Stockholms universitet

Performance Optimization of Phylogenetic Tree Inferring Python Implementation Using C++

Mazen Mardini

Prestanda-optimering av Python-implementation som utför fylogenetisk träd-inferens med C++

Handledare: Lars Arvestad

Examinator: Marc Hellmuth

Inlämningsdatum: 2022-11-09

Kandidatuppsats i datalogi

Bachelor Thesis in Computer Science

Abstract

I have researched to what degree C++ could be used to help optimize a phylogenetic tree inferring tool that is currently implemented in Python. The tool is called **dnctree**. It is well known that there is a significant overhead involved in using CPython, the reference interpreter of the Python language. Luckily, there are many ways to optimize Python code. We will especially focus on a central strategy called vectorization. I have also integrated a distance estimation tool, called **PaHMM-Tree** that is written in C++ into dnctree. This was done by turning PaHMM-Tree into a Python library, which allowed for easier interfacing and integration into the existing dnctree Python code. Lastly, I have re-implemented Neighbor Joining (NJ) from dnctree in C++, and compared its performance with dnctree. The C++ implementation showed promising results. This, among other things, has lead to the conclusion that C++ can significantly boost the performance of dnctree in ways that would be very difficult, if not outright impossible to achieve with Python+NumPy.

Sammanfattning

Jag har undersökt till vilken grad C++ can användas för att optimera ett verktyg för inferens av fylogenetiska träd som är implementerat i Python. Verktyget heter **dnctree**. Det är väl känt att det finns betydliga prestandakostnader involverade i användningen av CPython som är referensimplementationen av språket Python. Som tur är finns det många sätt att optimera Pythonkod. Vi kommer speciellt att fokusera på en central strategi som heter vektorisering. Jag har också integrerat ett avståndsestimeringsverktyg som heter **PaHMM-Tree** och som är skrivet i C++, med dnctree. Detta gjordes genom att omvandla PaHMM-Tree till ett Python-bibliotek, vilket möjliggjorde enkel anslutning och integrering med dnctrees existerande Pythonkod. Slutligen har jag skrivit om Neighbor Joining (NJ) från dnctree i C++, och jämfört dess prestanda med dnctree. C++-implementationen visade lovande resultat. Detta, bland annat, har lett till slutsatsen att C++ kan betydligt öka prestandan hos dnctree på sätt som hade varit väldigt svåra, om inte omöjliga att uppnå med Python+NumPy.

Contents

1. Acknowledgments

I would like to thank my supervisor Lars Arvestad for helping me in choosing a thesis subject, as well as his invaluable support while I have been working on this thesis. I would also like to thank my family for their support and patience during my studies, including this work.

2. Introduction

2.1. A Computational Problem in the Field of Biology

In biology, scientists are interested in finding evolutionary relationships between organisms. An evolutionary common ancestry between organisms is established by finding a set of similarities between them called *homologies*. They could be for example structural or behavioral features. Comparing more than two organisms with each other, one could create a so called phylogenetic tree that provides a visualization of the genetic relationship between the organisms. See Figure 2.1 for an example of a phylogenetic tree.

In earlier studies of homologies, one would look at morphological characters such as bone-structures [21]. Today, homologies are also looked for at a molecular level, in the organisms' genome as well as in proteins. This has helped researchers by providing a larger amount of independent characters. Homologies at this level may refer to structural or functional relationships [4]. These genomes or proteins are oftentimes large molecular sequences of nucleotides or amino-acids.

Homologies, which are found through molecular similarities between sequences, can be identified by aligning two or more sequences with each other. This used to be performed by hand. But as the quantity of such sequences grew, there became an increasing need for automation of the process. Thereby a wide variety of algorithms have been introduced to solve this problem. Commonly, these algorithms are based on some kind of Multiple Sequence Alignment (MSA) where more than two sequences are aligned together based on a scoring system. Sadly, computing such an MSA is an NP-complete problem [26], meaning any optimal solver would scale very badly [4]. Therefore, heuristics and approximative methods are employed instead. The result of this estimated MSA can then be used to estimate the distances between each pair of sequences. These distances are in turn used to build a phylogenetic tree by using an algorithm such as Neighbor Joining (NJ) or variations thereof [19]. This latter step is computationally taxing as well. For example, with NJ a correct tree can be constructed in cubic time [29], if the distances are correct enough [1]. There are also so called "alignment-free" methods for approximating phylogenetic trees, in which case distances between sequences are determined using some simpler similarity measures rather than aligning the sequences. A tool called *PaHMM-Tree* employs such a method, a tool which we will be talking more about in the coming sections. I will also show how it can be integrated with a scalable tree construction tool called *dnctree*. By doing so I am hoping that we can get a phylogenetic tree estimation solution that has good performance and accuracy.

2.2. Estimating Phylogenetic Trees

Constructing phylogenetic trees, to help us visualize evolutionary history between genomes or proteins, can be done in many different ways. A common strategy is to use distance-based methods. Generally, these comprise of three steps:

1. Calculate a multiple sequence alignment (MSA).
2. Calculate all distances d_{xy} between each pair of sequences x and y given the alignment.
3. Use the pair-wise distances to construct the tree, most commonly this will be done using Neighbor Joining (NJ). [4]

The first and second steps are central to the whole process, and are also

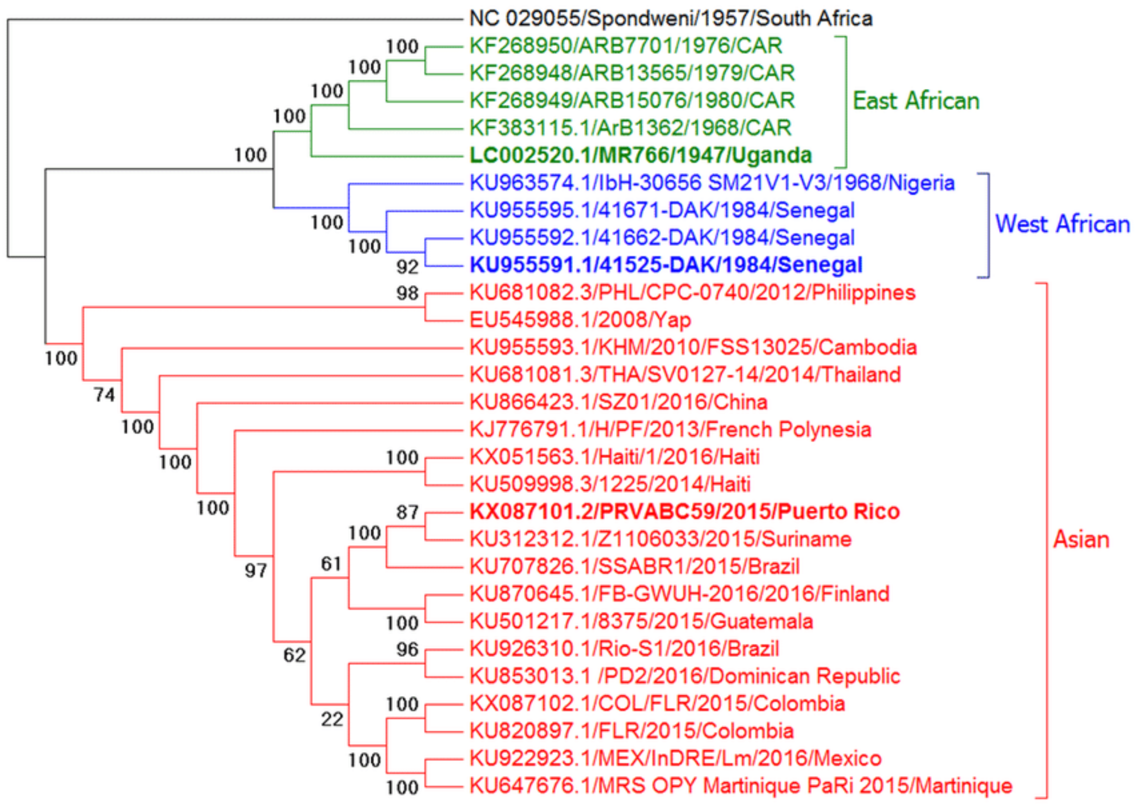
NC 029055/Spondweni/1957/South Africa
KF268950/ARB7701/1976/CAR
KF268948/ARB13565/1979/CAR
KF268949/ARB15076/1980/CAR
KF383115.1/ArB1362/1968/CAR
LC002520.1/MR766/1947/Uganda
East African
KU963574.1/IbH-30656 SM21V1-V3/1968/Nigeria
KU955595.1/41671-DAK/1984/Senegal
KU955592.1/41662-DAK/1984/Senegal
KU955591.1/41525-DAK/1984/Senegal
West African
KU681082.3/PHL/CPC-0740/2012/Philippines
EU545988.1/2008/Yap
KU955593.1/KHM/2010/FSS13025/Cambodia
KU681081.3/THA/SV0127-14/2014/Thailand
KU866423.1/SZ01/2016/China
KJ776791.1/H/PF/2013/French Polynesia
KX051563.1/Haiti/1/2016/Haiti
KU509998.3/1225/2014/Haiti
KX087101.2/PRVABC59/2015/Puerto Rico
KU312312.1/Z1106033/2015/Suriname
KU707826.1/SSABR1/2015/Brazil
KU870645.1/FB-GWUH-2016/2016/Finland
KU501217.1/8375/2015/Guatemala
KU926310.1/Rio-S1/2016/Brazil
KU853013.1 /PD2/2016/Dominican Republic
KX087102.1/COL/FLR/2015/Colombia
KU820897.1/FLR/2015/Colombia
KU922923.1/MEX/InDRE/Lm/2016/Mexico
KU647676.1/MRS OPY Martinique PaRi 2015/Martinique
Asian

