlastX
maxMatches=100
minScore=50.0
maxExpected=0.01
minPercentIdentity=0.0
topPercent=10.0
minSupportPercent=0.05
lcaAlgorithm=naive
minPercentReadToCover=0.0
minPercentReferenceToCover=0.0
minReadLength=0
useIdentityFilter=false
readAssignmentMode=readCount
fNames=;
```

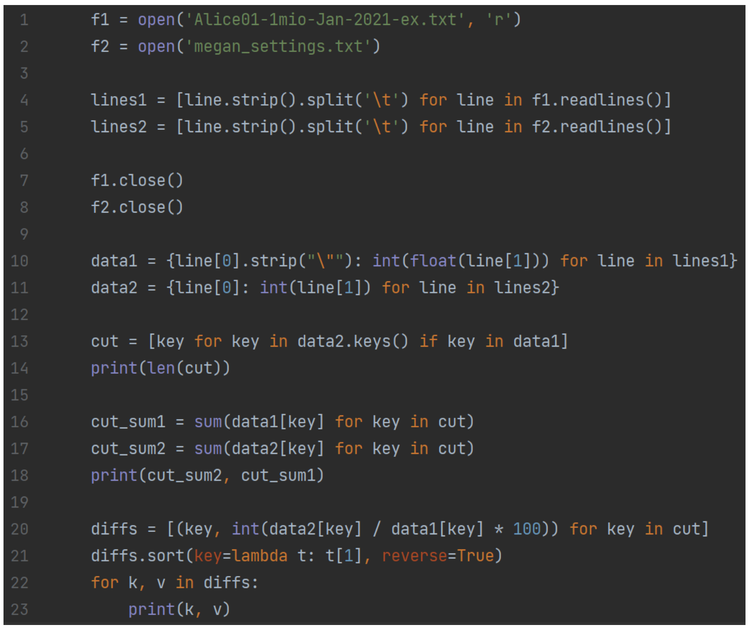
Figure A.1: MEGAN command used to generate taxonomy as seen in figure 4.1.

```
minScore=0.0
maxExpected=100000.0
minPercentIdentity=0.0
topPercent=10.0
minSupportPercent=0.0
minSupport=383
lcaAlgorithm=naive
lcaCoveragePercent=100.0
minPercentReadToCover=0.0
minPercentReferenceToCover=0.0
minReadLength=0
longReads=false
pairedReads=false
useIdentityFilter=false
readAssignmentMode=readCount
fNames=;
```

Figure A.2: MEGAN command used for the LCA algorithm.

```
tre_file = 'resources/ncbi.tre'
map_file = 'resources/ncbi.map'
megan_map_file = 'resources/megan-map-Jan2021.db'
blast_file = 'daafiles/Alice01-1mio-Jan-2021.txt'
blast_map = {'qseqid': 0, 'sseqid': 1, 'bitscore': 2}
top_score_percent = 0.1
db_segment_size = 10000
db_key = 'Taxonomy'
ignore_ancestors = False
min_support = 383
only_major = False
exclude = []
project_mode = ''
project_rank = ''
cluster_degree = 0
out_file = 'megan_comparison.txt'
prefix_rank = False
show_path = False
list_reads = False
```

Figure A.3: PYGAN parameters used to imitate the MEGAN parameters for the LCA algorithm from figure A.2.



```python
f1 = open('Alice01-1mio-Jan-2021-ex.txt', 'r')
f2 = open('megan_settings.txt')

lines1 = [line.strip().split('\t') for line in f1.readlines()]
lines2 = [line.strip().split('\t') for line in f2.readlines()]

f1.close()
f2.close()

data1 = {line[0].strip("\""): int(float(line[1])) for line in lines1}
data2 = {line[0]: int(line[1]) for line in lines2}

cut = [key for key in data2.keys() if key in data1]
print(len(cut))

cut_sum1 = sum(data1[key] for key in cut)
cut_sum2 = sum(data2[key] for key in cut)
print(cut_sum2, cut_sum1)