Figure 2.1.: Phylogenetic tree of 29 samples of the Zika Virus constructed through a distance-based method and Neighbor Joining. *This diagram is distributed under the CC BY 4.0 license through the PLOS Neglected Tropical Diseases journal by Weger-Lucarelli et al. Copyright © 2016 Weger-Lucarelli et al.* [27]

together the hardest problem to solve due to various pitfalls in attempting to estimate proper distances, such as the introduction of biases. They are also many times the most computationally intensive, especially when the lengths of the sequences are much larger than their number. In the case of PaHMM-Tree, the distance estimation is an $O(N^2L)$ algorithm where N is the number of sequences and L is the length of each sequence, assuming they have similar lengths [6].

The third step is (usually) Neighbor Joining, an $O(N^3)$ algorithm [19]. There exist improved variations of NJ with seemingly better scalability than the original algorithm, for example Relaxed Neighbor Joining (RNJ) which has a typical time complexity of $O(N^2 log(N))$ "without any significant qualitative difference in output" [18].

2.3. The Goal of This Thesis

Marcin Mogusz and Simon Whelan from Uppsala University (UU) introduced in 2016 a novel distance estimator called PaHMM-Tree (pronounced "palm-tree") which is based on an HMM-model. It uses a statistical method to calculate pair-wise distances between sequences. The reference implementation is done in C++ and is released as open-source software under the GPL license. Its input is a set of sequences in the FASTA format [5], and its output is a corresponding distance matrix. PaHMM-Tree has shown promising performance in terms of accuracy when compared against other distance inference methods such as MAFFT and PRANK [6]. The tool serves as an excellent candidate to fulfill the role in the second step described above, which is to reliably generate pair-wise distances. The first step is not needed when using PaHMM-Tree, because it is employing an alignment-free method.

Lars Arvestad from Stockholm University (SU) has developed a tool called dnctree which can perform distance-based tree building faster by not always requiring every single distance pair. Because of this, not every single pair of sequences has to be compared, and we save some execution time. It is written in Python. Its input is a set of sequences in various formats (including FASTA), and its output is an estimated phylogenetic tree in Newick format (a machine-readable string). Dnctree is not open-sourced, but I do have access to it with permission from Arvestad. More specifically, the version that I am going to focus on is from July 2020. This tool is very well suited for performing the third step of the tree building process, which is to construct the tree given the (required) set of distances. No formal complexity analysis has been done on this tool, but with experimentation we will see that it seems to have significantly better scalability than NJ.

My first task is to integrate PaHMM-Tree into dnctree to unite their advantages. There are several problems with regards to interoperability that have to be solved:

- PaHMM-Tree is written in C++, and there needs to be a way to make its distance-calculating functionality accessible from Python, which is the language that dnctree is written in.
- The tree building part of dnctree requires the ability to select specific distances. Always giving it all the distances like the PaHMM-Tree tool does would defeat the whole point of dnctree's tree estimation algorithm.
- Solving the two points above might require some modifications to PaHMM-Tree's source code. We need a sufficiently reliable way to test these modifications to make sure that the functionality remain unaffected.

My second task is to answer the following **research questions**:

- **(RQ1)** The user must feed the tool sometimes large inputs of data, and at the end receive a phylogenetic tree as output. Should the reading of sequence input and writing of tree output be done in Python or C++?
- **(RQ2)** What are the performance benefits of using C++ for central calculations (the tree building and distance calculations) rather than using Python+NumPy?
- **(RQ3)** Is it possible to estimate how much faster dnctree would have been if it was implemented entirely in C++ rather than Python?

By answering these questions we will be comparing C++ and Python in terms of computational performance, the focus will be mostly on execution time. But I will touch upon the subject of memory consumption and management as well, as this is an important aspect of computations that involve larger amounts of data, as phylogenetic tree building sometimes does. But before that, we will take a closer look at Neighbor Joining to gain a better understanding of what it really does.

2.4. Neighbor Joining

Just to illustrate what Neighbor Joining (NJ) is and how it works, I will dedicate this section to explain the algorithm.

Assume that there is a weighted tree representing the true evolutionary relationship between a set organisms. The weights represent evolutionary distances between connected organisms in the tree. Let us say that the leaves are organisms from which we have sampled genetic material, and the other nodes are common ancestors for their respective child nodes. From this weighted tree we can construct a distance matrix D with the size $N \mathrm{x} N$ where N is the number of leaves. Each element d_{xy} represent the evolutionary distance between the organisms x and y. Neighbor Joining is able to reconstruct this tree from D, even if the distance matrix has small errors [1]. We are hoping that by using a distance matrix, which we have computed with the help of an alignment based method (or otherwise), estimate a phylogenetic tree with NJ that corresponds with reality.

Neighbor Joining is an iterative algorithm. We keep track of all nodes considered for joining and join two nodes on each iterative step by introducing an auxiliary node which connects the pair, see **(b)** in Figure 2.2. Below we will go through each step in the algorithm.

(Step 0) Let us call the initial set of nodes that are considered for joining K_0. This set refers to the leaves of our tree, or rather the set of genomes that we have the pair-wise distances (D) for. Therefore, $|K_0| = N$.

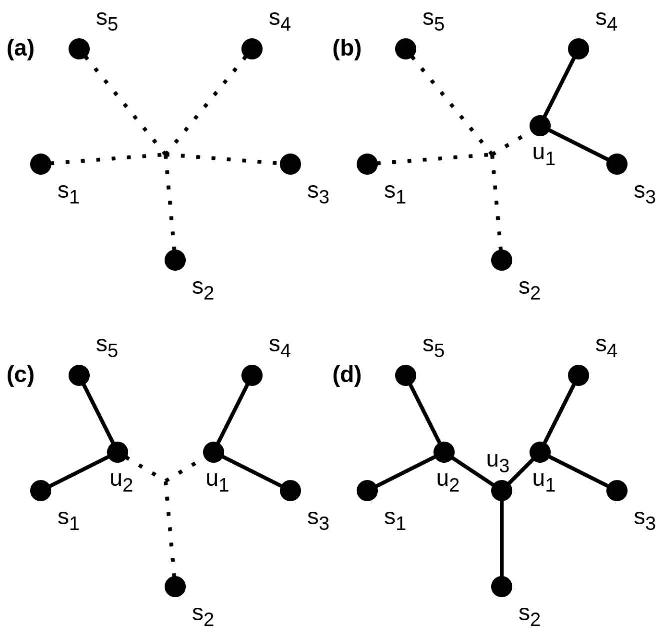

Figure 2.2.: Neighbor joining done in three steps. **(a)** Initially, the phylogenetic tree is completely unresolved and all nodes can be seen as joined in a star-shaped topology. **(b)** Taking one step with the algorithm joins s_3 and s_4 by introducing an auxiliary node u_1. Only nodes connected by dotted lines will be considered in the coming steps. **(c)** s_1 and s_5 are joined. **(d)** When there are only 3 nodes left, they are joined together by a final auxiliary node, in this example we call it u_3. Notice that the final tree is *unrooted*.

(Step 1) Next, we need to begin joining a pair of nodes from K_i. To choose which nodes to join we use the following formula [24]:

$$\operatorname*{argmin}_{x,y \in K_i} \left[(|K_i| - 2) d_{xy} - \sum_{k \in K_i} d_{xk} - \sum_{k \in K_i} d_{yk} \right] \tag{2.1}$$

The index i refers to the iteration step. Initially, $i = 0$.

After choosing our pair (x, y) we introduce an auxiliary node u and connect x and y to it. Then we replace these two nodes with the auxiliary node in K_i, in other words we get $K_{i+1} = (K_i \backslash \{x, y\}) \cup \{u\}$. Notice that $|K_{i+1}| = |K_i| - 1$.

(Step 2) After having introduced the auxiliary node u, we calculate the distances between u and each other node in K_{i+1} using [24]:

$$d_{uk} = \frac{1}{2}(d_{xk} + d_{yk} - d_{xy}) \quad \text{for} \quad k \neq x, y \tag{2.2}$$

If $|K_{i+1}| = 3$ we proceed to step 3, otherwise we go back to step 1 and continue the algorithm.

(Step 3) The last step is to connect the last three nodes in K_{i+1}. This is done by introducing a final auxiliary node and connecting it with the remaining nodes. See **(d)** in Figure 2.2. If we do not care about the distances of all edges in the tree, we can consider ourselves done at this point.

2.4.1. Dnctree

A shortcoming of Neighbor Joining is the requirement of having to utilize every single pair-wise distance in D. Since distance estimation can be relatively expensive, it would help to minimize the number of distances needed during the tree estimation process. The dnctree algorithm is an improvement over NJ in this regard. It uses a recursive randomized algorithm that is able to omit some distance computations when estimating trees from a larger number of sequences. Each recursion step splits the problem into three sub-problems. Eventually, a recursion step will find that the number of nodes to join are ≤ 100 and will use the NJ algorithm to create a sub-tree of these nodes. In other words, dnctree does use NJ, but only as the base case of its recursive algorithm. After each of the three recursive calls have been made in a recursion step, the resulting sub-trees from each call are joined together with the help of an auxiliary node.

Note that the resulting tree in NJ ends up being unrooted. In the case of dnctree, the same is also true, but the tree is presented as having the final auxiliary node as its root in the output.

2.5. The C++ Programming Language

C++ is a multi-paradigm, compiled and statically typed programming language. It allows for manual memory management and has no Garbage Collection (GC). It features classes, function overloading, operator overloading, templates (aka generics) and more. The standard library is extensive but mostly features general-purpose routines, such as for threading, string manipulation, various common data structures, file manipulation and a number of others. Its primary strength is that it is close to the hardware, yet rich in features.

When a language is *statically typed* it means that all types must be deducible while the program is being compiled (compile time). Languages that do not have this property are commonly referred to as *dynamically typed*, which implies that in general the program must be executed before all types can be deduced, and thus inhibit certain optimizations that could have been done to enhance performance.

To get an idea of how C++ code is compiled to an executable binary, see Figure 2.3.

The language allows you to *manually manage memory*, which means that you can allocate space in the heap and in the stack as you see fit, and you are responsible for not accessing any addresses outside of that space. It also means that you are responsible for freeing space in the heap once you are done with it. Data in memory can be interpreted however you like, a string, an array of integers, or whatever other predefined or custom data types. This freedom allows you to manage your resources in detail and gives you the ability to perform very interesting optimizations. It is worth noting that there are caveats to these freedoms. There is an increased risk of introducing faulty behavior and security vulnerabilities, like buffer overflows or memory leaks.

By following best-practices such as using smart pointers rather than raw pointers, and using containers from the standard library whenever possible,

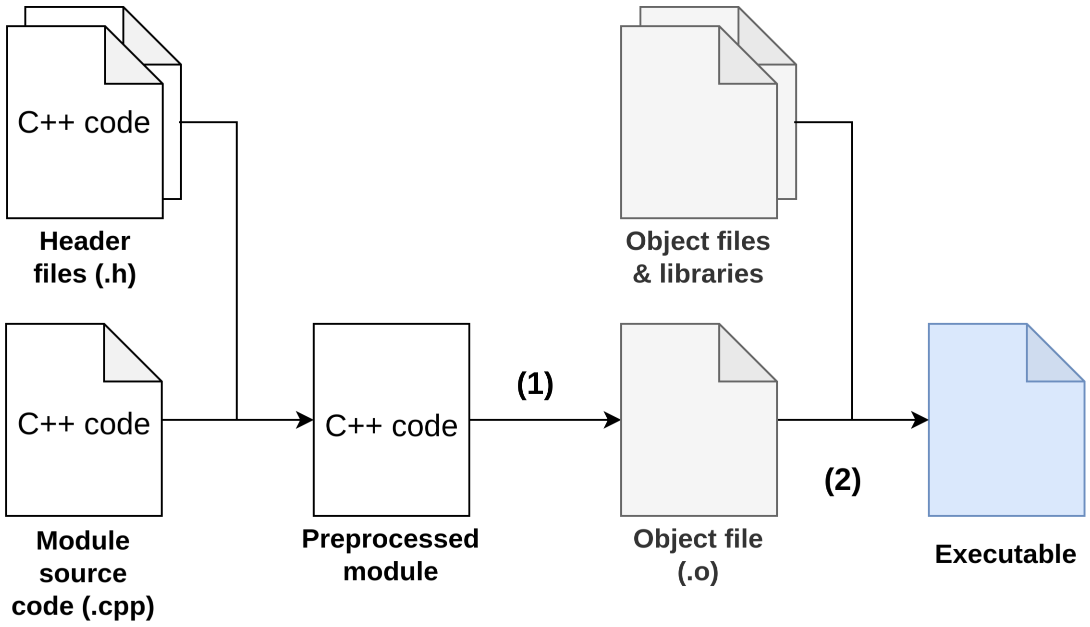

Figure 2.3.: A diagram that summarizes how C++ code is compiled into an executable binary. The code is usually split into source files (.cpp) that represent modules and header files (.h) that contain supplementary code. Both types of files contain C++ code. Preprocessing has to happen to merge the supplementary header files with the modules before compilation. The marked steps are **(1)** compilation of source code into unlinked machine code, and **(2)** linking of several object files (and libraries) into an executable.

many such errors can be mitigated.