diffs = [(key, int(data2[key] / data1[key] * 100)) for key in cut]
diffs.sort(key=lambda t: t[1], reverse=True)
for k, v in diffs:
    print(k, v)
```

Figure A.4: Script to compare results from a PYGAN and a MEGAN taxonomic analysis.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. *SIAM Journal on computing*, 5(1):115–132, 1976.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[3] T. Bald, J. Barth, A. Niehues, M. Specht, M. Hippler, and C. Fufezan. pymzML—Python module for high-throughput bioinformatics on mass spectrometry data. *Bioinformatics*, 28(7):1052–1053, 02 2012.

[4] S. Bassi. A primer on python for life science researchers. *PLOS Computational Biology*, 3(11):1–6, 11 2007.

[5] S. Boyer, A. Dalke, and P. Bruneau. AstraZeneca uses python for collaborative drug discovery. https://www.python.org/about/success/astra/, 2021.

[6] F. P. Breitwieser, J. Lu, and S. L. Salzberg. A review of methods and databases for metagenomic classification and assembly. *Briefings in bioinformatics*, 20(4):1125–1136, 2019.

[7] B. Buchfink, C. Xie, and D. H. Huson. Fast and sensitive protein alignment using diamond. *Nature Methods*, 12(1):59–60, Jan 2015.

[8] P. J. A. Cock, T. Antao, J. T. Chang, B. A. Chapman, C. J. Cox, A. Dalke, I. Friedberg, T. Hamelryck, F. Kauff, B. Wilczynski, and M. J. L. de Hoon. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423, 03 2009.

[9] S. Federhen. The NCBI Taxonomy database. *Nucleic Acids Research*, 40(D1):D136–D143, 12 2011.

[10] M. Fourment and M. R. Gillings. A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics*, 9(1):82, Feb 2008.

[11] J. Handelsman. Metagenomics: Application of genomics to uncultured microorganisms. *Microbiology and Molecular Biology Reviews*, 68(4):669–685, 2004.

[12] D. H. Huson, B. Albrecht, C. Bağcı, I. Bessarab, A. Gorska, D. Jolic, and R. B. Williams. Megan-lr: new algorithms allow accurate binning and easy interactive exploration of metagenomic long reads and contigs. *Biology direct*, 13(1):1–17, 2018.

[13] D. H. Huson, A. F. Auch, J. Qi, and S. C. Schuster. MEGAN analysis of metagenomic data. *Genome research*, 17(3):377–386, 2007.

[14] D. H. Huson, S. Beier, I. Flade, A. Górska, M. El-Hadidi, S. Mitra, H.-J. Ruscheweyh, and R. Tappu. MEGAN community edition - interactive exploration and analysis of large-scale microbiome sequencing data. *PLOS Computational Biology*, 12(6):1–12, 06 2016.

[15] D. H. Huson, S. Mitra, H.-J. Ruscheweyh, N. Weber, and S. C. Schuster. Integrative analysis of environmental sequences using MEGAN4. *Genome research*, 21(9):1552–1560, 2011.

[16] T. Junier and E. M. Zdobnov. The newick utilities: high-throughput phylogenetic tree processing in the unix shell. *Bioinformatics*, 26(13):1669–1670, 2010.

[17] V. Kunin, A. Copeland, A. Lapidus, K. Mavromatis, and P. Hugenholtz. A bioinformatician's guide to metagenomics. *Microbiology and Molecular Biology Reviews*, 72(4):557–578, 2008.

[18] M. Madigan, J. Martinko, K. Bender, D. Buckley, and D. Stahl. *Brock biology of microorganisms, 14th ed.* Pearson, 2012.

[19] S. S. Mande, M. H. Mohammed, and T. S. Ghosh. Classification of metagenomic sequences: methods and challenges. *Briefings in bioinformatics*, 13(6):669–681, 2012.

[20] J. Mcneill, F. Barrie, W. Buck, V. Demoulin, W. Greuter, D. Hawksworth, P. Herendeen, S. Knapp, K. Marhold, J. Prado, W. Prud'homme van Reine, G. Smith, J. Wiersema, and N. Turland. *International Code of Nomenclature for algae, fungi and plants (Melbourne Code) adopted by the Eighteenth International Botanical Congress Melbourne, Australia, July 2011*, volume 154. Koeltz Botanical Books, 12 2012.

[21] P. C. Ng and E. F. Kirkness. *Whole Genome Sequencing*, pages 215–226. Humana Press, Totowa, NJ, 2010.

[22] C. I. of Technology. Flight software and embedded systems framework. https://nasa.github.io/fprime/, 2020.

[23] T. E. Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[24] T. E. Oliphant. Python for scientific computing. *Computing in Science Engineering*, 9(3):10–20, 2007.

[25] D. H. Parks, M. Chuvochina, P.-A. Chaumeil, C. Rinke, A. J. Mussig, and P. Hugenholtz. A complete domain-to-species taxonomy for bacteria and archaea. *Nature Biotechnology*, 38(9):1079–1086, Sep 2020.

[26] J. M. Perkel. Julia: come for the syntax, stay for the speed. *Nature*, 572(7767):141–142, 2019.

[27] Python Software Foundation. General python FAQ 3.9.6. https://docs.python.org/3/faq/general.html, 2021.

[28] C. Quast, E. Pruesse, P. Yilmaz, J. Gerken, T. Schweer, P. Yarza, J. Peplies, and F. O. Glöckner. The SILVA ribosomal RNA gene database project: improved data processing and web-based tools. *Nucleic Acids Research*, 41(D1):D590–D596, 11 2012.

[29] A. Roghult. Benchmarking python interpreters: Measuring performance of CPython, Cython, Jython and PyPy, 2016.

[30] Rust Core Team. Announcing rust 1.0. https://blog.rust-lang.org/2015/05/15/Rust-1.0.html, 2015.

[31] Rust Docs contributors. Safety and guarantees. https://doc.rust-lang.org/1.0.0/style/safety/README.html.

[32] G. van Rossum. [python-dev] re: Speeding up cpython. https://mail.python.org/archives/list/python-dev@python.org/message/WVURDCWRH7F5UDLU5ZLT5P35ZO6TIBYA/, 2021.

[33] J. C. Venter, K. Remington, J. F. Heidelberg, A. L. Halpern, D. Rusch, J. A. Eisen, D. Wu, I. Paulsen, K. E. Nelson, W. Nelson, D. E. Fouts, S. Levy, A. H. Knap, M. W. Lomas, K. Nealson, O. White, J. Peterson, J. Hoffman, R. Parsons, H. Baden-Tillson, C. Pfannkoch, Y.-H. Rogers, and H. O. Smith. Environmental genome shotgun sequencing of the sargasso sea. *Science*, 304(5667):66–74, 2004.

[34] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[35] J. C. Wooley, A. Godzik, and I. Friedberg. A primer on metagenomics. *PLOS Computational Biology*, 6(2):1–13, 02 2010.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum	Unterschrift