There are very mature compilers available such as GCC and LLVM that are able to perform a number of optimizations automatically. Examples are constant folding, constant propagation, dead code removal, loop invariant code movement and tail call optimizations [16]. You also have the ability to use very low-level routines such as ones that invoke SIMD instructions in your CPU to perform many operations at once [11]. This allows you to further optimize your code in ways that the compiler cannot do on its own. You have fine-grained control over which optimizations to apply, not only because of the language and the libraries that are available, but also because of the advanced selection of compiler flags that we have at our disposal. It is possible to stick to the defaults by omitting any optimization related flags, or enable a whole set of optimizations through the "-O" flag, or individual optimizations like e.g. “-finline-functions” that lets the compiler consider all functions for inlining [10]. In C++, functions can be *inlined*, this means that the function body is "placed" where the function has been called, so no actual call happens. Rather, the function body is executed as if the body was located where the function had been called. This eliminates some overhead that is intrinsic to function invocation.

Much can be said about C++ and its vast set of features, libraries and tooling. The most important thing to understand is that this is a low-level language, it gives the developer much control over how to optimize their software but at the cost of an increased responsibility from the developer's side.

2.6. The Python Programming Language

Python is easy to use, has a very rich development ecosystem and provides us with a solid foundation to work with through a well-designed and well-documented standard library [14], and a helpful set of coding standards [23]. However, when it comes to performance of *pure* Python code, a lot is left to be desired. This does not mean that we cannot write fast Python programs, indeed we can! We will discuss the probably most effective and common way to do so, which is to offload much of the computational burden unto fast compiled libraries. But before that, we will go through some details about Python as a language and how performance is affected, then we will start looking at how vectorization with the Python library NumPy can help us in this regard.

Python is a multi-paradigm, interpreted dynamically typed programming language. The most commonly used interpreter is called CPython, this is also the reference implementation of Python. This implementation uses reference counting based garbage collection. The language features classes, introspection, reflection, operator overloading, and more.

From now on, when talking about Python I will implicitly refer to CPython *(version 3), unless specified otherwise.*

Unlike C++, Python takes care of memory management automatically by using reference counting. This means that whenever an object is referenced, a counter for said object will be incremented, and when the object is forgotten (seizes to be referenced) somewhere in the code, the counter is decremented. When the object is finally no longer referenced anywhere, this unused object is cleaned up from memory (garbage is collected). Python controls allocation and freeing of memory tightly, and therefore leaves little room for segmentation faults, buffer overflows, double freeing, and a myriad of other faulty behaviors that can be wrongfully implemented with lower-level languages.

2.6.1. Python's dynamism and performance

Python is a highly dynamic language in the sense that a program's structure can be altered at runtime in more ways than usually possible. To illustrate this, let us start with a simple example:

```
def my_function ( value = 0 ):
        my_function.__defaults__ = (my_function.__defaults__[0] + 1,)
        print(value)

my_function()        # Prints: 0
my_function()        # Prints: 1
my_function()        # Prints: 2
my_function("cat")   # Prints: cat
```

Listing 1: Python function that modifies its own argument's default value.

In Python, functions are objects, they have a type "function", and have their own attributes. One such attribute is `__defaults__`, a tuple which houses all argument default values of a function, and yes, as seen in the code above you can set that attribute to something else at runtime. In general, neither the type of an argument, nor its default value (if it exists) can be accurately deduced before actually running the code.

Let us look at an even more important example involving classes:

```
from random import randint

class Triangle:
        def __init__(self, point1, point2, point3):
                self.point1 = point1
                self.point2 = point2
                self.point3 = point3

random_triangles = []  # Create an empty list

# Fill the list with 100 random triangles
for i in range(100):
        p1 = (randint(-5, 5), randint(-5, 5))
        p2 = (randint(-5, 5), randint(-5, 5))
        p3 = (randint(-5, 5), randint(-5, 5))
        random_triangles.append(Triangle(p1, p2, p3))
```

Listing 2: Creates a list of Triangle-objects. Because of how the constructor is defined, Triangles are always initialized with 3 points as their attributes.

In this example, we create our own data type (class) called "Triangle" composed of 3 points. Each point is a tuple with 2 integers representing a position in a 2D coordinate system. If we want to iterate over all the triangles in `random_triangles` to perform some computation using their attributes, we will suffer many cache misses. These cache misses happen when the processor is unable to locate data in the CPU cache, and has to consult lower-level cache/memory such as RAM which results in higher latencies. When data is fetched from RAM, it is transferred to the cache in so called cache lines. These lines are sequential blocks of data that contain what was requested, and more [17]. This is why sequential reads from RAM are faster than random reads, because we utilize more of already loaded cache lines. The reason why we will suffer from many cache misses with the aforementioned list is because the triangles (and their attributes) are not located sequentially in memory, and cannot ever be located sequentially due to Python objects' dynamic nature. You could for example expand an object like this:

```
random_triangles[49].color = 0xFFFFFF
```

Listing 3: Give the 50th triangle a color attribute and assign the value 0xFFFFFF to it.

The flexibility seen in Listing 3 is incompatible with having the objects placed sequentially, because the objects can have variable sizes and can be expanded arbitrarily at runtime. Therefore, the objects must be stored non-sequentially, and that is in fact what CPython's list data structure does. Internally, a list holds an array of `PyObject` pointers. When the list is populated with elements, the pointers will point at `PyObjects` (Python objects) that reside at arbitrary locations in the heap. In Python 3.10.4, a list is defined approximately as follows:

```
struct PyListObject {
        // Lists are Python objects as well,
        // so they inherit the same properties:
        PyObject ob_base;
        // The internal array where pointers to all elements
        // are stored:
        PyObject **ob_item;
        // The number of elements in ob_item:
        Py_ssize_t ob_size;
        // The actual size of ob_item (could be larger than ob_size):
        Py_ssize_t allocated;
};
```

Listing 4: An approximative definition of `PyListObject` which represents list-objects internally in the CPython interpreter. Comments have been added for clarity. For the actual definition see [12] and [13].

Both Listing 1 and Listing 3 above are examples of reflection, the ability of a program to introspect and structurally modify itself at runtime. Apart from Python's dynamic typing and reflection capabilities causing poor CPU cache utilization, important optimizations such as the use of SIMD become difficult, if not outright impossible to perform reliably. In fact, **type related optimizations are completely absent in CPython (version 3.9)** [30]. Even though Python has support for type annotations, these are not mandatory and are only there to for example help development tools perform static analysis [15], they do not help the interpreter in optimizing code at all.

2.6.2. Other noteworthy expenses in Python

Relevant when speaking about the performance penalties in Python are the following expenses:

- The reference counter itself has been shown to slow down execution. This could possibly be due to cache misses [30].
- Function calls have a significant overhead [30].
- Using global variables from a function can be significantly slower than using local variables. This has to do with dict-style lookup vs. array-style lookup, different opcodes are used in these respective scenarios [30].

2.6.3. Vectorization with NumPy

One way to unburden the Python interpreter is to vectorize our operations, and one of the most common ways of doing so is to use the NumPy library. It is quite famous, and is known to be very performant. Internally, it is powered by compiled C code that uses SIMD instructions [22] among other optimizations to deliver the aforementioned performance. It is also able to further boost the speed of certain operations such as the dot product by utilizing external libraries such as BLAS or LAPACK if they are present on the system [7].

The way we use NumPy efficiently is by storing the numbers that are to be processed in instances of NumPy's own array data structure, called ndarray [8]. Then we call NumPy's functions or methods to perform the operations we want. It could be for example summing up all numbers of each column in a matrix, finding an inverse matrix, calculating the dot product of two vertexes, or many others.

Here is an example how to use another highly performant library called Pandas to load integers from a large CSV file into an ndarray, and then use NumPy to calculate the average of these numbers:

```python
# scores.csv content:
# UserID,Score
# 1, 15
# 2, 142
# 3, 63
# ...
import numpy as np
import pandas as pd
print(np.average(pd.read_csv("scores.csv").Score.to_numpy()))
```

Listing 5: Python program that reads all scores from the Score column of a CSV file and calculates the average.

As shown in Listing 5, if we have libraries that support the input format and want to perform operations that NumPy support, then we can create a high performance program with very little code.

The idea of operating on entire arrays rather than on each element individually is called vectorization. If we want to perform high performance mathematical operations with large amounts of data in Python, then it is worth taking a look at NumPy. But it is important to understand that not all types of algorithms can be easily represented with vectorized code. If it is not possible, then another option would be to use special-purpose libraries to offload work from Python. Otherwise, due to the relatively slow performance of pure Python code, you either have to implement these routines (in a compiled language) yourself and interface with them from Python, or even use another programming language entirely such as C++.

In light of all of this, as part of answering the research questions I will discuss how dnctree could be optimized and which strategies might best apply to its algorithm.

3. Method

In this section I will describe how I have solved the main task as well as how I have re-implemented the base case of the NJ algorithm within dnctree in C++ to demonstrate a real-world comparison of performance between C++ and Python.

3.1. Creating the PaHMM-Tree wrapper library: libpahmm

PaHMM-Tree is implemented in C++11 and uses Make for building/compilation. It has a dependency called dlib which is shipped with the source code [28]. This version of dlib is old (v18.17, released in August 2015), and had to be updated due to past incompatibilities with CMake (the build system that I use). I have also made sure to check if any custom modifications to dlib were made by the developers of PaHMM-Tree. Diffing the directory tree of v18.17 with the one included in PaHMM-Tree revealed 0 modifications, all files were identical and updating the library should be okay in this regard. Dlib has many components which themselves have their own dependencies such as libjpeg, CUDA and others. I have explicitly set build-constants to disable many of these to keep the library as light and portable as possible. The C++ version (C++11) had to be updated to C++17 so that I would be able to utilize more modern parts of the standard library, this required some modifications to the PaHMM-Tree source code, but none that would affect the output of the program. Given these modifications, I was able to proceed to turn PaHMM-Tree into a C library. This C library would then be interfaced with using C Foreign Function Interface (cffi) from Python.

To write the library code, the wrapper around PaHMM-Tree to interface with it, I had to understand how the PaHMM-Tree code was supposed to be used. The best reference for that would be the command line part of the code, which instantiates the right objects, and uses them to create the distance matrix given some arguments sent by the user. One problem however, was that there was no function that one could call to compute each distance individually, you could only compute everything at once using the method `BandingEstimator::optimizePairByPair()`. Luckily, this method was easy to refactor and I was able to create a new method called `BandingEstimator::optimizePair(int i)` that would compute the distance of a single pair. `BandingEstimator::optimizePairByPair()` was made to call the new method to compute every distance.

As you can see, the `optimizePair` method takes only a single index, not a pair of indexes which would have referred to a pair of sequences. The reason is that distances inside PaHMM-Tree are represented as the elements of an upper-triangular distance matrix (excluding the diagonal). They are actually stored sequentially in an array, starting from the first row of the distance matrix. We have to figure out which element in the list corresponds to the distance between a given pair of sequences. The following formula give us the right index: $((2N-3)*i-i^2)/2+j-1$ where N is the total number of sequences and (i,j) is a position in the distance matrix upper-triangular part $(i<j)$. This is equivalent to the formula that is found in the command line code which does the same thing, it is just slightly reformulated. The formula was used there to print the distance matrix, and the code needed to access the elements given an (i,j) position, which is exactly what we want to do. From there on, finishing the C-part of the library code was pretty straightforward. C++ is almost a super-set of C, and creating a C API using C++ is quite easy. Primarily, there are two things to keep in mind when writing a C library in C++:

1. Only use features that are supported by C in the public library header file (pahmm.h). In the module files (*.cpp), where the implemen-

tations actually are, and in private header files, we may use C++-exclusive features.

2. In the public header file, we have to use the `extern "C" { ... }` syntax to disable C++ name mangling, because otherwise we will have issues exposing the API to a C program.

Once the C-part of the library was finished, I prepared a setup.py file (something that is standard when deploying Python projects) that is capable of triggering CMake to build the C-library (called libpahmm) and a special binary related to cffi that links to this library. That special binary can then be installed in an appropriate location to expose the C API to Python. However, that was not enough. Due to limitations in C, we do not have classes. Instead, essentially all we have are structs (data types without methods) and functions, and they are not that pleasant to use. Manual memory management is also necessary. We need to call functions to create objects, and we need to call other accompanying functions to clean up these objects, otherwise we get memory leaks. All such details must be abstracted away. The end-user should be exposed to an object oriented interface that is easy to use, and which follows Pythonic coding patterns. So more wrapper code in Python had to be written, and in the end 3 classes were exposed, with memory management being done automatically.

3.1.1. Testing

Modifying the PaHMM-Tree code was risky, because if done incorrectly, we might accidentally introduce bugs that affect the output. Even though I have been careful while modifying the code, testing is necessary to prove that the library produces the same results as the original PaHMM-Tree tool. My strategy when testing libpahmm has been to compare each distance produced from the library with the tool, given a set of sequences. If we get different distance values, then something is wrong. Some error tolerance had to be set, and I have decided that it should be 4 correct decimal places because of the floating-point precision in the output of the tool.

One problem with testing PaHMM-Tree this way was that there were some non-deterministic parts of the code. The function rand() (from cstdlib) was invoked a few times, and the seed was set to be the current time, which is hardly reliable if the tests run too slowly. Another problem was that by using a newer version of dlib, PaHMM-Tree produced slightly different values, but in rare cases radically different. To solve this, I had to do two modifications to the original source code and libpahmm. The first was to replace dlib with the same version that I had used in the library, and the second was to make PaHMM-Tree deterministic by replacing the randomization-code with code that uses the `std::mt19937_64` [3] RNG seeded with a fixed number, and sampled from in a way such that we get a uniform distribution of numbers in the interval $[0, \mathrm{RAND\_MAX}]$, the same interval that the rand() generate numbers from [3, 2]. I have been extra careful to not modify any other code than what is related to randomization. These modifications can be found in tests/paHMM-dist/core/Maths.hpp, cpp. In the end, the tests ran successfully. They use a variety of input parameters and sets of sequences, which should create enough coverage to give us confidence that the behavior of PaHMM-Tree has been sufficiently preserved. The test-suite is available under tests/ in the libpahmm repository [20].

3.2. Modifying dnctree to use libpahmm for distance estimation

In the command line code of dnctree, an instance of a class called `MSA` is created. This represents the sequence aligner that feeds the tree building algorithm its distances. I have decided to utilize Python's duck typing *(if it looks like a duck, quacks like a duck, then it is a duck)*, and create another class called MSApaHMM that mimics the `MSA` class' interface. This newer class will use (lib)pahmm to calculate the distances. It has approximately the same methods, enough to keep the rest of the code satisfied. The pahmm library reads the input FASTA file itself, and supplies distances per the request of the caller. If a distance has already been calculated before, then it is just fetched from an array. I have also added a command line option to use pahmm as the distance estimator, rather than the preexisting one.

3.3. Re-implementing Neighbor Joining in C++

The best way to answer **(RQ3)** would have probably been to re-implement the entire dnctree tool in C++. But such an endeavor would have taken too much time for it to be practical for the sake of this thesis. The other extreme is to approach the question in a purely theoretical manner. I have decided to take the middle ground by doing some analysis on the code of dnctree in light of the facts which we have discussed about Python and C++ in the introduction. But I want to also give a real-world example of how the *base case* of dnctree's algorithm would have performed in C++ by re-implementing and benchmarking it. The tree building algorithm in dnctree is recursive, if it is processing 100 sequences or less, then it will execute normal NJ (the base case). If we have more than 100 sequences, then the algorithm will take extra steps to process the sequences faster. It is the re-implementation of this base case in C++ that I will describe here. In the Results section I will show the benchmarking results.

I wanted the C++ implementation to mimic dnctree as closely as possible, and ideally produce identical results every time for a number of sequences $N \leq 100$. This makes it easier for us to make comparisons later on, and it also makes testing relatively trivial. To implement NJ in C++, I began by analyzing the part of the dnctree code that performs normal NJ, it is a function called `dnc_neighborjoining`. It appears that it has 3 branches, one for $N = 1$, one for $N = 2$, and one for every other case

$N > 2$. The last branch is the most interesting one, and that is where we find a while-loop which does most of the work.

The algorithm depend primarily on two data structures, a Python list to keep track of nodes to join (`Current_Leaves` in Listing 6), and a general Tree class which represent the tree being built. Internally, the Tree class is composed of an adjacency list to keep track of the graph, and an attribute to specify which node is the root. The adjacency list is a Python dictionary, the keys are the nodes and the values are Python lists of connected nodes. Nodes are identifier strings of each sequence, and this is how sequences are referred to throughout the algorithm.

The third branch with the while-loop look something like this:

```
Tree <- "Empty tree"
Distances <- "Lazy loaded distance matrix"
Current_Leaves <- "List of all sequence IDs"
while |Current_Leaves| > 3:
    x, y <- Select(Distances, Current_Leaves)
    Current_Leaves.FindAndRemove(x)
    Current_Leaves.FindAndRemove(y)
    auxiliary_node <- CreateRepresentative(x, y, Current_Leaves)
    Current_Leaves.Append(auxiliary_node)
    Tree.AddEdge(x, auxiliary_node)
    Tree.AddEdge(y, auxiliary_node)
```

Listing 6: Pseudo code describing in abstract terms how NJ is done. Note: The final 3 nodes will be joined together, but this is not described here.

Given how the algorithm works, I was able to reach the following conclusions about the choices of data structures and their performance implications:

- Using a list to keep track of current leaves (aka current neighbors) is costly for larger numbers of sequences because you have to find and remove two elements that could exist anywhere in the list, this is an $O(N)$ operation. I have decided to use a hash table in my implementation instead.

- Sequences are referred to using a string as an identifier, this is costly because we need to use plenty of hashing and string comparisons to work with them. Instead, I have decided to use indexes to refer to the sequences throughout the algorithm, which is cheaper and allows us to use arrays efficiently.
- The tree is built in a very specific way: the tree is (apart from the root) binary, and the size is predictable from the start. With this knowledge in hand, rather than using a general-purpose Tree-class, I have built one based on `std::vector` that has a fixed size, where each element is a node that knows its sequence ID (an index) and its two children. The tree does not keep track of the root, it is not even explicitly specified in my algorithm and it did not have to be.
- The distance matrix is expanded dynamically in dnctree to leave room for distances to auxiliary nodes. Since we know exactly how many auxiliary nodes will exist, I have pre-allocated enough space from the start, and no dynamic expansion is necessary. All elements are stored sequentially.

It is worth noting that more sequential storage should theoretically lead to less cache misses, and better performance.

In every other way, the NJ algorithm should be very similar.

Note: Since I am only interested in the performance of the tree building itself (and not the distance estimation), my C++ NJ program only takes precomputed distances as input rather than sequences. The responsibility of estimating distances, or computing them randomly lies elsewhere.

3.3.1. Testing

To ensure that my re-implementation worked correctly, the safest way was to compare the output with dnctree's output and make sure that they were identical. The way I had done this was by feeding dnctree and the C++ NJ implementation several randomly generated distance matrices.

Dnctree was originally not able to take external distances as input, I had to create a new `MSA`-like class (similar to what I did when implementing pahmm support). Instead of the distances being computed, they are loaded from a file, stored in a list and accessed by the tree building algorithm when needed.

I have created a program that for each sequence count $10, 20, ..., 100$, create a random distance matrix with the seeds $100, 101, ..., 109$. So in total, we have 100 test cases of various sizes. The program would for each matrix compare the Newick tree output of dnctree with the one from C++ NJ.

One problem that caused some test cases to fail initially was when two similarly suitable pairs of sequences were selected by the algorithm's selection function (see Listing 6). The result was either based on how current leaves were stored, or by extremely small errors at the end of the distances that were difficult to control or emulate in C++. To solve this problem, I firstly made the algorithm select nodes similar to how they were stored in dnctree by taking advantage of the fact that they were stored in order (first all sequences in ascending order, and then all auxiliary sequences from new to old). Then I reduced the precision in dnctree a small bit to eliminate the minor numerical differences, and in C++ NJ I adjusted the tolerance when comparing distances accordingly.

After these adjustments, all test cases passed.

3.4. Benchmarking

To help us answer **(RQ3)** I have constructed a couple of benchmarks that show the performance of tree building in Python (dnctree) and in C++ (NJ). The benchmarks ran on an Intel i7 6700k processor with Linux

5.18.16-xanmod as the kernel. For C++ I used the perf tool for profiling with 1 kHz sampling frequency, and I sampled the running time of the `NeighborJoining::build()` method to only include the NJ algorithm itself. GCC compiler flags were "-O3 -DNDEBUG -std=gnu++20". The same idea was used for dnctree where I used the cProfile profiler with Python 3.10. There I sampled from the `dnc_tree` function. This function not only calls `dnc_neighborjoining` but also performs the full tree building algorithm (if $N > 100$). The idea was to also compare dnctree's tree building algorithm in Python with basic NJ in C++. For both programs I benchmarked using 6 different random distance matrices of sizes 50, 100, 200, 300, 400 and 500. For every distance matrix I sampled the running time of the aforementioned functions 20 times. That is in total $6 * 20 = 120$ executions for each program. For exact running times see Appendix A.

4. Results

For a comparison between dnctree's and C++ NJ's running time see Figure 4.1.

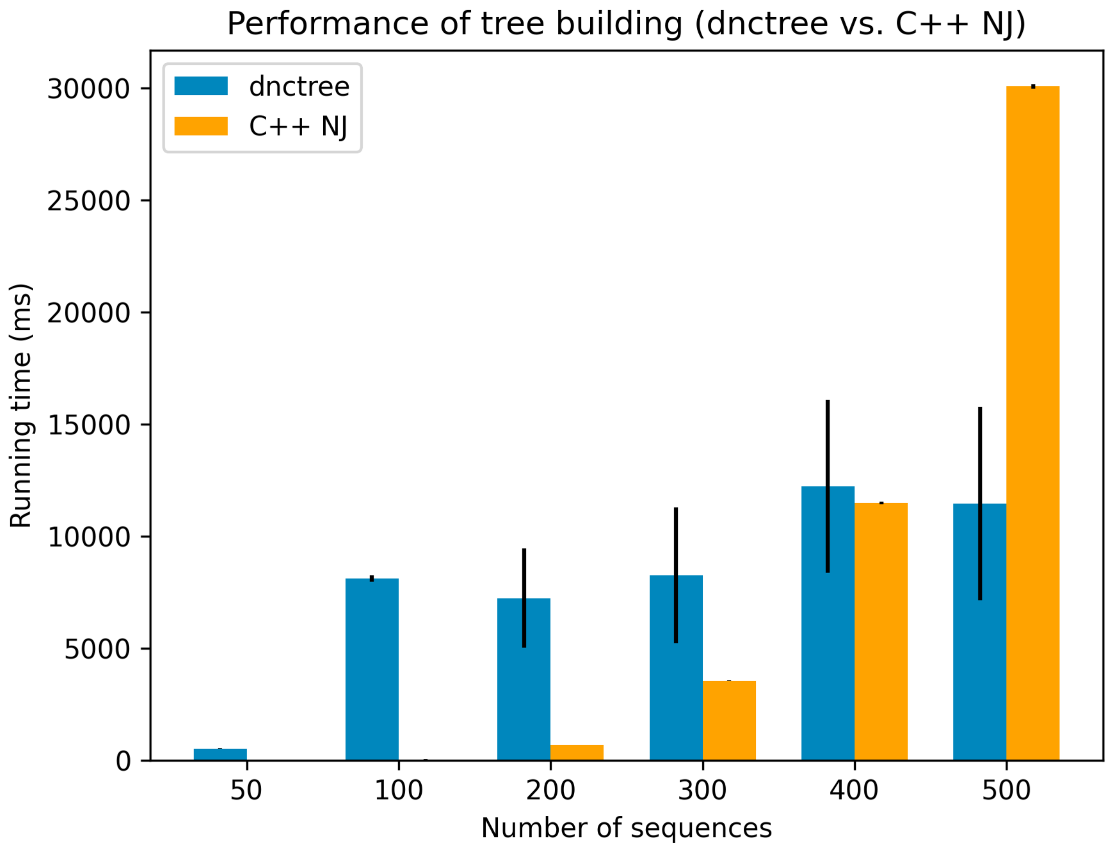

Figure 4.1.: Benchmark of dnctree's and C++ NJ's runtime performance in milliseconds. The vertical black lines represent the standard deviations of each respective set of running time samples.

For $N \leq 400$ the C++ NJ implementation clearly outperforms dnctree, while also having a more predictable running time (see the lower spread in Figure 4.1). Its memory consumption was also significantly lower. But the cubic growth of basic NJ in the C++ implementation is obvious. For $N = 100$ where both implementations use basic NJ, the C++ implementation is more than 250 times faster! But for $N = 400$ the C++ implementation is only 1.08 times faster. For $N = 500$ dnctree was 2.61 faster than the C++ NJ! Clearly the improved tree building algorithm of dnctree scales significantly better than normal NJ, even with the huge performance improvements gained from using C++.

5. Discussion

In this section I will summarize this thesis by answering the research questions and reflect upon the findings that have been made.

5.1. Answering RQ1: Input and Output

RQ1: *The user must feed the tool sometimes large inputs of data, and at the end receive a phylogenetic tree as output. Should the reading of sequence input and writing of tree output be done in Python or C++?*

Now that dnctree has support for estimating distances with pahmm, it is better to let pahmm handle the input. The reason is that if we process the input in Python, and then pass it on to the pahmm library, we will be creating redundant copies of the data. In actuality, if I/O is done in Python the entire process will be a bit slower due to Python's inherent flaws when it comes to performance. It is inevitable that redundant copies will be made, because the data structure that would have been created with Python would be different from the ones used by pahmm, and because memory management is handled differently.

When it comes to output, the size of a tree in Newick format is relatively small, and printing it to the Standard Out Stream (stdout) or to a file would likely account for only a tiny fraction of the entire phylogenetic tree building process. But if one wishes to save some computational resources here as well, and if a library is performing the NJ, then one could let it handle the outputting of the tree if possible.

5.2. Answering RQ2: C++ for distance estimation and tree building

RQ2: *What are the performance benefits of using C++ for central calculations (the tree building and distance calculations) rather than using Python+NumPy?*

There are clear performance advantages of using C++, both in practice and in theory. As we saw with the NJ implementations (dnctree's NJ vs. C++ NJ), by using C++, using the right data structures and eliminating some redundant computations (strings vs. indexes) we were able to reach more than 250 times better performance. Much of this is owed to the flexibility in how we perform our computations which we get with C++, including our ability to program in a more cache-friendly way to reduce cache misses. The C++ compiler is also able to perform many interesting optimizations, and the performance increase when using the "-O3" flag was very noticeable. Even in the C++ profiler, I noticed aggressive inlining of functions, which is a type of optimization that reduces the overhead of function calls.

Because of the results and what we know from past literature about the performance of C++, it is easy to say that the distance estimation would benefit (is benefiting in the case of pahmm) from being implemented in C++.

However, we can definitely achieve similar performance with NumPy for some applications due to its heavy optimizations and vectorization. But it is not immediately clear how NumPy could be useful for NJ. We could replace the Distances data structure seen in Listing 6 with a NumPy matrix. Current leaves could be implemented as an array of indexes, given that we use indexes rather than strings. This index array would then have elements added to/removed from it $O(N)$ times in Python (see the while-loop in Listing 6). This is an okay compromise, because the array is useful for optimizing the summation of distances which occur many times in the selection function. From my experiments, these account for a significant part of the runtime overhead of dnctree. When it comes to the tree itself, it is unclear for me how to utilize NumPy to optimize it.

Regardless, it does seem that we have to do non-constant amount of work in pure Python, and C++ could help (at least) in this regard given its performance advantages.

5.3. Answering RQ3: The Performance of a C++-based Dnctree

RQ3: *Is it possible to estimate how much faster dnctree would have been if it was implemented entirely in C++ rather than Python?*

To be fair, I have only re-implemented the basic NJ part in C++, and to say that dnctree's NJ would have been 250 times faster if it had been written in C++ is a bit of a stretch. So we need some more analysis. Given that the distance estimation code is purely implemented in C++ (pahmm), and that I/O is relatively insignificant in terms of time complexity, I will only talk about the tree building. Neighbor Joining is only a part of dnctree, the entire algorithm would possibly need more complex data structures than the ones I have used, which might reduce performance. It is also worth noting that the distance matrix that dnctree uses is lazy-loaded, for the reason that dnctree does not need all distances. I do not think that NumPy's ndarray data structure has support for lazy loading, let alone from an external source. So using NumPy for the distance matrix for the entire algorithm may prove challenging, if not outright impossible. A different approach here would probably have been to only build a NumPy distance matrix when executing the base case (NJ), then taking the same approach mentioned in my answer to **(RQ2)**. This does require $O(N^2)$ copying every time it is done, but it could be worth it.

With that said, the benchmarks do show very large differences in performance, even when the algorithms are almost identical (when $N \leq 100$).

It is clear that even if we decide to implement the entirety of the dnctree algorithm in C++, with $O(N^2)$ copying for NJ, we would get a significant performance boost. With regards to the base case alone, it is safe to say that once a full distance matrix has been created, its performance would by huge (the benchmarks can attest to that), and it would help dnctree significantly.

5.4. Possible Improvements

I will dedicate this section to mentioning some things that could have been done better when it comes to coding and analysis within this project.

Firstly, the C API's naming was not very clear because the names were immediately based on the PaHMM-Tree code base, and without some context it is difficult to understand what everything means. This could have been improved. Secondly, when implementing a getter (or a setter) in C, it is a good idea to leave the return value for error codes, and use a pointer parameter to pass the return value, otherwise it might be difficult to express many different errors for the same routine (unless we use the ugly errno pattern). Lastly, the code is not thread-safe. In fact, I have not mentioned threading at all until now. The reason is that this adds another layer of complexity which is beyond the scope of this thesis. In reality, threading support could be very useful and significantly boost performance, especially if the improved NJ algorithm in dnctree could be parallelized (and C++-based).

I feel confident about my analysis given the scope of this thesis. However, it is worth mentioning that there is plenty of depth when it comes to analyzing the runtime performance of phylogenetic tree building software such as this. One could conduct this analysis in a much more in-depth manner, by among other things utilizing more information about the hardware it runs on, about the various algorithms involved and the C++ toolchain used to build and debug the applications.

It is also worth noting that there are other Python interpreters out there than just CPython. Another example is PyPy which implements Just-In-Time (JIT) compilation to achieve (many times) better performance than CPython [25]. There is also Cython, which has its own typing system and is able to use it to optimize code [9].

6. Conclusion

I have managed to turn the PaHMM-Tree tool into a Python library, and make dnctree use it as its distance estimator to hopefully enhance dnctree's accuracy. I have also managed to implement basic NJ in C++ based on dnctree's algorithm, and by doing so getting empirical data that helped me answer the research questions. My hope is that this thesis has proven that C++ can enhance dnctree's performance significantly. I also hope that my analysis of the data structures and how they are used within dnctree could aid in further improvements to its runtime performance as well as memory usage, and also provide some useful hints that could be useful for optimizing other similar programs as well.

Bibliography

[1] Kevin Atteson. "The performance of neighbor-joining algorithms of phylogeny reconstruction". *Computing and Combinatorics*. Vol. 1276. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 101–110. ISBN: 9783540633570.

[2] Various authors. *Programming languages - C*. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf (visited on 09/03/2022).

[3] Various authors. *Working Draft, Standard for Programming Language C++*. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4849.pdf (visited on 09/03/2022).

[4] Marcin Bogusz. "Evolutionary Approaches to Sequence Alignment" (2018).

[5] Marcin Bogusz & Simon Whelan. *paHMM-dist FASTA parser*. URL: https://github.com/arvestad/paHMM-dist/blob/bb48e399fc025a1250939c52bb7dd5e27065ac53/src/core/FileParser.cpp (visited on 09/20/2022).

[6] Marcin Bogusz & Simon Whelan. "Phylogenetic Tree Estimation With and Without Alignment : New Distance Methods and Benchmarking" (2017).

[7] NumPy Developers. *Installing NumPy*. URL: https://numpy.org/install/ (visited on 09/03/2022).

[8] NumPy Developers. *The N-dimensional array (ndarray)*. URL: https://numpy.org/doc/1.22/reference/arrays.ndarray.html (visited on 09/03/2022).

[9] Victor Epain. *Examples Mandelbrot*. URL: https://github.com/cython/cython/wiki/examples-mandelbrot (visited on 09/03/2022).

[10] Free Software Foundation. *Options That Control Optimization*. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (visited on 09/03/2022).

[11] Free Software Foundation. *Using Vector Instructions through Built-in Functions*. URL: https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html (visited on 09/03/2022).

[12] Python Software Foundation. *listobject.h*. URL: https://github.com/python/cpython/blob/v3.10.4/Include/cpython/listobject.h (visited on 08/01/2022).

[13] Python Software Foundation. *object.h line 97*. URL: https://github.com/python/cpython/blob/v3.10.4/Include/object.h#L97 (visited on 08/01/2022).

[14] Python Software Foundation. *Python 3.10.6 documentation*. URL: https://docs.python.org/3/ (visited on 09/03/2022).

[15] Python Software Foundation. *typing — Support for type hints*. URL: https://docs.python.org/3/library/typing.html (visited on 08/01/2022).

[16] Matt Godbolt. “Optimizations in C++ compilers”. In: *Communications of the ACM* 63.2 (2020), pp. 41–49. ISSN: 0001-0782.

[17] John L. Hennessy & David A. Patterson. *Computer Architecture - A Quantitative Approach (4th Edition)*. Elsevier, 2007. ISBN: 978-0-12-370490-0. URL: https://app.knovel.com/hotlink/toc/id:kpCAAQAE02/computer-architecture/computer-architecture.

[18] Sheneman Luke et al. “ClearCut: a fast implementation of relaxed Neighbor Joining”. In: *Bioinformatics* 22.22 (2006), pp. 2823–2824. ISSN: 1367-4803.

[19] Thomas Mailund et al. “Recrafting the Neighbor-Joining method”. In: *BMC bioinformatics* 7.1 (2006), pp. 29–29. ISSN: 1471-2105.

[20] Mazen Mardini & Marcin Bogusz et.al. *libpahmm*. URL: https://github.com/mazen-mardini/libpahmm (visited on 09/03/2022).

[21] Oxford Reference. *Pierre Belon*. URL: https://www.oxfordreference.com/view/10.1093/oi/authority.20110810104440666 (visited on 09/03/2022).

[22] Jérôme Richard & Matti Picus. *ENH: help compilers to auto-vectorize reduction operators - Pull Request 21001*. URL: https://github.com/numpy/numpy/pull/21001 (visited on 09/03/2022).

[23] Guido van Rossum, Barry Warsaw & Nick Coghlan. *PEP 8 – Style Guide for Python Code*. URL: https://peps.python.org/pep-0008/ (visited on 09/03/2022).

[24] J. A STUDIER & K. J KEPPLER. “A note on the neighbor-joining algorithm of Saitou and Nei”. eng. In: *Molecular biology and evolution* 5.6 (1988), pp. 729–731. ISSN: 0737-4038.

[25] The PyPy Team. *PyPy*. URL: https://www.pypy.org/ (visited on 09/03/2022).

[26] Lusheng Wang & Tao Jiang. “On the complexity of multiple sequence alignment”. In: *Journal of Computational Biology* 1.4 (1994), pp. 337–348. ISSN: 1066-5277.

[27] James Weger-Lucarelli et al. “Vector Competence of American Mosquitoes for Three Strains of Zika Virus”. In: *PLoS Neglected Tropical Diseases* 10.10 (2016), e0005101–e0005101. ISSN: 1935-2727.

[28] Marcin Bogusz, Simon Whelan & Lars Arvestad. *paHMM-dist*. URL: https://github.com/arvestad/paHMM-dist (visited on 09/03/2022).

[29] Leonid Zaslavsky & Tatiana A. Tatusova. “Accelerating the Neighbor-Joining Algorithm Using the Adaptive Bucket Data Structure”. *Bioinformatics Research and Applications*. Vol. 4983. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 122–133. ISBN: 9783540794493.

[30] Qiang Zhang et al. “Quantifying the interpretation overhead of Python”. In: *Science of computer programming* 215 (2022), p. 102759. ISSN: 0167-6423.

A. Benchmarking data

Detailed results from the C++ NJ vs. dnctree benchmarking.

50	**100**	**200**	**300**	**400**	**500**
527	7973	4161	13346	15273	12535
519	7934	4877	7221	10352	17602
506	7790	7339	6509	13341	8949
507	8238	8525	10898	13713	10565
502	7840	9975	13329	15491	20500
504	8047	8499	6173	12899	11486
578	8126	7881	2752	15187	7734
503	8343	5003	4959	6478	13991
548	8126	7323	7650	15132	5802
533	8289	10561	9743	8580	16680
565	7998	10075	7163	12311	8701
502	8230	6610	4527	11014	5303
495	8159	5987	14505	22028	11777
497	8179	5477	6696	11884	8600
512	8025	7786	6975	13706	9642
512	7930	2623	7745	6516	4957
518	8118	8470	10800	5415	9307
504	8365	9755	12119	12542	15374
497	8022	2989	10637	15999	15975
504	8177	7596	6312	9639	14363

Table A.1.: Running times in milliseconds for dnctree's tree building. The headers show the number of sequences the distance matrix described ($N = 50, 100, ...$), and the numbers beneath them are samples generated by measuring how much time it takes to process a random distance matrix.

50	100	200	300	400	500
1	31	671	3519	11505	30053
1	31	674	3546	11508	30037
1	29	667	3515	11462	30247
1	31	665	3520	11463	30037
1	26	674	3530	11408	29970
1	28	662	3507	11499	30118
1	30	670	3546	11706	30287
1	32	672	3549	11396	30083
1	32	660	3571	11444	30304
1	30	672	3529	11536	30087
1	31	665	3547	11431	30009
1	30	664	3540	11430	30014
1	32	667	3542	11443	30008
1	29	659	3540	11483	30122
1	31	650	3550	11465	30002
1	29	666	3575	11461	29994
1	29	668	3542	11435	29914
1	32	662	3525	11431	29925
1	29	676	3540	11456	30002
1	30	664	3554	11432	30025

Table A.2.: Same as Table A.1, but for the C++ NJ implementation.

Datalogi
November 2022

www.math.su.se

Beräkningsmatematik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm