Novel Resource-Efficient Methods for Robust and Accurate Taxonomic Profiling of Metagenomic Data

Dissertation

zur Erlangung des
Doktorgrades der Naturwissenschaften (Dr. rer. nat.)

der

Naturwissenschaftlichen Fakultät III
Institut für Informatik

der Martin-Luther-Universität
Halle-Wittenberg,

vorgelegt

von Silvio Weging
geb. am 09.06.1989

Gutachter:
Prof. Dr. Ivo Grosse
Prof. Dr. Burkhard Morgenstern

Tag der Verteidigung: 20.10.2022

Acknowledgements

I would like to thank Ivo Große and Andreas Gogol-Döring for believing in me and the topic I have chosen. They showed me how fun the use of my knowledge can be and that bioinformatics could be something to stick with for a while. Adding to this, I would really like to thank Ivo Grosse for including me in his group of tremendous people: Alex, Benjamin, Claudius, Claus, Francesco, Ioana, Jan, Jördis, Lasse, Martin, Samar, and Yvonne. When I started this project, he and the group gave me the support I needed, so: many thanks and cheers to you!

Furthermore, I thank Jan Grau and Yvonne Pöschl-Grau for their valuable advice and mentoring. Especially teaching together with Jan Grau was a pleasure, our courses rock!

My thanks also goes to Matthias Müller-Hannemann for providing me with the much-needed time as well as to Stefan Posch for understanding and accepting my quirks when working together, as well as to Steffen Schüler for listening to my crazy ideas even though I felt like never giving him enough context.

Last but not least, my everlasting gratitude goes out to my best friends and family without whom I would not be who I am today.

Abstract

English version:
Examining the taxonomic composition of sequenced data is a necessary step in almost any metagenomic analysis. Most existing and widely used programs prioritize speed over accuracy and robustness, while consuming large amounts of memory. As an alternative, we have developed and implemented new methods in a program called `kASA`, which is able to efficiently identify DNA or protein sequences using k-mers to build a metagenomic profile. We ensure high accuracy and robustness by using an amino acid-like encoding together with an interval of k's while using at most the amount of memory specified by the user. Algorithms and data structures specifically adapted to the use of secondary memory allow a complete taxonomic analysis of metagenomic data without compromises on HPC clusters, desktops or even laptops.

German version:
Die Untersuchung der taxonomischen Zusammensetzung von sequenzierten Daten ist ein notwendiger Schritt in fast jeder metagenomischen Analyse. Die meisten existierenden und weit verbreiteten Programme priorisieren Geschwindigkeit über Genauigkeit oder Robustheit und verbrauchen dabei große Mengen an Arbeitsspeicher. Als Alternative haben wir neue Methoden entwickelt und in einem Programm namens `kASA` implementiert, das in der Lage ist, effizient DNA- oder Proteinsequenzen mit k-meren zu identifizieren, um ein metagenomisches Profil zu erstellen. Dabei wird eine hohe Genauigkeit und Robustheit sicher gestellt, indem es eine aminosäureähnliche Kodierung zusammen mit einem Intervall von k's verwendet, wobei dabei maximal die vom Benutzer angegebene Speichermenge verbraucht wird. Algorithmen und Datenstrukturen, die speziell an die Verwendung von Sekundärspeicher angepasst sind, ermöglichen eine vollständige taxonomische Analyse von metagenomischen Daten ohne Kompromisse auf HPC-Clustern, Desktops oder sogar Laptops.

Contents

Chapter 1

Introduction

Since the discovery of DNA as the basis of terrestrial life [1], molecular biology has been devoted to the study of the genetic information encoded in it. It has become apparent that DNA can serve as a unique identifier, in the form of genomes. Different living beings, even individuals [2], could thus be identified and their genome stored as a reference. If, for example, there were similarities in the DNA, it would be possible to draw conclusions about a kind of molecular kinship. However, in order to enable this reference-based research into the earth's biodiversity, methods and technologies are needed that can determine the specific sequence of bases in the DNA.

Nowadays, this is made possible by "Next generation sequencing" and "Third-generation sequencing" techniques [3]. They are able to read, or "sequence", DNA and translate it into a digital format. One problem, however, is that DNA has been found to be very long in some organisms (up to several million base pairs [4]). Manual processing and searching for commonalities and differences in DNA sequences, or genomes formed from them, is infeasible.

This is where a new discipline comes into play: bioinformatics. Processing large amounts of data is one of the main tasks of computers and a significant subfield of computer science. Accordingly, it is obvious to process biological data with methods from computer science as well. One of the main goals is to prepare data in such a way that they can be digitally processed. The result: a digital database of reference genomes, which can be searched using known algorithms [5]. Another advantage of this is that newly sequenced DNA can now be compared against these references. This makes it possible to separate the known from the unknown data and to determine the frequency of certain sequences.

The latter is particularly important for another field of research in the life sciences: metagenomics. It deals with the analysis of samples from the environment, which usually contain microbial life that cannot be cultured in the laboratory (for more on the history of metagenomics, see Section 1.1). The comparatively high proportion

of unknown DNA makes it much more difficult to deal with this type of data. In addition, it is not only important to identify which known organisms are found in a sample, but also how many (the "abundance"). A variety of conclusions can be drawn from this. An example from medical diagnostics would be the recognition of the pathological spread of an intestinal germ based on its abundance in a stool sample [6].

Therefore, it is important to have software that is able to quickly and versatilely as well as accurately determine the presence and abundance of the relevant organisms in a digitized DNA sample. There are numerous such programs that have been developed in recent years [7, 8]. However, almost all of them require the presence of hardware that is not necessarily available to every person: High-Performance Computer Clusters (HPCC). Especially with an ever-growing number of reference genomes, it seems reasonable to switch to low-cost memory and not try to load everything into primary memory.

This is the goal of this dissertation: to develop and use a software capable of identifying known strings of DNA with a customizable primary memory footprint. This software is called kASA - k-mer Analysis of Sequences based on Amino acid-like encoding and is able to run on many commercially available computers and spares neither accuracy nor performance. Some of the content of this dissertation has already been published in *Nucleic Acids Research* with open access [9]. Usage of published content is permitted via the Creative Commons CC BY license. The source code can be found online [10, 11, 12].

The dissertation is structured as follows: The first chapter gives an introduction into the research field of metagenomics [13] and why we chose it. Starting with a brief history in Section 1.1, it then depicts a typical metagenomic study. On that basis, our chosen topic of interest is derived at the end. It also explains the challenges we faced while developing `kASA` in Section 1.2.1 and describes the main task of this dissertation in Section 1.2.2 further.
This introduction is then followed by Chapter 2 which outlines in Section 2.1 what solutions and software already exist and what their limitations are. Based on this, we show new ideas in Section 2.2 which aim to overcome those limitations and at the same time achieve some of the goals we defined in Section 1.2.2.
Chapter 3 revolves around the implementation of `kASA`. It shows the algorithmic solutions we developed alongside some code that illustrates how the ideas we described in the second chapter were implemented. The different sections refer to the modes in which `kASA` operates. Section 3.4.2 for example shows how genomic data is transformed and saved into a data structure better suited for identification. This is then used in Section 3.4.5 to compare sequenced data to this data structure containing the reference. Additional modes shown in Sections 3.4.3, 3.4.4 and 3.5 aim to improve usability.
Chapter 4 verifies our theoretical ideas with various experiments with both simu-

lated and real data in Sections 4.3 and 4.4, respectively. Section 4.3.1 for example, contains comparisons of the performance and accuracy of `kASA` with software mentioned in Chapter 2.
Since `kASA` can also be used to gain new scientific insights, we give an overview of what we learned in Chapter 5.
In the last Chapter 6, we conclude our findings and discuss our results concerning our goals. We also sketch possible future work.

1.1 Introduction to metagenomics

The study of microbial communities started in 1674 with the observation through self crafted microscopes by Leeuwenhoek [14]. With samples gathered, for example from his teeth, he saw that tiny creatures lived inside his mouth. Another method was employed later when Robert Koch isolated *Bacillus anthracis* by making it grow on a nutrient in 1876 [15]. Both methods aimed at visualizing and understanding bacteria and their physiology. Over time, microscopes got better at magnifying the world of microbes. This lead to the discovery that the observed microbiome did not match the number of cultivatable organisms from the same sample [16]. Among others, Winogradsky drew the conclusion that different microbes only grow under specific circumstances in the late 1800s [17]. This was followed by a period of studies on microbial pathogenesis and biochemical abilities of certain microbes [17]. In the year 1977 Carl Woese used ribosomal RNA (rRNA) named 16S (for prokaryotes) and 18S (for eukaryotes) as molecular markers to distinguish microbial life [18]. The classification via 16S or/and 18S rRNA is a method still used due to its simplicity [19]. In order to detect if certain known rRNA was present, amplification with specific primers was performed [20]. This way, it was deducible that if the amplification of a certain 16/18s rRNA worked, the corresponding species was in the sample as well. This marked the beginning of reference based analysis: Known 16/18S sequences were saved and used to later re-identify them. Amplification was first done via "cloning vectors" which were mostly plasmids containing foreign DNA that were then introduced into a host species for duplication (in most cases *Escherichia coli*) [21, 22]. This lead to the possibility of studying the metagenome in a broader way: Handelsman et al. used DNA cloning vectors to directly study the microbial community of soil samples [23] by copying all found DNA. After isolating the DNA polymerase enzyme found in *Thermus aquaticus* in 1969 [24], polymerase chain reactions (PCRs) were used for amplification instead due to the shorter time needed for amplification [25]. However, this introduced an amplification bias which could distort the estimated amount of microbial species in a sample [26]. A method of directly sequencing genetic material was introduced by Sanger et al. in 1975 [27]. It was able to sequence DNA with a length of sometimes more than 500 nucleotides. Today, these sequenced pieces of DNA are called "reads". However, it suffered from low quality in the first few bases and therefore required quality checks [28]. After

30 years of further development, so called "Next generation sequencing" technologies emerged in 2005 [3] which were able to sequence millions reads (depending on the technology used) with fewer errors and reduced prices. This enabled the analysis of whole metagenomic samples by shotgun sequencing [29, 30] which works by sequencing quasi-randomly grouped fragments analogue to the shot grouping pattern of a shotgun. This era is now being complemented and even replaced by "Third-generation sequencing" techniques [3] such as those from Oxford Nanopore Technologies [31] or Pacific Biosciences [32] which enable sequence lengths of kilo- or even megabase level.

Obtaining metagenomic samples can be quite easy due to their various habitats [33]. Typical metagenomic samples come from sea- [34] / ground- [35] and waste [36] water, soil [23], gut [37], feces [38], surfaces [39], biopsy [40] and many more. With this data and the mentioned techniques available, it is currently estimated that earth's biodiversity, especially microbial life, is almost completely unknown [41].

This gives us a chance to find new solutions to existing problems given that over the last centuries, humanity made vital discoveries while studying microbial life. Prominent examples are the discovery of penicillin (produced by fungi from the genus *Penicillium*) [42] and the vaccination against smallpox (*Variola major*) with *Cowpox virus* [43]. More modern examples would be gene therapy based on viral vectors (usually *Adeno* viruses) [44], the development of the CRISPR/Cas-Method for gene editing [45], and the use of the thrombolytic enzyme Streptokinase found in some bacteria of the genus *Streptococcus* for eliminating blood clots [46]. These inventions were only possible due to the research of our biodiversity and its effect on nature. Given the current trend of loss in biodiversity, however [47], the ability to gain further insights which might solve current or future problems could get lost. It is therefore crucial to identify as many new organisms as possible in the most accessible and easiest way. To do this, researchers must be able to differentiate between already known genomes and new genetic material. This process is called taxonomic profiling and is part of almost all metagenomic studies.

The "taxonomic" part in taxonomic profiling refers to taxonomy - the classification of things (organisms in our case) based on shared characteristics [48]. Elements of a taxonomy are called "taxa" or "taxon" in singular. This branch of science started with Linnaeus in the 18th century [49]. About 100 years later, Charles Darwin based his classification on evolutionary relationships [50]. From this, a system still in use was developed. It is modeled like a tree with two domains of life (prokaryota and eukaryota) [51]. In 1977, Carl Woese did not only use 16S/18S to distinguish organisms but also discovered "Archaea" as the third domain of life [52]. Since then, multiple changes and new proposals for reordering the taxonomic relationships were given [53]. For example, with the rise of molecular biology, taxonomic order was not only given by phylogeny but also molecular similarities [54]. One of the most widely used taxonomy database is the NCBI taxonomy [55] which maps taxa to unique numbers - so called "taxonomic IDs". This way, genomes, genes, etc. are directly

mapped to a name and an identifier. In this dissertation, we will use this to identify known species by their genomic data linked to a taxonomic ID.

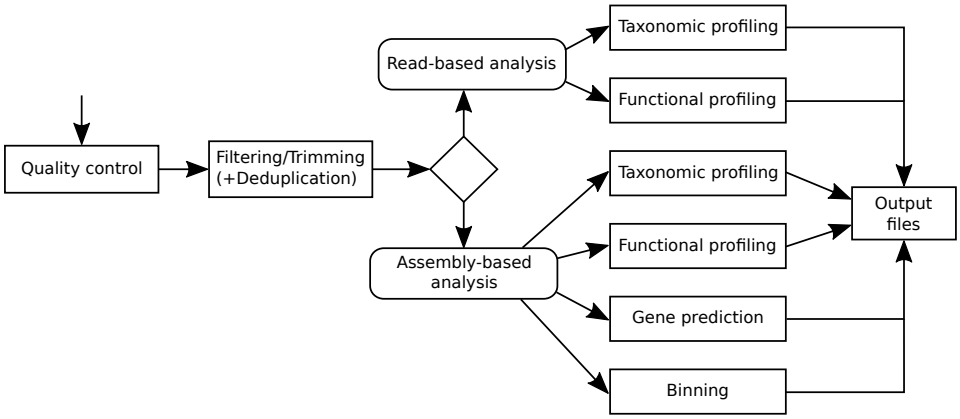

Figure 1.1: Flowchart of a typical metagenomic study.

With this, we are able to describe how a typical metagenomic study is performed [56, 57]. Let us assume that the DNA or RNA has already been sequenced. Then the typical approach can be seen illustrated in Figure 1.1 and is as follows:
The reads are first analyzed for per-base quality. Then, reads of low complexity (also called "dust") or artificial sequences belonging to adapters or primers are cut from the raw read files. In some cases, reads are exact copies from another, usually when they were amplified via PCR enrichment. Depending on the study's aims, they can be deleted as well. After that, the processing goes in two directions:
The first path uses a method called "assembly" which tries to piece together reads to create longer pieces of DNA or even whole genomes. This can be done *de novo* by trying to find reads that overlap (by having identical bases at one of the ends) [58]. Another method is called "reference-based" which uses existing genomes as a guideline to assemble similar genomes of, for example, related organisms [59]. Afterwards, these assembled reads, called "contigs", can be used for taxonomic profiling. However, assembling reads is a very computationally intensive task with no guaranteed success [56]. The second path is therefore viable as it tries to directly identify the reads by comparing them with a reference and reporting found similarities.
Both paths ultimately create a "taxonomic profile". A taxonomic profile is a list of all organisms or taxa that have been identified in a dataset together with their respective abundance [60]. This means that depending on the chosen path, the profile is derived either directly from the reads or from the contigs. During taxonomic profiling, the reads or contigs which are similar to reference genomes are assigned the respective taxonomic IDs belonging to these known species. After summarizing the abundances, they are also able to spot the most prominently identified species of the whole dataset. The profile created from the reads aims to describe the composition of the dataset in terms of the organisms identified. With this, researchers are able to inspect the whole metagenome of a sample. The profile derived from the con-

tigs is mainly used to check which contigs belong to known and which to unknown metagenome-assembled genomes (MAGs). This helps to estimate the amount of unknown data in the dataset as well as partition the reads into known and unknown sequences.
It then continues with the "annotation" which tries to find genes or non-coding parts and biological information like function [56]. If multiple samples were gathered (like human gut microbiota from pre- and post-spaceflight), statistical analysis might also give vital insight in similarities or differences between samples. Our software `kASA` can also be part of such a pipeline, as can be seen in Section 5.2.

A problem current pipelines are facing is that the number of reference genomes is rising quickly. The reason is that the cost of sequencing has gone down significantly in the last years [3] and whole genome sequencing techniques [61] are adding more and more genomes to databases such as the NCBI's nt [5, 62]. This leads to increasing space consumption and required computing time. We therefore decided to focus our research on this problem and develop a program capable of taxonomic profiling of data from both reads and contigs, while being scalable in terms of the number of reads and reference genomes. To better differentiate between the assignment of taxonomic IDs to a read and the creation of a table summarizing the whole data set, we call the former "r-identification" and the latter "profiling" from now on.

For the readers information: Usually, programs like `kASA` are called "tools" which use "classification" to taxonomically profile a metagenomic dataset. We will not be using those terms however, because even though they might be widely used, they are slightly incorrect. In software development, a "tool" is usually part of a software and used to support this software [63]. It therefore does not produce something on its own. An example would be the defragmentation tool used by most operating systems. Because `kASA` is not part of a software and produces its own output, we call it and all related programs "software" or "program" and not "tool". The term "classification" on the other hand, is usually used when something is categorized according to their type [64]. It could be argued that taxonomic profiling is a form of classification, as sequenced DNA is assigned to taxonomic IDs. However, the process behind this is more like recognition than a predetermined classification. Therefore, a program "identifies" the sequenced DNA based on similarities with known genomes. The human brain does the same thing for example, when it tries to match faces to recognize people it has seen before in order to identify them as a known person [65, 66]. This is the reason why we (and others [67]) will use "identification" instead of "classification".

1.2 Requirements and main objectives

1.2.1 Requirements

Before and during taxonomic profiling, some things need to be considered.

At first, the database used as a reference must be chosen. Depending on the scope of the study, this might range from a couple of genomes up to every currently known genome. Additionally, the reference must be kept up to date for future studies. With increasingly more species being discovered, there is also a risk of redundancy: It may be that two different researchers discover the same species but name it differently or known species are updated, as has happened for example, to many subspecies of *Shigella* and *Escherichia coli* [68, 69].

Secondly, the data that needs analyzing has certain specifics which a program performing taxonomic profiling must handle appropriately. Some sequencing technologies offer paired-end sequencing [70] which means that DNA was sequenced from both ends at the same time [71]. Another issue would be the length of the reads because some methods [72] are not optimized for large sequences (more than 1000 bases). The size of the files combined with the size of the reference database can pose quite a challenge even for high performance computing clusters (HPCC). Files containing reads can be as large as multiple gigabytes and the reference can comprise several terabytes. Finding similarities between the two could take weeks if done naively.

Thirdly, a problem often underestimated is the existence of errors in either the reference or sequenced data. The ability to deal with mutations, insertions, deletions, and sequencing errors is called "robustness". Some methods assume a high sequence quality and discard sequences with low quality even though they might have contained valuable information. Given a more robust method, these sequences could still be processed. Other methods try to balance between accepting errors and generating false positives, which naturally occur when the matching process is relaxed. Being able to deal with errors is important because some species mutate at a relatively rapid rate (e.g., RNA viruses from the genus *Orthomyxoviridae* [73]), therefore diverging fast from a reference genome. This leads to false conclusions about the actual presence of a taxon.

Fourthly and finally, there are problems of technical nature such as output formats and standards, the compatibility to operating systems and hardware, dependencies on third-party software, and the general ease of use. Most programs have their own output format and offer the translation from one format to another. Currently, no standard exists although the CAMI [8] format might emerge as one. It is also apparent that most programs are written for a Linux environment and work optimally on an HPCC platform, effectively limiting the user base to those who have access to both. Experience of users also varies, resulting in most people using software that is often cited, although it does not necessarily offer the best user experience.

1.2.2 Main objectives

Given the challenges mentioned in Section 1.2.1 we have to define what our software must be capable of when taxonomically profiling sequence data.

As we mentioned at the end of Section 1.1, the number of reference genomes is growing quite fast. Therefore, using a reference means writing a program that can handle a small as well as a very large number of genomes. It also means that the user must be able to add new reference genomes. This leads to goals Nr. 1 and 2:

Goal 1: *kASA has to work, independently of the number of reference genomes.*

Goal 2: *kASA must have a functionality to add new genomes to its existing reference database.*

It is also apparent that we need to support long reads since third-generation sequencing techniques [31, 32] are becoming more widely used [3]. But next generation sequencing is still used, so we also need to consider techniques like paired-end sequencing mentioned in Section 1.2.1.

Goal 3: *kASA needs to support current techniques and also accept any length of sequences.*

Because metagenomic data also contains viruses which mutate at a faster rate than most organisms [73], we have to think about robustness.

Goal 4: *kASA must be robust to nucleotide changes in either the reference or sequence.*

Another goal is to provide as much compatibility and ease of use as possible and useful. We also have to avoid needing admin privileges or a specific operating system.

Goal 5: *kASA must be easy to install and to use on common hardware and operating systems.*

This culminates in the final objective of this dissertation:

Goal 6: *kASA should:*

1. *correctly and efficiently identify any valid sequence belonging to a known species,*
2. *not use more memory than given by the user,*
3. *provide easy access as well as a variety of options for any user,*
4. *be as independent of platform and other software as possible,*
5. *provide significant scientific value for metagenomic and thematically related studies.*

Chapter 2

Taxonomic profiling

This chapter contains an overview of existing ideas and software as well as new ideas for taxonomic profiling. Section 2.1 starts by giving a mathematical formulation of the underlying problem. Then a brief introduction into existing algorithmic solutions is given accompanied by a description of selected software categorized by method. Section 2.2 presents new ideas building on conclusions from Section 2.1 and expanding them. These ideas form the basis for the implementation presented in Chapter 3.

2.1 Algorithmic solutions and existing software

The problem of taxonomic profiling can be abstracted and formulated as following:

Let M be the set of all sequences from sampled (meta-) genomic data (called "reads") and R be the set of IDs describing the order of these sequences. Then M_R is the set of tuples containing sequences in the form of strings with their respective read IDs from R. Furthermore, let S_R be a relation [74] between all possible non-empty substrings of sequences in M together with the respective IDs from R. This means that should different reads contain identical substrings, all corresponding IDs are part of the tuple. This way, the substring can function as the attribute name or "key" [74].

Let furthermore DB be the set of all genomic strings derived from a database of known genomes and T be the set of all taxonomic IDs of known genomes in DB. Then DB_T is the set of tuples where each string in DB has its respective taxonomic ID from T as second component. The relation G_T then contains all possible non-empty substrings of all genomes in DB mapped to the taxonomic IDs of their origin. Should multiple genomes share a substring, the taxonomic IDs are grouped inside the corresponding tuples as well.

Each of these sets of tuples S_R and G_T represents a relation [74] with the first

ACGTA	{1, 2}
CGATCGT	{3}
GTTCAA	{4}

Table 2.1: S_R with two reads having ACGTA as a substring

ACGTA	{9157}
CGATCGT	{9606, 9598}
AACAT	{562}

Table 2.2: G_T with two genomes having CGATCGT as a substring

ACGTA	{9157}	{1, 2}
CGATCGT	{9606, 9598}	{3}

Table 2.3: Natural join of S_R and G_T excluding keys that are not in both relations.

component, the substring, as key. The question is now:

Given S_R and G_T, what are the entries in $S_R \bowtie G_T$?

Where $\bowtie$ is the natural join operation [74]. The resulting tuples represent links between read IDs and taxonomic IDs. An example can be seen in Tables 2.1, 2.2 and 2.3.

However, generating such large sets and searching for matching keys is quite costly in terms of computational speed and space requirements. Therefore, it is preferable to avoid creating all possible substrings. In the following, methods that work directly with the original strings are explained.

A specialization of the aforementioned problem is to compare one string from the sampled data M_R with one string from the database DB_T. This problem can then be seen as a "longest common substring problem" [75]. If solved and applied to all tuples in $M_R \times DB_T$, it can be used to answer the general question stated above. In bioinformatics, the longest common substring problem translates to finding the longest match between at least two genomic strings. This can be solved via finding an "alignment" [76] between every pair of two strings. An alignment is an arrangement of two strings that is intended to achieve the highest match of equal letters by appropriately shifting one of these strings. Optimal global alignment algorithms like the Needleman-Wunsch [77] dynamic programming solution or local ones, e.g. developed by Smith & Waterman [78], however have a worst case time complexity of $\mathcal{O}(n \cdot m)$ for two sequences of lengths n and m. Therefore, they often become inefficient for large data sets. Examples of software using alignments to answer the question stated above can be found in Section 2.1.1.

A better way would be to create "suffix trees" [79] for all sequences in DB_T and then search for every string of M_R in them. A suffix tree is an efficient data structure that stores all suffixes of a string or text and their positions. Creating a suffix tree can be done in $\mathcal{O}(|DB| \cdot n)$ with n being the length of the longest genome. Searching is then quite fast with a time complexity of $\mathcal{O}(m)$ where m is the length of a string from

M_R. However, it typically consumes much more space than the original string when constructed (up to $28 \cdot m$ bytes vs m bytes [80]) and is thus impractical for usage in metagenomics with large comprehensive databases. This problem was reduced by "suffix arrays" [81] which use less space while still maintaining a good search time complexity. A method using these suffix arrays together with the Burrows-Wheeler-Transformation (BWT) [82] is the FM-Index [83]. Let db_{all} be the length of all strings in DB_T combined, then this can be done in $\mathcal{O}(db_{all})$ time and space complexity at most [82]. As this has to be done only once, the time complexity for building this index is irrelevant for the search itself. Another advantage of the FM-Index is that random access to the underlying suffix arrays is still possible. Therefore searching a string from M_R in this index can be done in $\mathcal{O}(m + o)$ time where m is the length of that string and o is the occurrence, meaning how often that string is found. Assuming that every string in M_R has the same length (which is not unusual in DNA sequence data), that the search is done for every string in M_R, and that every string only matches once, we arrive at $\mathcal{O}(|M_R| \cdot m)$ time. That run time complexity has a major drawback however: As mentioned in Section 1.1, the third-generation sequencing techniques allow for sequences on a megabases level, which would make m a rather large factor for this approach. Therefore, the FM-Index is usually only efficient for short strings [84]. Software that uses the FM-Index for taxonomic profiling can be seen in Section 2.1.2.

Another solution to the problem of taxonomic profiling would be to create substrings with a small fixed size from the strings in DB_T and/or M_R. More generally:

Definition 1 (k-mer)**.** *Given a string of length $L \in \mathbb{N}$, then a k-mer is a substring with length $0 < k \leq L$ of that string.*

When retaining as much information as possible is important, this substring sampling can be done with overlapping substrings: For example, the string "ACGAGT" has four overlapping substrings of length three "ACG", "CGA", "GAG", and "AGT". These overlapping substrings of length k are called "overlapping k-mers".

For the following argument, let K be the multiset containing all k-mers sampled from strings of DB_T. When using overlapping k-mers, the number of strings in K increases by $L-k+1$ for every string of length L in DB_T. Therefore, $|K| \geq |DB|$ for every $k \leq L$ which seems like an obvious disadvantage. However, now that all strings in K have equal length, a hash table can be used to efficiently store and search for these strings. Building the index of hashes can be done in $\Theta(|K|)$ time and space if only unique k-mers are saved. This is because the size of the hash table can also be calculated upfront so no rehashing/resizing has to be done. Not using unique k-mers increases the space and time required for building and searching, depending on the collision resolution strategy used. Also sampling k-mers from strings in M_R and then searching in this table can be done in $\Theta(1)$ time for every k-mer, if no collisions occur. This results in a time complexity of $\mathcal{O}(L-k)$ with L being the longest string in M_R. For all strings in M_R we get a time complexity of $\mathcal{O}(|M_R| \cdot (L-k))$. By

using the same assumptions of equal length m for all strings in M_R as above for the FM-Index, we get a total time complexity of $\mathcal{O}(|M_R| \cdot (m - k))$. We see that although the hash table based index uses more space than the FM-Index, the factor after $|M_R|$ is smaller since $m - k < m$, $\forall\, k > 0$. It is therefore a valid alternative solution. See Section 2.1.4 for software using this approach. Further applications of k-mers are for example assembly [85] and alignment-free analysis [86].

If retaining all information from strings in DB_T is not needed or desired, hashes can also be derived from these strings directly. Approaches like minhashing [87] generate hash values from k-mers that characterize these strings best and saves those. This reduces the space used to store the table and if also applied to strings from M_R, speeds up the process of searching significantly (see benchmarks in Section 4.3.3). Existing software using hash tables without overlapping k-mers can be seen in Section 2.1.3.

We would also like to elaborate how the results can be given to the user in meaningful ways. The usual form of an r-identification output (meaning the output generated for every read) would be a list containing matching taxonomic IDs for every read ID. Because having multiple taxonomic IDs for one read ID can be too much information, some programs use a so called "lowest common ancestor (LCA)" approach [88]. The taxonomy of all organisms can be seen as a directed acyclic graph (see Section 1.1). It is thus possible to traverse the tree upwards and look for the lowest node that still summarizes the matched taxonomic IDs. Let, for example, five species belonging to the same genus match the same sequence. Then the output could be either all five species' IDs or one ID: the genus of those species (since genus is only one node above species). Some methods to be discussed in the sections below use this LCA tree for their r-identification output. We would like to mention beforehand that we do not employ this idea in `kASA`. The reason is best explained with another example: assume a genus of 100 species and a match in five species but not in the other 95. Most LCA based software would then go upwards the tree to the genus and lose information. Because most output is post-processed by scripts anyway, we found that imposing such a decision for the initial output to be disadvantageous for the user.

Finally, the following sections describe the solutions of other authors which inspired some ideas we have incorporated into `kASA`. Given the diversity of metagenomic data and studies with different requirements, it is not surprising that multiple solutions emerged. Due to the sheer number of them, we are presenting only a subset of all existing software chosen by ideas that contributed to `kASA`. Also please note, that the classification based on method is not strict. Some methods use k-mers as well as hashing, for example. Rather, it is meant to group software that are similar in their core method.

Adding to that, we introduce two terms often used when comparing software: sensitivity and precision. Sensitivity is the fraction gained by dividing the number of

correctly identified reads by the total number of reads, whereas precision divides it by the number of identified reads. The formulas can be seen in Section 4.1.2. This means that sensitivity is high if a software is able to identify as many reads as possible. On the other hand, precision is high if the reads are truly from the reported genomes. Some programs choose a trade-off between the two measurements, for example decreasing sensitivity but increasing precision by applying thresholds to their scoring schemes. It is also possible to relax the matching process and allow for errors which can decrease the precision but increase sensitivity.

2.1.1 Software using alignments

The best known alignment based software is BLAST [89] or megablast [90]. It is using a heuristic to search for short matches, called "seeds". Gathering these seeds from a string of M_R is done by first creating overlapping k-mers. Second, from each k-mer a list of similar words is generated by using a similarity matrix [91]. Every element of this list gets a score and the ones with the highest score are saved in a search tree data structure. This helps to quickly compare the high-scoring words to the database sequences later. Usually, a BLOSUM62 scoring scheme is used [91, 92]. Afterwards, the strings in DB_T are searched. Matching seeds are elongated until a score given to the alignment falls below a certain threshold or the sequence matches entirely. Gapped alignment is allowed but influences the score negatively for every gap needed. Extending the match is done in both directions. While BLAST is good for single sequence search, megablast is more suited for multiple input sequences. It concatenates every string in M_R and then uses BLAST on this long new string. At the end, individual alignments are gathered in a post-processing step. Both algorithms are still widely in use, not least because of the easy access via a web interface [93]. The big disadvantage of BLAST and megablast however, is the high computational effort needed to process large numbers of sequences against a large database [94].

This problem was handled by another software we would like to present: DIAMOND [95]. It also uses a seed-and-extend strategy as done in BLAST, although with protein sequences only. The main differences are the usage of spaced seeds [96, 97] and the "double indexing" scheme [95]. Spaced seeds are seeds which contain relevant and irrelevant characters (that match to everything) but at the same time are longer than their contiguous part. For example, a seed of length 10 can be transformed to a spaced seed by considering two positions as irrelevant and then adding two other characters at the end, making it a spaced seed of length 12. This increases the probability of a match, since these space seeds now match strings that had other characters in the irrelevant positions previously. Double indexing describes the process of creating not only seeds for the strings in M_R as done in BLAST, but also those in DB_T. Should two seeds be similar, the position inside the string from DB_T is then fetched and the match is extended via calculating a Smith-Waterman alignment [78]. These sets of seeds generated from strings in M_R

and DB_T are sorted so that the comparison for equality can be performed with a time complexity that is linear in the size of these sets [98]. Because the number of seeds is significantly smaller than the number of sequences or genomes, this speeds up the process of finding alignments and at the same time ensures at least the same sensitivity as BLAST [95].

Using alignment-based software has the advantage that it is possible to get the exact positions where strings from M_R and DB_T are similar. This helps in deciding whether a string from M_R has a known biological meaning. We did not incorporate an alignment-based approach into `kASA` but the match-and-extend strategy was a good inspiration. The double indexing strategy of DIAMOND is something we also took ideas from, especially the comparison of two sorted sets in linear time.

2.1.2 Software using the FM-Index

As described above, the FM-Index [83] is a fast method for taxonomic profiling as long as the strings in M_R are relatively short. Because there still are sequencing technologies that generate short reads [99] (250 bp), this method remains relevant. We present two implementations, each having different approaches in using the FM-Index.

The first program is called Centrifuge [100]. It creates an FM-Index from strings in DB_T. This index is compressed by discarding genomic data that is $\geq 99\%$ similar to other genomes inside the index. Similarity is calculated by counting identical randomly sampled 53-mers. Segments belonging to two or more species are labeled correspondingly. Taxonomic profiling of strings from M_R is done with seeds of length 16 bp which are matched exactly and extended as far as possible. This methods was also used in BLAST, see Section 2.1.1. In case of a mismatch, the letter(s) is(/are) skipped and the matching starts anew. Based on the matched length, a score for every matched species is calculated. The species with the highest score then get assigned to the matched read. Should more than five species match the same read equally, the LCA method is applied. Additionally, Centrifuge can calculate a profile of the whole file which contains the abundances for every matched species. Here, an Expectation-Maximization(EM) [101] algorithm is used to calculated the likelihood of an abundance in the profile.

The second FM-Index based program that significantly influenced `kASA` is Kaiju [102]. What characterizes Kaiju, is its usage of amino acids instead of DNA. Therefore, its FM-Index is composed of protein sequences translated from genes of the genomes from DB_T. This reduces the index size and adds robustness against mutations or sequencing errors due to codon degeneracy [103]. For identification, the input sequences are translated in all six frames and fragmented at stop codons (for more details on "frames" and translation, see Section 2.2.2). To speed the querying process up, fragments below a certain length are discarded. Afterwards, the fragments are matched exactly using a modified backwards search. Should two or

more taxa score equally, the LCA method is used as well. The software can use a "Greedy" mode which works similar to the seed-and-extend strategy of Centrifuge except substitutions of amino acids in the BLOSUM62 matrix [91] are allowed.

We took three main ideas from these programs: presenting multiple matched taxa per read, the calculation of a profile, and the inherent robustness of the genetic code when using protein sequences.

2.1.3 Software using hashing

We would first like to present two notable programs using hashing: MetaCache [104] and Kraken2 [105]. Then, we present Ganon [106] as an interesting alternative to the other two.

In general, minhashing [87] tries to quickly estimate the Jaccard similarity coefficient [107] $J(A,B) \coloneqq \frac{|A \cap B|}{|A \cup B|}$ in order to measure equality between two sets of strings A and B. The closer the value is to 1, the more similar they are. This is done via creating a unique set of hash values per "window" - substrings of a fixed length. These hash values DB_h are generated for every window of every string in DB_T and stored into a hash table. Should now the same method be applied to all windows of every string in M_R resulting in M_h for every window, the number of collisions gives an estimator on $|M_h \cap DB_h|$. Because of the fixed windows, the number of elements in $|M_h \cup DB_h|$ are quickly calculated. It is now easy to estimate $J(M_h, DB_h)$ for every window. If done for all windows, it can be deduced how similar the two strings are. Now follows the implementation of this principle in MetaCache and then a similar method is presented in Kraken2.

In MetaCache, the windows are overlapping substrings of strings from DB_T with a pre-defined length l. They are decomposed into k-mers ($k \leq l$) and then hashed to gather the k-mers belonging to the smallest unique set of hash values. Uniqueness is measured by a resemblance function that decides if two k-mers are similar and if so, puts them into the same multiset. Of all multisets generated from the string, the smallest multiset is then used to characterize the string and the corresponding hash value is saved into the table together with information about the genome. For an identification, the same method is applied to the reads in M_R which are queried against the hash table. The scoring function includes results from every window and prints out the taxon with the most k-mer counts. If two or more taxa score equally, the LCA method is used to deduce the result. Values for l, k and the number of counted hash values s per window are user definable but need to be fixed when building the index and not changed afterwards for that index.

In Kraken2, the authors do not directly use minhashing, although their approach resembles it strongly. Therefore, a few differences exist. For example, unique hashes representing strings are called minimizers. They are built from m-mers which are the lexicographically smallest strings of all k-mers sampled from windows of length

l with $m \leq k \leq l$. These minimizers can also use a spaced approach as mentioned in context of DIAMOND in Section 2.1.1. During the building step, reference genomes from DB_T are converted into distinct minimizers and together with the taxonomic ID used to populate a hash table. If a collision occurs and the taxonomic IDs are close to each other inside the taxonomic tree, the taxon is updated to the next higher level (LCA method). Should the minimizer be different during collision, linear probing is used to calculate the next possible position inside the table. The identification works by creating minimizers from strings of M_R. Here, the k-mers used to create the minimizers are kept. If the hash value of a minimizer is found, the respective taxonomic ID is assigned to the k-mer. For every set of k-mers per sequence from M_R the LCA method is applied and the resulting taxonomic ID is written to the output file. A downside of the minimizer approach is that if originally a minimizer refers to multiple different k-mers, the found LCA would be assigned to all of them even though they are not identical.

Another interesting hashing approach was implemented in Ganon. A feature often neglected, is the ability to add genomes to an existing index without having to build it anew. Since new genomes are still being sequenced, a user might want to keep their index up-to-date. In Ganon, this is achieved by using so called "interleaved Bloom filters" [108] as a data structure for its index. In short, a Bloom filter is a hash table which, instead of saving pairs of keys and values, sets n bits to 1 where n is the number of hash functions applied. For a search, one has to just check if all positions equal 1 and if not, the key could not have been saved into the hash table. However, this introduces a chance for false positives since collisions are not handled and the search could return true even though the key has never been entered. Adding entries to this kind of hash table is an incremental and highly parallel process thus creating and updating this index is not only fast but trivial. Keys in this case are k-mers belonging to a preclustered group of taxonomically close genomes. During identification, k-mers from input data are searched against this index and scored depending on the number of mismatches. Should a high enough count of matched k-mers be reached, a read is considered identified. Afterwards, an LCA algorithm *can* be used to group multiple matches together.

All presented methods that use hashing to minimize the number of saved identifiers from DB_T have high speed and good scalability. However, due to the usage of hashing together with exact matching, there is no real robustness against mutations or sequencing errors. We nevertheless took a lot of good ideas from these approaches. The ability to update the index, for example, is something we think is necessary for modern software. Furthermore, while not explicitly said but observed during testing, MetaCache has a very good usability regarding installation, documentation and parameters. We took much inspiration in that. Lastly, Kraken2 allows for fixed sized hash tables, limiting the usage of memory. Especially for researchers with limited disk space, this can be a good choice which we found useful as well.

2.1.4 Software using *k*-mers

The main difference between software from the former Section 2.1.3 and k-mer based methods we will now present, is the amount of information stored. Usually, k-mer based methods try to combine fast searching with high confidence in their prediction. Therefore, almost all k-mers generated from strings in DB_T are saved to the index. Two approaches using this strategy are Kraken [109] and CLARK [110].

Kraken follows a strategy which assigns k-mers of length 31 (by default) on the DNA level to a lowest common ancestor (LCA) tree. This tree is built in a preprocessing step which takes taxonomic information from the NCBI [55] and genomic information from a fasta file. It then counts all k-mers with a third party library and assigns the respective taxonomic labels to them. If a k-mer is counted in multiple species, it uses the LCA method. Furthermore, Kraken clusters similar k-mers together to optimize cache usage. Similarity is measured by a so called "minimizer" which consists of the first 15 letters of a k-mer. k-Mers with the same minimizer are grouped together in a lexicographically sorted part of the hash table representing the index. During identification, k-mers from reads are generated, counted, and searched in the index. Following the resulting path inside the LCA-tree with the highest score yields the most probable taxon for each read. Because Kraken is a selective classifier which discards k-mers not belonging to the most probable identification path, precision is prioritized at some cost to sensitivity. Unfortunately, for this method to work, Kraken needs to load the entire index into primary memory.

In contrast to saving all k-mers from a string from DB_T, CLARK is discarding identical k-mers. What remains are "discriminative" k-mers, which are unique for every species. This significantly reduces memory requirements while increasing confidence in a match. For the index creation step, CLARK saves 21-mers of every taxon (called *targets*) together with information of its origin in a hash table. Collisions are handled with separate chaining [111]. Afterwards, any k-mer belonging to multiple taxa is removed. The identification step is simply the search of generated k-mers from strings of M_R in the index. For every read in M_R, the taxonomic ID with the highest number of matched k-mers is assigned to that read. Additionally, every assignment gets a confidence score. The authors argue that due to the index consisting of discriminative k-mers, no conflict of two or more equally high scored taxa should arise which would need resolving. This is why CLARK operates on a single taxonomic level as opposed to an LCA method.

The minimizer principle from Kraken inspired us to use it to check, whether a k-mer can be in the index before actually searching for it. The use of a single taxonomic level as done in CLARK is something we also employ in `kASA`. We argue that if the taxonomic profiling, for example on species level, is successful, higher order classification can always be done afterwards to increase readability of the output. More generally, we also use a k-mer approach for `kASA`, although not with a hash table as index. The reason for that is given in Section 2.2.3. The main benefit

in using k-mers is that they can be sampled from sequences of any length. This implicitly achieves the last part of Goal 3 that required our software to accept any length of sequences. We now introduce the new ideas of `kASA` in Section 2.2.

2.2 New ideas

Most of the programs presented in Section 2.1 have one or more of the following four major drawbacks:

Problems:

1. *Focus on either sensitivity or precision [102, 104, 105, 109, 110]*
2. *No built-in robustness against mismatches [104, 106, 109, 110]*
3. *Primary memory usage scales with index size (all except Ganon)*
4. *Dependency on the operating system (almost all of them run exclusively on Linux)*

The following sections further describe the problems and present solutions to them. As for a solution to Problem 4, we would like to refer the reader to Section 3.1 in Chapter 3.

2.2.1 Dynamic *k*

As mentioned in Section 2.1.4, the user must choose a fixed k for the index and input in a k-mer based approach. However, which value for k should be used is not always straightforward: If k is too large, it may not be possible to exactly match the k-mers. If k is too small, two related species may be more difficult to distinguish by the resulting score, leading to false ambiguity. We will now elaborate further but for simplicity's sake, assume the viewpoint of only one k-mer.

Let D denote a DNA string of k letters from the alphabet $\mathcal{B} \coloneqq \{A, C, G, T\}$ so $D \in \mathcal{B}^*$ with * being the Kleene star [112]. The elements of $\mathcal{B}$ represent the four bases which DNA consists of [113]. Let furthermore $G_{S1} \in \mathcal{B}^*$ be the genome of a species $S1$ with $|G_{S1}| = n$. Assuming that $k = n$, the probability that $D = G_{S1}$ is:

$$p_{eq} \coloneqq \frac{1}{4^k} \tag{2.1}$$

This is because we search for exactly one combination of letters in the space of possibilities given by the alphabet times the number of letters. However, k is usually smaller than n and therefore we have to try every position of G_{S1} as a possible starting position for a match.

The following approach stems from a publication that tries to approximate the probability of finding one string of a fixed length k in a longer string of length n at least once [114] (see Proposition 2). One of the assumptions necessary for this is that all positions where a match could start are independent of each other. With this, the number of Bernoulli trials we use to determine whether D is completely matched is $K \coloneqq n - k + 1$. We saw that formula before in Section 2.1 when calculating how many k-mers can be sampled from a string. The probability that G_{S1} contains D as a substring at least once can therefore be approximated with:

$$P(G_{S1} \text{ contains } D) \approx \sum_{i=1}^{K} \binom{K}{i} {p_{eq}}^{i} (1 - p_{eq})^{K-i} \tag{2.2}$$

$$= 1 - (1 - \frac{1}{4^k})^K \tag{2.3}$$

This means that the larger k is, the less likely it is for G_{S1} to contain D. If a match between a fairly large D and the genome G_{S1} occurs, it is quite unique and very likely, that the DNA sequence D belongs to the species $S1$. Let us assume now, that a species $S2$ exists and that it is biologically related to $S1$ so its genome G_{S2} is similar but not equal to G_{S1}. What is the minimum length D must have to distinguish whether it belongs to $S1$ or $S2$? As the reader can imagine, this is not easy to answer and depends on the data. Therefore, choosing a large enough k is crucial.

If k is too large however, we run into another problem. Assume that a base $b \in \mathcal{B}$ changes to another letter of $\mathcal{B}$ (for example, A to C) randomly. This event is usually called "mutation" in accordance to its biological pendant. The probability is here independent of the actual base for simplicity's sake and given as p_{mut}. The probability of at least one such mutation occurring in D is then:

$$P(D \text{ mutates}) = 1 - (1 - p_{mut})^k \tag{2.4}$$

We see that with increasing length of D, the probability of a mutation increases as well. This means that if D originally belonged to $S1$ but mutated before that matching process, we cannot determine that origin should k be too large. This leads to the problem of choosing an appropriate k which is representative enough to make it possible to uniquely assign D to a species and at the same time minimizes the chance of a mismatch due to mutations that occurred before. This trade-off also determines whether sensitivity or precision is the priority, as a large k increases precision but decreases sensitivity, for a small k the reverse is true.

To counteract this Problem (which corresponds to Problem 1), we reinterpret the extend strategy of BLAST and use a range of several k's $[k_L, k_H] \subseteq \mathbb{N}$. Should the smallest k_L-mer match, we increase $k \in (k_L, k_H]$ until either a mismatch happens or

$k = k_H$. This ensures that at least some inference to a group of species can be made should the k_H-mer not match. If repeated for all overlapping k_H-mers, the count of matched k-mers for every species results in a taxonomic profile. Furthermore, we only need to save the largest k_H-mers. This is because smaller k-mers can be sampled from larger l-mers ($k \leq l$) when the DNA string is padded by ($k_H - k_L$) letters. See Figure 2.1 for a visual example.

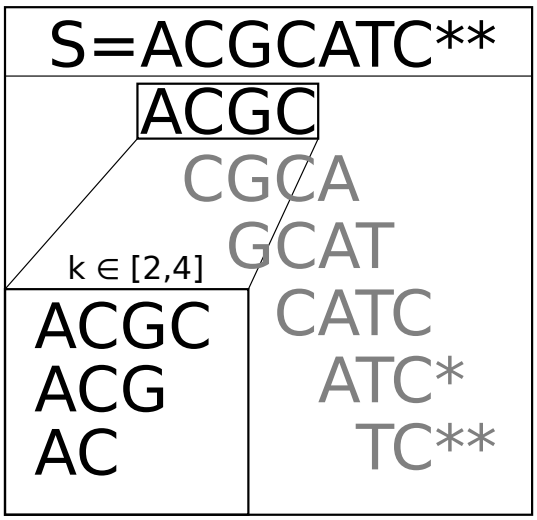

Figure 2.1: Example for $k \in [2, 4]$ in which smaller k-mers are sampled from larger ones. Due to the padding, only the prefix with the respective length of k needs to be sampled. Note that a k-mer containing a padding symbol will only match up to this symbol but never the symbol itself.

If $k_L = 1$ and $k_H = \max_{S \in \{\text{all species}\}} |G_S|$ is the length of the largest genome of all all species, this approach resembles an alignment-based strategy. However, k_H is usually limited by external factors like machine word size or memory size and thus much smaller (< 100 bp). Although this increases the execution time linearly in the length of the interval, the information gained should prove worthwhile. The user is able to freely change this interval at the beginning of every start of `kASA`, hence we name this approach: dynamic k.

2.2.2 Translation

The next problem we wish to address is Problem 2 which refers to the lack of robustness against mutations or sequencing errors in many programs. The necessity for this was already mentioned in the section above. While developing a solution to this, we were being inspired by nature and Kaiju (see Section 2.1.2) to use the so called "codon degeneracy" [103]: While translating an RNA sequence to a protein consisting of amino acids, two or more triplets of bases C, G, U, A often code for the same amino acid. See Table 2.4 for the standard codon table often seen in nature [115].

This introduces the possibility of "silent mutations" where the base changes but the amino acid does not. Therefore, matchings on amino acid level would be possible even though two letters mismatch on DNA/RNA level. Furthermore, in contrast to the translation in nature and Kaiju which starts at a certain triplet (e.g., AUG in Eukaryotes and Archaea), we include non-coding regions as well since they are known to be conserved as well (transcription factor binding sites for example [116]). For simplicity, we will use DNA instead of RNA as base for our arguments in this section.

Table 2.4: Standard codon table for the translation from RNA triplets (codons) to the respective amino acid (AA) as well as the stop codons (St).

Codon	**AA**	**Codon**	**AA**	**Codon**	**AA**	**Codon**	**AA**
UUU	F	UCU	S	UAU	Y	UGU	C
UUC	F	UCC	S	UAC	Y	UGC	C
UUA	L	UCA	S	UAA	St	UGA	St
UUG	L	UCG	S	UAG	St	UGG	W
CUU	L	CCU	P	CAU	H	CGU	R
CUC	L	CCC	P	CAC	H	CGC	R
CUA	L	CCA	P	CAA	Q	CGA	R
CUG	L	CCG	P	CAG	Q	CGG	R
AUU	I	ACU	T	AAU	N	AGU	S
AUC	I	ACC	T	AAC	N	AGC	S
AUA	I	ACA	T	AAA	K	AGA	R
AUG	M	ACG	T	AAG	K	AGG	R
GUU	V	GCU	A	GAU	D	GGU	G
GUC	V	GCC	A	GAC	D	GGC	G
GUA	V	GCA	A	GAA	E	GGA	G
GUG	V	GCG	A	GAG	E	GGG	G

Reducing three letters from the DNA alphabet (total size of $4^3 = 64$) to one letter from the amino acid alphabet (size of 22) decreases memory usage. The former needs six bits per triplet, whereas the latter only needs five per letter. This however leads to an inherent information loss. To counter this, we use the first three bases as a starting in each case and sample k-mers in "frames" (see Figure 2.2). We now show that this really is conserving all information needed for taxonomic profiling by proving that the translation can be inverted.

Let $\mathcal{A}$ be an alphabet of size n with $1 \leq n \leq 31$, $n \in \mathbb{N}$ and $\mathcal{B} \coloneqq \{A, C, G, T\}$. Let furthermore S be a word consisting of at least three letters from $\mathcal{B}^*$, so $S \in \mathcal{B}^*$ and $|S| \geq 3$. S therefore represents a DNA sequence.

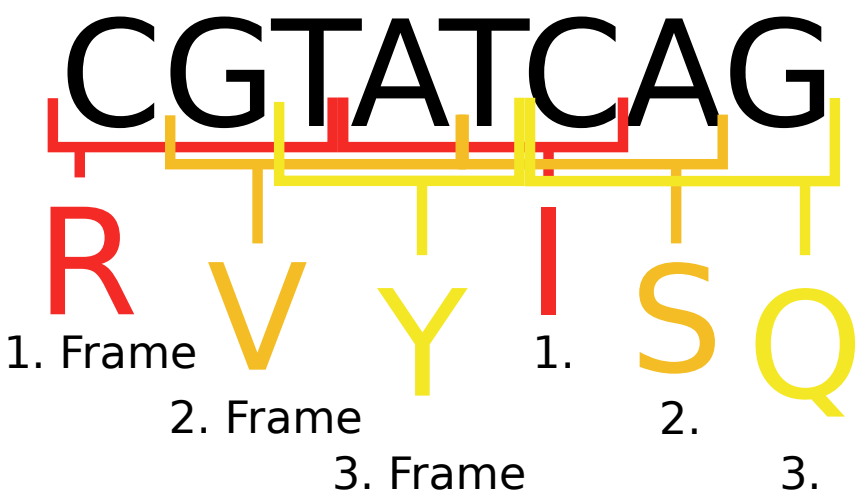

Figure 2.2: Translation in three frames resulting in three words coded with amino acid letters: "RI", "VS", and "YQ".

Let $\texttt{code} : \mathcal{B} \times \mathcal{B} \times \mathcal{B} \to \mathcal{A}$ be a function with the following property:

$$\begin{aligned} \texttt{code}((b_0, b_1, b_2)) &\mapsto a \in \mathcal{A} \text{ as given in the codon table.} \\ \text{If } \texttt{code}((b_0, b_1, b_2)) &= \texttt{code}((c_0, c_1, c_2)) \\ &\Rightarrow b_1 = c_1, \\ (b_0, b_1, b_2),\ (c_0, c_1, c_2) &\in \mathcal{B} \times \mathcal{B} \times \mathcal{B} \end{aligned} \tag{2.5}$$

This means that every two triplets of bases which code for the same amino acid have the same base in the middle. A look at Table 2.4 shows that this property is mostly true for the standard codon table with exceptions for "S" which has two middle bases and the stop codons. We therefore use a slightly modified codon table in `kASA` we now name C. Next, we model the translation itself.

$\texttt{translate} : \mathcal{B}^* \to \mathcal{A}^* \times \mathcal{A}^* \times \mathcal{A}^*$ is now a function for translating the DNA sequence S into three amino acid-like sequences with iterative application of `code`. If `code` is applied with a shifted start, we get the conversion in three frames mentioned above. So the resulting words w_0, w_1, $w_2 \in \mathcal{A}^*$ are as follows:

$$w_j := \bigoplus_{i=0}^{\lfloor \frac{|S|-j}{3} \rfloor} \texttt{code}(S\,[j + 3 \cdot i, j + 3 \cdot i + 2]),\ j = 0, 1, 2$$

where $\bigoplus$ is the string concatenation and $S[a, b]$ is the substring of S starting at position a and ending at position b.

Observation 2. *w_0, w_1, w_2 are created from overlapping triplets, which means that for a DNA sequence b_0, b_1, b_2, b_3, b_4, ... the triplet of the first frame b_0, b_1, b_2 shares*

two letters b_1, b_2 with the second frame and one letter b_2 with the third frame. Furthermore, the second frame b_1, b_2, b_3 shares b_2, b_3 with the third frame which starts with b_2, b_3, b_4. b_3 and b_4 are now again the bases forming the first and second letter for the next iteration in the first frame.

The translation is now fully modeled. If we can show that this translation can be reversed, then the original sequence can be reconstructed. This means that all relevant information is stored inside the translated sequences. The taxonomic profiling on amino acid level would therefore report the same taxa as on DNA level.

Lemma 3. *Apart from the first and last letter of S, no information loss occurs when using* `translate`.

Proof. To show that no information loss occurs, except for the first and last letter of S, we construct the functions $\texttt{code}^{-1}$ and $\texttt{translate}^{-1}$ that we will later use to show that `translate` is reversible. Let t_0, t_1, t_2 be words consisting of amino acids from which we wish to reconstruct the DNA sequence.

$\texttt{code}^{-1}$ is created by determining each triplet $b \in \mathcal{B} \times \mathcal{B} \times \mathcal{B}$ associated with the respective letter $a \in \mathcal{A}$ from the codon table C and storing it in a set.

$$\begin{aligned} \texttt{code}^{-1}(a) = T \subseteq \mathcal{B} \times \mathcal{B} \times \mathcal{B} \\ \forall (b_0, b_1, b_2), (c_0, c_1, c_2) \in T \Rightarrow b_1 = c_1 \end{aligned} \tag{2.6}$$

This means, that for example for $t_0 = a_0, ..., a_i, ..., a_l$ with $0 \leq i \leq l \leq \lfloor \frac{|S|}{3} \rfloor$, $\texttt{code}^{-1}(a_i)$ is a set of ordered triples with the same middle component according to Property 2.5.

$\texttt{translate}^{-1} : \mathcal{A}^* \times \mathcal{A}^* \times \mathcal{A}^* \to \mathcal{B}^*$ is now constructed as follows: We apply $\texttt{code}^{-1}$ to the letters in each word t_0, t_1 and t_2 which results in multiple of these ordered sets. These sets are implicitly ordered and grouped by word.

$$T_i \coloneqq (Tr_{\{i,0\}}, Tr_{\{i,1\}}, ..., Tr_{\{i,|t_i|\}}),\ i = 0, 1, 2 \tag{2.7}$$

$$\textit{where } Tr_{\{i,j\}} = \texttt{code}^{-1}(t_i[j]),\ j = 0, ..., |t_i| \tag{2.8}$$

$$\texttt{translate}^{-1}(t_0, t_1, t_2) \mapsto \bigoplus_{j=0}^{|t_0|} \bigoplus_{i=0}^{2} tr[1],\ tr \in Tr_{\{i,j\}} \tag{2.9}$$

with $t_i[j]$ being the j-th position in the string t_i and $tr[1]$ being the middle position in any string from $Tr_{\{i,j\}}$. This means, that for example, $Tr_{\{0,0\}}, Tr_{\{1,0\}}$ and $Tr_{\{2,0\}}$ must each contain at least one triplet where the bases match in the positions given in Observation 2. See Figure 2.3 for an example. Without loss of generality, accessing a string out of bounds returns an empty string during construction. This can happen if the original DNA string has a length that is $mod\, 3 \neq 2$ which results in different lengths for t_0, t_1, t_2.

Since the first base in the first frame and the last base of the last frame are not necessarily unique and cannot be checked by the other frames, they are considered ambiguous. Therefore, the sequence S can be reconstructed with just the amino-acid-like encoded frames except for its first and last base. This means that $\texttt{translate}^{-1}$ is a valid inverse function as:

$$\texttt{translate}^{-1}(\texttt{translate}(w)) = w[1, |w| - 1],\ w \in \mathcal{B}^*$$

□

Remark 4. *During the proof, we made the implicit assumption that the ordering of the frames are as* `translate` *created them. However, should this ordering be disturbed, we can try to use the constructed* $\texttt{translate}^{-1}$ *anyway because the reconstruction will fail if the ordering is not correct due to Observation 2. Therefore no other combination would generate a full-length sequence in the reconstruction process. Since there are only six possible combinations of* t_0, t_1 *and* t_2 *we get the correct DNA string eventually.*

Remark 5. *As written above, the standard codon table cannot be constructed with* `code`*, because prerequisite 2.5 is not satisfied ("S" has two middle bases). To fix that, one must first split the amino acid "S" into two letters ("AGT" and "AGC") and second give the stop codon "TGA" an additional letter as well. The default codon table used in* `kASA` *implements the additional stop codon but does not introduce a new letter to split "S" to be compatible with already converted amino acid sequences. Benchmark results for both the standard codon table and the one with a split "S" did not differ noticeably, so an approximation of* `code` *(by the standard codon table) is sufficient.*

Remark 6. *One can reconstruct the first and last base of S, if the alphabet* $\mathcal{A}$ *is restricted further: The first base of every triplet must be identical for every assigned letter* $a \in \mathcal{A}$*. Secondly, six instead of three frames are used (so the reverse complement is created and translated as well). A possible function* `code` *using a 16-letter alphabet maps each combination of the first two bases in a triplet to a unique letter, so*

$$\texttt{code}_{16}((b_0, b_1, b_2)) = code_{16}((c_0, c_1, c_2)) \Rightarrow b_0 = c_0 \text{ and } b_1 = c_1,$$
$$(b_0, b_1, b_2),\ (c_0, c_1, c_2) \in \mathcal{B} \times \mathcal{B} \times \mathcal{B}.$$

Remark 7. *This proof offers another insight: Even though the frames together contain almost all information of the DNA sequence, the translated words* t_0, t_1, t_2 *themselves do not. This means that should we create k-mers from one or all of these amino acid strings, it may very well match erroneously. This can be seen when using a different coding alphabet like the 16 letter alphabet described in Remark 6. It implies that every third letter of the sequence is ignored for each frame which leads to higher robustness but also reduced precision. We therefore assume this to*

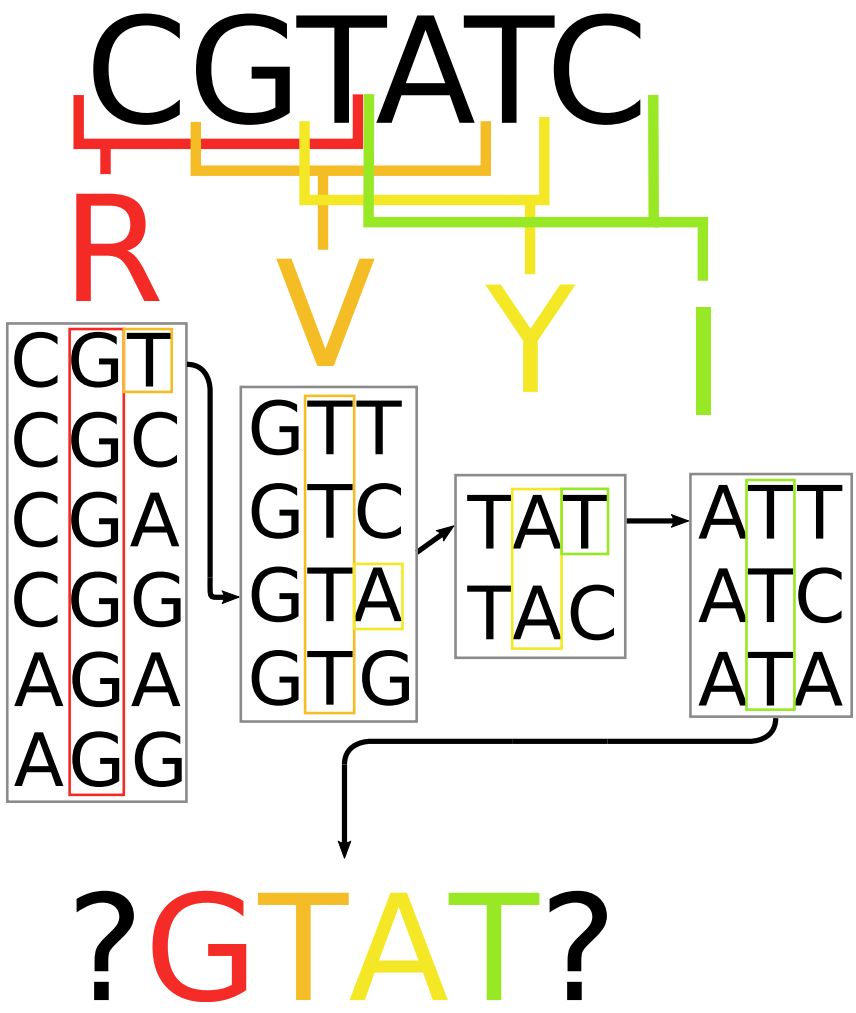

Figure 2.3: Translation and inversion based on the modified standard codon table C. It is visible that the DNA string is translated in three frames resulting in three different letters. For the reconstruction, the sets as given by $\texttt{code}^{-1}$ in Equation 2.6 are fetched from C. Then the middle letter of every set is written down and concatenated to the reconstructed DNA string. There is always a triplet in every set which has a last letter that maps to the first letter of the next set, these are marked in color. Because the first letters in "R" and the last letters in "I" are different, they cannot be reconstructed.

be a case of: "The sum is bigger than its parts". If the context which k-mer belongs to which frame is destroyed during lexicographical sorting, we might lose too much information and thus suffer a decrease in precision. We suspect that the standard conversion alphabet performs well because there might be an intrinsic balance between information conservation and robustness for each k-mer in it. We will see in Section 5.1 that there is empirical evidence to support this.

An implementation of the constructive proof can be found in `https://github.com/SilvioWeging/kASA/tree/master/scripts/reconstructDNA.py`.

This shows, that using an abstraction to amino acids together with using frames is a valid method of improving robustness to mutations per converted k-mer. At the same time, we do not really lose the information of the DNA string. So we can consider Goal 4 to be achieved. Because we designed `kASA` so that the translation table is changeable, the approach is called "amino acid-like encoding". The only disadvantage of this approach is that using three or six frames generates more k-mers than a DNA-based approach, resulting in higher memory or disk consumption. A solution to this problem will be discussed in the next section.

2.2.3 Memory restriction

All software introduced in Section 2.1 has a similar processing structure:

1. Create an index from a database of genomic information
2. Match new data to this index
3. Use a scoring scheme to determine the taxa most likely contained in this data
4. Print out the scores and taxa in a suitable way

We opted to use the same structure in `kASA`. The main argument for this is that preprocessing saves time later, especially if the same database needs to be searched often. However, most software assumes that hardware powerful enough is used to study metagenomic data. We thought this to not be a sufficiently satisfied assumption seeing that the amount of data is likely to increase in the near future [117]. Even if Moores Law [118] upholds, handling databases and building indices from them which fit into primary memory will get harder and more expensive [119]. We therefore added a technical robustness against the size of data: keeping the index on the disk/secondary memory. The user may then give `kASA` an amount of RAM via a parameter that is not exceeded. If enough primary memory and processing power is available, it can be used to load the index into it. If not, our software takes the available resources and streams the chunks needed from the index lying on the disk. `kASA` therefore provides the user with control over memory usage and avoids Problem 3, that the primary memory usage scales with the index size, while achieving Goal 1 which stated, that `kASA` has to work independently of the number of reference genomes.

To be able to do this, we use a k-mer based approach. As indicated in Section 2.1 on page 11, it offers the ability to calculate how many k-mers are created. Thus, we know how much primary memory will be used in both cases, the index creation and the identification.

Saving and accessing our indices is done by the STXXL [120] `C++` library (see Section 3.1.2). Combining this strategy with new technologies like SSDs [121] decreases

the bottleneck of "out-of-core" approaches [122] where not all data is saved in primary memory. This enables us to handle arbitrarily large files without much performance loss. The downside of this approach is that we cannot make use of hash tables as indices. Most hash table implementations are not really working well in secondary memory data structures as every call would be a random access which is quite slow [121]. We therefore use a sorted array as index data structure in order to utilize sequential accesses as much as possible (since even on SSDs, this is still faster [123]). More on this method can be read in Section 3.4.2.

Regarding large input files, we thought it best to process each file in chunks. The size of these chunks can be calculated from the given memory restriction parameter chosen by the user. This ensures that the software has a predictable upper bound on memory usage. In Section 4.3.3 we show experimentally that the overhead generated this way does not impact execution times by a large degree.

This strategy of letting the user choose an upper bound opens up the possibility of choosing hardware which is not part of an HPCC like a Laptop/Notebook or Desktop. In our opinion, this enables more scientists to run metagenomic studies since many have access to a Laptop/Notebook but not to an HPCC.

Summary
This chapter began with a mathematical formulation of the taxonomic profiling task. We then presented already existing algorithmic solutions using that formalism. Existing programs were categorized via approach and presented in relevance to our own program: `kASA`. We furthermore derived ideas from these programs and chose a k-mer based approach. The second section introduced new ideas, like the dynamic k, the amino acid-like encoding and the restriction of primary memory usage. They aim to solve the problems of existing programs that we stated at the beginning of the section. How we implemented these theoretical ideas and solutions is presented in the next chapter.

Chapter 3

Implementation details and modules of kASA

Chapter 2 introduced existing as well as new ideas how to solve the problem of taxonomic profiling. These ideas need to be implemented in order to validate them experimentally. This chapter shows how we set out to do exactly that. It starts with Section 3.1 where we introduce various functions from different libraries we used in kASA. This is followed by Section 3.2 which shows custom data structures that we use inside kASA. They are mostly customized versions of existing data structures which is why we did not include them in the previous chapter. Afterwards, two common input file formats often used in bioinformatics are briefly presented in Section 3.3. This serves the purpose of giving the reader context as to what kASA accepts as input in the different modules which are shown in Section 3.4. They are the main part of this chapter and our code. In particular, the identify module in section 3.4.5 is important for understanding how we generate a taxonomic profile as well as an r-identification. At the end, we also show additional features and modules in Section 3.5 which are not necessary but useful for any user of kASA.

We would like to point out, that we show snippets of code to visualize how exactly we implemented our ideas. Please beware that parts of our code will change over time. It is therefore more of a snapshot of our current version. The "[...]" in the code stands for the fact that there is more code inside this class or function but it is not shown. This is mostly because these parts are not relevant to the respective method. We hope that helps stress the important parts.

3.1 Libraries

We saw in Section 2.2.3 that a memory restriction is necessary to support large databases and is thus one core idea of kASA. We therefore chose to implement kASA in C++ because it offers such manual memory management. The standardization in

1991 [124] together with the development of a standard library in 1995 [125] also made it a very flexible language which offers many existing algorithms we incorporated into our code. It is also available for many operating systems which helps achieving Goal 5 which stated that kASA must be easy to install and to use on common hardware and operating systems. We would like to present three software libraries that made the development of kASA much easier: the Standard Template Library (STL) of C++ [125], the STXXL [120], and the zlib [126]. All libraries are either part of a C++ compiler or part of the source code so that these dependencies do not need to be installed in addition to kASA. If the user downloads our source code, everything necessary to run the program will already be available. This eliminates third-party or external dependencies, often limiting the possible user base.

3.1.1 Standard Template Library

The STL [125] is part of the C++ standard and offers various generic data structures and algorithms. Unfortunately, its implementation depends on the used compiler. The consequence is that some features are available for e.g. the Microsoft Visual Studio Compiler (MSVC) [127] but not for the GNU Compiler Collection (gcc) [128] or Clang [129]. We therefore made sure to use only features available to all these compilers or used preprocessor directives to ensure correct compilation. This can be done by making the source code compliant to a standard. The source code of kASA is compatible with a compiler supporting at least the C++11 standard but recommend using one that supports C++17. The reason for that is that the C++17 standard provides for example, generic multi-threaded algorithms and platform independent file management. These will be preferred in future updates of kASA. On UNIX systems like all Linux distributions and macOS, cmake [130] is needed as well (which is not provided but only needs a C++ compiler on the system to run).

The STL gives access to "containers" like vectors (arrays of dynamic size), unordered maps (hash tables) and many more. The interfaces to these classic data structures provided us with ample possibilities to implement our modules and at the same time enough abstraction to even use custom classes. We would also like to mention a few algorithms provided by this library without kASA would not work:

- `std::sort` - uses a hybrid sorting algorithm called "introsort" [131] which switches from quicksort to heapsort and then to insertion sort depending on the size of the parts that need to be sorted. In C++17 this algorithm is parallelized with a three-way-quicksort implementation (at least in MSVC).

- `std::unique` - needs a sorted container and then deletes all redundant entries from that container.

- `std::lower_bound` - a binary search algorithm determining the smallest element in a container that is greater than or equal to the searched element. It also takes a custom comparison function which makes it very versatile.

- `std::set_intersection` - detects identical entries between two sorted containers. It also inspired our algorithm used in `identify` in Section 3.4.5.

This library is therefore essential and recommended for any `C++` programmer. Its elements are referred to with `std::` before the name of either the container or an algorithm.

3.1.2 STXXL

The Standard Template Library for Extra Large Data Sets (STXXL) [120] is an extension of the STL. Data inside the implemented containers resides in secondary memory so very large sizes can be used as STL containers do not go beyond the available primary memory. It also includes custom algorithms to handle such containers and apply algorithms like `stxxl::sort` to them. However, many of those algorithms are not in-place so a temporary file is created every time a program using this library is started. We therefore use STL containers and algorithms as much as possible but the index has to lie in secondary memory. The only container from this library that can persist on secondary memory after termination of the program is the `stxxl::vector`. We therefore save the index into an `stxxl::vector` container together with a file containing the number of elements. This is necessary because the index will be in binary format and the `stxxl::vector` interface does not calculate the number of entries from the size of the file automatically after opening it again. When the index file is opened in read-only mode, multiple arrays are created which can hold a block of data from this file in primary memory. This "paging" makes use of the caching behavior of modern PCs and decreases the latency when accessing the file sequentially [132]. The container also allows random access albeit with larger latencies. In the beginning of the development of `kASA` we used more containers and algorithms from this library, but over time we found that only the `stxxl::vector` is necessary. Because the library was last updated 2018 and planned updates are not mentioned, we will either develop our own solution to keep `kASA` up to date or use a more recent interface in the near future.

3.1.3 zlib

Many users use files compressed with gzip [133] to reduce space usage. It is therefore not desirable to decompress them before starting a software like `kASA`. Since the decompression algorithm works like a stream, we can incorporate this and directly use the decompressed data stream for our program. An interface providing this can be found in the zlib library [126] together with the `gzstream` [134] header (provides an interface to `std::ifstream`). With this, we can read compressed files without needing to save them to secondary memory first. This feature is optional, of course, and compressed files are detected automatically.

3.2 Custom data structures and classes

This section contains more in-depth information about the data structures and classes we designed or customized and implemented specifically for usage inside `kASA`. One of them is the trie data structure which is used as search space reduction to limit the number of random accesses to secondary memory in `identify`. We also created a data structure based on an array containing only true and false as values, named "Bit Array Sets". Its purpose is to use only as much memory as necessary to minimize primary memory usage. Finally, we developed a queue of tasks inside threads to enable multi-threaded calculations inside multiple modules.

3.2.1 Trie

A prefix tree or "trie" [135] is a directed, acyclic graph data structure using strings as nodes. It is used to store and search for prefixes of strings. This can be done in a time complexity that is linear in the length of the string. We wrote a modified version in which we save the edges in the form of a lookup table containing pointers, inside the nodes. This enables fast traversal because the corresponding ASCII value [136] of the letter is exactly the position inside the lookup table and only the status of a pointer at that index needs to be checked (if it is a `nullptr` or a valid address). The consequence is a larger memory consumption (256 bytes for a node and 372 bytes for a leaf) than a compact trie would have, with a full trie using at most 11.9 GB RAM. However, in all our experiments including the real world example in Section 4.4 using all fully sequenced genomes as a database for its index, the trie's RAM usage did not exceed 1.8 GB. The leaves are larger because instead of pointers, their lookup tables contain integers describing the search range of the suffixes residing in the index on the secondary memory (see Code 3.1 below). This reduces the number of I/O operations significantly because only a subset of the index must be searched (see Figure 4.19 in the following Chapter 4). A visualization of our version of a trie can be seen in Figure 3.1.

A trie is also a valid replacement for a hash table because the search needs to be interruptable. If, for example, k is from an interval from five to ten, we traverse the trie until the fifth level (in case of a match). Then we look for the lexicographically smallest and largest leaf to determine the remaining search space for the suffix (see Code 3.1 below). This rules out using a hash table because emulating this behavior with it would use too much space.

Source Code 3.1: Leaf

```
    ////////////////////////////////////////////////////////////////
struct Leaf {
  // Sizeof(Leaf): 372 Byte
  uint64_t _vRanges1[32]; // starting positions
  uint32_t _vRanges2[32]; // number of elements inside range
```

```
  Leaf() {
    for (uint8_t i = 0; i < 32; ++i) {
      _vRanges1[i] = numeric_limits<uint64_t>::max();
      _vRanges2[i] = 0ul;
    }
  }

  inline uint64_t GetFirst() {
    for (uint8_t i = 0; i < 32; ++i) {
      if (_vRanges1[i] != numeric_limits<uint64_t>::max()) {
        return _vRanges1[i];
      }
    }
    return numeric_limits<uint64_t>::max();
  }

  inline uint64_t GetLast() {
    for (int8_t i = 31; i >= 0; --i) {
      if (_vRanges1[i] != numeric_limits<uint64_t>::max()) {
        return _vRanges1[i] + _vRanges2[i];
      }
    }
    return 0;
  }

};
```

The prefixes of every k-mer from the index saved inside the trie are of length six. This value is chosen because on one hand the number of possible nodes is small enough to create a trie which fits into primary memory (as the trie size grows exponentially with every level); on the other hand, it offers the possibility of reducing the size of the index by omitting the prefix already saved in the trie. See Section 3.4.3 for more information about this.

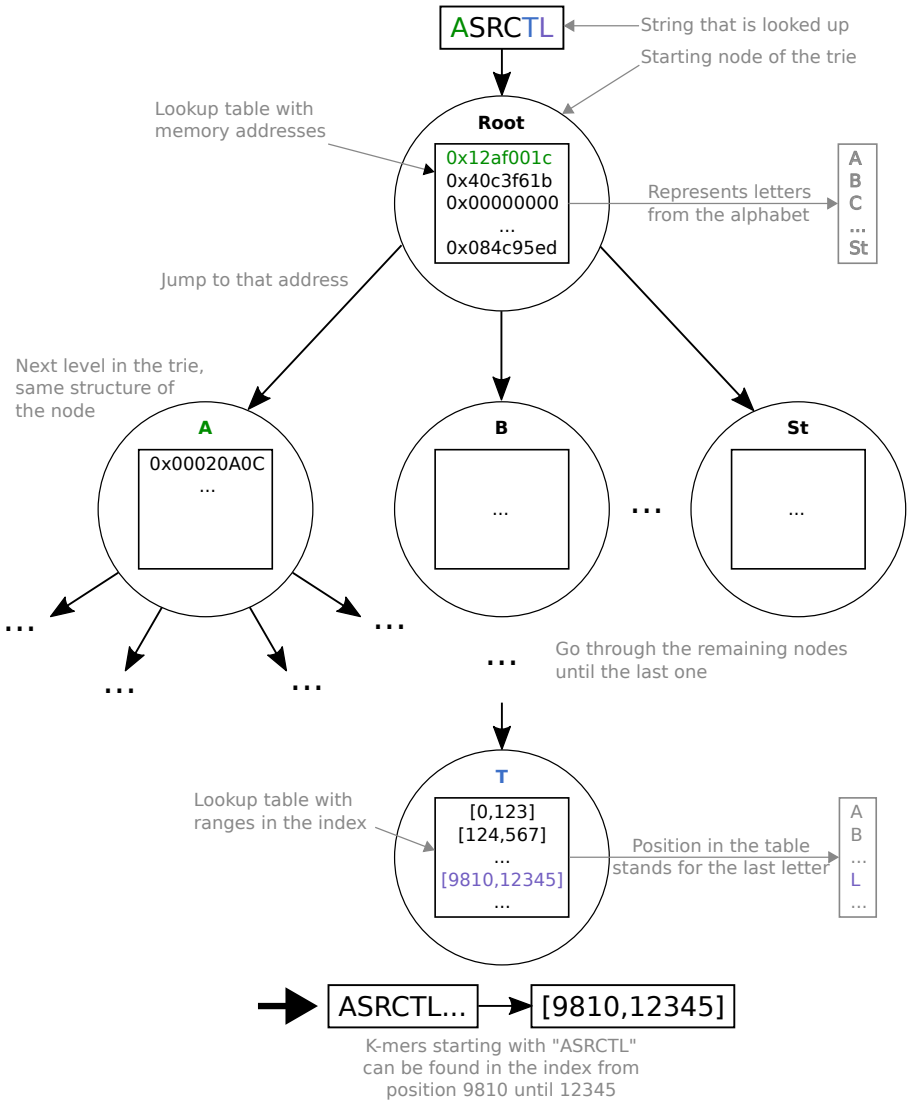

Figure 3.1: The first letter of prefix/6-mer determines the position in the lookup table of the root node. Should that memory address/pointer not be zero, this address is visited. It points to the next node which contains such a table as well. This is repeated until the penultimate letter. The leafs contain lookup tables containing the ranges. There, looking at the position of the last letter yields the range in which k-mers with that prefix can be found in the index.

3.2.2 Bit array sets

We define a bit array as an array data structure in which every element is a bit having either true or false as value. A set is here a collection in which every element only occurs once. Therefore, a bit array set is a set that uses a bit array to track, whether an element has already been inserted or not. This reduces the space requirements to a minimum and provides quick access without having to sort or search through the set. In order to implement such a data structure, we need an array of bits, a container to save the added elements in, and a pointer keeping track of the current element in the container. This pointer is useful because setting it to the beginning of the container is easier than emptying the container while resetting the data structure. The number of bits needed must be known beforehand so this data structure only makes sense if the maximum capacity *Cap* of the set is known. Because `C++` does not have a data type `bit` and the smallest one (`char` aka `int8_t`) uses one byte, a different method is used. We reinterpret an integer of size 64 bits (`uint64_t` to be precise) as an array of 64 bits. The elements in this new array can be accessed via bitwise operators, for example bit shifts. If the size of the set should be larger than 64, an array of integers is used. Access to specific bits is then calculated by dividing by 64 together with the modulo operation. See Code 3.2 for the implementation. Another advantage is that the bit array set can be reset without deleting it, avoiding de- and reallocation of memory. When resetting, the pointer is set to the beginning and every integer in the array is set to zero. This effectively sets the bit array to zero as well. If we would have taken a simple array, the number of operations would be *Cap*. With this data structure, it only takes $\frac{Cap}{64}$ operations.

Our application for this data structure is to save identified taxonomic IDs. As this must be done for every k-mer from the input, it is often reset. Furthermore, we only need to know if and which ID has been identified. Because we also know the number of possible taxonomic IDs beforehand, this data structure is therefore a fitting choice. We also enumerate these taxonomic IDs in order to map the ID to the index in the bit array (see Section 3.4.1). Should a taxonomic ID be identified during taxonomic profiling, the corresponding bit is set to one and the entry is saved into a container (an `std::vector` in our case). The pointer pointing to the current element of the container is incremented as well. If the bit already has the value one, the entry is not saved into the container. Gathering all saved taxonomic IDs in order to link the matched read IDs to the taxonomic IDs (see Section 2.1) is done by iterating over the container up to the pointer, indicating the end of the set. This way we do not need to look at the entries in the bit array as this would require |Taxa| operations, which in the average use case is much more than the number of identified taxa.

Source Code 3.2: Bit Array Set

```
////////////////////////////////////////////////////////
class sBitArray {

  unique_ptr<uint64_t[]> _arr;
  uint64_t _iNumOfBlocks = 0;
  vector<uint64_t> _vElementsArray;
  size_t _currentPositionInArray = 0;

public:
    ////////////////////////////////
    sBitArray(const uint64_t& iSize) {
      _iNumOfBlocks = (iSize + 63) >> 6; // idx / 64
      _arr.reset(new uint64_t[_iNumOfBlocks]);
      memset(_arr.get(), 0, _iNumOfBlocks * sizeof(uint64_t));
      _vElementsArray.resize(iSize);
    }

    ////////////////////////////////
    inline void set(const uint64_t& idx) {
      const uint64_t& blockID = idx >> 6; // idx / 64
      const uint64_t& fieldID = idx & 63; // idx % 64

      // if there already is a 1, this is skipped
      if (!((_arr[blockID] >> fieldID) & 1)) {
        _vElementsArray[_currentPositionInArray++] = idx;
        _arr[blockID] |= (1ULL << fieldID);
      }
    }

    ////////////////////////////////
    inline void reset() {
      memset(_arr.get(), 0, _iNumOfBlocks * sizeof(uint64_t));
      _currentPositionInArray = 0;
    }

    ////////////////////////////////
    inline vector<uint64_t>::const_iterator begin() const {
      return _vElementsArray.cbegin();
    }

    ////////////////////////////////
    inline vector<uint64_t>::const_iterator end() const {
      return _vElementsArray.cbegin() + _currentPositionInArray;
    }
[...]
};
```

3.2.3 `WorkerThread` and `WorkerQueue`

In order to execute tasks in parallel, we first used OpenMP [137] which is an API capable of parallelizing parts of code with the help of preprocessor pragmas (commands that are evaluated by the compiler). However, we soon realized that this would mean an additional dependency and compiler compatibility problems (MSVC supports only OpenMP 2.0 whereas Clang only supports versions starting from 3.1). We therefore switched to `std::thread` introduced with C++11. Unfortunately, the overhead of spawning threads soon outweighed the advantages. We therefore decided to create a pool of threads to which tasks can be handed over and the threads inside switch between work and sleep without needing to create or destroy them. This class is called `WorkerThread` and is used for the task of translating DNA sequences, converting them into k-mers or identifying them in parallel. More on the specific parallelization strategy in `identify` is described in Section 3.4.5. The class was designed for uniform tasks (those who take about the same time). So one task is assigned one thread. Having heterogeneous tasks like operating on multiple files concurrently as done in `identify_multiple` (see Section 3.5.1), motivated another class: `WorkerQueue`.

It is a variation of the thread pool principle because now tasks are put inside a queue where larger tasks get more threads. This process needs a function for priority calculation and since there is a correlation between file size and time consumption, we put identification tasks with large files at the head of the queue. This way all given cores are used since large tasks run in parallel to multiple smaller tasks. This solution approximates an optimal solution to the task scheduling problem [138].

Apart from the parallel implementation of the `std::sort` algorithm, these classes form the core for parallelization in `kASA`, but can also be used outside of it. The code for the `WorkerThread` class can be seen in Code 3.3, the code for `WorkerQueue` is not presented as it is very similar to the one shown.

Source Code 3.3: WorkerThread

```
////////////////////////////////////////////////////////
class WorkerThread {
    vector<thread> T; // hold exactly one thread
    int32_t ID = 0;
    vector<function<void(const int32_t&)>> tasks;
    condition_variable cv_worker;
    condition_variable cv_master;
    mutex taskMutex;
    bool stop = false, bStarted = false;

public:
    ////////////////////////////////////////////////////////
```

```
    inline WorkerThread() {
      tasks.reserve(1);
      startWorker();
    }
    ////////////////////////////////////////////////////////
    inline void pushTask(const function<void(const int32_t&)>& task) {
      tasks.push_back([task](const int32_t& id) { task(id); });
    }
    ////////////////////////////////////////////////////////
    inline void startThread() {
      if (tasks.size()) {
        bStarted = true;
        cv_worker.notify_one();
      }
    }
    ////////////////////////////////////////////////////////
    inline void waitUntilFinished() {
      if (bStarted) {
        unique_lock<std::mutex> lock(this->taskMutex);
        this->cv_master.wait(lock,
          [this] { return this->tasks.empty(); });
      }
    }
    [...]
private:
    ////////////////////////////////////////////////////////
    inline void startWorker() {
      T.emplace_back(
         [this](){
           while (true) {
             {
               std::unique_lock<std::mutex> lock(this->taskMutex);
               this->cv_worker.wait(lock, [this]() {return
               ↪  !(this->tasks.empty()) || this->stop; });
             }

             if (this->stop) {
               tasks.clear();
               this->T[0].detach();
               this->T.pop_back();
               this->cv_worker.notify_one();
               return;
             }

             for (const auto& entry : tasks) {
               entry(this->ID);
             }

             tasks.clear();
```

```
                bStarted = false;
                std::unique_lock<std::mutex> lock(this->taskMutex);
                cv_master.notify_one();
            }
        }
    );
  }
};
```

3.3 Input file formats

The following two file formats are the de-facto standard in bioinformatics regarding the storage of DNA or protein sequences. We briefly describe both formats to give the reader more context, how exactly sequences are put into `kASA`. They are used in almost all modules presented in Section 3.4.

3.3.1 FASTA

The FASTA format (from now on written in lower case) was initially used in a software with the same name [139] in 1985. Its simplicity quickly led to a wide adoption by succeeding programs like BLAST (see Section 2.1.1). It can easily be read and edited by various scripting languages as well as text-processing programs. The format itself consists of commenting lines, a sequence identifier line starting with '>', and the sequence itself. The sequence identifier line is usually used to refer to the source of the sequence, for example, in the accession.version format [140] of the NCBI. The genome of *Escherichia coli str. K-12 substr. MG1655* would for example, contain the following sequence identifier line: ">NC_000913.3 Escherichia coli str. K-12 substr. MG1655, complete genome". The NC_000913.3 stands for the nucleotide database [5, 62] (NC), followed by the accession (000913) which is used to uniquely identify that genome, and its version (3). This is then followed by line-by-line sequences of letters from the DNA or amino acid alphabet that make up the sequence or genome. The lengths of these lines are usually 70 or 80 letters for historical reasons. Another thing to consider is that fasta files often contain more than one sequence, for example if the genome consists of chromosomes. In these cases, they are separated by the sequence identifier lines and sometimes by empty lines. Now follows an example of a fasta file:

```
>NC_000913.3 Escherichia coli str. K-12 substr. MG1655, complete genome
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
TTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACCAA
TATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTACACAACATCCATGAAACGCATTAGCACCACC
ATTACCACCACCATCACCATTACCACAGGTAACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAG
CCCGCACCTGACAGTGCGGGCTTTTTTTTTTCGACCAAAGGTAACGAGGTAACAACCATGCGAGTGTTGAA
GTTCGGCGGTACATCAGTGGCAAATGCAGAACGTTTTCTGCGTGTTGCCGATATTCTGGAAAGCAATGCC
```

```
AGGCAGGGGCAGGTGGCCACCGTCCTCTCTGCCCCCGCCAAAATCACCAACCACCTGGTGGCGATGATTG
...
```

3.3.2 FASTQ

The fasta format inspired another format for storing sequences: the FASTQ format (from now on written in lower case as well) [141]. It was developed for next generation sequencing in order to save the sequencing quality alongside the sequenced read. As mentioned in Section 1.1, different sequencing technologies have different sequencing qualities. When analyzing the reads, the base quality gives the probability that the base was incorrectly called. This can then be considered when identifying this read to ignore mismatches for example. It is coded via a Phred quality score [142]. These scores are ASCII characters usually ranging from '!' to 'I' (quality +33 in the ASCII table [136]) with 'I' being the best quality. The format itself is structured into four parts: The sequence identifier line starting with '@', the sequence, a '+' which can include the sequence identifier as well, and the quality string. Usually, many millions of sequences make up a fastq file so this format is repeated many times. Because the origin of the sequenced DNA is usually unknown, the sequence identifier instead contains information about the machine and its parameters. An example of a fastq file would be:

```
@A00654:16:HCVKCDRXX:1:2101:1081:1016 1:N:0:NAATGTGTCT+GTAAGGCATA
AGTCTCGCTCATCAATCCAAAAGGAATACCGGACATCTCCACCAACCAAAAATCAGGAGTATTA
+
FFFFF:FFFFFFFFFFFFFFFFFFFFFF,F:FFFFFFFFFF:FFFFFF:FFFF:FFF,FFF
@A00654:16:HCVKCDRXX:1:2101:8169:1016 1:N:0:NAATGTGTCT+GTAAGGCATA
CTCCACCAGTTGTCATCAACGGGTAGGGTTTGTTGGAAATTACTGTCATTCTCCTGCCACGCTT
+
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF,FFFFFFFFFF:FFFFFFFFFFFFFF
```

3.4 Modules of kASA

The following sections describe the most important modules of our software. The module `build` creates the index, `shrink`, `update`, `delete` and `merge` can modify it. `identify` uses said index and is described afterwards. Figure 3.2 shows how the modules are interconnected. Analogue to Section 2.1, let DB_T be the set of genomes DB together with their respective taxonomic ID T, X_T be the sorted index generated from DB_T and M_R be the set of sequenced reads M together with their read IDs R. Furthermore, let the k's be from the interval $[k_L, k_H] \subset \mathbb{N}$ with k_L being called the "lower k" and k_H being called the "higher k".

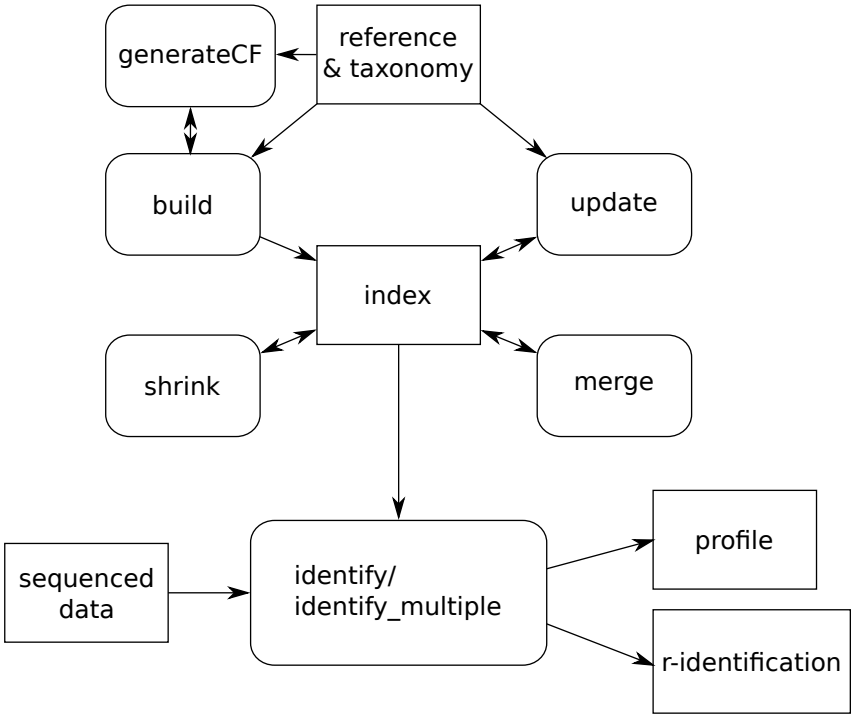

Figure 3.2: Directed graph showing how the modules are intertwined. `generateCF` creates a content file from the fasta file(s) together with the taxonomy. It can also be called inside `build`. Both `build` and `update` also need reference data from fasta files to either create or update the index. `shrink` and `merge` modify the index (the latter needs two indices) and `identify` uses the index to find similarities with new sequenced data. It then puts out two different files: one containing a broad overview of the identified taxa inside the input (profile) and the other a more thorough analysis of every read (r-identification).

3.4.1 `generateCF`

This module is mostly a precursor to `build`, but can be called independently. It contains functions for the creation of a "content file" which is a list of taxa from T together with information from the sequence identifiers found in the fasta file(s) (see Section 3.3.1).

First, we go through all given fasta files and gather information about the genomes or proteins from DB. As seen in Section 3.3.1, the fasta format contains sequence identifier lines starting with '>'. These lines also correspond to taxonomic IDs which are mapped to them via a given table. Depending on the chosen taxonomic level, the taxonomic tree, given via the "nodes.dmp" file from the NCBI taxonomy [55], is traversed upwards and IDs belonging to the same nodes are joined together. For example, two *Escherichia coli* (*E. coli*) strains are found in the fasta files and the

targeted taxonomic rank is species. Then the two taxonomic IDs corresponding to these strains are replaced by the one given for the species *E. coli*. This new taxonomic ID together with the IDs and accessions of the strains as well as the name of the species are saved into a "content file". It is a tab-separated, human-readable table which can also be created by other programs outside of `kASA` and then given as a parameter as well. This enables, for example, the use of taxonomic databases other than the one used by the NCBI. The content file is used in almost every module of `kASA` and necessary to fetch, for example, the name of a matched taxonomic ID after identification. A small example would be:

```
Histophilus somni  731  228400;205914  CP000947.1;CP000436.1;CP000019.1
Escherichia Coli  511145  511145  NC_000913.3
Sulfolobus solfataricus  2287  2287  NZ_LT549890.1
```

3.4.2 `build`

After creating the content file (see Section 3.4.1), we go through the fasta files (see Section 3.3.1) once more and gather all DNA or already translated protein sequences from DB that belong to recognized identifiers. See Code 3.4 for the translation from a DNA string to a string of amino acid letters. While doing so, we convert and translate the DNA to overlapping k-mers on the amino acid level (see Section 2.2.2). The translation uses one, three, or six frames depending on the user's choice. If the sequences have already been translated, we simply create overlapping k-mers from them. The k in this context is always k_H because as seen in Figure 2.1, smaller l-mers can be sampled from larger k-mers with $k_L \leq l \leq k_H$. Thus, we store only k-mers of maximum length.

Source Code 3.4: DNA to amino acids translation

```
////////////////////////////////////////////////////////////////////
// Converts a string of dna to a string of aminoacids by triplets;
// sDNA is a reference to the dna string, length is the input length,
↪  pointer position is the starting position in the dna string
inline void dnaToAminoacid(const string& sDna, const int32_t& iLength,
↪  const int32_t& iPointerPosition, string* outString) {
  // map A,C,T,G,X,Z to 0,1,2,3,4,5 by & with 1110 = 14 and then shift to
  ↪  the position in 000 000 000, this gives the position in the array
  ↪  where the respective amino acid is written
  // small letters are also correctly converted
  const uint32_t& iDnaSize = iLength / 3;

  for (uint32_t i = 0, j = 0; i < iDnaSize; ++i, j += 3) {
    const int32_t& iIndex =
      ((sDna[iPointerPosition + j] & 14) << 5)
      | ((sDna[iPointerPosition + j + 1] & 14) << 2)
      | ((sDna[iPointerPosition + j + 2] & 14) >> 1);
```

```
    (*outString)[i] = _sAminoAcids_bs[iIndex]; //_sAminoAcids_bs is the
    ↪  coding table
  }
}
```

The largest currently supported k is determined by the size of an integer in C++ because every translated k-mer is saved in its integer representation. See Code 3.5 for our implementation of this. Every letter in an alphabet the size of at most 31 needs five bits of space because $2^5 - 1 = 31$. The largest integer currently supported by the C++ standard has a size of 64 bits. This value divided by five results in 12 letters without information loss. We also offer a custom 128-bit integer which consists of two 64-bit integers together with the necessary arithmetic. This way we can raise the maximum k to 25.

Source Code 3.5: Conversion of amino acid strings to their integer representation

```
////////////////////////////////////////////////////////////////////
// Converts one aminoacid string to a coded kMer, using 5 bits per amino
↪  acid letter (max k = 12 => 60 bits used)
template<class intType> // uint64_t or uint128_t
static inline intType aminoacidTokMer(const string& aminoacid) {
  intType ikMer = 0;

  const int32_t& iLengthAA = int32_t(aminoacid.length());
  for (int32_t i = 0; i < iLengthAA - 1; ++i) {
    uint32_t iConvertedChar = aminoacid[i] & 31;
    ikMer |= iConvertedChar;
    ikMer <<= 5;
  }
  uint32_t iConvertedChar = aminoacid[iLengthAA - 1] & 31;
  ikMer |= iConvertedChar;

  return ikMer;
}
```

The k-mers generated from DB are saved together with their respective taxonomic ID in a ascending order into a data structure called "STXXL vector" (see Section 3.1.2) on secondary memory. The sorting is necessary to achieve linearithmic time complexity during `identify` because naively comparing two arrays (with sizes n and m) is of $\mathcal{O}(n \cdot m)$ time complexity. When an array the size of m is already sorted (since the index only needs to be sorted once) and the other array the size of n is sorted after creation as well, we achieve a time complexity of $\mathcal{O}(n \cdot \log(n) + 2 \cdot (n + m))$ [98] which is in $o(n \cdot m)$.

After this step, we create a minimal index so that two identical entries are merged. If only the k-mer is identical but the taxonomic ID is not, we keep all entries with different taxonomic IDs. This introduces a certain redundancy to our index but

ensures the highest sensitivity. This can be seen in Chapter 4 of this dissertation, where CLARK uses distinctive k-mers and suffers sensitivity loss because of it (see Section 4.3.1).

The process of sorting and storing minimal pairs efficiently under the restriction of minimal memory and space requirements is a problem with multiple possible approaches. Our first and naive approach was to translate, convert, and save all k-mers into one large temporary `stxxl::vector` (referred to as "vector" in the following text) which is then sorted by `stxxl::sort` and lastly made unique with duplicate k-mer removal. This approach was dismissed because the sorting algorithm is not working in-place, hence, an additional temporary file the size of our vector was created in the background. Furthermore, we discovered that the number of I/O accesses of the implemented merge sort algorithm inside the STXXL was unacceptable for systems without an SSD.

Therefore, we developed a less straightforward but much more efficient solution. Instead of sorting the whole vector, we gather as much k-mers as possible given the memory restriction by the user. The resulting pairs are then sorted in primary memory with `std::sort` (see Section 3.1.1) and afterwards put into a temporary vector residing on secondary memory. Then the local container in RAM is reset and the process starts anew. When all data is processed, we apply the following strategy: Create a new temporary vector and keep a pointer for each file and its topmost (and therefore smallest) element. These topmost elements are compared inside a priority queue, the smallest is added to the new temporary file, and the pointer for this vector containing the smallest element is incremented. If two or more topmost elements are equal in both k-mer and taxonomic ID, all but one are discarded. If a temporary vector reaches its end, it is deleted. The merging step is repeated until only one temporary file is left. This last file then represents the index. Our strategy avoids using more memory than given by the user and uses only the necessary amount of temporary space. The reason for that is that the STXXL supports growing and shrinking vectors when data is added or removed. Therefore the elements are truly moved instead of copied. A graphical interpretation can be found in Figure 3.3.

After creating the index, we can use it to add some further preprocessing steps: Narrow the search space and count the number of all k-mers for later use. The first step of narrowing the search space is motivated by the fact that the input which needs to be identified is usually much smaller than the index. Therefore going through the index sequentially would stream a lot of useless data until a match would occur. To solve this, we use the fact that we sorted our index beforehand. This way, we can narrow the search space every time a letter matches since no further match can occur in a lexicographically smaller word. Therefore, the index can be split into ranges, depending on the shared prefix. Should a k-mer from the input M_R not match such prefix, it cannot be found inside the index and is therefore discarded. Should a match occur, the search space has at most $|\text{Alphabet}|^{k_{max}-|\text{prefix}|}$ entries

which is usually much smaller than the whole index. To save these ranges for usage in `identify`, we use a trie as described in Section 3.2.1. It is created from the index and saved alongside our index to allow for much shorter run times during `identify`. Because the trie data structure cannot be saved directly, it is "linearized". This process saves the nodes in a list to efficiently reconstruct the trie during `identify` and works as follows:

The leaves of our trie contain the ranges in the index in which every k-mer has the same prefix of length six (see Section 3.2.1). A path through the trie represents that prefix. Because the index is sorted, we can directly count the number of k-mers that have the same prefix. The index is also contiguous so summing up those counts gives the ranges when reconstructing the trie in `identify`. For now, the prefixes together with their respective count are saved as tuples into an `stxxl::vector`.

Lastly, a "frequency file" is constructed. This file contains the absolute number of k-mers for every taxon and k in the built index. These numbers are used in `identify` in Section 3.4.5 and multiple other modules.

With this, we have created an index and associated files from DB_T and preprocessed them for use in other modules.

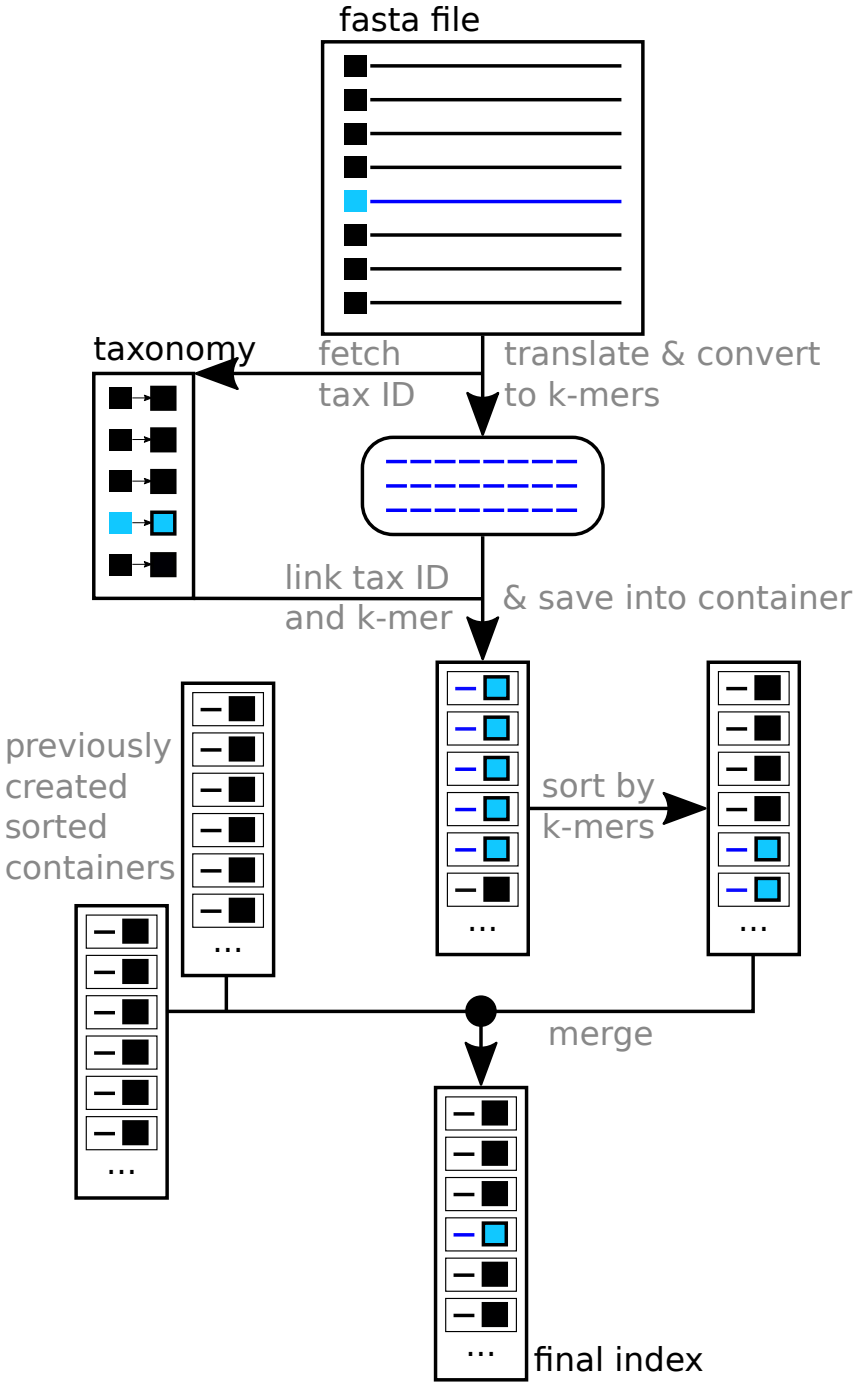

Figure 3.3: Scheme of the algorithm building the index. Multiple temporary files containing sorted pairs of k-mers and taxonomic IDs are created until no genomic information is left in the fasta file(s). Afterwards, these files are merged concurrently into the final index.

3.4.3 `shrink`

As mentioned in Section 3.4.2, the index is designed with no information loss in mind. This introduces a certain redundancy because two or more identical k-mers can have different taxonomic IDs but they still use the space of two or more whole pairs. The reason for that is that the underlying data structure of an `stxxl::vector` requires uniform space for every entry. In our case this means 12 or 20 bytes per entry, depending on whether a 64-bit integer or a 128-bit integer is used. So even if we could remove the redundant integer representing the k-mer and just save all corresponding taxonomic IDs, the space would still be wasted. This leads to high space requirements for large, comprehensive databases (e.g. Terrabytes). While we argue, that `kASA` was not designed to use compact indices because space on secondary memory is more abundant than on primary memory, we decided to give the user the possibility of shrinking the index. In the following paragraphs, we present our current methods for achieving exactly that.

Lossless strategy

Our first approach was finding constrains that enabled us to make assumptions which would shrink the index without information loss. In Section 3.2.1, we mentioned that prefixes saved into the trie are of length six. This approach uses the fact, that when the maximum k is 12, these prefixes already contain the first half of information of every k-mer from the index. If we assume that the lower k, named k_L, is always larger than six, we only need to look at the suffixes if the prefix was found in the trie. Removing the prefix from every 12-mer in the index would therefore make no difference in this case so we can omit it. Since the entries in the index are pairs containing the k-mer and the taxonomic ID, the second component must be restricted in size as well to fully halve the size of the index. To code the taxonomic ID, a 32-bit integer is used by default because humanity is far from having identified $2^{32} - 1$ species yet. Coupling the content file to an index by using the line number of every entry instead of the taxonomic ID and restricting the number of entries to 65535 taxa (the highest number that can be coded in a 16-bit integer) would enable us to change the tuple size from 64+32 bits to 32+16 bits, effectively halving it. Afterwards, the index is marked as shrunken, so the user can be reminded of these constrains.

Deleting k-mers

We also found out that lossy concepts work well enough if the compromise of losing information is justified by the space gained. The second approach is a removal of k-mers based on a user-specified percentage, so that large genomes lose more k-mers than small ones. To calculate the number of k-mers that are deleted per taxon, we need the total number of every taxon first. Fortunately, we can use the frequency file created while building the index as it contains the total number of k-mers of every taxon and every k including the largest. Then two arrays, each the size of all taxa

in the index, are initialized. One will count upwards should a 12/25-mer of a taxon be hit during the loop over every 12/25-mer of the index. The other contains step sizes which are derived from the given percentage and the total number. Should the count of the former array be the same as the value in the latter while going through the index iteratively, the current 12/25-mer is removed and the counter is reset. All others are written to a new index file which is still sorted but needs a new frequency and trie file which are both generated afterwards. The code can be seen in Code 3.6.

Source Code 3.6: Deleting k-mers with a given percentage

```
////////////////////////////////////////////////////////////////////
// Throw out kMers for every ID where fN is the percentage of how many are
↪  thrown out of the respective ID (e.g. fN = 60 -> throw out 60%,
↪  iCount[ID] = 1200 -> 480 stay, 720 are thrown away)
template<class vecType, class elemType>
inline void deleteEveryNth(const unique_ptr<const vecType>& vLib,
↪  unique_ptr<vecType>& vOut, const float& fN, const
↪  unordered_map<uint32_t, uint32_t>& mContent) {
  const double& dStepSize = 100. / fN;
  vector<uint64_t> vSteps(mContent.size(), 1);
  vector<double> vNextThrowOutIdx(mContent.size(), dStepSize);

  auto itOut = vOut->begin();
  uint64_t iCounter = 0, iCurrentPercentage = 0;
  for (auto libIt = vLib->cbegin(); libIt != vLib->cend(); ++libIt) {
        const auto& entry = *libIt;
        const auto& idx = Utilities::checkIfInMap(mContent,
        ↪  entry.second)->second;
        if (vSteps[idx] != vNextThrowOutIdx[idx]) {
          *itOut = entry;
          itOut++;
          ++iCounter;
        }
        else {
        vNextThrowOutIdx[idx] += dStepSize;
        }
        ++vSteps[idx];
  }

  vOut->resize(iCounter, true);
}
```

This approach can also be applied while building the index. For example: if a percentage of 50% is given, every second overlapping k-mer is ignored. Additionally, setting the memory restriction parameter in this mode reduces the index size to its value in GB by calculating the necessary percentage.

Deleting based on entropy
The third approach uses information theory to calculate the normalized metric entropy [143] of every k-mer and removes those falling below a certain threshold. For example, a k-mer like "AAAAAABBBBBB" would be removed while "ABBCADDEAFAR" would be kept. While this approach does not shrink the index to a large degree, it helps removing k-mers which do not contribute much to the identification and thus saves space and time without much loss of accuracy. The reasoning is the same as in any low-complexity filter for e.g. DNA sequences [144]: low-complexity sequences can cause artificial hits. The process is as follows:

For every k-mer from the index, we calculate the number of occurrences of every letter l from the alphabet $\mathcal{A}$ and divide these counts by the total size of the k-mer resulting in the relative frequency rf for the letter l. The Shannon entropy [145] is then determined by:

$$H(k\text{-mer}) \coloneqq -\sum_{l \in \mathcal{A}} \mathrm{rf}(l) \cdot \log_2(\mathrm{rf}(l))$$

The normalized metric entropy is then defined as:

$$H_{\mathrm{N}}(k\text{-mer}) \coloneqq \frac{H(k\text{-mer})}{\log(|\mathcal{A}|)}$$

Should this value be larger than a user given threshold between zero and one (0.5 for example) the k-mer will be kept or discarded if not.

3.4.4 `update`, `delete`, and `merge`

Building the index can take some time, especially if it uses a large database. If some taxonomic IDs should change or data must be added to the index, building it anew is not preferable. In order to update the index by either adding data to it or removing deprecated taxa, we wrote the modules `update` and `delete`.

In `update`, the user can add data from one or more fasta files to the index. Should the fasta file(s) contain taxa or accessions which are not part of the content file (see Section 3.4.1) already, it must and will be updated as well. The process is similar to the one in `build` (see Section 3.4.2): After converting the sequences to k-mers and saving them into a sorted temporary file, the existing index and this temporary file will be merged while either overwriting the existing index or creating a new version. Afterwards, the frequency and the trie file will be updated as well. Therefore, the update step can be seen as another iteration of the building algorithm. The last part of merging two indices is also available via the module `merge`. This can be useful if two indices were created for different studies but now need to be united and also saved onto the drive for another study. With the availability of `update`, we achieve Goal 2 for `kASA` in which we require the software to be able to add data to its index.

In case the user wants to delete entries by their taxonomic ID entirely from the index, `kASA` can be given a list of such taxonomic IDs via `delete`. Internally, this list is converted to a hash table for faster lookup. For every entry in the index, it determines if either the entries' taxonomic ID is inside this table meaning that the entry can be discarded, or if the entry can be written to a new index file. All other files belonging to the index (e.g. trie, frequency file, ...) are updated as well.

We initially developed methods for in-place updating as well but found out that reading from an external file and writing into another is much faster than doing both in the same file due to cache thrashing. Unfortunately, this forces the user to provide enough space for the new index to be saved alongside the old one. It is future work to find a way to avoid this behavior.

3.4.5 `identify`

This module calculates the links between DB_T and M_R as described in Section 2.1. The result of this process is an r-identification output and/or a taxonomic profile. It is therefore the main part of `kASA` and solves the problem of taxonomic profiling with the ideas presented in Chapter 2.

Loading of preprocessed files
In the building step (see Section 3.4.2) we preprocessed data from DB_T to allow for a faster identification of data from M_R. This part describes what information is gathered from these preprocessed files and how they will be used.

The frequency file (see Section 3.4.2) gives the number of k-mers per k of all taxa in the index. These values are saved into an array so that an index of this array corresponds to its line number in the file. It also contains the total number of taxa in the index which is needed for primary memory space allocation beforehand. The content file (see Section 3.4.1) is also read and a hash table containing the links between the taxonomic IDs and the corresponding line numbers is generated. The names of the taxa are saved into an array as well. The line numbers in the hash table correspond to the indices in this array as well as to those in the array created from the frequency file. This simplifies output generation because an identified taxon can then be referred to its line index and thus to its name and k-mer frequency. We therefore put the identified taxa into further context, making the output files human readable as well (reading “9606” is not as informative as reading ”Homo sapiens” for example).

The index itself can be loaded either via the STXXL interface (see Section 3.1.2) for secondary memory access or into primary memory via the `std::vector` data structure (see Section 3.1.1), depending on the users choice. The latter speeds up the identification process but reduces available primary memory space. As described in Section 3.4.2, we also created a flattened trie. It is now expanded and loaded into primary memory to speed up the identification process.

Reading the input

After loading all preprocessed files and data structures, the input files in the form of fastq or fasta files (see Sections 3.3.2 and 3.3.1), gzipped or not (see Section 3.1.3), can be processed to M_R and matched against our index. For this to happen, the DNA or protein sequences need to be converted into a format comparable to k-mers in our index:

The DNA from M is translated to overlapping k-mers of length k_H via a user given translation alphabet (the standard translation table is used by default, see Table 2.4) in one, three, or six frames. The latter means that the reverse complement is created and translated as well. This may be necessary if the direction of the sampled data is unknown. At this point we would like to advise the reader and user that using six frames for `build` and three for `identify` could skew the identification as single k-mers from the input could be identical to some of the reverse complement of the genomes in the index.

The sampled k-mers are saved together with IDs belonging to their original read, which in turn are created by incrementing a variable each time a new read is processed. Duplicate k-mers can be removed if it is necessary for the experiment. After converting and saving those pairs into an `std::vector` container, it is sorted in ascending order via `std::sort` (see Section 3.1.1). Let this sorted container be named I_R from now on.

All input files are read in chunks if the memory restriction does not allow the whole file to be processed. If a read can only be processed partially due to these conditions, we called it an "overhanging read". These overhanging reads are scored but the score is not yet written to the output. Instead the rest of that read is processed in the next iteration and the new score is added to the saved one. Only when a read has been completely processed, its result is written to the output file.

Since access to the index via secondary memory is a bottleneck regarding time, we avoid trying to match k-mers which are not even inside the index. This is done with the help of the trie built prior to this step: We try to match the prefixes of our newly constructed k-mers up to the smallest k (k_L) provided. If that fails, the k-mers cannot be inside the index since not even the prefixes are found. However, if a match occurs we get the range of the index in which the suffixes lies (see Figures 3.1 and 3.4). This range is saved alongside the k-mer and the read ID. Should k_L be larger than or equal to the depth of the trie (which is currently six), the trie is traversed completely and the matching process is continued within the suffixes. Should k_L be smaller than six, the traversal only gets to the k_Lth level and from there the lexicographically smallest and largest prefixes are determined. See Section 3.2.1 and in the Code 3.1 the two functions `GetFirst()` and `GetLast()` for further details on this last case. We are traversing the trie after sorting because lexicographically sorted accesses to our trie enables it to prefetch the next nodes and therefore save a lot of time.

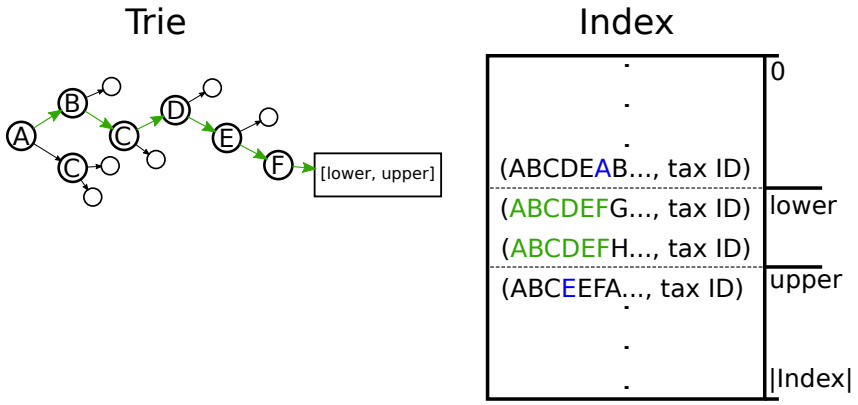

Figure 3.4: Ranges inside the index for which the k-mers have identical 6-mer prefixes are gathered from the trie.

Identification algorithm

After fetching the ranges, the identification algorithm can run in parallel with as many parts of I_R as threads are available, should the user have multi-threading enabled. The vector I_R is split to take advantage of the caching effects. These caching effects come into play when the areas where the suffixes are searched within the index are close together. The reason for that is that sequential access to SSD and RAM is faster than random access (a principle called "locality of reference", see [132]). Therefore we ensure that ranges close to each other are in the same part of the vector and thus computed by the same thread. More information regarding the parallelization can be seen further down below (see here: 3.4.5). The algorithm applied to every k-mer of the input inside the vectors is inspired by the `set_intersection` algorithm from the STL (see Section 3.1.1). See Figures 3.5 and 3.6 for a graphical example and a flowchart of this process. The pseudocode is given in Algorithm 3.1. The process is as follows:

Let $i \in I_R$ be a pair of k-mer and read ID from the input and $Y_{Ran} \subseteq X_T$ be pairs of k-mer and taxonomic ID inside the range $Ran \subseteq [0, |X_T|]$ from the index X_T determined by the trie for i. At the beginning, the first matching $m \in Y_{Ran}$ for the smallest k (k_L) inside the range is searched inside the range Y_{Ran} via a binary search. Should this fail, we try it again with the next element of I_R. If such a match m is found however, the sequential matching with pairs from Y_{Ran} starts. The $k \in [k_L, k_H]$ is increased until a mismatch occurs or the highest given k (k_H) is reached. In the latter case, the index is searched for identical k-mers in Y_{Ran} with different taxonomic IDs than the matching one, since we included such duplicates to increase sensitivity. Afterwards, the algorithm starts anew with the next entry in I_R.

Should a mismatch occur because the k-mer from i is larger than that of m for the current k, Y_{Ran} is searched sequentially for a k-mer that is either equal to or

larger than the current k-mer from i. If such an element is found, the process of elongating the k-mer by increasing k resumes. In case the i is smaller than m for a k, the process is interrupted and the next element of I_R is chosen. We also keep a copy of all currently considered k-mers. Should the next element of I_R be equal to the one before for any value of k, their read IDs will be saved as well without going through the index again. Furthermore, should a new k-mer be different from the saved copy for a k, the algorithm will consider i to be finished for this k because no other k-mer will match the copy in the future due to the lexicographical order. Therefore, the algorithm knows that it can score the links between the saved read IDs and taxonomic IDs. This works as follows:

For every match of the k-mer in i and k, the read ID and matching taxonomic ID are saved. We use the bit array set data structure (see Section 3.2.2) to save the matching taxonomic IDs $T_M(i,k)$ and an `std::vector` for the matching read IDs $R_M(i,k)$. The number of matches for every $i \in I_R$ and k is saved into an array of integers with length k named `counts`(i,k). Keeping all this information until the end is neither useful nor necessary. Instead we calculate a score and save it into a matrix the size of $T \times R$ named $Score_{TR}$ (implemented as an array of pointers pointing to arrays). This means that for every pair in $(t,r) \in T_M(k) \times R_M(k)$ the score derived from the counts in `counts`(i,k) is added to the entry in $Score_{TR}[t][r]$. It is initialized to zero and reset every time a new part of the input M_R is put into the identification algorithm described above. The identification output is generated from this matrix.

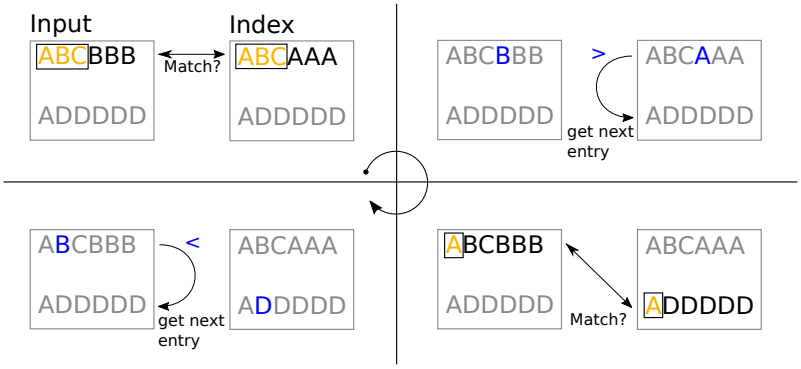

Figure 3.5: Example of a matching during identification. Picture order is left top, right top, right bottom, left bottom. First, a match for $k = 1, 2, 3$ occurs until for $k = 4$ the entry in the index is smaller than that of the input. We therefore take the next entry from the index and try to match anew. Here, only for $k = 1$ a match can be found until the input k-mer is smaller than that of the index, so we take the next one.

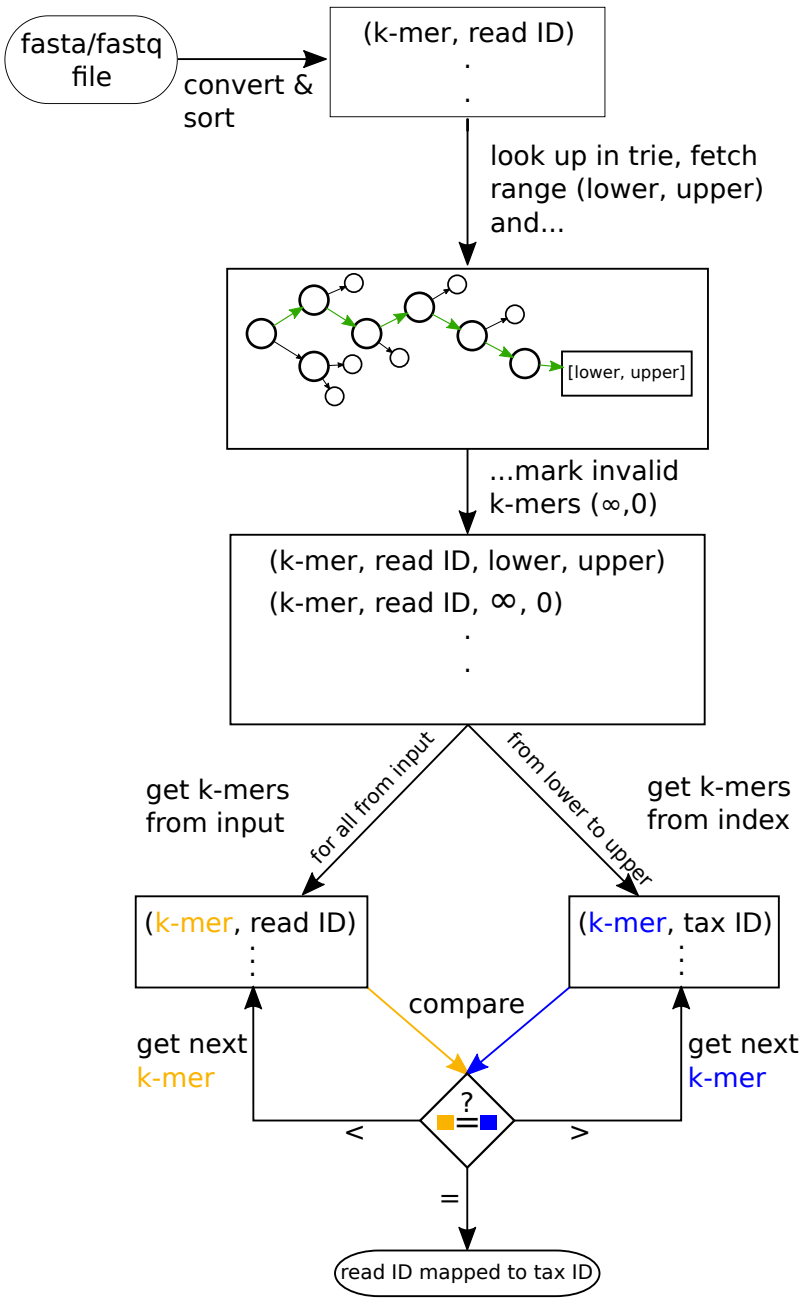

Figure 3.6: Flowchart of the program flow inside `identify`. First, k-mers are converted, then sorted and searched in the trie to narrow the search space inside the index. The remaining ones are compared to those from the index. Either a match was found or the next k-mer needs to be read from either the input or the index, depending on the result of the comparison.

Scoring
Overall, we calculate six scores. The first is called the "k-mer Score" which describes, how many k-mers a taxon shares with a read. Based on this, an error score is computed which indicates how far away the k-mer Score for a read is from the best possible one. The "Relative Score" is calculated at the end of the identification process for every read when it has been fully identified. These scores are part of the r-identification output. The "unique relative frequency", "non-unique relative frequency" and "overall relative frequency" are part of the taxonomic profile. Now follows a more detailed description of these scores:

The aforementioned score that is saved into $Score_{TR}$ is called the "k-mer Score" which is a weighted sum of the counts in $\texttt{counts}(i,k)\,\forall\, i \in I_R$. The weights w_k depend on k and are calculated by squaring k and normalizing these values of k^2 to $(0,1]$ because just using k rewards short matches too much. Instead, a polynomial function was chosen because it ensures that longer matches gain a higher significance than shorter ones. Should more than one taxon share the same k-mer with a read, a penalty on the score is necessary to decrease the importance for those k-mers so we divide the count of matches by the number of matched taxa for that k ($T_M(i,k)$). The resulting formula for the k-mer Score of the taxon $t \in T$ and read $r \in R$ is as follows:

$$k\text{-mer Score}_{t,r} \coloneqq \sum_{i \in I_R} \sum_{k=k_L}^{k_H} w_k \cdot \frac{\texttt{counts}(i,k)}{|T_M(i,k)|} \tag{3.1}$$

Given a k-mer Score and a read $r \in R$ consisting of DNA that was translated in three frames, the error is calculated via:

$$\text{Best score}_r \coloneqq \sum_{k=k_L}^{k_H} (\text{length}(r) - k \cdot 3 + 1) \cdot w_k \tag{3.2}$$

$$\text{Error}_{t,r} \coloneqq \frac{\text{Best score} - k\text{-mer Score}_{t,r}}{\text{Best score}} \tag{3.3}$$

This normalizes the error to [0,1] which means that should the error be equal to 1, no k-mer hit that taxon for this read. A zero indicates that all sampled k-mers were found in the index. The r-identification output only contains the error for the highest scoring taxa for every read given by the "Relative Score". The motivation behind this error score is that taxa matching reads with a low error score increase confidence that this read truly belongs to its matching taxon. On the other hand, it can be used to filter out specific reads. During decontamination, an index of unwanted taxa is created and everything that matches "good enough" regarding this error (e.g., everything below 0.5) can be filtered out. Afterwards, two files can be created, one containing the clean and the other one the filtered sequences. This filtering is optional and can be enabled with a few parameters in `kASA`.

The "Relative Score" takes k-mer Scores for every taxon and read and normalizes them. This is necessary because of two reasons: 1. To make a fair comparison between reads of different lengths and 2. smaller genomes generate fewer matches than longer ones, making them more important. It is calculated by taking the k-mer Score for every read and taxon and normalizing them to the length of the read $r \in R$ and the number of k-mers called the frequency of the taxon $t \in T$. The latter was saved into memory prior to the identification (see "Loading of preprocessed files" above). The formula is inspired by the E-value of the BLAST algorithm [89] but in our case, higher means better and can be used to stress matches of small genomes.

$$\text{Relative Score}_{t,r} \coloneqq \frac{k\text{-mer Score}_{t,r}}{1 + \log_2 (\text{length}(r) \cdot \text{frequency}(t))} \tag{3.4}$$

For the r-identification output and every read ID in it, taxa are sorted in decreasing order by this relative score. If a read has multiple identified taxa, the leftmost one has the highest value. We furthermore separate between "Top Hits" and "Further Hits" to make interpretation of the per-read results easier. A taxon t is a "Top hit" if $\frac{k\text{-mer score of } t}{\text{highest } k\text{-mer score}} > 0.8$. This value seemed to correspond best to what would intuitively be considered "relevant" when reporting several results for one read. It was therefore determined empirically.

To calculate the scores for the profile, `kASA` manages two counts per k and taxon: The "unique" and the "non-unique" k-mer count. The unique count of the taxon $t \in T$ is increased during Algorithm 3.1 by the value in $\texttt{counts}(i,k)$ if $|T_M(i,k)| = 1$. The non-unique count for each taxa in $T_M(i,k)$ divides the value in $\texttt{counts}(i,k)$ by the number of taxa in $T_M(i,k)$ beforehand in order to give each taxon their fair share. If for example a k-mer found in five reads from the input matched to four different taxa, each taxon would receive $\frac{5}{4}$ to their non-unique count. In case $T_M(i,k) = 0$, nothing is added to both counts as $\texttt{counts}(i,k)$ would be zero as well.

$$\text{uniqueness}(i,k,t,T_M(i,k)) \coloneqq \begin{cases} \texttt{counts}(i,k), & T_M(i,k) = \{t\} \\ 0, & \text{else} \end{cases} \tag{3.5}$$

$$\text{unique count}(t,k) \coloneqq \sum_{i \in I_R} \text{uniqueness}(i,k,t,T_M(i,k)) \tag{3.6}$$

$$\text{non-unique count}(t,k) \coloneqq \sum_{i \in I_R} \frac{\texttt{counts}(i,k)}{|T_M(i,k)|} \tag{3.7}$$

The unique count helps to differentiate between two related species. For example, species A and B are related so their genomes are similar. This means the non-unique counts will be similar as well. However, there will for example be k-mers that match exclusively to A and not to B. Therefore, the unique count of species A will be higher than that of B. Measuring the non-unique counts on the other hand gives an approximation on the composition of known taxa. It also protects from database redundancies like entries that are identical on amino acid level but have a

different taxonomic ID which would receive no unique count. These counts can also be put in relation to either the total number of non-/unique counts or to all sampled k-mers from I_R. These scores are then called the unique relative frequency (h_U), the non-unique (h) relative frequency, and the (unique-) overall relative frequency (h_O), respectively. The latter is primarily used if, for example, 90% of the input was unidentified but one taxon has a unique relative frequency of 1. Because the overall relative frequency relates to all sampled k-mers from the input, the identified taxa are put into relation to the whole dataset and this would be visible.

Let $T_M(i,k)$ be the set of matched taxa for $i \in I_R$ and k as above and T be the set of all taxa. Let also $\mathit{frames} \in \mathbb{Z}$ be the number of frames in which the reads are translated (one, three or six). Then the formulas are as follows:

$$h_U(t,k) \coloneqq \frac{\text{unique count}(t,k)}{\sum_{\hat{t}\in T} \text{unique count}(\hat{t},k)} \tag{3.8}$$

$$h(t,k) \coloneqq \frac{\text{non-unique count}(t,k)}{\sum_{\hat{t}\in T} \text{non-unique count}(\hat{t},k)} \tag{3.9}$$

$$h_O(t,k) \coloneqq \frac{\text{non-unique count}(t,k)}{|I_R| - |R| \cdot (k - k_L) \cdot \mathit{frames}} \tag{3.10}$$

The denominator in h_O calculates the total number of k-mers for that k sampled from the input. Since I_R only contains maximum length k_H-mers, it is necessary to subtract the number of smaller k-mers which were sampled from those long ones.

After all data from a file has been processed and scored, the results are written to two output files. The r-identification output contains information which taxa scored highest for each read, the profile forms a taxonomic profile showing the above described counts and relative frequencies per taxon. The r-identification output can be formatted as `json`, `jsonl`, Kraken-style [109], or tab separated with one line per read. The profile is given as a `csv`-file for best compatibility with scripts and spreadsheet software.

Parallelism

Since 2000, increasingly more CPUs started to include two or more cores (with IBM® POWER4 being the first [146]). Writing software that uses only one core is therefore not future-proof. We set out to parallelize every part of `kASA` that can be parallelized. The only exceptions are reading from and writing into a file. Although, when using multiple drives or an SSD for example, reading from a file while writing to another can be run concurrently. Whenever parallel code is written, the programmer has to keep in mind if there are parts that need synchronization or "locking". Locking must be used if only one thread should compute certain parts of the code because a race condition could occur otherwise. A race condition occurs when the correctness of the program depends on the order of execution of the threads [147]. In our context, this happens if two threads access the same area

of memory. For example, the value of a variable is read, another value is added, and the result is then written back to memory. In a race condition, one thread might access the value before the other thread adds its value and so the new value is overwritten. This distorts the true value in the end and could lead to a false result and interpretation. “Locking” [148] can prevent this by instructing the other thread to wait until the variable is freed again by the locking thread. The result is therefore calculated sequentially and correctly. However, locking increases execution time because it creates overhead and the waiting thread wastes time. Optimally, no locking should occur in the first place due to "embarrassingly parallel" [149] code with little to no dependency between the tasks. There are some parts of code in `kASA` which fall into that category, namely, the translation from DNA to amino acid-like sequences, reading from the trie, counting the number of k-mers, and sorting. The latter is parallelized in the `C++17` standard in the STL (see Section 3.1.1), the other parts use `WorkerThread` (see Section 3.3). Translation (as seen in Section 2.2.2) can be done in parallel if chunks of DNA are saved into a `std::vector` container first and then translated in parallel with another container for the resulting k-mers (what we defined as I_R earlier). Because the number of k-mers can be calculated directly from the length of the DNA, we can set specific ranges for every task in the result container. More generally, splitting a container like `std::vector` into equal parts that are then processed in parallel (using the class `WorkerThread`) is a typical method when each entry of the container is independent of all others. It is therefore also used for the other mentioned cases above.

The parallelization of `identify` however requires a bit more synchronization. Because both the index X_T and the container holding the input I_R are sorted in ascending order of the k-mers, the attached read IDs and taxonomic IDs are scrambled. Splitting the input container in as many parts as threads are available is therefore possible, but every thread would need its own matrix $Score_{TR}$ mentioned earlier. Because this matrix is of size $|T| \times |R|$ and uses float as data type (the k-mer score in Equation 3.1 is a floating point value), this is not viable. A common data set of a million reads and an index with about a thousand taxa would use more than 3 GB of memory for each thread. Assuming we have just two threads that would mean more than 6 GB just for the matrix. Of course many entries will be zero but note, that a dynamically growing matrix is much less efficient and could in the worst case use more memory than a pre-allocated array due to the operating system allocating incoherently more than necessary. We thus conclude that using just one matrix or rather two-dimensional array containing matches between read ID and taxonomic ID must suffice. To avoid a race condition, locking becomes necessary because our score could be influenced should two or more threads access the same entry in the array. After some experiments however, we noticed that locking introduced a large overhead and decreased performance by a large degree, making parallelization useless. Out of curiosity, we disabled locking entirely for this part and expected our results to change but they did not. This motivated the following observation:

Let $|R|$ be the number of reads, $|T|$ be the number of taxonomic IDs, $J_T \subset \mathbb{N}$ be the index set generated by enumerating the elements in T, and $p_r = (p_{r,1} \ ... \ p_{r,|T|}),\ 1 \leq r \leq |R|$ be the probability vector of taxa for each read r. This means that the probability, that a taxon $t \in T$ with the position $j_t \in J_T$ is present in read r is p_{r,j_t}. The values inside the probability vector are usually unknown in metagenomics although for the following arguments, this does not bother us. We further assume that every read is statistically independent from all other reads. This means that p_r and $p_{r'}$ do not depend on one another for all $r, r' \in R$, $r \neq r'$. Therefore, the probability that an entry of the matrix $Score_{TR}$ is hit can be calculated as follows:

$$P(\text{Hit between taxon } t \text{ and read } r) = P(\text{Hit}_{tr}) = \frac{p_{r,j_t}}{|R|},\ j_t \in J_T \tag{3.11}$$

Since access to such an entry by two or more threads simultaneously is a binary event (happening or not), it is modeled as a Bernoulli trial. Let X_{tr} be therefore the event that for $N \geq 2$ threads at least $M \leq N$ threads access the same read ID r and taxonomic ID t in $Score_{TR}$. The probability of that happening is:

$$P(X_{tr}) = \sum_{M=2}^{N} \binom{N}{M} \left(P(\text{Hit}_{tr})\right)^M \left(1 - P(\text{Hit}_{tr})\right)^{N-M} \tag{3.12}$$

Which results in a higher probability, the more threads we start and in a lower probability the more reads and/or taxa we need to consider. Finally, the probability across all taxa and reads can be calculated by:

$$P(\text{Race condition happens}) = 1 - \prod_{(t,r) \in T \times R} \left(1 - P(X_{tr})\right) \tag{3.13}$$

Just taking an unrealistically small example of three taxa, 100 reads, eight threads, and $p_r = (0.5\ 0.3\ 0.2)$ for every read results in $P(\text{Race condition happens}) \approx 0.1$. As this is already quite small, we conclude that locking would unnecessarily impact performance and is thus omitted. Even if this race condition should occur, the worst case is a reduction of the k-mer score (see Equation 3.1) for a pair of read ID and taxonomic ID. This reduction can at most be $w_k \cdot \frac{\texttt{counts}(i,k)}{|T_M(i,k)|}$ for one k-mer from I_R if it happens once.

3.5 Additional features and modules

3.5.1 `identify_multiple`

We have also developed a version of `identify` that makes more efficient use of the available hardware (for example on an HPCC) if the user wants to process multiple files at the same time. Since reading and writing a file can only be performed by a single thread, there are limits to the acceleration of our parallel algorithm by more processors. We therefore use the given resources to process multiple input files in

parallel instead of sequentially as this uses given cores much more efficiently. This mode is also used for most of our experiments, since they often consist of multiple input files. The informed reader may recognize this to be a typical job scheduling problem [138] for which we approximate a solution.

It starts by sorting the files by size so that larger files get more cores. If more cores than files are available, all files will be processed in parallel asynchronously while larger files get more cores. If more files than cores are available, then as many files as possible are processed, each using only one core. We use the `WorkerQueue` class (see Section 3.2.3) to start calculations asynchronously (see next chapter for more details). The given RAM is divided by the number of concurrently running processes so that the memory barrier is not exceeded. Furthermore, loading the index and expanding the trie is done once so that all subsequent calls to `identify` use these data structures instead of creating them anew every time.

3.5.2 Retrieving lost or damaged Files

Currently, `kASA` needs certain files in order to be useful (see Section 3.4.5). These files are the index in a binary file format, the info file containing its size and additional information about the type of index (see Section 3.4.2), the flattened trie in binary file format and its corresponding info file (see Section 3.2.1), the frequency file containing all k-mer counts for every taxon (see Section 3.4.2), and lastly the content file (see Section 3.4.1). Should any of these files go missing or are corrupted for unknown reasons, we developed ways to compensate for that by rebuilding them.

- The trie file can be generated again with a call to `trie` if the index was not halved (see Section 3.4.3). This will use the existing index to create the trie again together with its info file. The algorithm is the same as in `build` (see Section 3.4.2).
- Should the frequency file be affected, it can be created with `getFrequency` from the index.
- The info file for an index can simply be retrieved by creating a .txt file, dividing the binary size of the index by 20, 12 or 6 Bytes and writing this number to the first line. Depending on the choice of the maximum k of 25 or 12, or if it was halved by the respective shrink mode, “128” or “3” has to be written on the second line, respectively. This sounds like it could be automated but unfortunately, STXXL files contain no metadata so it cannot be bootstrapped from the binary file alone.
- The index file itself can only be retrieved by rebuilding it. The user can give the pre-computed content file to the building routine in order to skip the creation step. Afterwards, the k-mers need to be rebuild from the database and the other additional files are overwritten. If the content file was affected, it has to be rebuild from scratch as well.

3.5.3 Measuring the redundancy

Comprehensive databases suffer the disadvantage of species having many k-mers in common. We therefore introduced multiple scoring strategies for our profiling step. To help the user interpreting the results, we added a mode to check the redundancy of the index. If over 99% of the k-mers have a unique taxonomic ID, the unique relative frequency should give a very good profiling result. In contrast: if almost all k-mers belong to two or more taxa, the non-unique relative frequency gives a better impression of the composition of the data.

The redundancy is calculated by creating a histogram with the number of taxa per k-mer as counts. Starting from one, the numbers of k-mers are summed up per number of taxa. When this sum reaches 99% of all k-mers, almost all k-mers have at most this number of taxa. If, for example, 50% of all k-mers have unique taxonomic IDs and 49% have two then two will be returned as output. Afterwards, this number and an interpretation is given to the user so that the person may decide how important the non-unique relative frequency might be. See Code 3.7 for the implementation.

Source Code 3.7: Calculating the redundancy histogram

```
////////////////////////////////////////////////////////////////////
// count occurrence of taxIDs per kMer and locate 99% of kMers cutoff. Most
↪  kMers have only one taxon.
template<class vecType, class intType>
inline uint32_t histogram(const unique_ptr<const vecType>& vLib, const
↪  uint32_t& iNumOfTaxIDs) {
  intType iSeenkMer = (vLib->at(0)).first;
  vector<uint32_t> taxIDs(iNumOfTaxIDs + 1);
  uint32_t iCounter = 0;
  uint64_t iUniquekMerCounter = 0;

  for (const auto& entry : *vLib) {
    const auto& kMer = entry.first;
    if (kMer == iSeenkMer) {
      iCounter++;
    }
    else {
      taxIDs[iCounter]++;
      iCounter = 1;
      iSeenkMer = kMer;
      ++iUniquekMerCounter;
    }
  }
  taxIDs[iCounter]++;

  double percentage = 0.0;
  for (uint32_t i = 1; i < iNumOfTaxIDs; ++i) {
```

```
    if (percentage >= 0.99) {
      return i - 1;
    }
    percentage += static_cast<double>(taxIDs[i]) / iUniquekMerCounter;
  }
  return 4; // Some magic number that signals something went wrong
}
```

Summary

In this chapter we introduced the data structures and modules of which `kASA` consists. First, the more technical side was presented in form of library functions, customized data structures and our parallelization scheme optimizing program execution time. It was shown that `kASA` has minimal dependencies when installing, thus requiring almost no further work from the user, which broadens the potential user base. The implementation of our trie and the bit array set offers significant reduction of execution time and the Worker-Thread/Queue classes enable parallelization of many parts of the code. Lastly, we saw how locking would reduce performance without being necessary and therefore was omitted making `kASA` almost completely parallelizable.

Second, we presented the modules which serve as the core functionality for the software and contain all ideas which we had during development. We saw that in order for `identify` to work we need `build` and to offer versatility, additional modules like `shrink`, `merge`, and `update` were necessary. This gives the user and reader the power as well as context to use `kASA` in various ways mentioned. Additionally, we presented procedures and algorithms as well as figures explaining how our software works.

Algorithm 3.1: Identify

input : The index and the sorted input with converted k-mers and ranges
output: Scores and counts

```
// Part One
```
list of matched read IDs per k: $R_M(i,k) \leftarrow []$;
list of matched taxonomic IDs per k: $T_M(i,k) \leftarrow []$;
list of last known k-mer per k: $known\_k \leftarrow []$;

for every *entry* with *range* in input **do**
    reset all lists;
    $k\text{-}mers_{[lower,higher]}$ = gather all k-mers with the same range;
    for all k-mers in $k\text{-}mers_{[lower,higher]}$ **do**
        $i \leftarrow$ current k-mer;
        if range invalid **then**
            `continue`;
        currKMerShifted $\leftarrow$ i with lowest value for k;
        start $\leftarrow$ *lower*;
        end $\leftarrow$ *higher*;
        do once
            if *currKMerShifted* is in *range* **then**
                use binary search to find the *start*;
            else
                `continue`;
        if i has been *seen before* **then**
            for all k's **do**
                add *read ID* to $R_M(i,k)$ if i_k matches entry in $known_k$;
            `continue`;
        else
            seen before $\leftarrow i$;
        `// Part Two`
        `// see below`
    `// Part Three`
    `// see below`
return Scores and counts

```
// Part One:  see above
for every entry with range in input do
    for all k-mers in k-mers_[lower,higher] do
        // Part Two
        for x in index from start to end do
            for all k's from lowest to highest do
                i_k ← i shortened for this k;
                x_k ← x shortened for this k;
                if i_k < x_k then
                    for all remaining k's from k to highest do
                        add read ID to R_M(i,k) if i_k matches entry in known_k but
                         avoid duplicates;
                    break out of the x loop and get next i;
                else
                    if i_k == x_k then
                        if i_k matches entry in known_k then
                            add taxonomic ID from x to T_M(i,k) and read ID to
                             R_M(i,k) but avoid duplicates;
                        else
                            // all possible matches for old i_k found ->
                               save
                            for all entries in T_M(i,k) do
                                count unique or non-unique match;
                                for all entries in R_M(i,k) do
                                    save match of read ID and taxonomic ID;
                                reset R_M(i,k) and add the current read ID;
                                reset T_M(i,k) and add the current tax ID;
                                known_k ← i_k;
                    else
                        // i_k > x_k
                        iterate x further through index and add taxonomic IDs to
                         T_M(i,k) if x_k matches entry in known_k;
                        stop if no full match occurs;
                        break out of k loop;
            if highest k was reached then  get next x ;
        // Part Three:  see below
```

```
for every entry with range in input do
    for all k-mers in k-mers_[lower,higher] do
        // Part One and two
        // <see above>
        // Part Three
        // look for any remaining x in case there are no more i
           left
        for any x left in range do
            for k from  lowest to highest do
                if x_k matches entry in known_k then
                    add taxonomic ID to T_M(i, k);
                else
                    break;
            if at least one k matched then
                get next x;
            else
                break;
        // all possible matches for old i_k found -> save
        for all entries in T_M(i, k) do
            if size of T_M(i, k) == 1 then
                save unique match;
            else
                save non-unique match;
            for all entries in R_M(i, k) do
                save match of read ID and taxonomic ID;
return Scores and counts
```

Chapter 4

Experimental design and results

The premise of this chapter is to apply `kASA` to data and compare the results with those of relevant software introduced in Section 2.1. We present results from multiple benchmarks in order to evaluate `kASA` in a reasonable way. First, we use data specifically designed for taxonomic profiling from other benchmarks to see if `kASA` is able to perform at least as good as established programs in Section 4.2. Then in Section 4.3 we are going to show our own benchmarks with synthetic data to prove that our statement that our software is indeed more robust than established methods, holds. Afterwards, real data experiments in Section 4.4 are used in comparison with other software to see if they agree on the composition of known genomes. This includes tests with third-generation sequencing techniques as well. We therefore check theoretic results empirically or present valuable conclusions made during our experiments.

4.1 Preliminaries

To better understand the experiments further down, we would like to explain the framework used for our tests and some measurement formulas used.

4.1.1 Snakemake

The Python programming language together with snakemake [150] offers the possibility to automatically test our and other software on the same data without needing to fine-tune or write scripts for every software. With snakemake we only need to set so called *rules* which describe in which order calls shall be made. It supports multi-threading and concurrent execution of rules which do not depend on one another. We used an HPCC environment for most of our tests, so this proved to be the most sophisticated and maintainable method. Furthermore, it allows recreation of our experiments on other computers by downloading them from our GitHub pages and rerunning them with just minor changes to a configuration file [11, 12].

4.1.2 Quality measurements

This section contains the frequently used measurements for assessing the accuracy of our and other software. In most benchmarks, the accuracy of the r-identification is evaluated. For the synthetic robustness test in Section 4.3 both the accuracy per read and per taxon is evaluated. The former is done by checking whether the original taxon of a read was identified correctly by their taxonomic ID. If it matched, it was marked as a correctly identified and assigned read, otherwise only as assigned (decreasing precision). Only the best hits were considered. If not specified otherwise, the json array for every read containing the "Top hits" was used for `kASA` (see Section 3.4.5). If two or more taxonomic IDs matched the read with an equal score, it was additionally considered ambiguous but correctly assigned if at least one taxon was correct. This ambiguity is printed as additional information but does not influence the evaluation. If the LCA-based algorithms (see Section 2.1) gave a higher taxonomic rank as result (while containing the correct ID), the assignment was also considered ambiguous but correctly assigned. If the taxonomic path given by backtracking the LCA-path did not contain the correct ID, it was considered incorrectly assigned. We added genomic reads from species not inside the database/indices to test the ability of every software to "ignore" reads. This is measured with the specificity via checking if a nonassignable read was correctly not assigned. The formulas were derived from the definition in [109, 151] and are as follows:

$$\text{Sensitivity} := \frac{|\text{Correctly assigned reads}|}{|\text{Reads}|} \tag{4.1}$$

$$\text{Precision} := \frac{|\text{Correctly assigned reads}|}{|\text{Assigned reads}|} \tag{4.2}$$

$$\text{F1 score} := 2 \cdot \frac{\text{Sensitivity} \cdot \text{Precision}}{\text{Sensitivity} + \text{Precision}} \tag{4.3}$$

$$\text{Specificity} := \frac{|\text{Correctly unassigned reads}|}{|\text{Nonassignable reads}|} \tag{4.4}$$

As these measures do not penalize the assignment of one read to multiple (sometimes wrong) taxa, we added the perspective of each taxon in our benchmarks with synthetic data. This can be done via a binary classification: The "original label" is the true taxonomic ID of the taxon the read originates from, the "reported label" is the taxonomic ID the software returns for that read.

- True positives (TP) - The original label and the reported label are identical.
- True negatives (TN) - The original label came from a taxon not inside the index and, correctly, no label was reported.
- False positives (FP) - The original label and the reported label are not identical or the original label came from a taxon not inside the index but a label was reported.

- False negatives (FN) - The original label came from a taxon inside the index, but no label was reported.

With this, we can calculate the four values for every expected taxon and get the Matthews correlation coefficient (MCC) [152]:

$$\text{MCC} \coloneqq \frac{(TP * TN - FP * FN)}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}} \tag{4.5}$$

We then averaged these MCCs for every file and got a measure of how specific software reports taxa per read.

We would also like to mention three measures used to calculate how close the reported distribution of taxa in a profile is to the true distribution. The log-odd score [91] for a taxon uses the relative frequencies introduced in Equation 3.8 is calculated by:

$$\log_2 \left(\frac{\text{relative frequency}_{\text{observed}}}{\text{relative frequency}_{\text{true}}} \right) \tag{4.6}$$

These values are summed for all taxa. The closer this sum is to zero, the better. It also offers the possibility of comparing profiles from different programs with each other.

The other two measures are the Pearson correlation coefficient (PCC) [153] and Spearman's rank correlation coefficient (SRCC) [154]. Both measure the linear correlation between two sets of data X and Y. For the PCC the values are taken directly, for the SRCC, the values of each set are sorted and then given a ranking rg. With Cov being the Covariance and sd being the standard deviation, the formulas are as follows:

$$\text{PCC}_{X,Y} \coloneqq \frac{Cov(X, Y)}{sd(X)sd(Y)} \tag{4.7}$$

$$\text{SRCC}_{X,Y} \coloneqq \text{PCC}_{rg(X),rg(Y)} \tag{4.8}$$

The closer both values are to one, the higher the correlation. Zero means no correlation and minus one shows an anti-correlation.

With this we can perform quality measures of our benchmarks.

4.2 Existing benchmark studies

Because taxonomic profiling is a common problem and no software has every feature desired, more are developed and tested in benchmarks. The problem is that these benchmarks are static as in they take a certain version of a software and evaluate data developed only for this test. The latter can be used to benchmark other software published later, but to fairly evaluate the other software, one would have to redo the study all over. Because software usually continues in development, this study is then

no longer as meaningful. There are efforts to unify this to avoid redundant tests whenever a new software is published [155]. However, researchers continue to rely on such static studies to decide on a software they use for their research (e.g.: [156] (in Section "Rapid and Moderate Comprehensive Sequencing Analysis Followed by Exhaustive Remnants Analysis"), [157] (in Section "Metagenomic sequence analysis")). We therefore would like to present two often cited benchmark studies and show how `kASA` performs with their data and measurements.

4.2.1 McIntyre et. al.

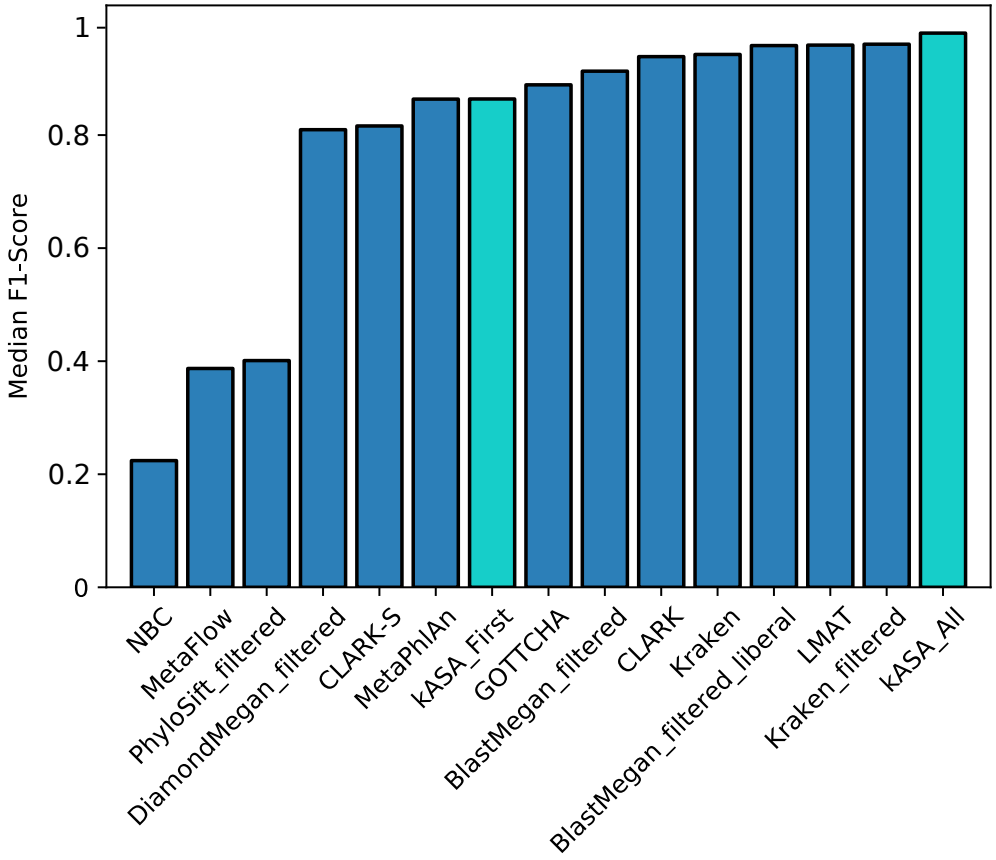

Figure 4.1: Median of the F1 scores from 10 data sets for all software evaluated in the study of McIntyre et. al. plus `kASA` (highlighted). The two results for `kASA` come from either taking only the best scoring taxon from the "Top Hits" per read (kASA_First) or all taxa from the "Top Hits" (kASA_All) (as described in Section 3.4.5). The latter scores highest.

The study of McIntyre et. al. [7] was primarily designed to test how false positives during a metagenomic study can be reduced. This included running combinations

of software on the same data to see if it would increase confidence (and thus reduce false positives) in the result to, e.g., make clinical decisions based on the composition of the sample. Multiple data sets were designed as a ground truth for the evaluation. Of the 35 simulated data sets from this study we selected 10: two files with high complexity and even distribution of species (named "HC" in the study) and eight files with low complexity and log-normal distribution of species (named "LC"). These files had therefore known abundances for every species contained, so we were able to check if that distribution was detected correctly by `kASA`. This served as an additional sanity check which is not presented here. The used quality measures of sensitivity (Equation 4.1), precision (Equation 4.2) and the resulting F1 score (Equation 4.3) are the same as described in Section 4.1.2. We chose the taxonomic level "species" because it strikes a balance between index size (the lower the taxonomic level, the larger the index) and complexity. The index was created in accordance with the default databases of the other programs. The publication offered results for subspecies, species and genus level so we did not have to rerun other software and could just add results for `kASA` to the table. Figure 4.1 shows the median of the F1 scores from the results for the selected data sets. For `kASA`, we had to choose two different methods for the evaluation. The first is similar to how software that does not support multiple assignments per read operates and takes only the taxon with the highest relative score (see Equation 3.4) for every read (thus named `kASA_First`). The second takes all taxa with almost the same score as the best one into consideration (the "Top hits") which is equivalent to printing out multiple assignments per read (named `kASA_All`).

Using only the first best taxon for every read results in a sufficiently high F1 score in comparison to other software. However, the real strength of `kASA` is the ability to print out multiple assignments per read with an equal or slightly lower score. This leaves the user with more information to work with and we see that it results in the best F1 score of all software tested.

4.2.2 Lindgreen et. al.

The benchmark study designed by Lindgreen et. al. [151] contains, among other evaluations, a benchmark of the profiling capabilities of multiple software. Together with their design of the data sets which are modeled after real data, we concluded that it would give `kASA` the opportunity to be evaluated before doing experiments with real data. The test files are composed of two different simulated bacterial communities and for each community, three replicates were created which sums up to six data sets. These replicates have the same relative abundance for each phyla but the genomes and the reads from those genomes were chosen randomly. The targeted taxonomic levels are genus and phylum to make the test more robust against differences in the standard databases of the programs (e.g. different names for the same species). We adopted this strategy for our real data test in Section 4.4.2 as well. The reference database for `kASA` and CLARK was custom made due to deprecation of genomes since publication date and consisted of all known bacteria, viruses, and archaea (2019-07-22). Eukaryotes were neglected because not all software in the benchmark had them in their standard databases. We then calculated the profile for all six data sets with `kASA` and CLARK. CLARK was used here as an indicator if our method is valid. We expect the scores to be worse due to the deprecation of some species used then. However, should the values of CLARK be almost equal to those reported in the publication we know that our custom database did not introduce a bias. As done in the benchmark, we used the sum of the absolute log-odd scores (Equation 4.6) and the PCC (Equation 4.7) as a measure of similarity between the true and observed values for the taxa on the mentioned taxonomic levels. For `kASA`, we calculated the abundance of a taxon by taking the r-identification output and counting the number of reads that had the respective taxon as best scoring hit. If multiple high scoring taxa were reported for a read, each reported taxon got $\frac{1}{|\text{taxa}|}$ added to their count. This count is then divided by the total number of reads resulting in a read-based relative frequency for our evaluation. This was done in order to be consistent with the method used for other software in the publication. We also calculated the profile and looked at the relative frequencies calculated directly from the k-mers to see if they would also be viable.

Figure 4.2 shows that `kASA` is the best software regarding summed up log-odd scores and among the best for the PCC. The values for CLARK are almost equivalent to the ones reported in the publication which shows that our reference is valid. The (non-unique) relative frequency calculated from the r-identification output and the profile generated by `kASA` are quite similar with a PCC of 0.96 and an SRCC (Equation 4.8) of 0.98. We therefore conclude that `kASA` should be able to accurately profile real world-like data sets.

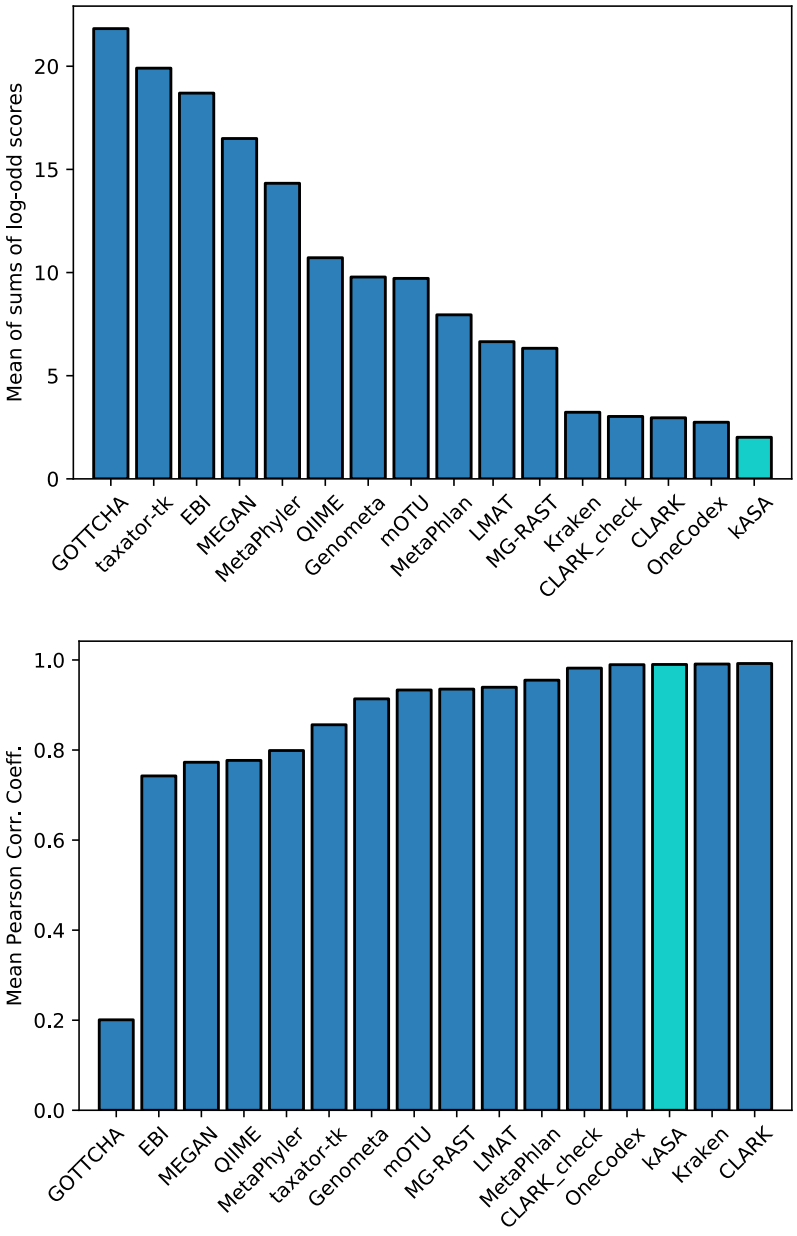

Figure 4.2: Top: Mean of the summed up log-odd scores across all six data sets, lower is better. Bottom: Mean of the PCC across all six data sets, higher is better. CLARK_check stands for the values we got after rerunning it with our reference. Results for `kASA` are highlighted.

4.3 Benchmarks with synthetic data

In order to evaluate if `kASA` can really identify the most likely taxon for a read as well as the composition of genomic data, we need to have a ground truth. This can be done by using reference genomes for which we know the taxonomic ID as input. Therefore the following two benchmarks use a set of genomes as a database from which the index as well as the input data is generated. This way we can assign the correct identifier to each read and afterwards check if it was correctly identified.

The first benchmark with synthetic data (named this way because data is synthesized from existing genomes) in Section 4.3.1 is not only checking the correctness of every read but also the robustness against mutations. This aims to empirically validate our theoretic ideas of a dynamic k (see Section 2.2.1) and an amino-acid like encoding (see Section 2.2.2). The second benchmark in Section 4.3.2 is derived from the CAMI Challenge [8] which tests software similar to `kASA` with various methods based on the taxonomic profile. Both benchmarks work within a snakemake framework and are easily verifiable by other researchers. They can be found in our GitHub repositories [11, 12] and a flowchart can be seen in Figure 4.3. Both benchmarks are also evaluated regarding speed and memory consumption in Section 4.3.3. Most of the software presented in Section 2.1 is used as comparison for `kASA`. Alignment based software (see Section 2.1.1) is not included as their execution times are too high to be comparable to the others. A benchmark regarding `kASA` and Kaiju [102] is given separately in Section 5.4 because Kaiju accepts only protein sequences as reference and the others do not.

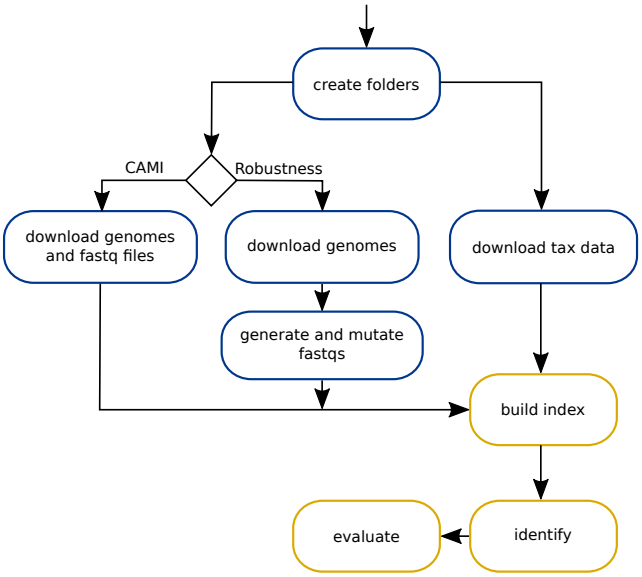

Figure 4.3: Workflow of both snakemake benchmarks. The steps 'build', 'identify' and 'evaluate' (yellow) are carried out for every tested software. The other steps (blue) are called only once.

4.3.1 Robustness

As mentioned, this benchmark uses a set of known genomes as database. Currently, we use the largest chromosome (or whole genome in case of prokaryotes) of seven model organisms and a negative control of three species to check the specificity. The names of the taxa are as follows (NEG marks the taxa used for negative controls):

- *Homo Sapiens chr1*
- *Mus musculus chr1*
- *Arabidopsis Thaliana chr1*
- *Drosophila melanogaster chr3R*
- *Saccharomyces cerevisiae chrIV*
- *Escherichia Coli*
- *Sulfolobus solfataricus*
- NEG: *Phaeodactylum tricornutum chr1*
- NEG: *Acinetobacter baumanii*
- NEG: *Tobacco mosaic virus*

We would like to point out that this list can be expanded at will, we just chose a minimal working example to keep the necessary computational requirements low. The genomic source was downloaded from the NCBI [62, 5] and then converted to indices until all data was available for every program tested. Input data was generated by randomly sampling reads of length 100 bases from the genomes and mutating them with single point mutations, insertions, and deletions until a given number of mutations per read is reached (ranging from 0 to 20). These values are much higher than found in nature [158] but used in this context to test the limits of our approach. The generated `fastq` files (see Section 3.3.2) were then tested for every software with the measurements given in Section 4.1.2. The index for `kASA` was built using the default three translation frames, all other indices were build as described in the software's respective documentation. In addition to the default three frames used for the input, we test `kASA` with one frame as well (named "`kASA_one`"). We suspect that one frame for the input may be sufficient since the genomic data was already saved in three frames inside the index. Note that using six frames for the index would be unnecessary since no reverse complement reads were generated.

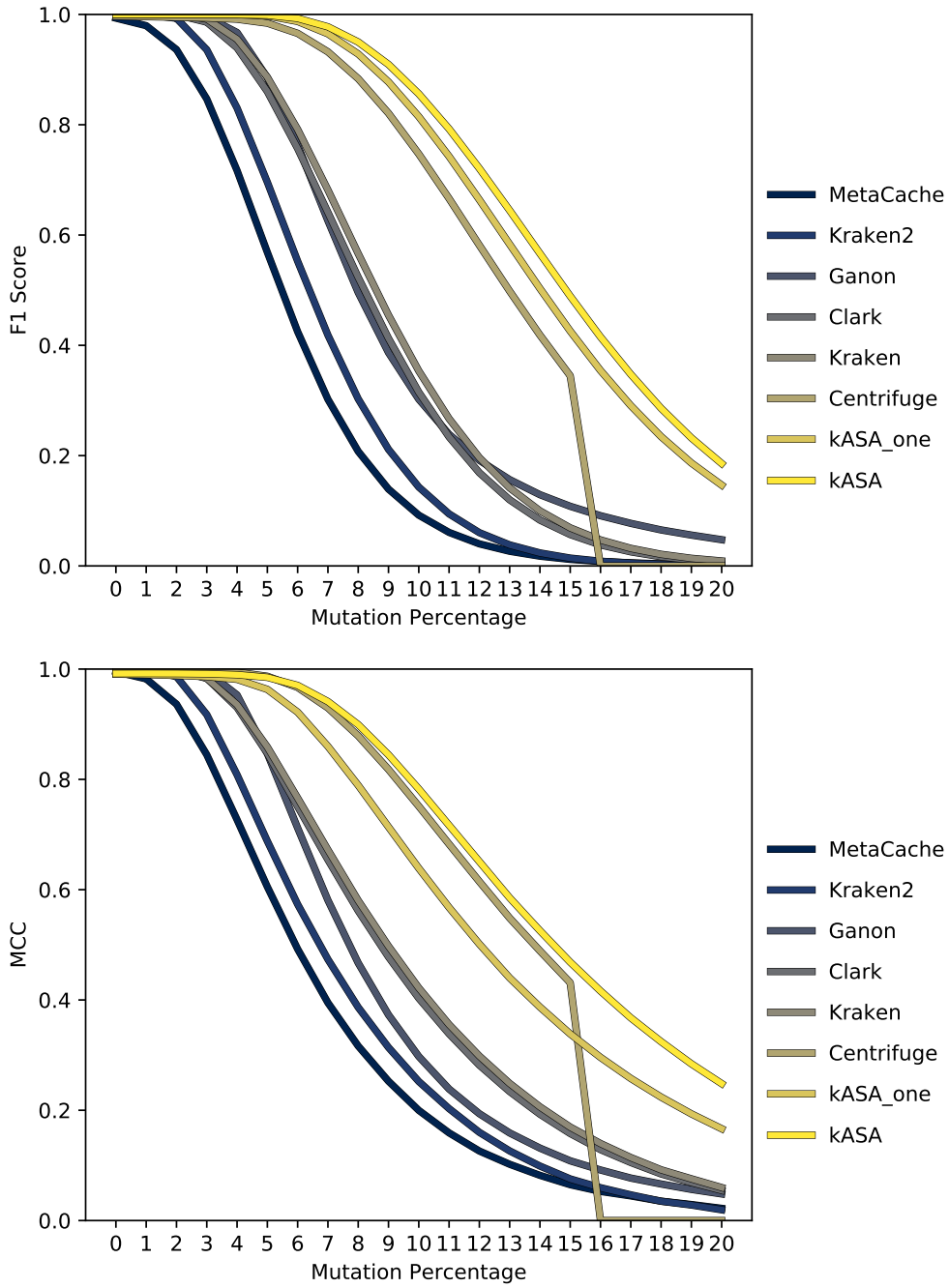

Figure 4.4: F1 score (top, see Equation 4.3) and MCC (bottom, see Equation 4.5) of tested software for our simulated data. Because all software was used with default settings, `kASA` is shown with default settings of $k \in [7, 12]$ and three/one frame(s) as well.

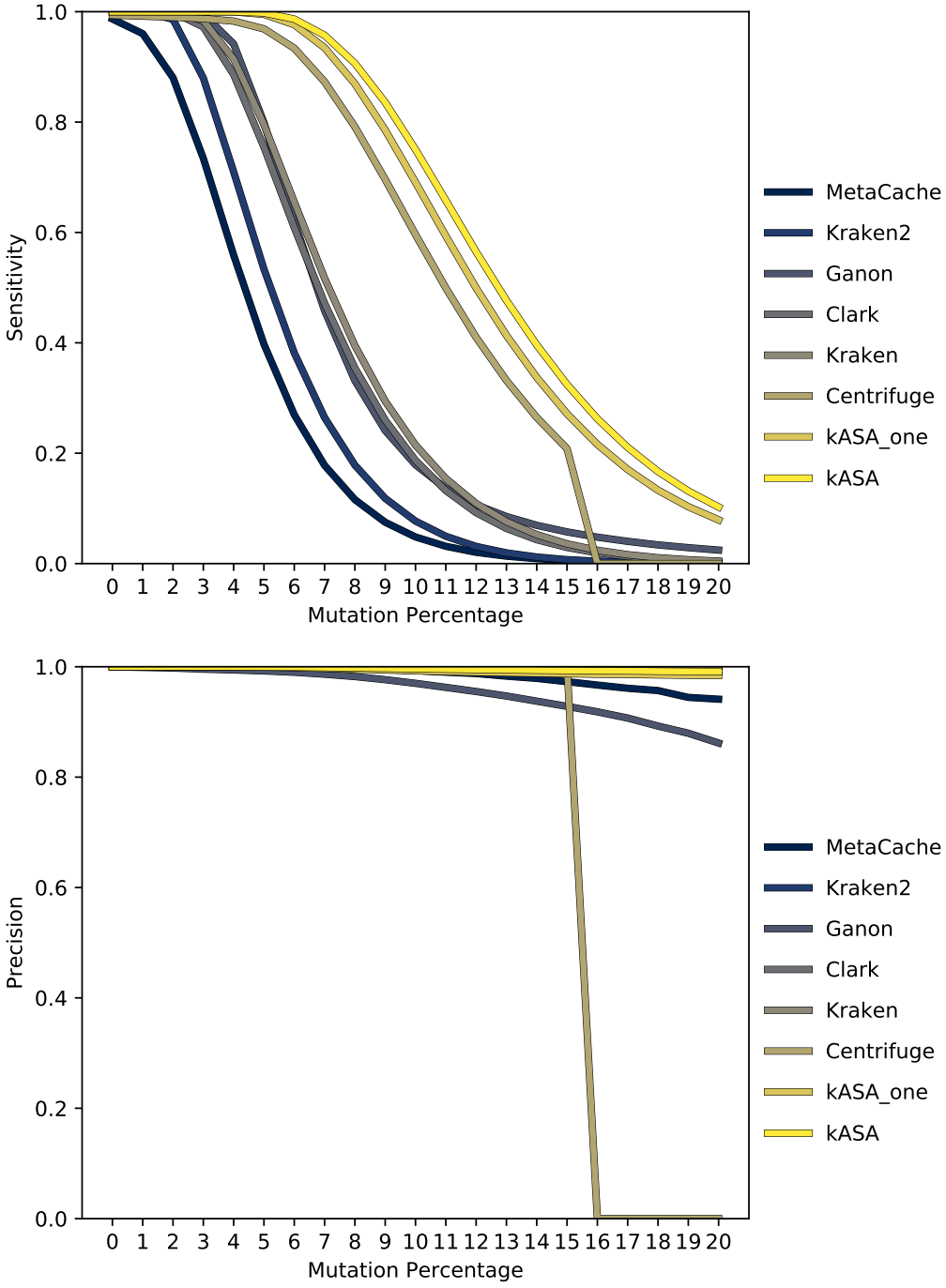

Figure 4.5: Sensitivity (top, see Equation 4.1) and precision (bottom, see Equation 4.2) of tested software for the synthetic data with different mutation percentages.

We see in Figures 4.4 and 4.5 that the sensitivity of `kASA` stays very high even if seven percent of every read differs from its reference. From there on, the robustness is visibly higher than that of all other tested software. The values for Centrifuge drop for two reasons: First, because the required minimum exact matching seed length of 16 bases is achieved in fewer and fewer cases. Second, an internal threshold is applied so that low scoring matches are ignored. The latter lets the score drop to zero at 16%. We suspect that this was implemented to increase specificity. The precision of `kASA` is comparable to those of the best software. Ganon and MetaCache lose precision when the number of mutations increases (due to hash table collisions or misses). We clearly see that our methods together increase the robustness against sequence errors without sacrificing precision. Even if only one translation frame is used, the sensitivity of `kASA` is still higher than that of other software. However, the MCC is visibly lower than in the three frames mode, which means that fewer correct taxa are reported.

Because `kASA` detects every similarity, we also need to look at the specificity (see Equation 4.4) to rule out that `kASA` would misidentify an unknown taxon with a high score. Figure 4.6 shows that the specificity for both translation frame settings is very low because even very low scoring hits are reported. Applying a low threshold on the relative score (see Section 3.4.5) resolves this as can be seen in Figure 4.7. This threshold has nothing to do with the classification in "Top Hits" and "Further Hits" mentioned in Section 3.4.5. It is rather used as a cutoff to separate identified reads with a high relative score (≥ 1) from those with a low relative score (< 1). We applied various thresholds to see how sensitivity and specificity are influenced by different mutation percentages (5 and 10 were chosen as representatives). For zero mutations, a threshold up to 1.0 increased specificity to almost 1 but did not influence sensitivity at all. For the two mutation percentages, there is a direct link between increasing specificity and decreasing sensitivity. This effect gets stronger the higher the mutation percentage per read is. This is expected because scores get lower overall if the number of matches decreases. We therefore recommend to decide on a threshold on a case-by-case basis if needed and are confident that users are able to interpret those results given that they know the amount of noise in their data.

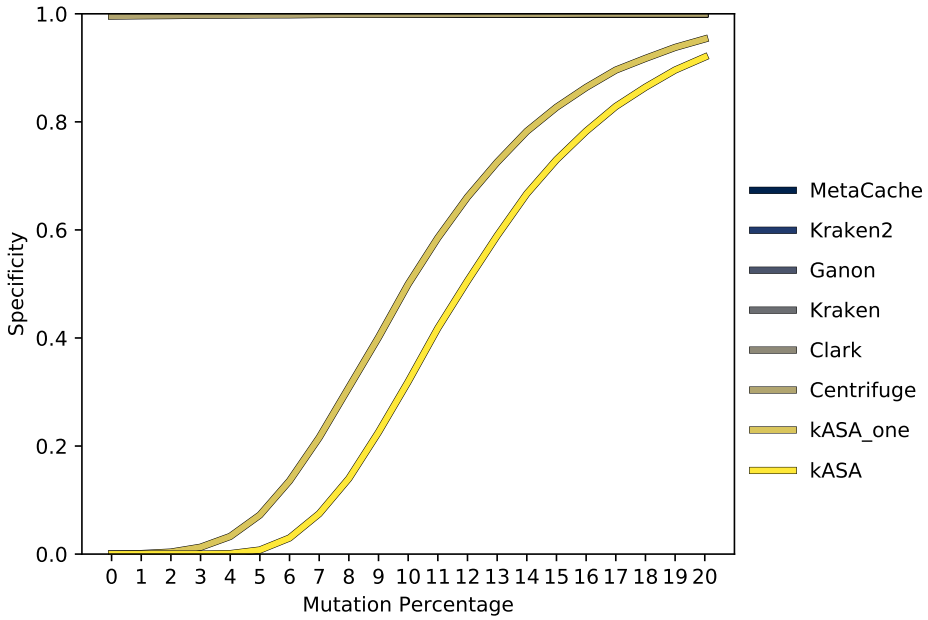

Figure 4.6: Specificity for all tested software, same color and settings as in Figure 4.4. The line at the top results from the graphs of multiple programs lying on top of each other. Only `kASA` and `kASA_one` have a lower specificity than all others.

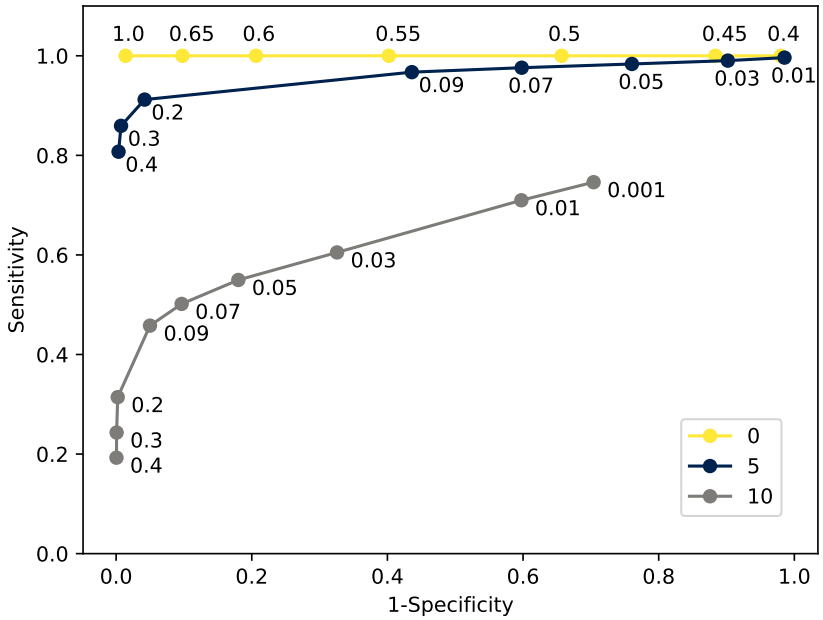

Figure 4.7: Receiver operating characteristic curve (ROC curve) [159] for `kASA` with zero, five and ten percent mutations and various thresholds.

We were also interested in the influence of the range of k's on the robustness. The shorter the k is, the more similarities can be found, but the longer `kASA` takes to compute them all. Using the largest k of 25 as an upper bound, we calculated F1 scores for all lower k's, the result can be seen in Figure 4.8. We see that for large k the F1 score drops considerably due to low sensitivity and ultimately due to no identification at all. For small k, the precision is impacted by the effect mentioned before that all scores get lower and thus many different taxa overshadow the true ones. However, there seems to be a subset $k \in [5, 6]$ where sensitivity and precision are optimal (see Figure 4.9). Unfortunately, finding these optimal lower k's is generally only possible with a ground truth and depends on the data and sequencing length. To show how `kASA` performs without such knowledge, we chose the default lower k of seven for our benchmark even if it may put us at a disadvantage.

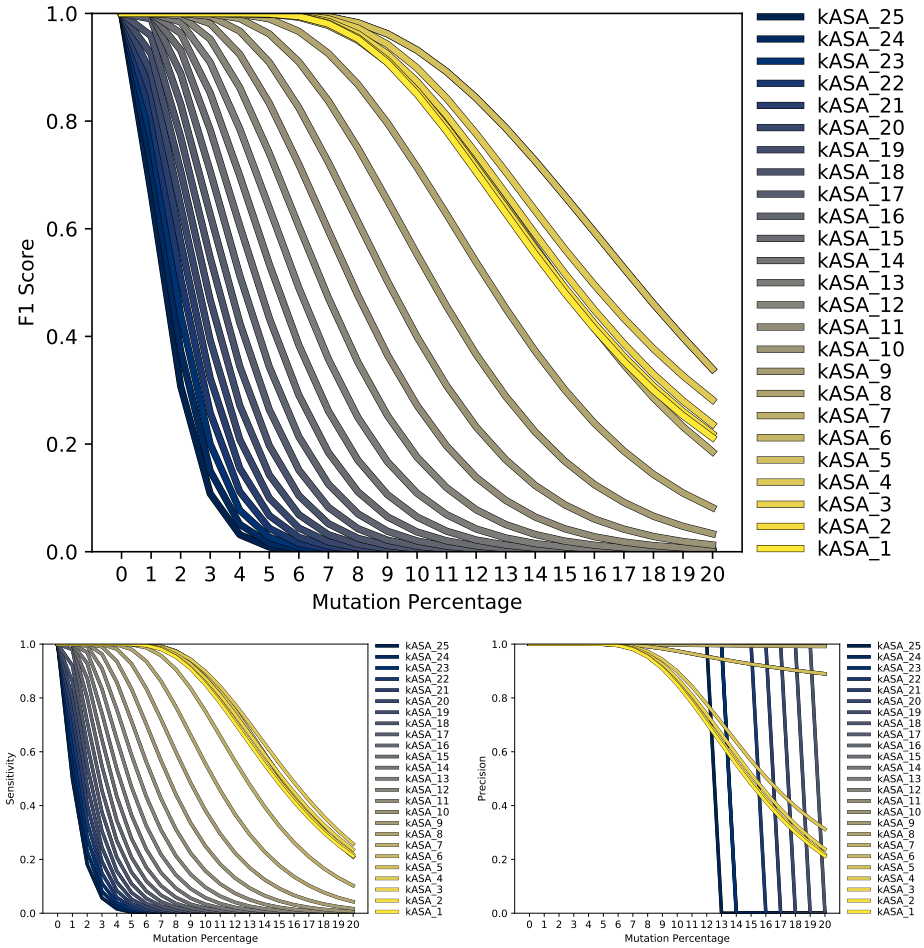

Figure 4.8: F1 score (top), sensitivity (bottom left) and precision (bottom right) of `kASA` for all lower k's with a higher k of 25.

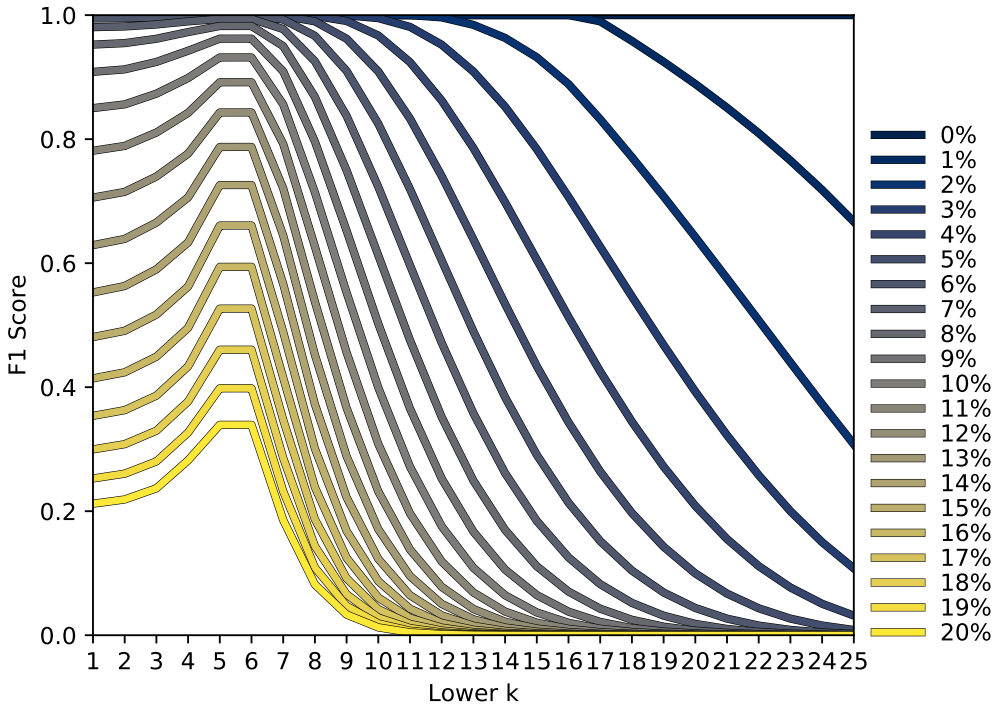

Figure 4.9: Inverted plot showing F1 score curves for all mutation percentages from Figure 4.8 of all lower k's.

The last thing to check is whether the amino acid-like encoding by itself could improve robustness already since the degeneration of the genetic code proved to be a successful evolutionary advantage up until now. To isolate this factor, we need to fix the upper and lower k to the same value. Other k-mer based software like CLARK or Kraken (see Section 2.1.4) use a fixed k of 21 and 31, respectively. They both use DNA instead of an amino acid-like encoding thus providing a base for comparison. To compare `kASA` with these programs we fixed k to 10, simulating a k of 30 on the DNA level. To fully utilize the amino acid-like encoding, we created the index with six frames and translated all input reads with six frames as well. The test data was modified to include reads in reverse complement to the genome.

Figure 4.10 shows how `kASA` performs in comparison to CLARK and Kraken. We see that even though the graphs are close together, there is a noticeable difference with `kASA` being better when the number of mutations increases. This confirms our hypothesis that the amino acid-like encoding has an inherent ability to handle mutations but at the same time shows why the dynamic k is necessary as well.

We also tested how the number of frames affects the robustness. After creating three indices, each with a different number of frames (one, three or six), `identify` translated the reads with one, three or six frames as well. Figure 4.11 shows how the

robustness is affected by the choice of frames for index and reads. As expected, the use of only one frame for the index and input is suboptimal. Using six frames for both, on the other hand, leads to the highest robustness. This is even more evident in Figure 4.12, where any setup with six frames is more robust than the rest.

We conclude that `kASA` outperforms the other tested software regarding robustness (see Figure 4.4). We also see in Figure 4.8 that the influence of the dynamic k is quite strong and thus fixing the value for k is disadvantageous. The effect of the amino acid-like encoding is shown in Figure 4.10 where there is a small but noticeable improvement compared to the DNA based k-mer approaches of Kraken and CLARK. These results corroborate our statements and show that our strategies for increasing robustness with `kASA` seem to indeed be viable in practice.

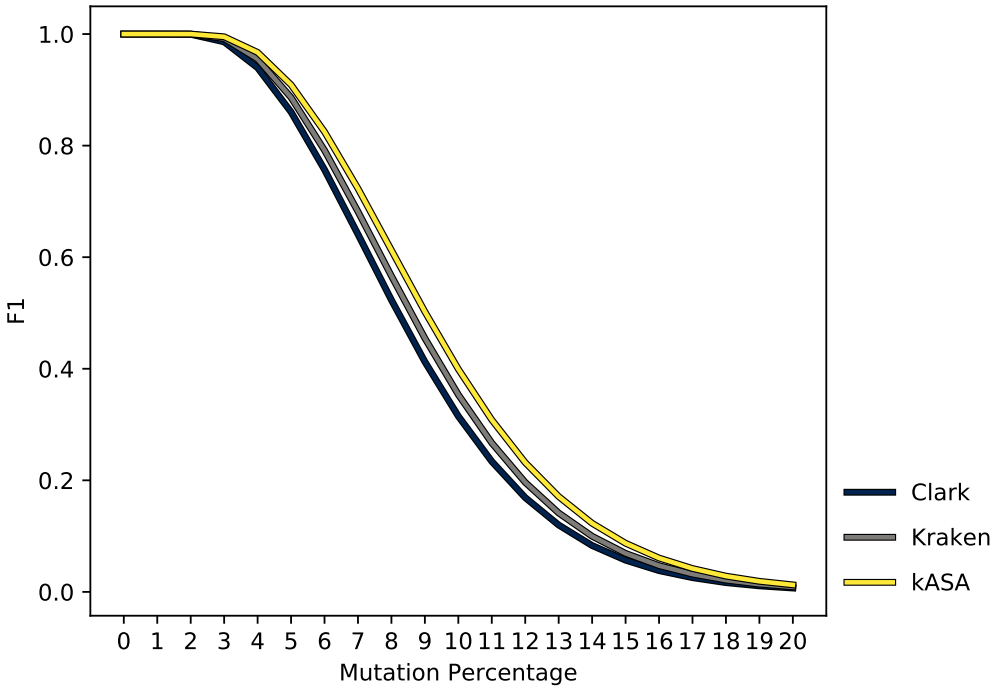

Figure 4.10: Comparison of fixed k DNA software and `kASA` using only the amino acid-like encoding with six frames for index and input.

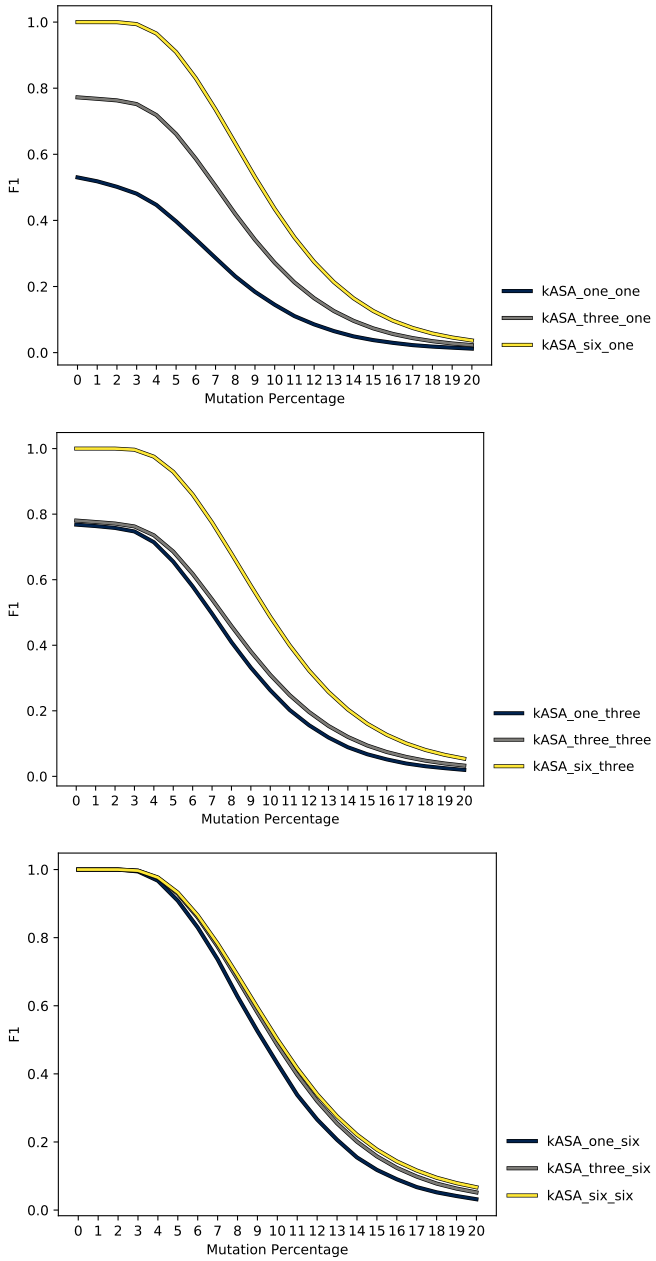

Figure 4.11: F1 score of `kASA` with a fixed k of 10. The index was built with one (top), three (middle), and six (bottom) frames. For the input, one, three, or six frames were used for each index.

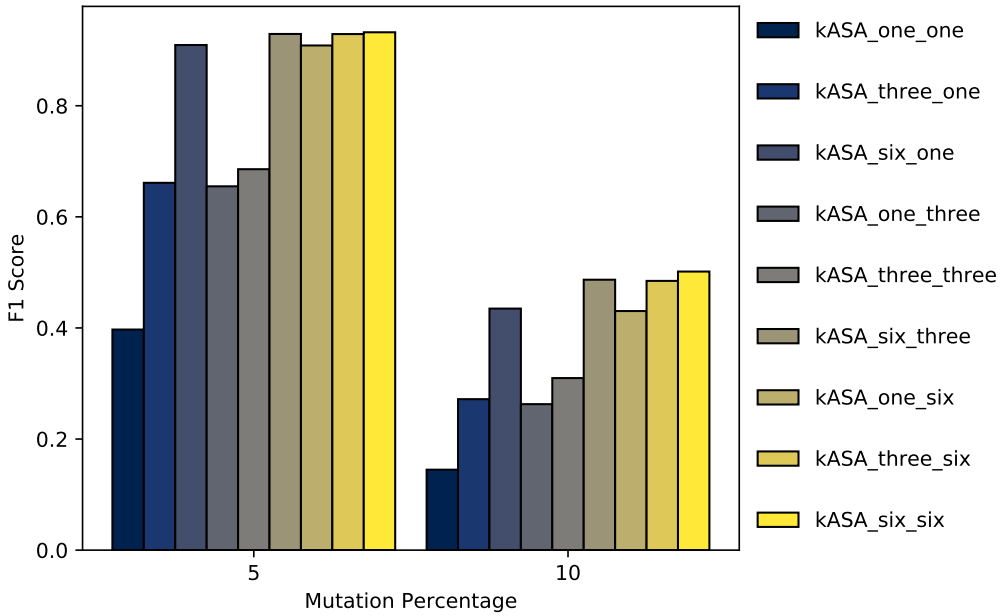

Figure 4.12: F1 score of `kASA` with a fixed k of 10 showing values for five and ten percent mutation for all combinations from Figure 4.11.

4.3.2 CAMI

This benchmark is inspired by the "Critical Assessment of Metagenome Interpretation (CAMI)" Challenge [8] in which several software was tested for their profiling capabilities, among other things. Together with OPAL [60] we were able to create a snakemake pipeline which allows a ranking of the profile quality. The data for this is taken from the "2nd CAMI Toy Human Microbiome Project Dataset (gastrointestinal tract)" for which a true "gold standard" profile is available. It contains a set of genomes for the indices and reads in a format used by PacBio [160] with long reads (2500 bases on average) and many sequencing errors (Phred score [142] of eight), making profiling a challenge for the software used in the previous benchmark. Note, that we do not include results from Ganon because it could not detect any taxa. Furthermore, OPAL reports entries with very low relative frequencies as false positives, so we set a threshold of 0.01 on the species level after some preliminary tests for all applicable software (Kraken and Kraken2 report many taxa with a value of zero so no thresholds were applied). The results of this test can be seen in Table 4.1.

First, we want to determine which relative frequency describes the real composition of the taxa best, since the profile contains counts and relative frequencies for all k's of the user given interval. We then compare the scores of all meaningful k's for both the 64 bit ($k \in [7, 12]$) as well as the 128 bit version ($k \in [10, 25]$). Because the real taxonomic composition of a dataset is usually unknown and the best k is dependent

on that data, we recommend users to use the largest k. Therefore the scores for the highest k of 25 are shown next to those of the best k. Afterwards we compare values for both the best and the highest k with the profiles of other software. We also include results for the fast but more inaccurate choice of one translation frame instead of three.

Table 4.1: Sum of scores (lower is better) given by OPAL for the unique, non-unique, and overall relative frequency for k's from 10 to 25.

k	**Unique**	**Non-unique**	**Overall**
10	1951	1945	1860
11	1651	1811	1775
12	1290	1603	1617
13	1124	1297	1343
14	1020	894	1126
15	706	676	914
16	585	577	868
17	489	502	789
18	445	412	657
19	462	416	642
20	551	492	615
21	718	609	628
22	809	728	668
23	824	794	720
24	813	824	695
25	950	929	849
Average	899.25	906.81	985.38
Median	811	761	819

We see in Table 4.1 that the values and the average score for the unique and non-unique relative frequencies are closer together than the ones of the overall relative frequency are to either of them. Furthermore, the median score of the non-unique relative frequency is lowest which shows that it will most likely describe the real composition best. In both the unique as well as the non-unique relative frequency, the scores for $k = 18$ are the lowest ones which shows the best k for this data which will be used in the comparison as well.

Figure 4.13 shows a comparison of scores from OPAL for the non-unique frequencies (with threshold) for both versions of our index (64 and 128 bit) and all relevant k's. As expected, the values for the same k's are about equal and the larger index is more useful because more information from the long reads can be used. We will therefore only use the results of the 128 bit index for our comparison.

The scores in Table 4.2 are shown for the best (18) and the default k (25) for `kASA`. The scores for other software are given with and without threshold if applicable.

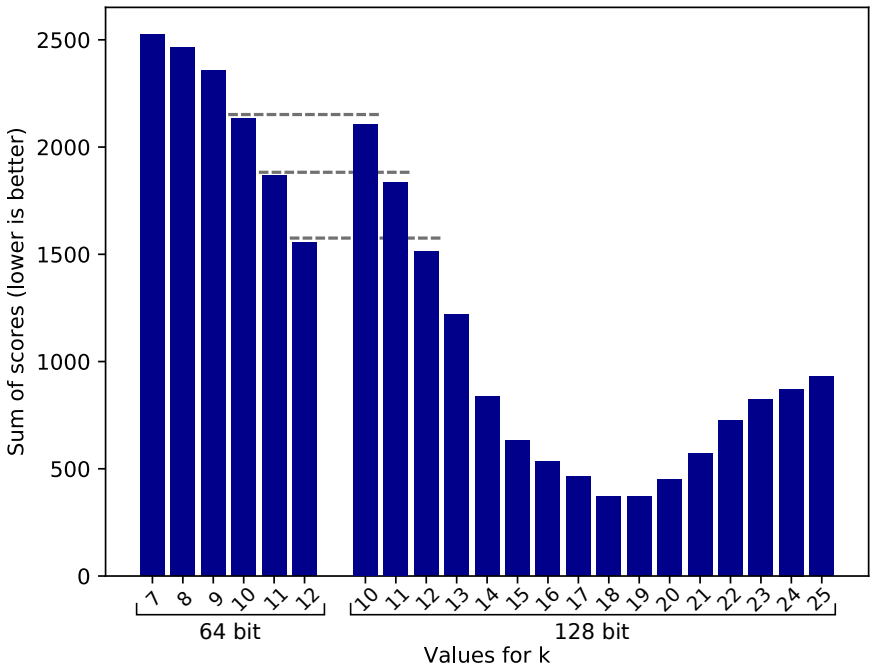

Figure 4.13: Sum of scores from OPAL for different k's for the 64 bit index (highest k is 12) and the 128 bit index (highest k is 25). The dashed lines show that the sums of the scores do not differ significantly when the same k is chosen from the 64-bit and the 128-bit indexes.

`kASA` is shown with the default three and with one translation frame as well. The normalization in OPAL ensures that the sum in every taxonomic rank is 1. This however benefits only Kraken and Kraken2 as can be seen when this normalization is turned off (score of 421 vs 883 for Kraken). The reason is that Kraken and Kraken2 report some taxa even if their abundance is zero. With normalization enabled, these taxa are counted regardless. We can only guess why Kraken and Kraken2 show this behavior.

Using only one translation frame comes at almost no cost in profile accuracy due to the index being created with three translation frames (see Section 2.2). The optimal determined k of 18 scores best but the largest k of 25 is not far behind. Applying a threshold proves to be beneficial, especially for CLARK and MetaCache as the number of false positives can be significantly reduced this way. Centrifuge was second best in the robustness benchmark (see Figure 4.3.1) but performs worst during profiling even with a threshold. We suspect that due to the noisy data, the required minimum exact match of 16 bases was not always met in each read. We conclude that `kASA` is able to profile even very noisy data with good accuracy, that the non-unique relative frequency should be preferred for diverse data and that the

Table 4.2: Sum of scores with and without (-n) normalization given by OPAL. Lower is better, the table is sorted by the first column. 0.01 is the threshold mentioned, and the 18 and 25 represent the k.

Software	Sum of scores	Sum of scores -n
kASA-18-0.01	408	430
kASA-18-one	415	437
Kraken	421	883
MetaCache-0.01	425	409
CLARK-0.01	542	508
kASA-25-0.01	649	595
kASA-25	751	687
Kraken2	882	1351
MetaCache	887	865
Centrifuge-0.01	977	945
CLARK	1106	1063
Centrifuge	1463	1292

highest k instead of the optimal still yields representative results.

4.3.3 Performance and memory consumption

To evaluate the performance of every tested software, we used the snakemake benchmark routine which shows the elapsed wall-clock time and the peak primary memory consumption during testing. Every software was given 40 GB of RAM, eight cores, 10 input files of the same size and ran with recommended default settings on our HPCC [161]. Experiments testing performance on different platforms were additionally run on personal hardware. The versions of all software and system specifications can be seen in Tables 4.3 and 4.4.

Table 4.3: Versions as well as the date of retrieval of used software.

Software	Version and retrieval date
Centrifuge	1.0.4, 2020-03-02
CLARK	1.2.6, 2020-06-01
Ganon	0.2.1, 2020-04-20
Kaiju	1.7.4, 2020-04-20
kASA	1.4, 2020-11-19
Kraken	1.1.1, 2020-03-02
Kraken2	2.0.8, 2020-03-02
MetaCache	1.1.1, 2020-04-20

Table 4.4: Hardware specifications of the testing systems.

Component	iDiv HPC	Desktop	Laptop
CPU	Intel® Xeon® Gold 6148 @ 2.40 GHz or E5-2690 @ 2.9 GHz	Intel® Core™ i7 7700K @ 4.5 GHz	Intel® Core™ i7-6500U @ 2.5 GHz
RAM	1480 GB DDR4	16 GB DDR4 2800 MHz	8 GB DDR3 1600 MHz
SSD	Connected via InfiniBand(TM)	Samsung T7 (USB 3.1G2, 2 TB)	Samsung T7 (USB 3.0, 2 TB)

Experiments performed on the HPCC ran in a Linux environment (CentOS) so `kASA` was compiled with gcc [128]. We tested if execution times vary with the operating system and/or compiler and found no correlation (on our Desktop System). The only thing we would like to mention is that the sorting algorithm for the MSVC compiler [127] is slightly faster due to better parallelization. When disabled and replaced with the same algorithm used for gcc and Clang [129], no significant difference was measurable.

The following figures 4.14 and 4.15 show the measured wall-clock time and the memory and space consumption for both the robustness as well as the CAMI benchmark.

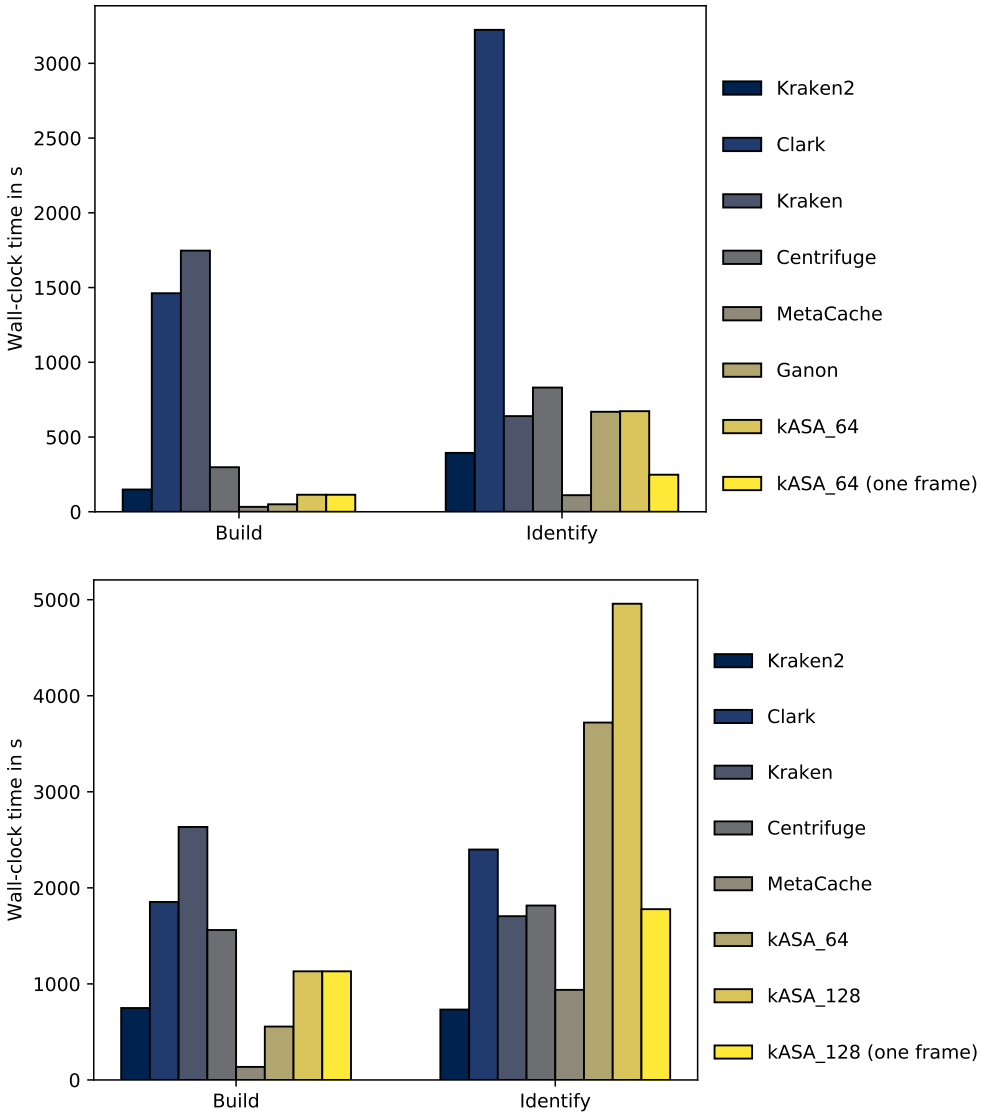

Figure 4.14: Wall-clock times for the robustness benchmark (top) and for the CAMI benchmark (bottom) of all tested software. Ganon is missing from the CAMI benchmark because the profiles were empty.

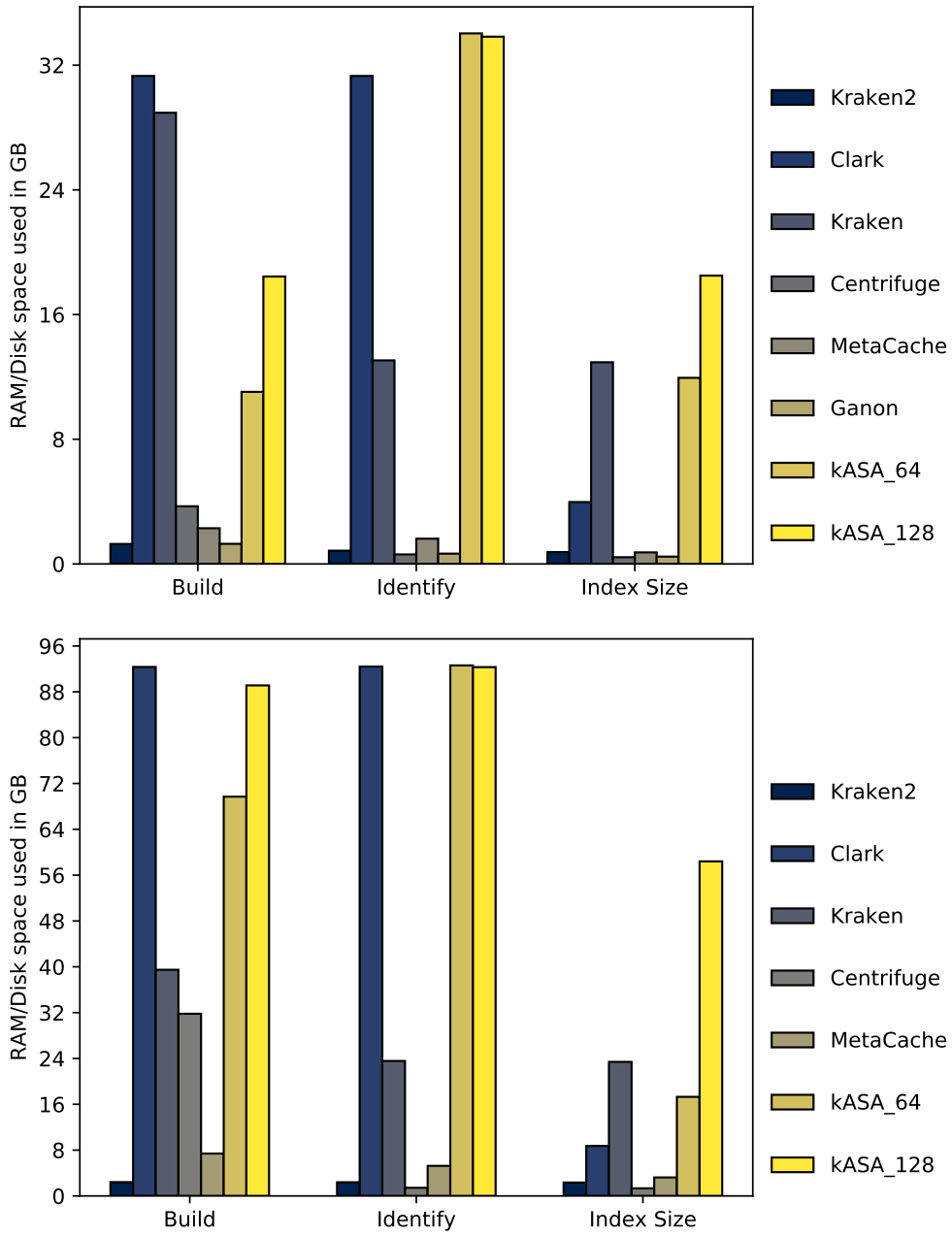

Figure 4.15: Memory and space consumption for the robustness benchmark (top) and the CAMI benchmark (bottom) of all tested software. Ganon is missing from the CAMI benchmark because the profiles were empty.

When looking at the wall-clock times of both benchmarks in Figure 4.14, we see that the hash table-based algorithms (used in Kraken2 and MetaCache) are indeed the fastest. From the one-frame-mode of `kASA`, we can deduce an obvious trade-off between time and accuracy. With default settings, `kASA` ranges midfield in the robustness benchmark and is the slowest software in the CAMI benchmark during identification. When building the index, `kASA` is one of the fastest software but this is because the available memory was larger than the index size. With larger databases, the partitioning of the input, which is almost entirely dependent on the throughput, may decrease performance. We would like to remind the reader that this is the result of our paradigm to be able to build a large index on a memory restricted environment. In Figure 4.15, which shows the primary memory usage in the CAMI benchmark, we can immediately see that some software scales beyond the primary memory capacity of our laptop. `kASA` is only using so much memory because it was available, but Kraken, CLARK, and Centrifuge, for example, would not have been able to build their index on our laptop regardless of the settings. We therefore conclude that `kASA` still needs some optimization but can already achieve good execution times given that it does not use a hash table based index. Together with the conclusions from both benchmarks we see that using only one translation frame reduces accuracy but still performs better than other presented software regarding sensitivity and precision. It also has a better performance than the three frames mode. This makes it a viable alternative mode should the user prefer performance without losing much accuracy.

In Section 2.2.3 we made the statement that `kASA` runs on major platforms (Cluster, Desktop, Laptop) due to a highly customizable primary memory footprint. In Section 3.4.5 we mentioned that due to this memory restriction, the input will be processed in chunks. Therefore, we expect that platforms with less available primary memory will take more time for the same input than systems with more resources. In order to test this, we created a benchmark by using the CAMI benchmark and one input file from it on multiple platforms (each time using four cores). Figure 4.16 confirms our expectation and shows that even on the same system, making more memory available decreases execution time. We also see that throughput is a large factor since it is the only bottleneck of our algorithm (USB 3.1G2 vs USB 3.0). Running this experiment on an external HDD for example led to a very large increase of execution time (see Figure 4.17). We therefore do not recommend using `kASA` on an HDD since its architecture is not optimal for random accesses to the index. Another interesting observation is that should the interface technology allow a high bandwidth, using secondary memory for the index directly can even be faster than the RAM version due to the overhead of loading the index into primary memory first. In case multiple input files need to be processed, this overhead is mitigated since it only has to be done once. Therefore `kASA` still scales better for an HPCC than on a desktop computer with an external SSD. However, the bandwidth for communication protocols has increased rapidly over the last years. For example, USB has gone from 3.0 (5 Gbit/s) to 3.1 (10 Gbit/s) in just three years (2011-2014)

and after another three years to 3.2 (20 Gbit/s) in 2017 [162]. USB 4.0 (released in 2019) offers speeds up to 40 Gbit/s [162]. We would therefore argue that the bottleneck to secondary memory will widen in the near future. One final conclusion from this figure is that CPU clock frequencies do have an influence on execution times: The wall-clock time for the Desktop 5 GB version is slightly lower than that of our laptop.

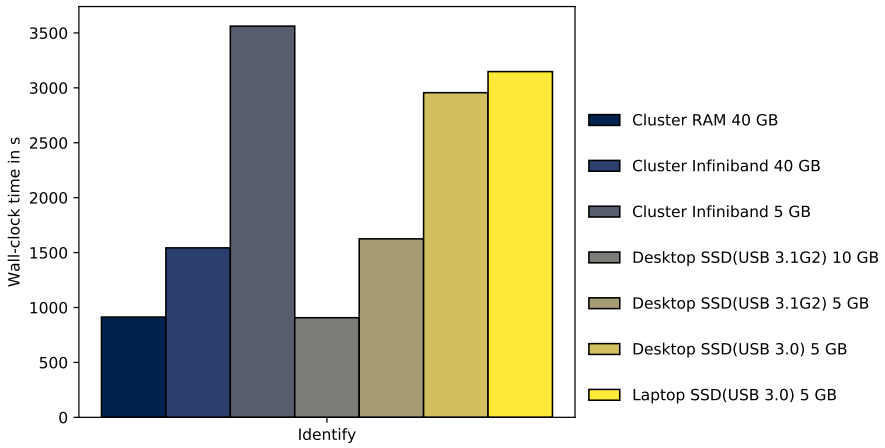

Figure 4.16: Results showing the wall-clock time while running the same test for `kASA` on different platforms and settings. The legend is structured as following: Platform, hardware on which the index lies and interface technology, amount of RAM given.

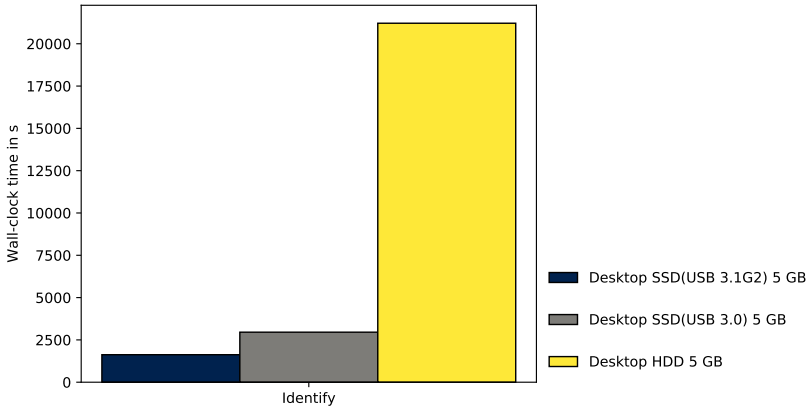

Figure 4.17: Wall-clock times on the same platform (Desktop PC) for SSD vs HDD. We do not recommend using the latter.

To see more clearly how `kASA` scales with increasing number of cores and amount of primary memory, we created Figure 4.18. We see that `kASA` does benefit from multi-core environments but that I/O restricts the parallelization potential. This non-parallelizable part of every program determines the minimum execution time which is called Amdahl's law [163]. In our case, increasing the number of cores has a non-linear decreasing effect. The value per core is still best when only using one core as for example you would need six cores to get the value of two. We conclude that even though most parts of `kASA` are parallelizable, there still is a rather large proportion of time spent on reading the input file and writing the output file which cannot be parallelized (see Section 3.4.5). If the bandwidth increases, this influence should decrease.
Regarding the use of more primary memory, `kASA` benefits more from multiple cores than from more primary memory. The lines in the middle of the figure decrease non-linearly but converge rather quickly. This is because the `C++ sort` algorithm has a linearithmic time complexity regarding the number of k-mers from the input but our set intersection algorithm has a linear time complexity regarding the number of k-mers from the input as well as the ones from the index. This means that sorting a small container in every iteration is faster than sorting one large container of k-mers in one go. However, calling the set intersection algorithm iteratively like this quickly mitigates that benefit due to the large latency while reading from the hard/solid state drive. We conclude that it is always preferable to keep the number of iterations that `identify` has to perform as small as possible and that providing more RAM is beneficial to this.

Finally, we want to prove our statement made in Section 2.2.1 that the execution time depends on the size of the range of k's. In Figure 4.19 we clearly see that the search space reduction of the trie (see Section 3.2.1) has a major effect (left side of the black line). We also see that in the algorithm of `identify` itself, the number of k's linearly affects the execution time (right side of the black line). We therefore consider this statement valid and choose the default value of seven for the lower k.

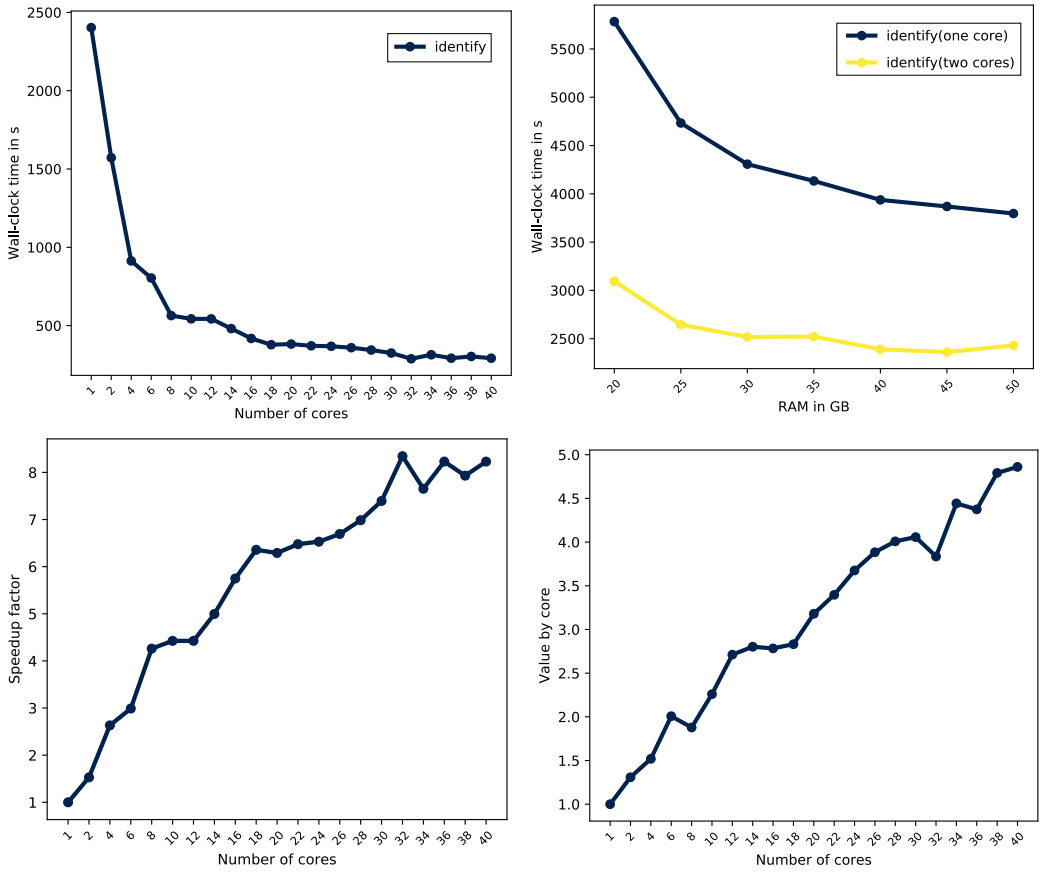

Figure 4.18: Wall-clock times for the same experiment but with different numbers of cores available for `kASA` every time (top left) or with more primary memory available (top right), as well as speedup (bottom left, higher is better) and value per core (bottom right, lower is better) calculated from the wall-clock times per core.

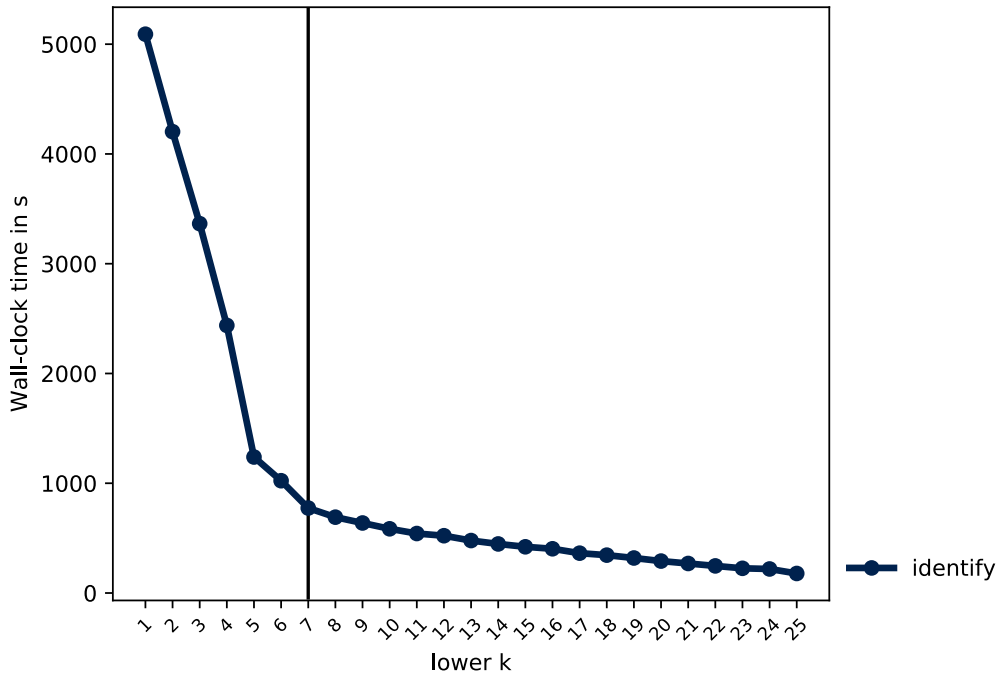

Figure 4.19: Wall-clock time of `identify` when used with the CAMI dataset on one fastq file and the 128 bit index for various lower k's. Starting with a lower k of seven, the trie fully reduced the search space in the index.

4.4 Real data

Because of the good results in our synthetic benchmarks, we set out to test `kASA` with real data where the ground truth is unknown. To nevertheless verify the output of `kASA` or at least put it in relation, software used in the previous benchmarks is also applied to the real data. Each presented experiment models a commonplace problem often encountered in metagenomics. The first experiment in Section 4.4.1 falls into the category of low presence detection, usually viruses. There, very sensitive matching is required to determine whether a virus is present in a sample and, if so, in what quantity. The second experiment in Section 4.4.2 is a classical analysis of the microbiota in a host (here: human). The goal is to detect present and related bacteria and/or viruses, so either the remaining sequences may be analyzed for novel organisms or the presence of a pathogen is detected. The latter case is usually done in a clinical study. Lastly, we use `kASA` in Section 4.4.3 on third-generation sequencing reads (generated with Nanopore [164]) to show that long reads pose no problem. Furthermore, we simulate a decontamination so that reads can be filtered out if necessary.

4.4.1 *Deformed wing virus* detection

The experiment shown here was part of a study conducted before `kASA` was written [165]. We were tasked with finding out whether the expected viruses *Deformed wing virus Type A (DWV-A)* and *Varroa destructor virus-1* also known as *Deformed wing virus Type B (DWV-B)* were present in pupae of *Apis mellifera*. Two data sets were generated by sequencing DNA with Illumina from crushed pupae infected with the viruses. These data sets were named M1 and M2 and contain sequences of *DWV-A* and *DWV-B*, respectively. We used the Bowtie2 mapper [166] to align the sequences to the host genome (*Apis mellifera*) and the viral genomes. The remaining reads were identified or discarded with megablast [90] using the nt [5, 62] database.

We re-evaluated these results later with `kASA` and Centrifuge [167] to verify our early findings. Centrifuge is used here because it was the second best performing program regarding sensitivity in our benchmarks with synthetic data (see Section 4.3.1 and Figure 4.5). Table 4.5 shows the relative frequency of mapped reads via Bowtie2, `kASA` and Centrifuge. We can see that the relative frequencies derived from the reads for `kASA` as well as for Centrifuge are close to those calculated with Bowtie2 for the M2 data set. It is also apparent that Centrifuge and Bowtie2 could not detect as many reads belonging to *Apis mellifera* as `kASA` did. Note, that we chose to use the frequency calculated from the read distribution to better compare it to Bowtie2.

Regarding the M1 data set, the difference in relative frequencies of `kASA` and Centrifuge to Bowtie2 is visible but not large. We looked at the reads that `kASA` and Centrifuge identified as belonging to *DWV-B*, since the difference to Bowtie2 was most obvious there for both programs. We found out that most of those reads were ambiguous with the "Top Hits" array often having two entries in case of `kASA`. There-

fore, they could not be solely identified as belonging to either *DWV-B* or *DWV-A*. In this case, each of the viruses got 0.5 to their read count before calculating the relative frequency via $\frac{\text{read count}(t)}{\text{number of reads}}$ with t being the taxonomic ID. We then used megablast [90] to see if the similarities were also visible there. These results showed that almost all of the reads were most likely from *DWV-A* with either exactly the same score as or a score close to *DWV-B*. Therefore `kASA` and Centrifuge could not differentiate between the two and thus have a higher relative frequency for *DWV-B* than Bowtie2.

Nevertheless, we were able to verify our results from 2016. For future experiments of this kind, `kASA` appears to be a good addition to or a substitute for Bowtie2.

Table 4.5: Relative frequencies of identified reads given in percent for Bowtie2, `kASA` and Centrifuge for both data sets.

M1	**Bowtie2**	`kASA`	**Centrifuge**
Apis mellifera	44.2	46.5	41.8
DWV-A	53.5	51.5	51.9
DWV-B	0.0032	1.83	0.87
M2			
Apis mellifera	74.5	77.5	68.9
DWV-A	0.03	0.029	0.051
DWV-B	22.2	22.4	22.1

4.4.2 Human microbiome project

The human microbiome has a large influence on the body [168] and thus determining the composition of microbes in a sample is vital for finding, e.g., pathogens. `kASA` offers exactly this via its profile, but because we have no ground truth in real world, we chose to use Kraken2 and Centrifuge to see if they can agree on the composition of known organisms. For our test, we downloaded existing open access data from the Human Microbiome Consortium [169], more precisely data sampled and sequenced from saliva (Sample ID: SRS147126, SRA ID: SRX154235). We then created indices for all three programs and created profiles in CAMI format [60]. As reference database, we used the RefSeq [170] from 2019-10-1. Table 4.6 shows that the programs mostly agree on the relative frequency for the genus level (except for Centrifuge which could not identify Porphyromonas with the expected frequency). Results of `kASA` are closer to those of Kraken2 as they are to those of Centrifuge. This is not surprising given that `kASA` and Kraken2 are both k-mer based methods. In addition, it is known that almost all detected genera with a high frequency occur in the human oral flora, which confirms our results.

Table 4.6: Selection of genera with the highest relative frequency (in %) found in data sampled from human saliva (SRS147126). For `kASA`, the non-unique relative frequencies for k=12 are shown. Below are the averaged absolute differences between the relative frequencies of the programs.

Scientific name	**kASA n.-u.**	**Kraken2**	**Centrifuge**
Prevotella	27.9	30.7	34.1
Streptococcus	16.5	13.5	17.8
Neisseria	15.2	11.5	9.1
Veillonella	7.7	7.7	6.9
Porphyromonas	6.6	6.8	0.25
Haemophilus	5.1	4.5	3.8
Campylobacter	1.6	1.4	1.1

Program names	**Average difference**
`kASA` vs Kraken2	0.0138
Kraken2 vs Centrifuge	0.0244
`kASA` vs Centrifuge	0.0293

4.4.3 Human genome assembly

This experiment used data from Jain et. al. [171] in which they sequenced the human genome from the GM12878 Utah/Ceph cell line (namely FAB42476) [172] with the Nanopore minION [164] technology. As reference database, we chose the RefSeq [170] from Section 4.4.2 as well. We set out to prove two things:

1. Even with strong similarities to other taxa (e.g., chimpanzee), it is possible to see the true host (human)
2. Contaminations with e.g. bacteria are visible in the profile

That the first point holds true can be seen in our multifaceted profile which presents among others, the unique and the overall relative frequency which help to separate true from false positives. For example, the difference in the unique relative frequency of *Homo sapiens*(0.29) vs *Pan troglodytes*(0.05) is clearly visible (see Table 4.7).

Table 4.7: Excerpt from the profile resulting from the experiment of `kASA` on data from the human cell line FAB42476 mixed with E. Coli. Only the columns for k=12 are shown due to the limited size of the page.

taxID	**Name**	**U. counts**	**U. rel. f.**	**N-u. counts**	**N-u. rel. f.**	**O. rel. f.**
9606	Homo sapiens	18270375	0.293796	2.78E+07	0.166454	0.0059195
9597	Pan paniscus	4481489	0.0720643	9.36E+06	0.0560577	0.00199355
9593	Gorilla gorilla	3648926	0.0586763	9.80E+06	0.0586692	0.00208642
9598	Pan troglodytes	3118585	0.0501482	8.66E+06	0.0518557	0.00184412
9601	Pongo abelii	2392786	0.038477	6.11E+06	0.0366143	0.00130209
61621	Rhinopithecus bieti	1804089	0.0290105	5.19E+06	0.0311037	0.00110612
9516	Cebus capucinus	1767579	0.0284234	4.99E+06	0.0298676	0.00106217
9541	Macaca fascicularis	1719200	0.0276455	6.28E+06	0.0376241	0.00133801
60711	Chlorocebus sabaeus	1651331	0.0265541	5.31E+06	0.0317964	0.00113076
9483	Callithrix jacchus	1614864	0.0259677	4.67E+06	0.027979	0.000995001
9545	Macaca nemestrina	1606200	0.0258284	4.74E+06	0.028377	0.00100916
9555	Papio anubis	1385434	0.0222784	5.63E+06	0.0337148	0.00119898
591936	Piliocolobus tephrosceles	1374825	0.0221078	4.31E+06	0.0258187	0.000918175
9531	Cercocebus atys	1299742	0.0209004	4.25E+06	0.0254758	0.00090598
9565	Theropithecus gelada	1207755	0.0194212	5.41E+06	0.0323893	0.00115184
9568	Mandrillus leucophaeus	1173074	0.0188635	4.16E+06	0.024895	0.000885325
37293	Aotus nancymaae	1113403	0.017904	3.47E+06	0.0207814	0.000739037
9544	Macaca mulatta	1068340	0.0171794	4.90E+06	0.0293279	0.00104297
27679	Saimiri boliviensis	1012049	0.0162742	3.31E+06	0.019793	0.000703886
562	Escherichia coli	23431	0.000376781	226570	0.00135674	4.82E-05

The second point can be proven by inserting, for example, bacterial reads from *Escherichia coli str. K-12 substr. MG1655* into the `.fastq` files, because a cell line usually has no contamination. We randomly generated 100 reads with length 1000 from this genome to ensure that the reads are unique. Both fastq files were concatenated to see if the corresponding amount will be visible in the profile (the identification per read is ignored here). With this setup, we expect to find at most 196000 k-mers (k is 12, forward and reverse complement are considered). With 4708900768 total k-mers and 62187376 unique k-mers in our dataset, we can calculate the expected unique and the expected overall relative frequency (see Table 4.8).

Table 4.8: Profiling results after the insertion of *E. coli* reads into the dataset for the unique and the overall relative frequency.

	Expected rel. freq.	Observed rel. freq.
Unique	0.0031	0.00037
Overall	0.000041	0.000048

After running `kASA` on the merged dataset, we inspected the generated profile and compared our observed values with the expected ones. We see that the values for the overall relative frequency are very close to each other. The unique relative frequency, however, differs greatly. This is not surprising given what we wrote in Section 3.5.3: Comprehensive databases suffer the disadvantage of species having many k-mers in common. With the usage of the RefSeq and the fact, that *E. Coli* and *Shigella* share many similarities [68], the uniqueness is reduced. We therefore conclude that `kASA` is capable of identifying contaminations in a dataset but admit that the profile will be hard to interpret if many related species overshadow these contaminations.

Because of this, we added the possibility to calculate the coverage for every taxon. This coverage is gained by counting every match during `identify` once for a taxon and dividing that count by the number of k-mers in the index for that taxon. The latter is gained from the frequency file (see Section 3.4.2). For this to be correct however, the input needs to be processed completely in one go. If it is processed in chunks, the counts will be distorted because two identical k-mers are counted twice or more and not once. The advantage is that the resulting profile could be sorted by this coverage and everything "high enough" (depending on the sequencing depth), be considered valid. This way, even small genomes with low k-mer counts but high coverage would be visible. Including the coverage in the profile is optional and can be activated via a parameter. For the experiment above, the coverage is too low to be noticeable. When inserting the complete genome of *E. coli* into the fastq file, the coverage is at 100% and $\approx$ 85% for *Shigella flexneri*. A Krona[173] visualization of our calculated profile can be seen in Figure 4.20.

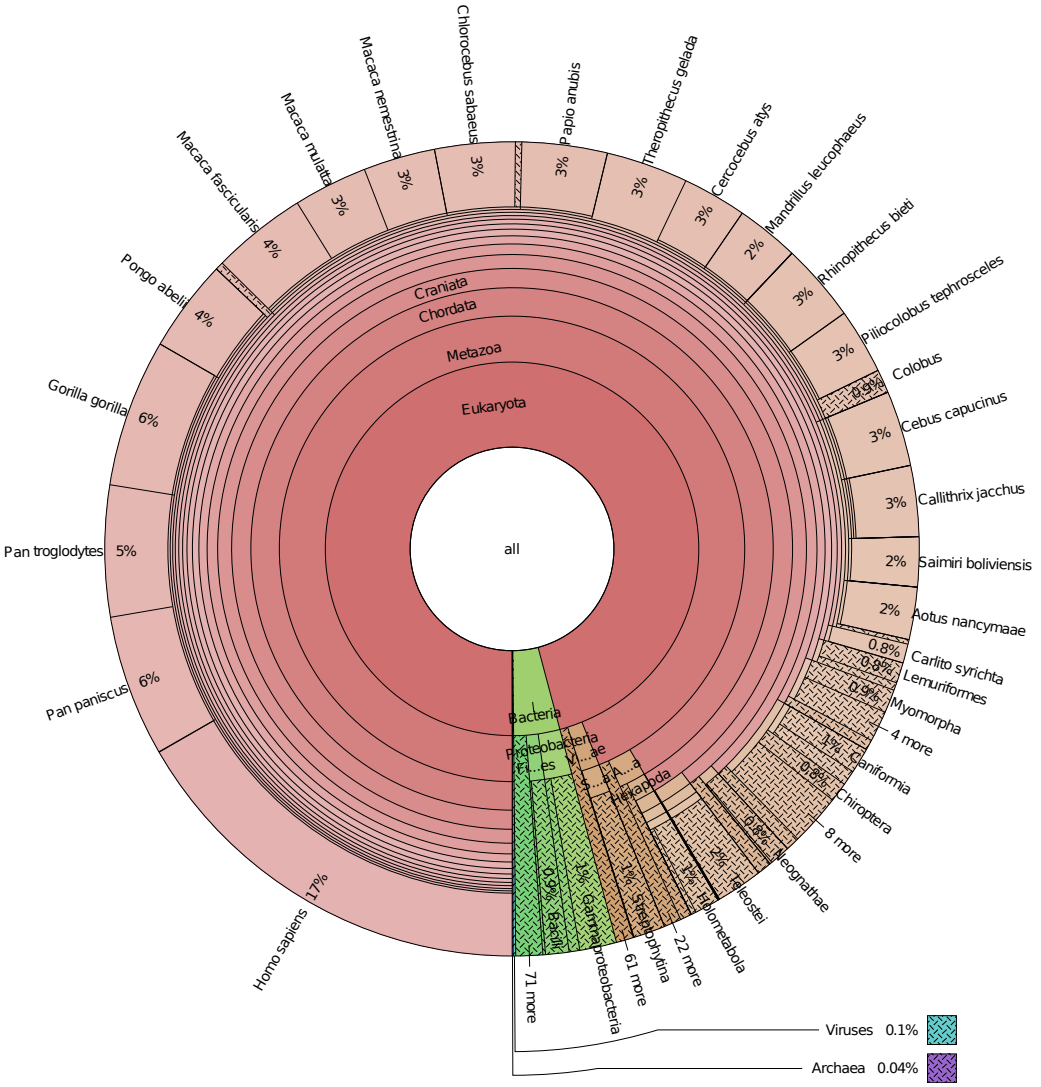

Figure 4.20: Visualization of our profiling result of the FAB42476 dataset together with *E. coli* insertions with Krona. Since species related to *Homo sapiens* were also reported, they are also visible in the pie chart. The green part representing Bacteria shows the Gammaproteobacteria to which *E. coli* belongs.

4.5 Discussion and summary

This chapter aimed at showing how `kASA` performs in various experiments. The first four experiments used data sets with known truths so that a systematic investigation of sensitivity, precision, and other measures of accuracy could be conducted. The last three experiments focused on real data to show that `kASA` can be used for actual studies as well. The performance of `kASA` was also evaluated and its technical limitations explored. For all experiments, existing relevant software was used as a comparison. We now discuss our results, a conclusion with regards to the theoretic ideas presented in Chapter 2 is given in the final Chapter 6.

The experiments with static data in Section 4.2 by McIntyre et. al. and Lindgreen et. al. made it possible to compare `kASA` with other software based on the same data. In both benchmarks, typical metagenomic data with pre-defined distributions was created. In the study of McIntyre et. al., the accuracy of the r-identification was benchmarked. With data from Lindgreen et. al., the quality of the profile gained from the r-identification output of `kASA` was tested. Both benchmarks showed that `kASA` performs very well in comparison to other software. We also noticed, judging from Figure 4.1, that it is advantageous for `kASA` to print out more than one taxon for every read. This is also reflected in the experiment from Lindgreen et. al. in which the calculation for the profile from the r-identification was made proportionally to the number of reported taxa for each read. This way multiple reported taxa got counted in the profile with their share of $\frac{1}{|\text{taxa}|}$. Figure 4.2 shows that this is a valid strategy. We therefore conclude that `kASA` is performing just as good or better than other tested software in those benchmarks. The ability to report multiple taxa in the r-identification is a useful feature that directly influences benchmark performance and helps users gain more knowledge about the data.

The major problem with the benchmarks of McIntyre et. al. and Lindgreen et. al. is that their data and necessary reference will become increasingly outdated over time. For the latter benchmark, we encountered this problem because some reference genomes have been updated or deleted since the benchmark was developed. This, together with the fact that more and more taxonomic profiling programs are being developed, makes it necessary to develop a benchmark that is able to generate its own data based on up-to-date references. We therefore created our own benchmark in Section 4.3.1 which is highly customizable and easy to use thanks to snakemake (see Section 4.1.1). Data is generated from reference genomes so the test files can be updated whenever the references change. The primary purpose of the benchmarks was to test the robustness of currently used software, in particular `kASA`, with respect to mutations and sequencing errors. One of the results was that `kASA` is more robust than all other tested programs. Furthermore, if three or six frames are used to create the reference, using one frame for the input sequences seems to suffice, as the robustness was still very high in this case (see Figures 4.4 and 4.12). One problem with `kASA` was that the specificity (see Figure 4.6) seems to be quite low and

thresholds on the relative score had to be applied to correct this (see Figure 4.7). This is a direct result of the fact that `kASA` reports every match and does not post-process the results inherently. However, we provide python scripts to do this afterwards if desired.

Another problem is that the choice of the optimal range of k's is not obvious. In Figure 4.9, the optimal lower k's were made visible because ground truth data were available. In general, however, it depends strongly on the data and the number of errors/mutations in the data. We must conclude that while we have chosen a reasonable default lower bound of seven k, which should be sufficient for most data, we cannot guarantee optimal performance in general. Should optimal behavior be desired, multiple runs with different lower and higher k's will be necessary in order to find the best range of k's. Judging from our experience however, the default ranges $[7, 12]$ or $[7, 25]$ perform well enough if no knowledge of the data is available. The problem of choosing the right k's can also be seen in the CAMI benchmark in Section 4.3.2. There, we tested how close the created taxonomic profile is to the real distribution of the taxa in the simulated data. In Figure 4.13 the same situation occurs: certain k's perform better than others. The difference however is, that here only one k is considered for the comparison. But even then, the standard pick of the largest k performs good enough as can be seen in Table 4.2.

All tested programs were also evaluated with regard to their performance in terms of wall-clock time and primary memory consumption, as well as disk space consumption. Section 4.3.3 showed that the wall-clock time of `kASA` is directly influenced by the number of frames used. Figure 4.14 shows that using the default three frames makes `kASA` the slowest program for large data sets. If one frame is used, wall-clock times for `kASA` range midfield next to Centrifuge and Kraken. If the index was built with three or six frames, the robustness stays high as described earlier. This is an advantage of `kASA` since Centrifuge was the only other program that combined high sensitivity with reasonable good performance. However, Centrifuge underperformed during profiling (see Table 4.2). This and the abrupt cutoff seen in Figure 4.5 make it a less attractive option in our opinion.
If performance is much more important than robustness, MetaCache and Kraken2 are valid choices since they were the fastest programs in the competition for both index building as well as identification. If, on the other hand, wall-clock time is irrelevant, choosing `kASA` with three or six translation frames for the input would be the best choice of all considered programs because it achieves the highest robustness.

In Figure 4.15, we can see that CLARK, Kraken and Centrifuge would not be able to create their index on a platform with only 8 GB RAM. This problem was countered by `kASA` with the usage of secondary memory for its index and the processing of the input in chunks which enables a user defined RAM usage. Using SSDs is an obvious optimization, since HDDs are not optimally suited for this task (see Figure 4.17). Especially external SSDs depend on the communication protocol speed, for example the USB standard (see Figure 4.16). Internal SSDs can nowadays be accessed via

PCIe [174] with bandwidths up to 7000 MB/s [175]. As both external and internal communication bandwidths increase, this bottleneck is eliminated eventually and only the clock speed of the CPU remains as the limiting factor.

At the end of the chapter, real data was used from three typical metagenomic studies. The first experiment was dedicated to detecting the frequency of viral genetic material in samples with a host species. The second dealt with identifying known taxa in a metagenomic sample and creating a taxonomic profile. The third and last experiment showed that third-generation sequencing techniques pose no problem for `kASA` and that filtering out contaminates is possible with it as well. In all experiments, the values of `kASA` were close to those of established programs. This functions as a validation since ground truths are not available for real data. The conclusion of these experiments is that `kASA` is excellently suited to handle real data and ready to be employed even in large field studies.

In the last two real data experiments, the non-unique relative frequency was especially useful and seemed to be close to the values of other software. As seen in Table 4.1 this makes sense when using a comprehensive database such as the RefSeq [170]. However, in Table 4.7 the unique relative frequency helps to see that *Homo sapiens* is the true host for this data. We did not present a data set with a large unknown fraction to motivate the overall relative frequency in this chapter but in Section 5.2 of the next chapter, the percentage of unidentified sequences was calculated by this frequency. This means that it depends on the study and its focus to choose the right type of relative frequency. The profile that `kASA` creates contains all of them nevertheless so that the user has all necessary information available. We see this as one major advantage in contrast to other software. Combined with the optional coverage described in Section 4.4.3, the taxonomic profile becomes a very useful output and summary of the examined data set.

Chapter 5

Further experiments and new insights

The experiments in Chapter 4 aim to empirically verify our ideas and methods chosen for `kASA`. Rather, this chapter is a collection of experiments conducted to explore the possibilities that `kASA` offers to answer some scientific questions and to gain new insights. It starts in Section 5.1 by checking if the codon table used in Section 2.2.2 for the translation from DNA to amino acid-like sequences has any influence on robustness. Section 5.2 shows that `kASA` can be used within a metagenomics pipeline and therefore replace existing programs. A wholly different experiment is shown in Section 5.3 where an often used method for increasing sensitivity is explored: Spaced k-mers [97]. The penultimate Section 5.4 compares `kASA` to Kaiju [102] as both can process protein sequences. Then lastly, Section 5.5 shows how strong the influence of the lossy shrinking method presented in Section 3.4.3 on the robustness really is. We would like the reader to know, that because this chapter has no real coherence, there will be no transitions from section to section as well as no summary at the end.

5.1 Influence of the codon table and its resistance to mutation

`kASA` has the capability of applying a user given translation alphabet in the NCBI format to the translation process. To test if the size of the alphabet matters more than the actual codon table, we experimented with multiple, randomly generated alphabets satisfying the requirements of `code` (see Section 2.2.2 and Equation 2.5). We chose alphabet sizes from 8 to 27. The lower bound of eight was chosen because 64 codons can be evenly mapped to eight letters with eight codons per letter. The only number below eight with such an even distribution of codons that can satisfy the requirements of `code`, is four but this produced nonsensical results. The upper

limit results from the fact that each letter has five bits available, which stand for an integer of at most 31 and `kASA` reserving four letters for internal use. For every alphabet size, a different conversion table was created by a Python script. The 8-letter alphabet and the 16-letter alphabet are special since they were designed by hand with the idea from Remark 6 in mind: If the first **and** second letter are identical for all codons that map to a letter from the alphabet, no information loss occurs when using six translation frames.

To simulate equal circumstances for all alphabets, we used the same benchmark as in Section 4.3.1. However, we only used mutation percentages of 0, 5, 10, and 15. We tested the sensitivity, precision, and resulting F1 score as well as the MCC described in Section 4.1.2. The results are presented as line plots to show the effects of different alphabet sizes, and as bar graphs to show how much the robustness is affected. `kASA_sta` represents the standard codon table 2.4.

Figure 5.1 shows that the sensitivity, precision, and the resulting F1 score profit from a larger alphabet, regardless of the conversion table. The plots showing the MCC paint a more clearer picture in which there is an optimum at 26 letters. The reason is a trade-off between robustness and conservation of information: More letters in the alphabet means that more triplets of DNA are uniquely identifiable which conserves information. In case of a mutation however, these triplets get unidentifiable or are falsely assigned. Since this depends on the frequency with which each triplet is included in the index, the equilibrium point depends on the data. Here, the sweet spot seems to be at 26 letters. In any case, we can conclude that the standard alphabet has no intrinsic property that makes it more powerful than a generic table with more available letters.

Regarding the special alphabet of 16 letters, Figure 5.2 shows that the precision is largely dependent on the number of available letters of the alphabet. As the simulated mutations increase, this observation only gets more obvious. Therefore, the information that could be recovered as described in Remark 6 has no measurable impact on robustness.

We therefore conclude that increasing the number of letters and thus the size of the alphabet for a translation table from nucleotide triplets to amino acid-like letters does influence the identification accuracy by increasing uniqueness but may decrease robustness. Decreasing the alphabet size leads to a clear loss of accuracy if less than 16 letters are used. There might be a mechanism to calculate translation tables which optimizes both aspects for a specific data set but this would require more sophisticated research and is thus future work. For now, we recommend using the standard codon table (see Table 2.4) if only for compatibility reasons.

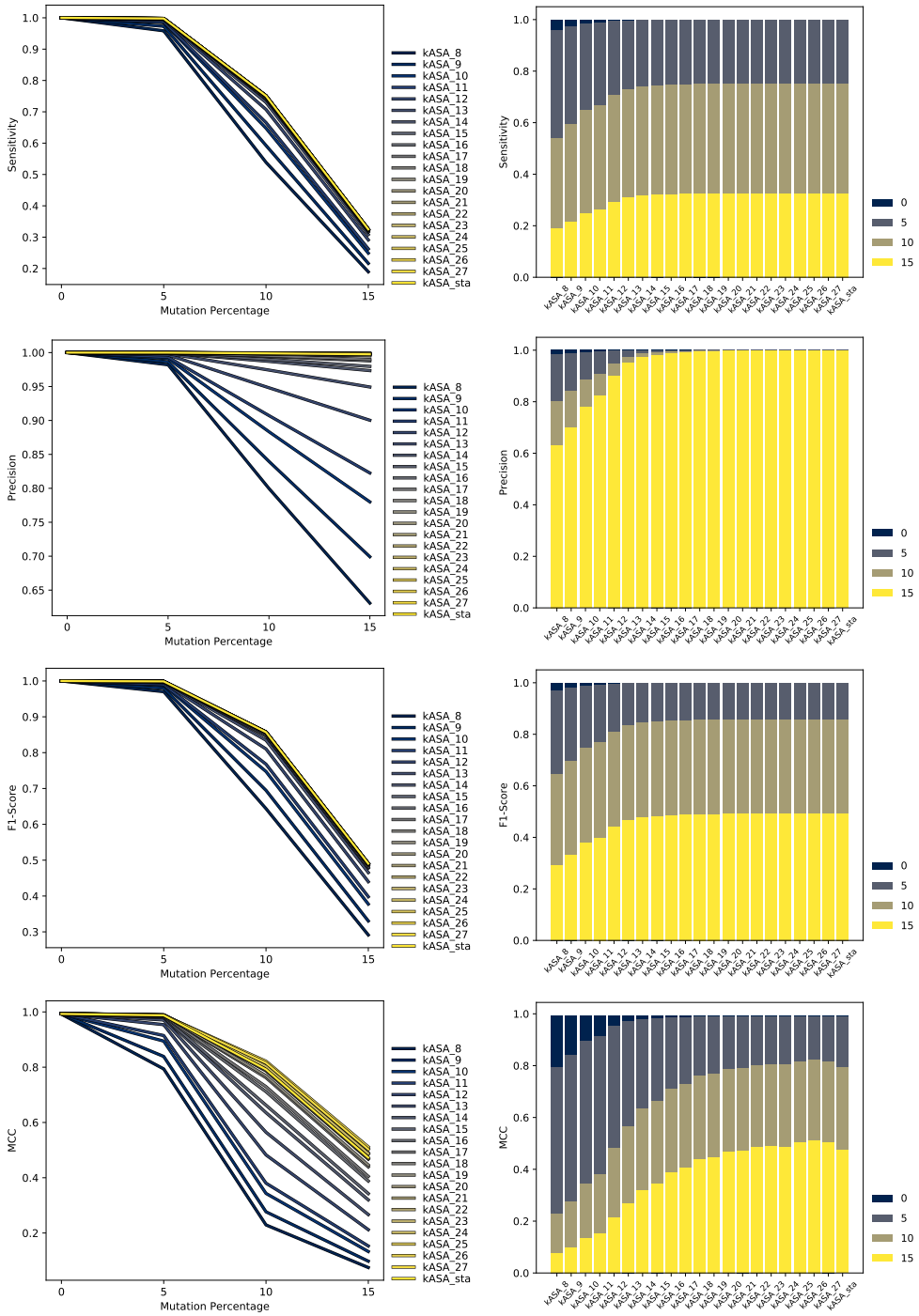

Figure 5.1: Results of the robustness benchmark for all tested alphabets displayed as line and bar plots.

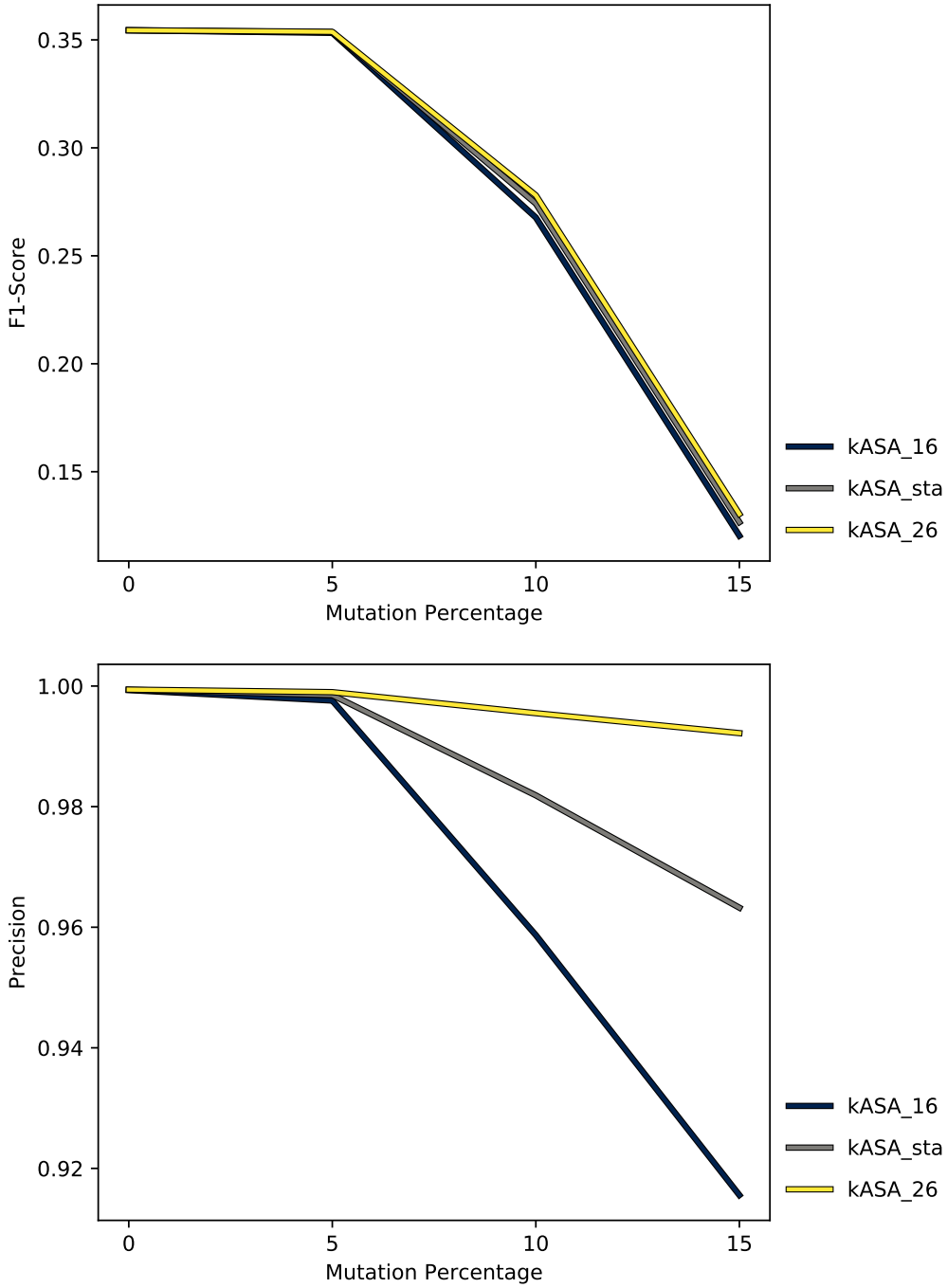

Figure 5.2: Benchmark from Section 4.3.2 with the 16-letter alphabet ("`kASA_16`") mentioned in Remark 6 versus the standard translation alphabet used by most organisms in nature ("`kASA_sta`") and the best performing 26-letter alphabet ("`kASA_26`"). Top: F1 score, bottom: Precision.

5.2 kASA as part of a metagenomics pipeline

In Section 1.1 the workflow of a typical metagenomic pipeline was described. We would like to reiterate that these pipelines usually have two different paths: One using assembly to generate longer strings of DNA from the reads and another using the reads directly. In both paths, taxonomic profiling is used to identify known taxa. The reader might now be suspecting a certain redundancy since the taxonomic profiling of the reads should be sufficient. However, the outputs of both paths differ: The k-mer/read frequencies of the raw reads give an indication on the abundance of the taxa, the r-identification of the contigs validates the found taxa. The latter is especially useful since not only the taxa which were truly sequenced but also related taxa could be given high scores (see Section 4.4.3). Another reason why both paths are used is that calculating the r-identification output for each raw read is usually computationally intensive, as the files are often several gigabytes in size. The profile is therefore much more useful in this case. On the other hand, after the assembly, most of the reads were assembled into one or more contigs. Without the information of how many reads contributed to the contig(s), an abundance analysis would certainly be biased. Therefore we recommend using the profiling output before and the r-identification output after an assembly. To show that `kASA` is able to do both, we included it in a metagenomics pipeline and evaluated the results.

The metagenomic analysis pipeline chosen was developed by the NASA GeneLab [176, 177] and is used to process data sampled from spaceflight experiments. There, the taxonomic analysis is performed with two programs: MetaPhlan [178] for the raw reads, and DIAMOND [95] for the assembled contigs. After an initial run to gather results for a comparison, we replaced both of them with `kASA` and analyzed exemplary data for differences and similarities. We would like to mention that the benefit of replacing these two taxonomic profiling programs is that it saves disk space (one instead of two indices) and installation effort. The paired-end data used was from the GLDS-249 dataset [38] (more specifically, the file "C57-6T_FCS_BSL_LAR_NxtaFlex_Rep1_B1"). This dataset was sampled with Illumina [3] from feces of mice (*Mus musculus*) which were in space flight. The database used for our index was the non-redundant protein database "nr" from the NCBI [62, 179] since it is used by other programs in the pipeline as well.

In total, about 74% of all k-mers could be identified. From those, 91% of all identified taxa are Bacteria, $\approx 8\%$ are Eukaryota and a small rest are Archaea $\approx 0.56\%$ or Viruses $\approx 0.11\%$ (see Figure 5.3 for a visualization). The largest share of Bacteria was made up of three phyla: Bacteroidetes (69%), Firmicutes (7%), and Proteobacteria (6%). These percentages roughly correspond to those also determined by MetaPhlan. This is not surprising given the fact that they make up the largest proportion of the gut microbiome of mice [180]. Adding to this, the mice were fed Nutrient Upgraded Rodent Food Bars (NuRFBs) which contain wheat flour, wheat gluten, and corn syrup [38, 181]. It has been shown that food high in carbohydrates

promotes growth of species from the Bacteroidetes phylum [182, 183]. This explains why Bacteroidetes make up the largest share.

Regarding the r-identification of the contigs, we present the top 10 species (with respect to the number of contigs) and their phylum in Table 5.1. This shows that not only was the host correctly identified, the profile also predicted the correct phyla. On closer inspection, however, we found it curious that none of species from the genus *Parabacteroides* present in the profile (see Figure 5.3) are in the top 10 or even the top 50 of the assembly output. We inspected the assembly report, the k-mer scores (see Equation 3.1) and the output of DIAMOND (which includes coverages) and saw that almost all reads originally belonging to these species were assembled into a few large contigs. This shows that calculating abundance based on assembled contigs should be done with caution.

We conclude that both, taxonomic profiling of the raw reads as well as the assembled contigs give valuable insight into the data. Using `kASA` for both analyses is not only possible but useful as well.

Table 5.1: Top 10 identified species from the contigs.

Tax ID	**Phylum**	**Name**	**Num. of contigs**
10090	Eukaryota	Mus musculus	9510
2026724	Chloroflexi	Chloroflexi bacterium	7641
83555	Chlamydiae	Chlamydia abortus	6405
2026735	Proteobacteria	Deltaproteobacteria bacterium	4819
1978231	Acidobacteria	Acidobacteria bacterium	2659
1913989	Proteobacteria	Gammaproteobacteria bacterium	2613
1496	Firmicutes	Clostridioides difficile	1712
1531	Firmicutes	Enterocloster clostridioformis	1588
1898104	Bacteroidetes	Bacteroidetes bacterium	1580
6335	Eukaryota	Trichinella nativa	1501

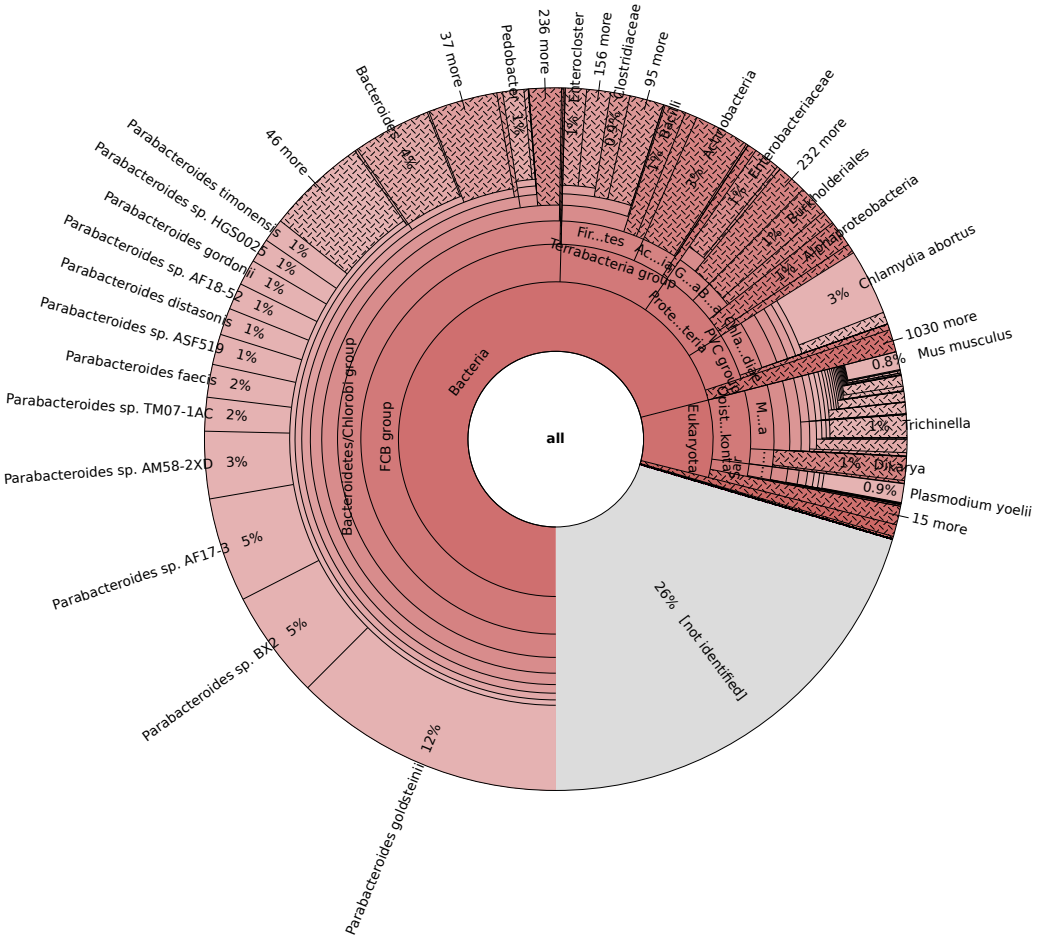

Figure 5.3: Krona [173] chart of the profile generated from the raw reads.

5.3 Usage of spaced *k*-mers

It has been shown that spaced words can increase sensitivity for seeded searches [96, 97]. The principle was mentioned in Section 2.1.1 and is as follows: A word S of length $L = |S| = r + i$ containing r relevant positions and i irrelevant positions is searched in an index of words. Only the relevant r positions need to match, the irrelevant i positions can match to everything. This breaks with the usual necessity of consecutive matches in exact matching algorithms, but the advantage is an implicit error management: If a letter in S would be a mismatch but is at one of the i positions, S is still considered a match. In the case of alignment software, this match could then be extended to a longer matching word that otherwise would not have been found. In the case of spaced k-mers, the number of relevant positions r is usually equal to k and $l = i + r$ is the new length of the l-mer [184]. An example would be:

$$\begin{aligned} \text{String from database: } & ABCDEFG \\ \text{Failing exact match: } & A\mathbf{C}CDE \\ \text{Spaced match: } & A * CD * FG \end{aligned}$$

In order to see if this approach works in `kASA` as well and how it performs in tandem with the dynamic-k (see Section 2.2.1) we also implemented an optional spaced k-mer approach. The first variant was to guess and apply a bit mask to the k-mers on DNA level and then translate these to the amino acid level. This way, any triplet containing an irrelevant position was translated to a dummy letter. However, this lead to significant information loss in case two DNA letters were not irrelevant. Adding a dummy letter for each of the seven cases avoided that problem but increased the alphabetical complexity and trie size (see Section 3.2.1).

We therefore opted for a different variant by applying a spaced mask to the amino acid-like encoded k-mers of the input during comparisons with those of the index in `identify` (see Section 3.4.5). In total, we created nine spaced masks. The largest one for example looks like this: 111111101101010010101011001. A "1" indicates a relevant, a "0" an irrelevant position. The first seven letters contain no zeroes because the trie needs an exact matching to fully decrease the search space. To work in tandem with the dynamic k approach, this spaced mask is used with a higher k of 25. There are nine zeroes in this mask so it corresponds to a 16-mer without spaces. That means that in order to compare the performance of the spaced approach to the default version of `kASA`, it is necessary to fixate the higher k's in both cases. If we set the higher k of the default version to $k_H = r$ then the spaced mode with this mask uses a higher l of $l_H = r + i$. The lower k is the same in both instances.

The downside of this approach is that if a k-mer truly matches S in all positions,

even the irrelevant ones, there is no benefit in using this approach. On the contrary, it could even add false positive matches that would not have matched without spaces. This can be seen in the publication of CLARK-S [184] Supplementary Table 2, where the precision of CLARK-S is lower than that of CLARK in almost all cases. To verify this for `kASA` empirically as well, we used the robustness benchmark pipeline from Section 4.3.1. Because we need, among others, a higher k_H of 25, the read length is increased to 1000 bases and the number of mutations in every read to 0, 160, 200, 240, 280 and 320.

In Figure 5.4 the sensitivity, precision, and F1 score (see Section 4.1.2) were calculated for all spaced masks and mutations. The sensitivity really is at times slightly better than the contiguous version, but the precision is worse for all spaced masks. This means that the scores for the taxa are inflated for the spaced versions, leading to a higher number of misidentifications of the taxa in every read.

In conclusion, using the spaced version for `kASA` only makes sense if the data is very noisy and the number of possible false positives is not of importance. There is no difference regarding performance and the same index is used in both cases.

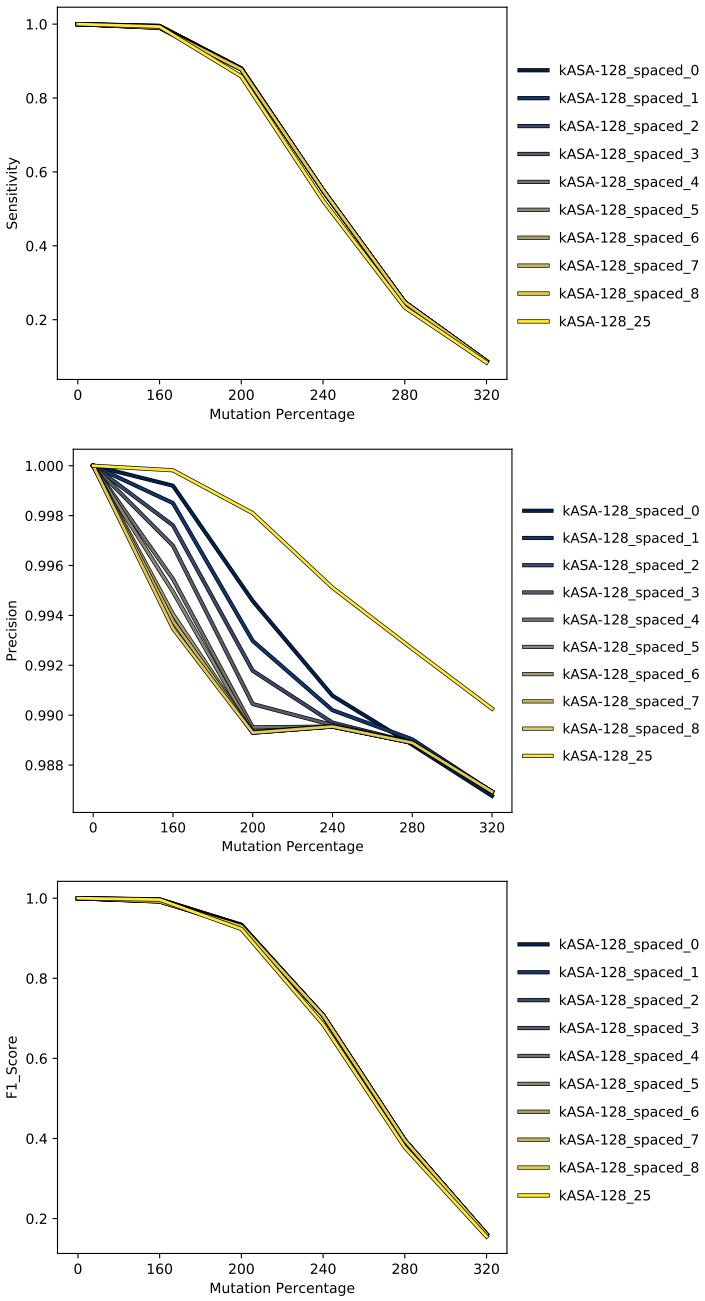

Figure 5.4: Sensitivity (top), precision (middle) and F1 score (bottom) of `kASA` for the nine spaced masks as well as for the fixed higher k of 25. Lower k for all runs was seven (default). Note, that the y-axis of the figures start at different values.

5.4 Kaiju vs kASA

Both the robustness and the CAMI benchmark in Sections 4.3.1 and 4.3.2 compare kASA only to software working on DNA level. Since kASA is also able to take proteins as input for both building the index as well as identification, we considered it interesting for the reader to include a benchmark working solely on protein level. As comparison, we used Kaiju [102] which we described in Section 2.1.2. As database we used the already translated genomes included in the "Fungi" pre-selected database from the website of Kaiju [185]. The translated genomes served as input for the index and from the genomes themselves, DNA sequences were generated the same way as in the robustness benchmark (albeit with fewer mutation percentages, see Section 4.3.1). Since the authors Menzel et. al. wrote that Kaiju benefits from longer reads, we sampled 200 base pairs (bps) instead of 100 bps randomly from the genomes. We let kASA use six translation frames for the input since the index consisted of already translated sequences without any indication of direction. Because most genomes contain non-coding regions, we do not expect both programs to be able to identify every read as these parts are not included in the database. Furthermore, Kaiju only uses open reading frames which are parts of the DNA that begin and end with specific codons [186]. In contrast, kASA translates everything including non-coding parts. This is something we think is the biggest difference between the two programs aside from their identification method.

Some technical details and settings (platform was HPCC [161]): Kaiju version 1.7.4, 5 cores used during build, 5 GB RAM used (30 GB given) for build, 2.8 GB resulting index size, $\approx$ 5 min needed for build, 8 cores used during identification, about 3 GB used for identification, about 7 min in total used for identification. kASA version 1.4, 5 cores during build, 19 GB RAM used (30 GB given), 18 GB resulting index size, 6 min needed for build, "species" as taxonomic level used, 8 cores given for identification, 28 GB RAM used (30 GB given, index loaded into RAM), `identify_multiple` used and about 13 min needed.

Results are visible in Figure 5.5 which shows that by default, kASA has a higher sensitivity as well as a better robustness than Kaiju. Precision for kASA was lower because very low scoring hits were also reported. True positives usually had k-mer scores of more than 15, false positives less than 1. We could therefore apply a threshold and improve precision. Everything with a k-mer score (see Equation 3.1) lower than 1.0 was discarded. However, this threshold would only apply to this data and with higher mutation percentages it starts to influence true positives as well. We are nevertheless confident that any user examining the r-identification output would be able to distinguish between a true hit and one with short, low scoring similarities. In kASA, such threshold can be given via a command line parameter.

We conclude that thresholds can be important in differentiating between short similarities and long, high scoring matches. After applying a threshold, kASA had not only a better precision than Kaiju but still a better sensitivity. Figure 5.5 also showed

how much data could not be identified because only protein coding segments of the genomes were used as reference. We therefore believe that our approach of also using non-coding parts of the genome is well justified.

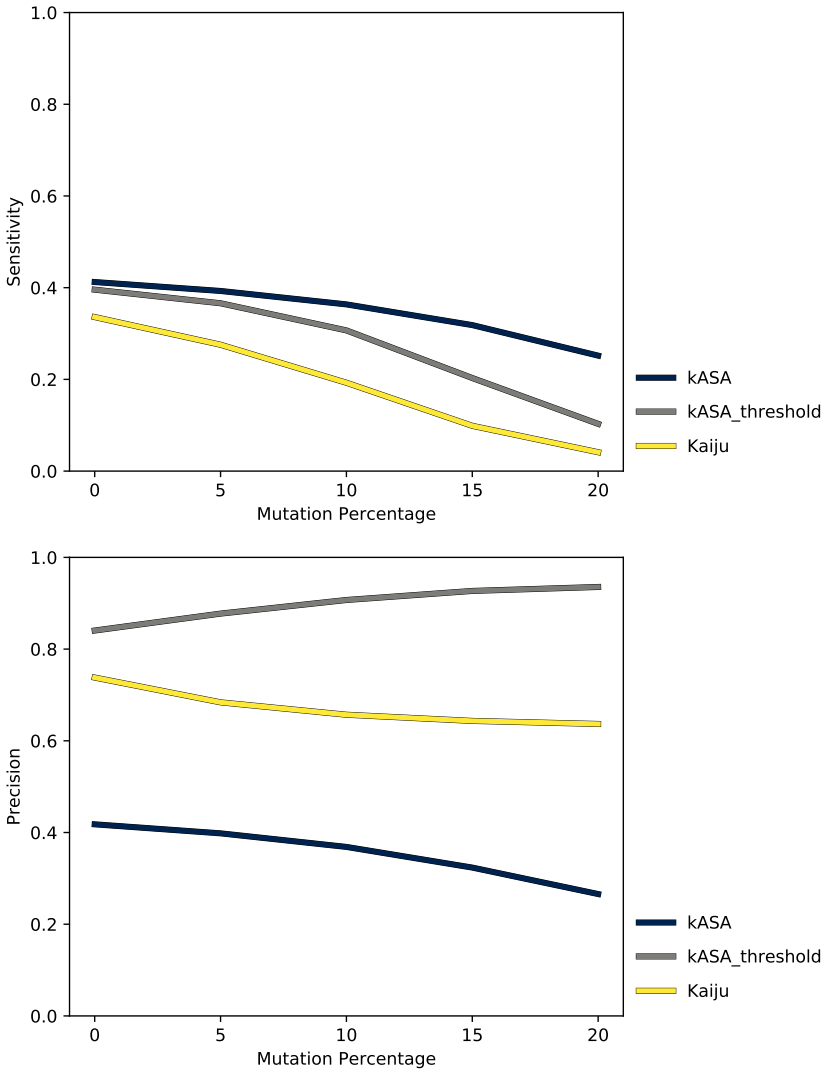

Figure 5.5: Sensitivity (top) and precision (bottom) of Kaiju and `kASA` with and without a threshold of 1.0 on the k-mer score.

5.5 Influence of `shrink` on sensitivity

In Section 3.4.3 part two, we described a way to reduce the index size by removing k-mers from the index via a calculated or given percentage. It is obvious that this influences sensitivity but the effect might be negligible when considering the gained disk space and identification speed (by reducing the amount of data that need to be streamed through the I/O). We took the robustness benchmark from Section 4.3.1 to see how the accuracy and robustness would be influenced when certain percentages of the index would be removed. Figure 5.6 shows the effect on the robustness should a given percentage be kept (aside from that, default settings were used). We see that the impact is measurable but not significant except for 10% and 20%. This conclusion was used for the real data experiments in Section 4.4.2 were we shrunk the index by about 60% first and then halved it with the lossless method. We therefore recommend using this method should index size matter. Together with the information of the redundancy from Section 3.5.3, we see this as a useful feature and addition to `kASA`.

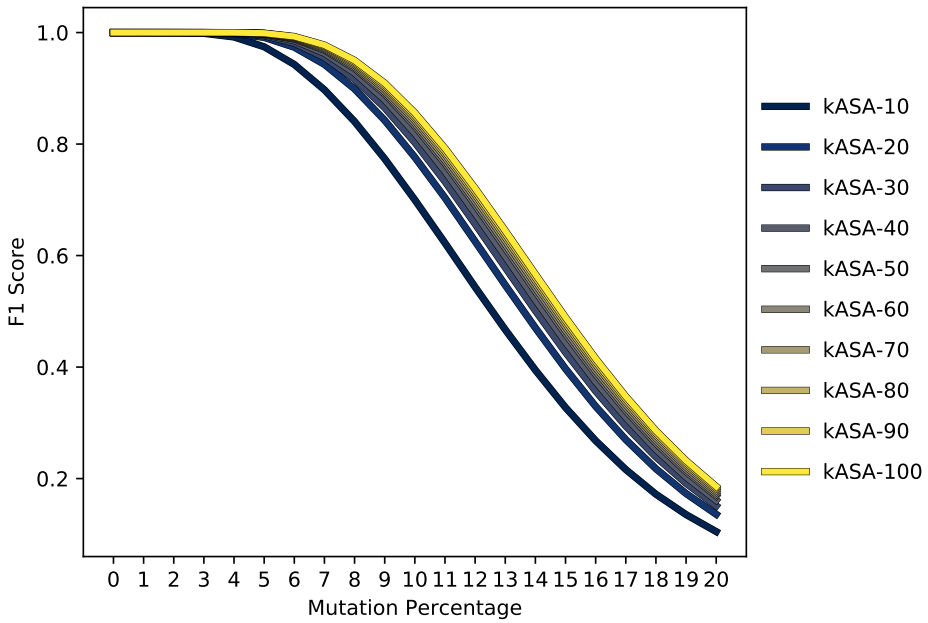

Figure 5.6: Robustness benchmark for shrunken indices via a given percentage. kASA-100 is the full index, kASA-90 is only 90% of the full index, and so on.

Chapter 6

Conclusions and future work

This chapter summarizes all results, theoretical and experimental, and answers the question whether `kASA` truly advances the field of taxonomic profiling and metagenomics. Afterwards, some ideas are presented which did not work out. We also give an outlook on what features are planned for `kASA` and what we might study in the near future.

6.1 Conclusions

In Chapter 2 three major new ideas were introduced that aimed to make `kASA` an outstanding software for analyzing metagenomics data. The first was the usage of a range of k's, the "dynamic k", which increases sensitivity while preserving precision as shown in Figures 4.5 and 4.8. The second idea was the translation of the genomic data, even the non-coding parts, from the database as well as new sequences with an "amino acid-like encoding" in multiple frame which increases robustness as well, as shown with Figures 4.10, 4.11 and 4.12. Especially the translation of the non-coding parts proves to be advantageous, which was covered in Section 5.4. The third idea was the restriction of primary memory. This was made possible by first holding the index in secondary memory and then processing the input in blocks. The impact on the performance was shown in Figure 4.16 and is significant. However, usage of high bandwidth communication protocols and hardware (SSDs) alleviates this bottleneck. The following paragraph explains how well `kASA` performs the task of taxonomic profiling and elaborates further on our new ideas.

As we have seen in Chapter 4, `kASA` is excellently suited to correctly identify and profile genomic or protein data. Both of our created snakemake benchmarks presented in Sections 4.3.1 and 4.3.2 show that `kASA` has not only a higher robustness but also a very good accuracy in contrast to most of the existing tested software from which we drew our inspiration for `kASA`. When using data from other benchmarks like in Sections 4.2.1 and 4.2.2 we were also able to outperform or draw level

with the existing software. Furthermore, the real data tests in Section 4.4 showed that `kASA` can calculate profiles which are very similar to those from Kraken2 or Centrifuge. This further proves that our methods work, even if the true distribution of taxa is unknown which is usually the case. We can therefore conclude that `kASA` is able to fulfill the task of taxonomic profiling of metagenomic data.

To see if our methods (dynamic k and amino acid-like encoding) would work not only together but by themselves, we added the experiments shown in Figures 4.8 and 4.10. It became clear that the strategy of using a dynamic k has a positive effect on the sensitivity without harming precision. Furthermore, enlarging the interval of k's has only a linear effect on the execution time as long as the lowest k is larger than six. This was predicted in Section 2.2.1 and seen in Figure 4.19. We can therefore only reiterate our statement and say that the additional execution time proves worthwhile for increasing accuracy.

Regarding the amino acid-like encoding, we see that it does increase robustness against mutations. As mentioned in Section 2.2.2, we lose information during translation and sorting. We also see in Figure 5.2 that the alphabet itself does not have a major impact. It is therefore questionable if the inherent balance of robustness and uniqueness of the standard alphabet as seen in Figure 5.1 is really an advantage in itself. What does influence robustness visibly is the usage of more than one translation frame, as can be seen in Figures 4.4 and 4.12. We conclude that the translation as a whole positively influences robustness against mutation as at least one k-mer in three or six frames is unlikely to be affected by a mutation.

From a technical viewpoint, the restrictability regarding primary memory is the major feature of `kASA`. The strategy described in Section 2.2.3 enables our program to perform taxonomic profiling even on mainstream hardware with a parameter restricting the amount of primary memory. It was also designed to use all given and available primary memory, as can be seen in Figure 4.15. Furthermore, we see in Figures 4.16 and 4.18 that both the amount of primary memory and the throughput influence the execution time so an increase in both benefits the program up to a certain point of saturation. Providing more cores reduces the execution time, but with an increasing number of cores, the value per core reduces significantly. We do not see that as an issue since `kASA` was primarily designed for systems with a small number of cores and usually, more than one file must be analyzed. This can be done with the `identify_multiple` mode (see Section 3.5.1) which uses the available resources as best as possible and therefore mitigates some of these effects. In conclusion, we made it possible for `kASA` to run on any system which offers at least 5 GB of RAM without loss of accuracy as long as enough secondary memory is available. It benefits from systems providing more resources and is therefore suited for most of the targeted user base.

This versatility is also reflected by the number of adjustable parameters for `kASA` which is quite high. Almost all of them have default values, but some depend on the

study and user preference. Therefore, a wiki and a comprehensive README file with various tips can be found on our GitHub page [10]. The three major parameters that need some attention are the interval of k's, the number of gigabytes for the primary memory usage and the number of frames. Depending on the lengths of the reads, a higher k of 25 my be preferred (default is 12). The number of gigabytes for primary memory usage that makes the most sense is usually system dependent so it is up to the user to configure this. Finally, the decision on the number of frames is a decision between speed and space requirements. Using six translation frames for `build` results in a larger index than using only three or one frame would have. Should six or three frames be used for the index, using only one frame for `identify` harms robustness only slightly (see Figure 4.12) thus increasing speed. However using only one frame for `build` and three or six for `identify` saves space but increases wall-clock time. What makes more sense depends on how often the index is used and how important robustness is. We argue that performance is not as important as quality and therefore recommend using three or six frames in both `build` and `identify`.

Another aspect is how representative our experiments are for the real world. The experiments with real data have been added to give the reader and user an insight into what studies `kASA` can be used for. Usually, metagenomic studies are undertaken as described in Section 1.1. Looking at the assembly path, decontamination may be necessary. The experiment in Section 4.4.3 shows how this can be done via filtering out reads based on the error score (see Equation 3.2). Analyzing the resulting assembled contigs via `kASA`'s r-identification output can be done like in Section 5.2. If the other path of directly calculating the taxonomic profile is chosen, our experiment in Section 4.4.2 shows what findings can be obtained and how it may be visualized. Therefore, our various experiments give a good overview of what real studies look like and how `kASA` can contribute to new studies.

What must be mentioned however, is that there may be some instances where `kASA` is not the best choice. Especially if enough primary memory is available, the lengths of the reads are small (< 100 bp) and/or the number of sequencing errors or mutations is close to zero. In these cases, methods using hash tables (see Section 2.1.3) are faster and still have a good sensitivity (see Figures 4.4 and 4.14). Another instance would be if the hardware to support high throughput is not available (SSDs for example) because `kASA` needs a high bandwidth for access to secondary memory (see Figures 4.16 and 4.17).

We also encountered some issues which must be mentioned as well. One major problem is the number of random matches from taxa which share some similarity with a sequence of amino acid-like encoded k-mers. Even though most of the time they can be discarded with a threshold, very noisy data might lower the highest possible score so that taxonomic assignments of reads become ambiguous. We would like to mention that such matches are not false matches or errors in the judgment of our software. `kASA` simply reports every similarity it encounters starting from the

lowest k. Because every living matter on earth is related to one another as far as we know [50], we are bound to find such similarities. However, we do admit, that due to the translation from DNA to amino acid-like strings, we inherently introduce the possibility of matches on that level, which would not be possible on the DNA level. That means that if the user wants to make sure if that assignment is correct, they can run the experiment again on a database containing, for example, the genes of the suspected taxon. Should the read still have the suspected taxon assigned to it, it is unlikely to be a random match. Note, that we do not advise the user to naively only use databases consisting of protein sequences because there are non-coding regions that are highly conserved as well (see our experiment in Section 5.4).

Another issue was the size of the indices we had to create for our experiments. Especially for the real world tests (see Section 4.4), the indices generated from the complete RefSeq [170] or even the bacterial database soon scaled even beyond the available secondary memory. In Section 3.4.3 we presented three methods of reducing the size of those indices, the second method (see here: 3.4.3) can even be applied during building. However, that implies a loss of k-mers and thus a decrease of accuracy. The user could use only one translation frame for the index and then six for every input file but this increases the execution time significantly. However, given the fact that in Figure 5.6 we see that the accuracy is not influenced unless a large percentage of k-mers is lost and that the price of every gigabyte continues to sink, we are confident that this problem can be resolved.

6.2 Goals reached

We would also like to reiterate our Goals from Section 1.2.2 and how we achieved them:

Goal 1 stated that `kASA` must be able to support even large numbers of genomes as reference. Due to our memory restriction presented in Section 2.2.3, most of the reference lies on secondary memory so we consider this goal achieved.

Adding to this, Goal 2 required our software to be able to add or delete genomes from our index. This is supported by the modules `update` and `delete` described in Section 3.4.4.

Goal 3 stems from the necessity of supporting all sequence lengths and current technologies. The first part was achieved by choosing a k-mer based approach which, due to its fixed size, only creates more k-mers the longer the sequence gets. We also implemented support for paired-end input [70], and therefore fully achieve this goal.

We presented two new methods “dynamic k” and “amino acid-like encoding” in Sections 2.2.1 and 2.2.2, and tested them in Section 4.3.1. Our conclusion from this, is that the robustness of `kASA` is high enough to achieve Goal 4. It stated that `kASA` must be robust to nucleotide changes in either the reference or sequence.

From a technical point of view, Goal 5 says that we should try to simplify the use of our software as much as possible. This includes availability on most platforms and operating systems. By employing various implementation techniques and compatibility checks mentioned in Section 3.1, we are as close to achieving this goal as possible and meaningful. We would like to also support the ARM64 architecture [187] in the near future.

The final Goal 6 stated in Chapter 1.2.2 said that `kASA` should:

1. correctly and efficiently identify any valid sequence belonging to a known species
2. not use more memory than given by the user
3. provide easy access as well as a variety of options for any user
4. be as independent of platform and other software as possible
5. provide significant scientific value for metagenomic and thematically related studies

We proved the first and second point to be the case in our experiments in Chapter 4 and described in Chapter 3 how we achieved the fourth requirement. For the third point, we provide several methods to balance wall-clock times, space usage and memory consumption. These are also explained on our GitHub page [10]. If users need a smaller index, they can reduce the index either with or without information loss (if the requirements are met) and use one or three frames instead of six during creation. This may reduce accuracy. If wall-clock time is critical, users can use more resources or reduce the number of frames and load the index into primary memory. This uses more RAM and leads to greater consumption of both primary and secondary memory. Finally, if primary memory is limited, users can set a limit that `kASA` does not exceed and the index can be moved to an SSD, for example, which slightly increases execution time.

Consequently, we find the fifth point to be true and with `kASA` virtually every life scientist with a laptop should now have the technical means to taxonomically profile metagenomic data. Hence, we fully described the development and use of the software `kASA`.

6.3 Failed ideas

While developing `kASA`, we also tried ideas which did not work out in the end. We now present some of them, give context, and then explain why they did not work out.

- Using a hash table instead of a trie
  As can be seen in Figure 4.19, using a k lower than seven impacts the execution

time quite severely. We therefore set the default lower k to seven. This could lead to users only choosing the default lower k. However, one benefit of the trie is, that is does not need to be fully traversed to get the range of the index. If this is not used, we could simply use a hash table of fixed length 6-mers instead. When we did that however, we found out that speed gain was marginal but primary memory usage increased by a large degree. It worsened by using a lookup table albeit it was a bit faster. At the end, using the trie for prefix matches was never the bottleneck of performance for `kASA` therefore optimizing this step ultimately seemed pointless.

- Going read-by-read instead of gathering data in a container
  This can be simulated by lowering the memory restriction so that only one read can be processed at a time. As can be seen in Figure 4.16 there is no benefit in doing that. One could argue that this would be much more parallelizable and we agree. However, the cache plays a major role for secondary memory access and changing pages every time due to random accesses converging to random noise is doomed to fail.

- Reducing the alphabet to increase k by using less than five bits per letter
  We actually tried to further translate the amino acid-like coded words into even smaller k-mers of length six to reduce the index to the trie entirely. This resulted in the F1 score dropping to 0.5 in the robustness benchmark even when no mutations occurred. It is therefore not suitable for our purpose. One could use the 16 letter alphabet and save one bit but this would destroy the compatibility with protein sequences.

- Using external sorting with `stxxl::sort`
  In our early development phase, we thought it best to convert both the reference database as well as the input to external containers and then stream both to keep below the memory restriction. This implied sorting the externally stored input with the sorting algorithm of the STXXL. Since this is not in-place due to the implementation being a merge sort algorithm, a file of the same size as the input was created. Not only did this take very long it was just not possible with large files due to our secondary memory running out after a while.

- Manual buffering of the index
  To counterbalance the bottleneck of streaming the index, we tried preloading the range given by the trie before comparing with the input. It turns out that most of the time, only a small percentage of that range really mattered and so we loaded much more than necessary into memory.

- Implementing a galloping search [188] instead of using binary search
  When fetching the range inside the index from the trie and searching for the first matching k-mer, a binary search would fetch the start and the end before halving the range. This could force the pager to load a page near the end

of the range even though it may never be used. We therefore thought that beginning at the start and using a galloping search would be more sequential than random access and thus benefit the streaming of the index. We found out that the galloping search needed more comparisons and took longer than just using binary search.

- Deleting k-mers with the BLOSUM matrix [91]
  The BLOSUM matrix can be used to score how close a protein sequence is to another, not-identical one. This means that if, for example, only one amino acid is a mismatch and the matrix shows that this substitution occurs often, it can still count as a match. We tried using the Levenshtein distance [189] together with the BLOSUM80 matrix, deleting all similar k-mers and thus only using discriminative k-mers like CLARK or removing all k-mers sharing the same prefix/suffix. But ultimately, none of these approaches either produced acceptable results or took too long to be considered useful.

- Updating the index in-place
  The downside of our algorithm used in `update` and `merge` is that it creates a new file instead of modifying the existing one. However, adding an element into an existing array is of linear time complexity in the length of the array. This means that the large external index would need to be shifted every time an element is added into its position. This is obviously not viable. There are also no implementations of an external list and with `stxxl::vector` being the only persistent data structure available, we abandoned this idea.

- Compressing the index
  When we designed the format of our index, we knew that the redundancy of saving identical k-mers with only a difference in the taxonomic ID would needlessly increase the size. Since the `stxxl::vector` class needs a fixed bit size beforehand, this space would be spend regardless so we considered it to be a necessary evil. Later, we tried to compress the index by packing k-mers and the respective taxonomic IDs next to each other. We used a run length encoding so that the first number would be the number of taxonomic IDs following the k-mer. This way we knew how many bits would belong to one entry and could save the entries next to each other. A visualization would be a text with line breaks. This compressed the index and saved about 20% but increased build time (as the large index would need to be built first anyway) and identification time as the decompression could only be applied sequentially. Saving space this way ultimately seemed not worth the effort and time, so it was scrapped.

6.4 Future work

In Section 5.1 we mentioned that there might be a codon table together with an alphabet, other than the standard one, that optimizes robustness and precision. Finding such a table and alphabet is not trivial however as the search space is very large. Sophisticated optimization techniques could be employed but creating a model is by no means straightforward. Our software together with our snakemake benchmarks offer a testing platform however. Therefore, if interest in further exploring this open question from the community rises, it would be possible to do so.

Inspired by software using alignments (see Section 2.1.1), we also saw potential in saving genome positions along with the k-mers generated from reference genomes. This would make it possible to distinguish random matches from more meaningful overlapping matches of k-mers. It would only work for a small number of reference genomes and require post-processing of the matches but might be worthwhile for increasing confidence in a reported taxon for a read.

After `kASA` was published [9], a proposal has been made on GitHub to support real-time streamed sequencing data that can be output with MinION [164]. Since `kASA` runs on most laptops, this combination seems quite useful for researchers who are frequently on the road (e.g., on a ship).

Lastly, using `kASA` not only for metagenomics but also metatranscriptomics is an obvious generalization. Using a protein database and the profile for gene calling and quantification seems to be a straightforward way of extending studies aiming for both research topics.

We sum up by saying that we are confident that `kASA` will stay relevant for the foreseeable future. It is currently used in multiple collaboration projects either for replacing existing software in pipelines or as a standalone to study for example the link between *Human gammaherpesvirus 4* and multiple sclerosis. Due to the time period in which these projects are being conducted, none have been published yet, but we are actively working to give `kASA` the attention it deserves.

Bibliography

[1] J. D. WATSON and F. H. C. CRICK. *Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid*. Nature **171** (Apr. 1953), 737–738. DOI: 10.1038/171737a0. URL: https://doi.org/10.1038/171737a0.

[2] C. E. Bruder, A. Piotrowski, A. A. Gijsbers, R. Andersson, S. Erickson, T. Diaz de Ståhl, U. Menzel, J. Sandgren, D. von Tell, A. Poplawski, M. Crowley, C. Crasto, E. C. Partridge, H. Tiwari, D. B. Allison, J. Komorowski, G.-J. B. van Ommen, D. I. Boomsma, N. L. Pedersen, J. T. den Dunnen, K. Wirdefeldt, and J. P. Dumanski. *Phenotypically Concordant and Discordant Monozygotic Twins Display Different DNA Copy-Number-Variation Profiles*. The American Journal of Human Genetics **82** (Mar. 2008), 763–771. DOI: 10.1016/j.ajhg.2007.12.011. URL: https://doi.org/10.1016/j.ajhg.2007.12.011.

[3] B. Adewale. *Will long-read sequencing technologies replace short-read sequencing technologies in the next 10 years?* African Journal of Laboratory Medicine **9** (2020), 5. DOI: 10.4102/ajlm.v9i1.1340. URL: https://ajlmonline.org/index.php/ajlm/article/view/1340.

[4] E. S. Lander et al. *Initial sequencing and analysis of the human genome*. Nature **409** (Feb. 2001), 860–921. DOI: 10.1038/35057062. URL: https://doi.org/10.1038/35057062.

[5] Bethesda(MD). *Nucleotide [Internet]*. National Library of Medicine (US), National Center for Biotechnology Information (2004). URL: https://www.ncbi.nlm.nih.gov/nucleotide/.

[6] J. Babickova and R. Gardlik. *Pathological and therapeutic interactions between bacteriophages, microbes and the host in inflammatory bowel disease*. eng. World journal of gastroenterology **21** (Oct. 2015). PMC4616208[pmcid], 11321–11330. DOI: 10.3748/wjg.v21.i40.11321. URL: https://doi.org/10.3748/wjg.v21.i40.11321.

[7] A. B. R. McIntyre, R. Ounit, E. Afshinnekoo, R. J. Prill, E. Hénaff, N. Alexander, S. S. Minot, D. Danko, J. Foox, S. Ahsanuddin, S. Tighe, N. A. Hasan, P. Subramanian, K. Moffat, S. Levy, S. Lonardi, N. Greenfield, R. R. Colwell, G. L. Rosen, and C. E. Mason. *Comprehensive benchmarking and ensemble approaches for metagenomic classifiers*. Genome Biology **18** (Sept.

2017), 182. DOI: 10.1186/s13059-017-1299-7. URL: https://doi.org/10.1186/s13059-017-1299-7.

[8] A. Sczyrba, P. Hofmann, P. Belmann, D. Koslicki, S. Janssen, J. Dröge, I. Gregor, S. Majda, J. Fiedler, E. Dahms, A. Bremges, A. Fritz, R. Garrido-Oter, T. S. Jørgensen, N. Shapiro, P. D. Blood, A. Gurevich, Y. Bai, D. Turaev, M. Z. DeMaere, R. Chikhi, N. Nagarajan, C. Quince, F. Meyer, M. Balvociute, L. H. Hansen, S. J. Sørensen, B. K. H. Chia, B. Denis, J. L. Froula, Z. Wang, R. Egan, D. Don Kang, J. J. Cook, C. Deltel, M. Beckstette, C. Lemaitre, P. Peterlongo, G. Rizk, D. Lavenier, Y.-W. Wu, S. W. Singer, C. Jain, M. Strous, H. Klingenberg, P. Meinicke, M. D. Barton, T. Lingner, H.-H. Lin, Y.-C. Liao, G. G. Z. Silva, D. A. Cuevas, R. A. Edwards, S. Saha, V. C. Piro, B. Y. Renard, M. Pop, H.-P. Klenk, M. Göker, N. C. Kyrpides, T. Woyke, J. A. Vorholt, P. Schulze-Lefert, E. M. Rubin, A. E. Darling, T. Rattei, and A. C. McHardy. *Critical Assessment of Metagenome Interpretation–a benchmark of metagenomics software*. Nature Methods **14** (Oct. 2017), 1063 EP. URL: http://dx.doi.org/10.1038/nmeth.4458.

[9] S. Weging, A. Gogol-Döring, and I. Grosse. *Taxonomic Analysis of Metagenomic Data with kASA*. Nucleic Acids Research (Apr. 2021). DOI: 10.1093/nar/gkab200. URL: https://doi.org/10.1093/nar/gkab200.

[10] *kASA GitHub Page*. URL: https://github.com/SilvioWeging/kASA.

[11] *kASA Robustness Benchmark GitHub Page*. URL: https://github.com/SilvioWeging/kASA_snakemake.

[12] *kASA CAMI Benchmark GitHub Page*. URL: https://github.com/SilvioWeging/kASA_cami.

[13] A. Escobar-Zepeda, A. Vera-Ponce de León, and A. Sanchez-Flores. *The Road to Metagenomics: From Microbiology to DNA Sequencing Technologies and Bioinformatics*. Frontiers in Genetics **6** (2015), 348. DOI: 10.3389/fgene.2015.00348. URL: https://www.frontiersin.org/article/10.3389/fgene.2015.00348.

[14] S. Hamarneh. *Measuring the Invisible World. The life and works of Antoni van Leeuwenhoek. A. Schierbeek. Abelard-Schuman, New York, 1959. 223 pp. $4*. Science **132** (1960), 289–290. DOI: 10.1126/science.132.3422.289. eprint: https://science.sciencemag.org/content/132/3422/289.full.pdf. URL: https://science.sciencemag.org/content/132/3422/289.

[15] M. S. B. Steve M. Blevins. *Robert Koch and the 'golden age' of bacteriology*. International Journal of Infectious Diseases **14** (2010), E744–E751. DOI: 10.1016/j.ijid.2009.12.003. URL: https://www.ijidonline.com/article/S1201-9712(10)02314-3/fulltext.

[16] J. T. Staley and A. Konopka. *MEASUREMENT OF IN SITU ACTIVITIES OF NONPHOTOSYNTHETIC MICROORGANISMS IN AQUATIC AND TERRESTRIAL HABITATS*. Annual Review of Microbiology **39** (1985). PMID: 3904603, 321–346. DOI: 10.1146/annurev.mi.39.100185.001541.

eprint: https://doi.org/10.1146/annurev.mi.39.100185.001541. URL: https://doi.org/10.1146/annurev.mi.39.100185.001541.

[17] M. McFall-Ngai. *Are biologists in 'future shock'? Symbiosis integrates biology across domains.* Nature Reviews Microbiology **6** (Oct. 2008), 789–792. DOI: 10.1038/nrmicro1982. URL: https://doi.org/10.1038/nrmicro1982.

[18] C. R. Woese and G. E. Fox. *Phylogenetic structure of the prokaryotic domain: The primary kingdoms.* Proceedings of the National Academy of Sciences **74** (1977), 5088–5090. DOI: 10.1073/pnas.74.11.5088. eprint: https://www.pnas.org/content/74/11/5088.full.pdf. URL: https://www.pnas.org/content/74/11/5088.

[19] S. K. Heijs, R. R. Haese, P. W. J. J. van der Wielen, L. J. Forney, and J. D. van Elsas. *Use of 16S rRNA Gene Based Clone Libraries to Assess Microbial Communities Potentially Involved in Anaerobic Methane Oxidation in a Mediterranean Cold Seep.* Microbial Ecology **53** (Apr. 2007), 384–398. DOI: 10.1007/s00248-006-9172-3. URL: https://doi.org/10.1007/s00248-006-9172-3.

[20] W. G. Weisburg, S. M. Barns, D. A. Pelletier, and D. J. Lane. *16S ribosomal DNA amplification for phylogenetic study.* Journal of Bacteriology **173** (1991), 697–703. DOI: 10.1128/jb.173.2.697-703.1991. eprint: https://jb.asm.org/content/173/2/697.full.pdf. URL: https://jb.asm.org/content/173/2/697.

[21] B. A. Lodish H and Z. S. et al. *DNA Cloning with Plasmid Vectors.* Molecular Cell Biology **4th edition** (2000), Section 7.1. URL: https://www.ncbi.nlm.nih.gov/books/NBK21498/.

[22] W. Gerlach and J. Bedbrook. *Cloning and characterization of ribosomal RNA genes from wheat and barley.* Nucleic Acids Research **7** (Dec. 1979), 1869–1885. DOI: 10.1093/nar/7.7.1869. eprint: https://academic.oup.com/nar/article-pdf/7/7/1869/7056022/7-7-1869.pdf. URL: https://doi.org/10.1093/nar/7.7.1869.

[23] J. Handelsman, M. R. Rondon, S. F. Brady, J. Clardy, and R. M. Goodman. *Molecular biological access to the chemistry of unknown soil microbes: a new frontier for natural products.* Chem Biol **5** (Oct. 1998), R245–249. URL: https://www.cell.com/cell-chemical-biology/pdf/S1074-5521(98)90108-9.pdf?_returnURL=https%5C%3A%5C%2F%5C%2Flinkinghub.elsevier.com%5C%2Fretrieve%5C%2Fpii%5C%2FS1074552198901089%5C%3Fshowall%5C%3Dtrue.

[24] T. D. Brock. *The value of basic research: discovery of Thermus aquaticus and other extreme thermophiles.* eng. Genetics **146** (Aug. 1997), 1207–1210. URL: https://pubmed.ncbi.nlm.nih.gov/9258667.

[25] R. Saiki, S. Scharf, F. Faloona, K. Mullis, G. Horn, H. Erlich, and N. Arnheim. *Enzymatic amplification of beta-globin genomic sequences and restriction site analysis for diagnosis of sickle cell anemia.* Science **230** (1985), 1350–1354. DOI: 10.1126/science.2999980. eprint: https://science.sciencemag.or

g/content/230/4732/1350.full.pdf. URL: https://science.sciencemag.org/content/230/4732/1350.

[26] R. Verhelst, H. Verstraelen, G. Claeys, G. Verschraegen, J. Delanghe, L. Van Simaey, C. De Ganck, M. Temmerman, and M. Vaneechoutte. *Cloning of 16S rRNA genes amplified from normal and disturbed vaginal microflora suggests a strong association between Atopobium vaginae, Gardnerella vaginalis and bacterial vaginosis*. BMC Microbiology **4** (Apr. 2004), 16. DOI: 10.1186/1471-2180-4-16. URL: https://doi.org/10.1186/1471-2180-4-16.

[27] F. Sanger and A. Coulson. *A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase*. Journal of Molecular Biology **94** (1975), 441–448. DOI: https://doi.org/10.1016/0022-2836(75)90213-2. URL: https://www.sciencedirect.com/science/article/pii/0022283675902132.

[28] C. Ledergerber and C. Dessimoz. *Base-calling for next-generation sequencing platforms*. Briefings in Bioinformatics **12** (Jan. 2011), 489–497. DOI: 10.1093/bib/bbq077. eprint: https://academic.oup.com/bib/article-pdf/12/5/489/747085/bbq077.pdf. URL: https://doi.org/10.1093/bib/bbq077.

[29] T. J. Sharpton. *An introduction to the analysis of shotgun metagenomic data*. Frontiers in Plant Science **5** (2014), 209. DOI: 10.3389/fpls.2014.00209. URL: https://www.frontiersin.org/article/10.3389/fpls.2014.00209.

[30] C. Quince, A. W. Walker, J. T. Simpson, N. J. Loman, and N. Segata. *Shotgun metagenomics, from sampling to analysis*. Nature Biotechnology **35** (Sept. 2017), 833–844. DOI: 10.1038/nbt.3935. URL: https://doi.org/10.1038/nbt.3935.

[31] J. J. Kasianowicz, E. Brandin, D. Branton, and D. W. Deamer. *Characterization of individual polynucleotide molecules using a membrane channel*. Proceedings of the National Academy of Sciences **93** (1996), 13770–13773. DOI: 10.1073/pnas.93.24.13770. eprint: https://www.pnas.org/content/93/24/13770.full.pdf. URL: https://www.pnas.org/content/93/24/13770.

[32] E. B. Fichot and R. S. Norman. *Microbial phylogenetic profiling with the Pacific Biosciences sequencing platform*. Microbiome **1** (Mar. 2013), 10. DOI: 10.1186/2049-2618-1-10. URL: https://doi.org/10.1186/2049-2618-1-10.

[33] A. Gupta, R. Gupta, and R. L. Singh. "Microbes and Environment." *Principles and Applications of Environmental Biotechnology for a Sustainable Future*. Ed. by R. L. Singh. Springer Singapore 2017, 43–84. DOI: 10.1007/978-981-10-1866-4_3. URL: https://doi.org/10.1007/978-981-10-1866-4_3.

[34] 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** (2004), 66–74. DOI: 10.1126/science.1093857. eprint: https://science.sciencemag.org/content/304/5667/66.full.pdf. URL: https://science.sciencemag.org/content/304/5667/66.

[35] R. Pedron, A. Esposito, I. Bianconi, E. Pasolli, A. Tett, F. Asnicar, M. Cristofolini, N. Segata, and O. Jousson. *Genomic and metagenomic insights into the microbial community of a thermal spring*. Microbiome **7** (Jan. 2019), 8. DOI: 10.1186/s40168-019-0625-6. URL: https://doi.org/10.1186/s40168-019-0625-6.

[36] D. Schneider, N. Aßmann, D. Wicke, A. Poehlein, and R. Daniel. *Metagenomes of Wastewater at Different Treatment Stages in Central Germany*. Microbiology Resource Announcements **9** (2020). Ed. by F. J. Stewart. DOI: 10.1128/MRA.00201-20. eprint: https://mra.asm.org/content/9/15/e00201-20.full.pdf. URL: https://mra.asm.org/content/9/15/e00201-20.

[37] R. Saxena and V. Sharma. "Chapter 9 - A Metagenomic Insight Into the Human Microbiome: Its Implications in Health and Disease." *Medical and Health Genomics*. Ed. by D. Kumar and S. Antonarakis. Academic Press 2016, 107–119. DOI: https://doi.org/10.1016/B978-0-12-420196-5.00009-5. URL: https://www.sciencedirect.com/science/article/pii/B9780124201965000095.

[38] G. JM, G. SJ, L. P. S, S.-B. AM, F. HW, B. NB, B. V, D. MT, C. Y, W. T, K. KJ, H. E, C. SV, G. SG, B. Y. T, and L. MD. *Metagenomic analysis of feces from mice flown on the RR-6 mission*. Version 13. 2019. DOI: 10.26030/h713-bd02. URL: https://genelab-data.ndc.nasa.gov/genelab/accession/GLDS-249/.

[39] K. Venkateswaran, P. Vaishampayan, J. Cisneros, D. L. Pierson, S. O. Rogers, and J. Perry. *International Space Station environmental microbiome — microbial inventories of ISS filter debris*. Applied Microbiology and Biotechnology **98** (July 2014), 6453–6466. DOI: 10.1007/s00253-014-5650-6. URL: https://doi.org/10.1007/s00253-014-5650-6.

[40] E. Höring, D. Göpfert, G. Schröter, and U. von Gaisberg. *Frequency and Spectrum of Microorganisms Isolated from Biopsy Specimens in Chronic Colitis*. Endoscopy **23** (2008), 325–327. DOI: 10.1055/s-2007-1010707. URL: https://www.thieme-connect.de/products/ejournals/abstract/10.1055/s-2007-1010707.

[41] K. J. Locey and J. T. Lennon. *Scaling laws predict global microbial diversity*. Proceedings of the National Academy of Sciences **113** (2016), 5970–5975. DOI: 10.1073/pnas.1521291113. eprint: https://www.pnas.org/content/113/21/5970.full.pdf. URL: https://www.pnas.org/content/113/21/5970.

[42] A. Fleming. *On the Antibacterial Action of Cultures of a Penicillium, with Special Reference to their Use in the Isolation of B. influenzæ*. eng. British

journal of experimental pathology **10** (June 1929). PMC2048009[pmcid], 226–236. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2048009/.

[43] S. Riedel. *Edward Jenner and the history of smallpox and vaccination.* eng. Proceedings (Baylor University. Medical Center) **18** (Jan. 2005), 21–25. DOI: 10.1080/08998280.2005.11928028. URL: https://doi.org/10.1080/08998280.2005.11928028.

[44] K. Lundstrom. *Viral Vectors in Gene Therapy.* eng. Diseases (Basel, Switzerland) **6** (May 2018). diseases6020042[PII], 42. DOI: 10.3390/diseases6020042. URL: https://doi.org/10.3390/diseases6020042.

[45] M. Jinek, K. Chylinski, I. Fonfara, M. Hauer, J. A. Doudna, and E. Charpentier. *A Programmable Dual-RNA–Guided DNA Endonuclease in Adaptive Bacterial Immunity.* Science **337** (2012), 816–821. DOI: 10.1126/science.1225829. eprint: https://science.sciencemag.org/content/337/6096/816.full.pdf. URL: https://science.sciencemag.org/content/337/6096/816.

[46] N. Sikri and A. Bardia. *A history of streptokinase use in acute myocardial infarction.* eng. Texas Heart Institute journal **34** (2007). PMC1995058[pmcid], 318–327. URL: https://pubmed.ncbi.nlm.nih.gov/17948083.

[47] G. Ceballos, P. R. Ehrlich, A. D. Barnosky, A. Garcia, R. M. Pringle, and T. M. Palmer. *Accelerated modern human–induced species losses: Entering the sixth mass extinction.* Science Advances **1** (2015). DOI: 10.1126/sciadv.1400253. eprint: https://advances.sciencemag.org/content/1/5/e1400253.full.pdf. URL: https://advances.sciencemag.org/content/1/5/e1400253.

[48] A. Cain. *Taxonomy* (2021). URL: https://www.britannica.com/science/taxonomy.

[49] C. v. Linné and L. Salvius. *Caroli Linnaei...Systema naturae per regna tria naturae :secundum classes, ordines, genera, species, cum characteribus, differentiis, synonymis, locis.* Vol. v.2. Holmiae :Impensis Direct. Laurentii Salvii, (1759), 564. URL: https://www.biodiversitylibrary.org/item/10278.

[50] C. Darwin and L. Kebler. *On the origin of species by means of natural selection, or, The preservation of favoured races in the struggle for life.* (1859). URL: https://lccn.loc.gov/06017473.

[51] C. A. Long. *Sokal, Robert R., and Peter H. A. Sneath. Principles of Numerical Taxonomy. W. H. Freeman and Co., San Francisco and London. Pp. xvi + 359, illus. 1963. Price $8.50.* Journal of Mammalogy **46** (Feb. 1965), 111–112. DOI: 10.2307/1377831. eprint: https://academic.oup.com/jmammal/article-pdf/46/1/111/2665769/46-1-111.pdf. URL: https://doi.org/10.2307/1377831.

[52] C. R. Woese, L. J. Magrum, and G. E. Fox. *Archaebacteria.* Journal of Molecular Evolution **11** (Sept. 1978), 245–252. DOI: 10.1007/BF01734485. URL: https://doi.org/10.1007/BF01734485.

[53] R. MA, G. DP, O. TM, B. N, B. T, and B. R. et al. *A Higher Level Classification of All Living Organisms*. PLoS ONE **10** (2015). DOI: 10.1371/journal.pone.0119248. URL: https://doi.org/10.1371/journal.pone.0119248.

[54] A. Mateos-Rivera, R. Skern-Mauritzen, G. Dahle, S. Sundby, B. Mozfar, A. Thorsen, H. Wehde, and B. A. Krafft. *Comparison of visual and molecular taxonomic methods to identify ichthyoplankton in the North Sea*. Limnology and Oceanography: Methods **18** (2020), 599–605. DOI: https://doi.org/10.1002/lom3.10387. eprint: https://aslopubs.onlinelibrary.wiley.com/doi/pdf/10.1002/lom3.10387. URL: https://aslopubs.onlinelibrary.wiley.com/doi/abs/10.1002/lom3.10387.

[55] S. Federhen. *The NCBI Taxonomy database*. Nucleic Acids Research **40** (2012), D136–D143. DOI: 10.1093/nar/gkr1178. eprint: /oup/backfile/content_public/journal/nar/40/d1/10.1093_nar_gkr1178/2/gkr1178.pdf. URL: http://dx.doi.org/10.1093/nar/gkr1178.

[56] T. Thomas, J. Gilbert, and F. Meyer. *Metagenomics - a guide from sampling to data analysis*. Microbial Informatics and Experimentation **2** (Feb. 2012), 3. DOI: 10.1186/2042-5783-2-3. URL: https://doi.org/10.1186/2042-5783-2-3.

[57] C. Y. Chiu and S. A. Miller. *Clinical metagenomics*. Nature Reviews Genetics **20** (June 2019), 341–355. DOI: 10.1038/s41576-019-0113-7. URL: https://doi.org/10.1038/s41576-019-0113-7.

[58] P. A. Pevzner, H. Tang, and M. S. Waterman. *An Eulerian path approach to DNA fragment assembly*. Proceedings of the National Academy of Sciences **98** (2001), 9748–9753. DOI: 10.1073/pnas.171285098. eprint: https://www.pnas.org/content/98/17/9748.full.pdf. URL: https://www.pnas.org/content/98/17/9748.

[59] K. Schneeberger, S. Ossowski, F. Ott, J. D. Klein, X. Wang, C. Lanz, L. M. Smith, J. Cao, J. Fitz, N. Warthmann, S. R. Henz, D. H. Huson, and D. Weigel. *Reference-guided assembly of four diverse Arabidopsis thaliana genomes*. Proceedings of the National Academy of Sciences **108** (2011), 10249–10254. DOI: 10.1073/pnas.1107739108. eprint: https://www.pnas.org/content/108/25/10249.full.pdf. URL: https://www.pnas.org/content/108/25/10249.

[60] F. Meyer, A. Bremges, P. Belmann, S. Janssen, A. C. McHardy, and D. Koslicki. *Assessing taxonomic metagenome profilers with OPAL*. Genome Biology **20** (2019), 51. DOI: 10.1186/s13059-019-1646-y. URL: https://doi.org/10.1186/s13059-019-1646-y.

[61] D. Morgensztern, S. Devarakonda, T. Mitsudomi, C. Maher, and R. Govindan. "11 - Mutational Events in Lung Cancer: Present and Developing Technologies." *IASLC Thoracic Oncology (Second Edition)*. Ed. by H. I. Pass, D. Ball, and G. V. Scagliotti. Second Edition. Elsevier 2018, 95–103.e2. DOI: https://doi.org/10.1016/B978-0-323-52357-8.00011-1. URL: https://www.sciencedirect.com/science/article/pii/B9780323523578000111.

[62] N. R. Coordinators. *Database resources of the National Center for Biotechnology Information*. Nucleic Acids Research **46** (Nov. 2017), D8–D13. DOI: 10.1093/nar/gkx1095. eprint: https://academic.oup.com/nar/article-pdf/46/D1/D8/23162308/gkx1095.pdf. URL: https://doi.org/10.1093/nar/gkx1095.

[63] *Definition "Tool"*. URL: https://www.pcmag.com/encyclopedia/term/tool.

[64] *Definition "Classification"*. URL: https://dictionary.cambridge.org/dictionary/english/classification.

[65] V. Bruce and A. Young. *Understanding face recognition*. British Journal of Psychology **77** (1986), 305–327. DOI: https://doi.org/10.1111/j.2044-8295.1986.tb02199.x. eprint: https://bpspsychub.onlinelibrary.wiley.com/doi/pdf/10.1111/j.2044-8295.1986.tb02199.x. URL: https://bpspsychub.onlinelibrary.wiley.com/doi/abs/10.1111/j.2044-8295.1986.tb02199.x.

[66] C. A. Nelson. *The development and neural bases of face recognition*. Infant and Child Development **10** (2001), 3–18. DOI: https://doi.org/10.1002/icd.239. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/icd.239. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/icd.239.

[67] V. R. Marcelino, P. T. L. C. Clausen, J. P. Buchmann, M. Wille, J. R. Iredell, W. Meyer, O. Lund, T. C. Sorrell, and E. C. Holmes. *CCMetagen: comprehensive and accurate identification of eukaryotes and prokaryotes in metagenomic data*. Genome Biology **21** (Apr. 2020), 103. DOI: 10.1186/s13059-020-02014-2. URL: https://doi.org/10.1186/s13059-020-02014-2.

[68] N. K. Devanga Ragupathi, D. P. Muthuirulandi Sethuvel, F. Y. Inbanathan, and B. Veeraraghavan. *Accurate differentiation of Escherichia coli and Shigella serogroups: challenges and strategies*. eng. New microbes and new infections **21** (Sept. 2017). S2052-2975(17)30074-4[PII], 58–62. DOI: 10.1016/j.nmni.2017.09.003. URL: https://doi.org/10.1016/j.nmni.2017.09.003.

[69] G. Zuo, Z. Xu, and B. Hao. *Shigella strains are not clones of Escherichia coli but sister species in the genus Escherichia*. eng. Genomics, proteomics & bioinformatics **11** (Feb. 2013). S1672-0229(12)00112-X[PII], 61–65. DOI: 10.1016/j.gpb.2012.11.002. URL: https://doi.org/10.1016/j.gpb.2012.11.002.

[70] *Paired-end sequencing in Illumina*. URL: https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/paired-end-vs-single-read.html.

[71] E. R. Mardis. *DNA sequencing technologies: 2006–2016*. Nature Protocols **12** (Feb. 2017), 213–218. DOI: 10.1038/nprot.2016.182. URL: https://doi.org/10.1038/nprot.2016.182.

[72] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. *SOAP2: an improved ultrafast tool for short read alignment*. Bioinformatics

25 (June 2009), 1966–1967. DOI: 10.1093/bioinformatics/btp336. eprint: https://academic.oup.com/bioinformatics/article-pdf/25/15/1966/561321/btp336.pdf. URL: https://doi.org/10.1093/bioinformatics/btp336.

[73] J. W. Drake. *Rates of spontaneous mutation among RNA viruses.* Proceedings of the National Academy of Sciences **90** (1993), 4171–4175. DOI: 10.1073/pnas.90.9.4171. eprint: https://www.pnas.org/content/90/9/4171.full.pdf. URL: https://www.pnas.org/content/90/9/4171.

[74] E. F. Codd. *A Relational Model of Data for Large Shared Data Banks.* Commun. ACM **13** (June 1970), 377–387. DOI: 10.1145/362384.362685. URL: https://doi.org/10.1145/362384.362685.

[75] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press (1997). DOI: 10.1017/CBO9780511574931.

[76] J. Pevsner. *Bioinformatics and Functional Genomics, Second Edition.* John Wiley & Sons, Inc. (2009). DOI: 10.1002/9780470451496. URL: https://onlinelibrary.wiley.com/doi/book/10.1002/9780470451496.

[77] S. B. Needleman and C. D. Wunsch. *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of Molecular Biology **48** (1970), 443–453. DOI: https://doi.org/10.1016/0022-2836(70)90057-4. URL: http://www.sciencedirect.com/science/article/pii/0022283670900574.

[78] T. Smith and M. Waterman. *Identification of common molecular subsequences.* Journal of Molecular Biology **147** (1981), 195–197. DOI: https://doi.org/10.1016/0022-2836(81)90087-5. URL: http://www.sciencedirect.com/science/article/pii/0022283681900875.

[79] P. Weiner. "Linear pattern matching algorithms." *14th Annual Symposium on Switching and Automata Theory (swat 1973).* (1973), 1–11. DOI: 10.1109/SWAT.1973.13.

[80] S. Kurtz. *Reducing the space requirement of suffix trees.* Software: Practice and Experience **29** (1999), 1149–1171. DOI: https://doi.org/10.1002/(SICI)1097-024X(199911)29:13<1149::AID-SPE274>3.0.CO;2-O.

[81] U. Manber and G. Myers. *Suffix arrays: a new method for on-line string searches.* siam Journal on Computing **22** (1993), 935–948.

[82] M. Burrows and D. J. Wheeler. *A block-sorting lossless data compression algorithm.* Tech. rep. 1994.

[83] P. Ferragina and G. Manzini. "Opportunistic data structures with applications." *Proceedings 41st Annual Symposium on Foundations of Computer Science.* (Nov. 2000), 390–398. DOI: 10.1109/SFCS.2000.892127.

[84] H. Li. *Fast construction of FM-index for long sequence reads.* Bioinformatics **30** (Aug. 2014), 3274–3275. DOI: 10.1093/bioinformatics/btu541. eprint: https://academic.oup.com/bioinformatics/article-pdf/30/22/3274

/7252262/btu541.pdf. URL: https://doi.org/10.1093/bioinformatics/btu541.

[85] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, S. Li, H. Yang, J. Wang, and J. Wang. *De novo assembly of human genomes with massively parallel short read sequencing*. Genome Research **20** (2010), 265–272. DOI: 10.1101/gr.097261.109. eprint: http://genome.cshlp.org/content/20/2/265.full.pdf+html. URL: http://genome.cshlp.org/content/20/2/265.abstract.

[86] G. E. Sims and S.-H. Kim. *Whole-genome phylogeny of Escherichia coli/Shigella group by feature frequency profiles (FFPs)*. Proceedings of the National Academy of Sciences **108** (2011), 8329–8334. DOI: 10.1073/pnas.1105168108. eprint: https://www.pnas.org/content/108/20/8329.full.pdf. URL: https://www.pnas.org/content/108/20/8329.

[87] A. Z. Broder. "Identifying and Filtering Near-Duplicate Documents." *Combinatorial Pattern Matching*. Ed. by R. Giancarlo and D. Sankoff. Springer Berlin Heidelberg (2000), 1–10.

[88] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. "On Finding Lowest Common Ancestors in Trees." *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. STOC '73. Association for Computing Machinery (1973), 253–265. DOI: 10.1145/800125.804056. URL: https://doi.org/10.1145/800125.804056.

[89] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. *Basic local alignment search tool*. Journal of Molecular Biology **215** (1990), 403–410. DOI: https://doi.org/10.1016/S0022-2836(05)80360-2. URL: https://www.sciencedirect.com/science/article/pii/S0022283605803602.

[90] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. *A Greedy Algorithm for Aligning DNA Sequences*. Journal of Computational Biology **7** (2000). PMID: 10890397, 203–214. DOI: 10.1089/10665270050081478. eprint: https://doi.org/10.1089/10665270050081478. URL: https://doi.org/10.1089/10665270050081478.

[91] S. Henikoff and J. G. Henikoff. *Amino acid substitution matrices from protein blocks*. eng. Proceedings of the National Academy of Sciences of the United States of America **89** (Nov. 1992). PMC50453[pmcid], 10915–10919. DOI: 10.1073/pnas.89.22.10915. URL: https://doi.org/10.1073/pnas.89.22.10915.

[92] S. R. Eddy. *Where did the BLOSUM62 alignment score matrix come from?* Nature Biotechnology **22** (Aug. 2004), 1035–1036. DOI: 10.1038/nbt0804-1035. URL: https://doi.org/10.1038/nbt0804-1035.

[93] *BLAST Webpage*. URL: https://blast.ncbi.nlm.nih.gov/Blast.cgi.

[94] G.-M. Tan, L. Xu, D.-B. Bu, S.-Z. Feng, and N.-H. Sun. *Improvement of Performance of MegaBlast Algorithm for DNA Sequence Alignment*. Journal of Computer Science and Technology **21** (Nov. 2006), 973–978. DOI: 10.100

7/s11390-006-0973-0. URL: https://doi.org/10.1007/s11390-006-0973-0.

[95] B. Buchfink, C. Xie, and D. H. Huson. *Fast and sensitive protein alignment using DIAMOND*. Nature Methods **12** (2015), 59–60. DOI: 10.1038/nmeth.3176. URL: https://doi.org/10.1038/nmeth.3176.

[96] B. Ma, J. Tromp, and M. Li. *PatternHunter: faster and more sensitive homology search*. Bioinformatics **18** (2002), 440–445.

[97] U. Keich, M. Li, B. Ma, and J. Tromp. *On spaced seeds for similarity search*. Discrete Applied Mathematics **138** (2004), 253–263. DOI: https://doi.org/10.1016/S0166-218X(03)00382-2. URL: https://www.sciencedirect.com/science/article/pii/S0166218X03003822.

[98] H. Cohen and E. Porat. *Fast set intersection and two-patterns matching*. Theoretical Computer Science **411** (2010), 3795–3800. DOI: https://doi.org/10.1016/j.tcs.2010.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0304397510003361.

[99] A. Modi, S. Vai, D. Caramelli, and M. Lari. *The Illumina Sequencing Protocol and the NovaSeq 6000 System*. Methods Mol Biol **2242** (2021), 15–42.

[100] F. P. Breitwieser, D. N. Baker, and S. L. Salzberg. *KrakenUniq: confident and fast metagenomics classification using unique k-mer counts*. Genome Biology **19** (2018), 198. DOI: 10.1186/s13059-018-1568-0. URL: https://doi.org/10.1186/s13059-018-1568-0.

[101] A. P. Dempster, N. M. Laird, and D. B. Rubin. *Maximum Likelihood from Incomplete Data via the EM Algorithm*. Journal of the Royal Statistical Society. Series B (Methodological) **39** (1977), 1–38. URL: http://www.jstor.org/stable/2984875.

[102] P. Menzel, K. L. Ng, and A. Krogh. *Fast and sensitive taxonomic classification for metagenomics with Kaiju*. Nature Communications **7** (Apr. 2016). Article, 11257EP–. DOI: 10.1038/ncomms11257. URL: http://dx.doi.org/10.1038/ncomms11257.

[103] U. Lagerkvist. *"Two out of three": an alternative method for codon reading*. Proceedings of the National Academy of Sciences **75** (1978), 1759–1762. DOI: 10.1073/pnas.75.4.1759. eprint: https://www.pnas.org/content/75/4/1759.full.pdf. URL: https://www.pnas.org/content/75/4/1759.

[104] A. Müller, C. Hundt, A. Hildebrandt, T. Hankeln, and B. Schmidt. *MetaCache: context-aware classification of metagenomic reads using minhashing*. Bioinformatics **33** (2017), 3740–3748. DOI: 10.1093/bioinformatics/btx520. eprint: /oup/backfile/content_public/journal/bioinformatics/33/23/10.1093_bioinformatics_btx520/3/btx520.pdf. URL: http://dx.doi.org/10.1093/bioinformatics/btx520.

[105] D. E. Wood, J. Lu, and B. Langmead. *Improved metagenomic analysis with Kraken 2*. Genome Biology **20** (2019), 257. DOI: 10.1186/s13059-019-1891-0. URL: https://doi.org/10.1186/s13059-019-1891-0.

[106] V. C. Piro, T. H. Dadi, E. Seiler, K. Reinert, and B. Y. Renard. *ganon: continuously up-to-date with database growth for precise short read classification in metagenomics*. bioRxiv (2019). DOI: 10.1101/406017. eprint: https://www.biorxiv.org/content/early/2019/03/19/406017.full.pdf. URL: https://www.biorxiv.org/content/early/2019/03/19/406017.

[107] P. Jaccard. *THE DISTRIBUTION OF THE FLORA IN THE ALPINE ZONE.1*. New Phytologist **11** (1912), 37–50. DOI: https://doi.org/10.1111/j.1469-8137.1912.tb05611.x. eprint: https://nph.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1469-8137.1912.tb05611.x. URL: https://nph.onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-8137.1912.tb05611.x.

[108] T. H. Dadi, E. Siragusa, V. C. Piro, A. Andrusch, E. Seiler, B. Y. Renard, and K. Reinert. *DREAM-Yara: an exact read mapper for very large databases with short update time*. Bioinformatics **34** (Sept. 2018), i766–i772. DOI: 10.1093/bioinformatics/bty567. eprint: https://academic.oup.com/bioinformatics/article-pdf/34/17/i766/25702473/bty567.pdf. URL: https://doi.org/10.1093/bioinformatics/bty567.

[109] D. E. Wood and S. L. Salzberg. *Kraken: ultrafast metagenomic sequence classification using exact alignments*. Genome Biology **15** (Mar. 2014), R46. DOI: 10.1186/gb-2014-15-3-r46. URL: https://doi.org/10.1186/gb-2014-15-3-r46.

[110] R. Ounit, S. Wanamaker, T. J. Close, and S. Lonardi. *CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers*. BMC Genomics **16** (Mar. 2015), 236. DOI: 10.1186/s12864-015-1419-2. URL: https://doi.org/10.1186/s12864-015-1419-2.

[111] P. Reviriego, L. Holst, and J. A. Maestro. *On the expected longest length probe sequence for hashing with separate chaining*. Journal of Discrete Algorithms **9** (2011). Selected papers from the 7th International Conference on Algorithms and Complexity (CIAC 2010), 307–312. DOI: https://doi.org/10.1016/j.jda.2011.04.005. URL: https://www.sciencedirect.com/science/article/pii/S1570866711000505.

[112] *Kleene star*. DOI: 10.1093/oi/authority.20110803100039649. URL: https://www.oxfordreference.com/view/10.1093/oi/authority.20110803100039649.

[113] B. Alberts, A. Johnson, and J. e. a. Lewis. *Molecular Biology of the Cell. 4th edition.* New York: Garland Science (2002). URL: https://www.ncbi.nlm.nih.gov/books/NBK26821/.

[114] E. Blais and M. Blanchette. "Common Substrings in Random Strings." *Combinatorial Pattern Matching*. Ed. by M. Lewenstein and G. Valiente. Springer Berlin Heidelberg (2006), 129–140.

[115] E. V. Koonin and A. S. Novozhilov. *Origin and evolution of the genetic code: The universal enigma*. IUBMB Life **61** (2009), 99–111. DOI: https://doi.org/10.1002/iub.146. eprint: https://iubmb.onlinelibrary.wiley.com

/doi/pdf/10.1002/iub.146. URL: https://iubmb.onlinelibrary.wiley.com/doi/abs/10.1002/iub.146.

[116] Q. He, A. F. Bardet, B. Patton, J. Purvis, J. Johnston, A. Paulson, M. Gogol, A. Stark, and J. Zeitlinger. *High conservation of transcription factor binding and evidence for combinatorial regulation across six Drosophila species*. Nature Genetics **43** (May 2011), 414–420. DOI: 10.1038/ng.808. URL: https://doi.org/10.1038/ng.808.

[117] Z. Zhao, A. Cristian, and G. Rosen. *Keeping up with the genomes: efficient learning of our increasing knowledge of the tree of life*. BMC Bioinformatics **21** (Sept. 2020), 412. DOI: 10.1186/s12859-020-03744-7. URL: https://doi.org/10.1186/s12859-020-03744-7.

[118] G. E. Moore. *Cramming more components onto integrated circuits*. Electronics **38** (Apr. 1965). URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf.

[119] A. Sboner, X. J. Mu, D. Greenbaum, R. K. Auerbach, and M. B. Gerstein. *The real cost of sequencing: higher than you think!* Genome Biology **12** (Aug. 2011), 125. DOI: 10.1186/gb-2011-12-8-125. URL: https://doi.org/10.1186/gb-2011-12-8-125.

[120] R. Dementiev, L. Kettner, and P. Sanders. *STXXL: standard template library for XXL data sets*. Softw., Pract. Exper. **38** (2008), 589–637. URL: https://stxxl.org/.

[121] S.-W. Lee, B. Moon, and C. Park. "Advances in Flash Memory SSD Technology for Enterprise Database Applications." *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Association for Computing Machinery (2009), 863–870. DOI: 10.1145/1559845.1559937. URL: https://doi.org/10.1145/1559845.1559937.

[122] J. S. Vitter. *External Memory Algorithms and Data Structures: Dealing with Massive Data*. ACM Comput. Surv. **33** (June 2001), 209–271. DOI: 10.1145/384192.384193. URL: https://doi.org/10.1145/384192.384193.

[123] M. Jung and M. Kandemir. *Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications*. SIGMETRICS Perform. Eval. Rev. **41** (June 2013), 203–216. DOI: 10.1145/2494232.2465548. URL: https://doi.org/10.1145/2494232.2465548.

[124] B. Stroustrup. "Evolving a Language in and for the Real World: C++ 1991-2006." *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. Association for Computing Machinery (2007), 4–1–4–59. DOI: 10.1145/1238844.1238848. URL: https://doi.org/10.1145/1238844.1238848.

[125] A. Stepanov and M. Lee. *The Standard Template Library*. Technical Report 95-11(R.1). HP Laboratories, Nov. 1995. URL: http://stepanovpapers.com/STL/DOC.PDF.

[126] *zlib*. URL: https://zlib.net/.

[127] *Microsoft Visual Studio*. URL: https://visualstudio.microsoft.com/.

[128] *GNU Compiler Collection*. URL: https://gcc.gnu.org/.
[129] *Clang*. URL: https://clang.llvm.org/.
[130] *CMake*. URL: https://cmake.org/.
[131] D. R. MUSSER. *Introspective Sorting and Selection Algorithms*. Software: Practice and Experience **27** (1997), 983–993. DOI: https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-\#. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/\%28SICI\%291097-024X\%28199708\%2927\%3A8\%3C983\%3A\%3AAID-SPE117\%3E3.0.CO\%3B2-\%23. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%5C%28SICI%5C%291097-024X%5C%28199708%5C%2927%5C%3A8%5C%3C983%5C%3A%5C%3AAID-SPE117%5C%3E3.0.CO%5C%3B2-%5C%23.
[132] W. Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall (2010). URL: https://books.google.de/books?id=-7nM1DkWb1YC.
[133] *gzip*. URL: https://www.gnu.org/software/gzip/.
[134] *gzstream*. URL: https://www.cs.unc.edu/Research/compgeom/gzstream/.
[135] P. Brass. *Advanced Data Structures*. Cambridge University Press (2008). DOI: 10.1017/CBO9780511800191.
[136] *American Standard Code for Information Interchange*. URL: https://www.sr-ix.com/Archive/CharCodeHist/X3.4-1963/index.html.
[137] *OpenMP*. URL: https://www.openmp.org//.
[138] E. F. Codd. *Multiprogram scheduling: parts 1 and 2. introduction and theory*. Communications of the ACM **3 (6)** (1960), 347–350. DOI: https://doi.org/10.1145/367297.367317. URL: https://dl.acm.org/doi/10.1145/367297.367317.
[139] D. Lipman and W. Pearson. *Rapid and sensitive protein similarity searches*. Science **227** (1985), 1435–1441. DOI: 10.1126/science.2983426. eprint: https://science.sciencemag.org/content/227/4693/1435.full.pdf. URL: https://science.sciencemag.org/content/227/4693/1435.
[140] *NCBI Sequence Identifiers*. URL: https://www.ncbi.nlm.nih.gov/genbank/sequenceids/.
[141] P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. *The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants*. Nucleic Acids Research **38** (Dec. 2009), 1767–1771. DOI: 10.1093/nar/gkp1137. eprint: https://academic.oup.com/nar/article-pdf/38/6/1767/16769834/gkp1137.pdf. URL: https://doi.org/10.1093/nar/gkp1137.
[142] B. Ewing, L. Hillier, M. C. Wendl, and P. Green. *Base-Calling of Automated Sequencer Traces UsingPhred. I. Accuracy Assessment*. Genome Research **8** (1998), 175–185. DOI: 10.1101/gr.8.3.175. eprint: http://genome.cshlp.org/content/8/3/175.full.pdf+html. URL: http://genome.cshlp.org/content/8/3/175.abstract.

[143] J. Yang and W. Qiu. *Normalized Expected Utility-Entropy Measure of Risk*. Entropy **16** (2014), 3590–3604. DOI: 10.3390/e16073590. URL: https://www.mdpi.com/1099-4300/16/7/3590.

[144] A. Morgulis, E. M. Gertz, A. A. Schäffer, and R. Agarwala. *A Fast and Symmetric DUST Implementation to Mask Low-Complexity DNA Sequences*. Journal of Computational Biology **13** (2006). PMID: 16796549, 1028–1040. DOI: 10.1089/cmb.2006.13.1028. eprint: https://doi.org/10.1089/cmb.2006.13.1028. URL: https://doi.org/10.1089/cmb.2006.13.1028.

[145] C. E. Shannon. *A mathematical theory of communication*. The Bell System Technical Journal **27** (1948), 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[146] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. *POWER4 System Microarchitecture*. White Paper. IBM Server Group, Oct. 2001. URL: ftp://public.dhe.ibm.com/software/mktsupport/techdocs/power4.pdf.

[147] D. A. Huffman. *The synthesis of sequential switching circuits*. Technical Report. Research Laboratory of Electronics, Massachusetts Institute of Technology, Jan. 1954. URL: https://dspace.mit.edu/handle/1721.1/4804.

[148] S. Peyton Jones. *Beautiful concurrency*. Beautiful code. O'Reilly (Jan. 2007). URL: https://www.microsoft.com/en-us/research/publication/beautiful-concurrency/.

[149] M. T. Heath. *Hypercube multiprocessors 1986* (Jan. 1986). URL: https://www.osti.gov/biblio/6055870.

[150] J. Köster and S. Rahmann. *Snakemake—a scalable bioinformatics workflow engine*. Bioinformatics **28** (Aug. 2012), 2520–2522. DOI: 10.1093/bioinformatics/bts480. eprint: https://academic.oup.com/bioinformatics/article-pdf/28/19/2520/819790/bts480.pdf. URL: https://doi.org/10.1093/bioinformatics/bts480.

[151] S. Lindgreen, K. L. Adair, and P. P. Gardner. *An evaluation of the accuracy and speed of metagenome analysis tools*. Scientific Reports **6** (Jan. 2016). Article, 19233 EP. URL: http://dx.doi.org/10.1038/srep19233.

[152] D. Chicco and G. Jurman. *The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation*. BMC Genomics **21** (Jan. 2020), 6. DOI: 10.1186/s12864-019-6413-7. URL: https://doi.org/10.1186/s12864-019-6413-7.

[153] K. Pearson. *Note on Regression and Inheritance in the Case of Two Parents*. Proceedings of the Royal Society of London Series I **58** (Jan. 1895), 240–242.

[154] C. Spearman. *The Proof and Measurement of Association Between Two Things*. 1904.

[155] M. Seppey, M. Manni, and E. M. Zdobnov. *LEMMI: A Live Evaluation of Computational Methods for Metagenome Investigation*. bioRxiv (2019). DOI: 10.1101/507731. eprint: https://www.biorxiv.org/content/early/2019/04/16/507731.full.pdf. URL: https://www.biorxiv.org/content/early/2019/04/16/507731.

[156] C. P. Oechslin, N. Lenz, N. Liechti, S. Ryter, P. Agyeman, R. Bruggmann, S. L. Leib, and C. M. Beuret. *Limited Correlation of Shotgun Metagenomics Following Host Depletion and Routine Diagnostics for Viruses and Bacteria in Low Concentrated Surrogate and Clinical Samples*. Frontiers in Cellular and Infection Microbiology **8** (2018), 375. DOI: 10.3389/fcimb.2018.00375. URL: https://www.frontiersin.org/article/10.3389/fcimb.2018.00375.

[157] A. C. Retchless, C. B. Kretz, L. D. Rodriguez-Rivera, A. Chen, H. M. Soeters, M. J. Whaley, and X. Wang. *Oropharyngeal microbiome of a college population following a meningococcal disease outbreak*. Scientific Reports **10** (Jan. 2020), 632. DOI: 10.1038/s41598-020-57450-8. URL: https://doi.org/10.1038/s41598-020-57450-8.

[158] L. Wang, Y. Sun, X. Sun, L. Yu, L. Xue, Z. He, J. Huang, D. Tian, L. D. Hurst, and S. Yang. *Repeat-induced point mutation in Neurospora crassa causes the highest known mutation rate and mutational burden of any cellular life*. Genome Biology **21** (June 2020), 142. DOI: 10.1186/s13059-020-02060-w. URL: https://doi.org/10.1186/s13059-020-02060-w.

[159] D. Green and J. Swets. *Signal Detection Theory and Psychophysics*. Wiley (1966). URL: https://books.google.de/books?id=fHR9AAAAMAAJ.

[160] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman, A. Bibillo, K. Bjornson, B. Chaudhuri, F. Christians, R. Cicero, S. Clark, R. Dalal, A. deWinter, J. Dixon, M. Foquet, A. Gaertner, P. Hardenbol, C. Heiner, K. Hester, D. Holden, G. Kearns, X. Kong, R. Kuse, Y. Lacroix, S. Lin, P. Lundquist, C. Ma, P. Marks, M. Maxham, D. Murphy, I. Park, T. Pham, M. Phillips, J. Roy, R. Sebra, G. Shen, J. Sorenson, A. Tomaney, K. Travers, M. Trulson, J. Vieceli, J. Wegener, D. Wu, A. Yang, D. Zaccarin, P. Zhao, F. Zhong, J. Korlach, and S. Turner. *Real-Time DNA Sequencing from Single Polymerase Molecules*. Science **323** (2009), 133–138. DOI: 10.1126/science.1162986. eprint: https://science.sciencemag.org/content/323/5910/133.full.pdf. URL: https://science.sciencemag.org/content/323/5910/133.

[161] *iDiv HPC Cluster*. URL: https://www.idiv.de/en/research/platforms_and_networks/hpc_cluster.html.

[162] *USB Implementers Forum*. URL: https://www.usb.org/documents.

[163] D. P. Rodgers. *Improvements in multiprocessor system design*. ACM SIGARCH Computer Architecture News **13** (June 1985), 225–231. DOI: 10.1145/327070.327215. URL: https://dl.acm.org/doi/10.1145/327070.327215.

[164] M. Jain, H. E. Olsen, B. Paten, and M. Akeson. *The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community*. Genome Biology **17** (Nov. 2016), 239. DOI: 10.1186/s13059-016-1103-0. URL: https://doi.org/10.1186/s13059-016-1103-0.

[165] D. P. McMahon, M. E. Natsopoulou, V. Doublet, M. Fürst, S. Weging, M. J. F. Brown, A. Gogol-Döring, and R. J. Paxton. *Elevated virulence of*

an emerging viral genotype as a driver of honeybee loss. Proceedings of the Royal Society B: Biological Sciences **283** (2016), 20160811. DOI: 10.1098/rspb.2016.0811. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspb.2016.0811. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspb.2016.0811.

[166] B. Langmead and S. L. Salzberg. *Fast gapped-read alignment with Bowtie 2*. Nature Methods **9** (Apr. 2012), 357–359. DOI: 10.1038/nmeth.1923. URL: https://doi.org/10.1038/nmeth.1923.

[167] D. Kim, L. Song, F. P. Breitwieser, and S. L. Salzberg. *Centrifuge: rapid and sensitive classification of metagenomic sequences*. Genome Research **26** (2016), 1721–1729. DOI: 10.1101/gr.210641.116. eprint: http://genome.cshlp.org/content/26/12/1721.full.pdf+html. URL: http://genome.cshlp.org/content/26/12/1721.abstract.

[168] J. C. Clemente, L. K. Ursell, L. W. Parfrey, and R. Knight. *The impact of the gut microbiota on human health: an integrative view*. eng. Cell **148** (Mar. 2012). S0092-8674(12)00104-3[PII], 1258–1270. DOI: 10.1016/j.cell.2012.01.035. URL: https://doi.org/10.1016/j.cell.2012.01.035.

[169] T. H. M. P. Consortium. *A framework for human microbiome research*. Nature **486** (June 2012). Article, 215 EP. URL: http://dx.doi.org/10.1038/nature11209.

[170] N. A. O'Leary, M. W. Wright, J. R. Brister, S. Ciufo, D. Haddad, R. McVeigh, B. Rajput, B. Robbertse, B. Smith-White, D. Ako-Adjei, A. Astashyn, A. Badretdin, Y. Bao, O. Blinkova, V. Brover, V. Chetvernin, J. Choi, E. Cox, O. Ermolaeva, C. M. Farrell, T. Goldfarb, T. Gupta, D. Haft, E. Hatcher, W. Hlavina, V. S. Joardar, V. K. Kodali, W. Li, D. Maglott, P. Masterson, K. M. McGarvey, M. R. Murphy, K. O'Neill, S. Pujar, S. H. Rangwala, D. Rausch, L. D. Riddick, C. Schoch, A. Shkeda, S. S. Storz, H. Sun, F. Thibaud-Nissen, I. Tolstoy, R. E. Tully, A. R. Vatsan, C. Wallin, D. Webb, W. Wu, M. J. Landrum, A. Kimchi, T. Tatusova, M. DiCuccio, P. Kitts, T. D. Murphy, and K. D. Pruitt. *Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation*. Nucleic Acids Res. **44** (Jan. 2016), D733–745.

[171] M. Jain, S. Koren, K. H. Miga, J. Quick, A. C. Rand, T. A. Sasani, J. R. Tyson, A. D. Beggs, A. T. Dilthey, I. T. Fiddes, S. Malla, H. Marriott, T. Nieto, J. O'Grady, H. E. Olsen, B. S. Pedersen, A. Rhie, H. Richardson, A. R. Quinlan, T. P. Snutch, L. Tee, B. Paten, A. M. Phillippy, J. T. Simpson, N. J. Loman, and M. Loose. *Nanopore sequencing and assembly of a human genome with ultra-long reads*. Nature Biotechnology **36** (Jan. 2018), 338 EP. URL: http://dx.doi.org/10.1038/nbt.4060.

[172] *GM12878 Utah/Ceph cell line*. URL: https://www.coriell.org/0/Sections/Search/Sample_Detail.aspx?Ref=GM12878&Product=CC.

[173] B. D. Ondov, N. H. Bergman, and A. M. Phillippy. *Interactive metagenomic visualization in a Web browser*. BMC Bioinformatics **12** (Sept. 2011), 385.

DOI: 10.1186/1471-2105-12-385. URL: https://doi.org/10.1186/1471-2105-12-385.

[174] D. Mayhew and V. Krishnan. "PCI express and advanced switching: evolutionary path to building next generation interconnects." *11th Symposium on High Performance Interconnects, 2003. Proceedings.* (2003), 21–29. DOI: 10.1109/CONECT.2003.1231473.

[175] *Example of a PCIe SSD.* URL: https://www.westerndigital.com/de-de/products/internal-drives/wd-black-sn850-nvme-ssd#WDS500G1X0E.

[176] E. G. Overbey, A. M. Saravia-Butler, Z. Zhang, K. S. Rathi, H. Fogle, W. A. da Silveira, R. J. Barker, J. J. Bass, A. Beheshti, D. C. Berrios, E. A. Blaber, E. Cekanaviciute, H. A. Costa, L. B. Davin, K. M. Fisch, S. G. Gebre, M. Geniza, R. Gilbert, S. Gilroy, G. Hardiman, R. Herranz, Y. H. Kidane, C. P. Kruse, M. D. Lee, T. Liefeld, N. G. Lewis, J. T. McDonald, R. Meller, T. Mishra, I. Y. Perera, S. Ray, S. S. Reinsch, S. B. Rosenthal, M. Strong, N. J. Szewczyk, C. G. Tahimic, D. M. Taylor, J. P. Vandenbrink, A. Villacampa, S. Weging, C. Wolverton, S. E. Wyatt, L. Zea, S. V. Costes, and J. M. Galazka. *NASA GeneLab RNA-seq consensus pipeline: Standardized processing of short-read RNA-seq data.* iScience **24** (2021), 102361. DOI: https://doi.org/10.1016/j.isci.2021.102361. URL: https://www.sciencedirect.com/science/article/pii/S2589004221003291.

[177] *GeneLab Data Processing Pipeline.* URL: https://github.com/asaravia-butler/GeneLab_Data_Processing.

[178] D. T. Truong, E. A. Franzosa, T. L. Tickle, M. Scholz, G. Weingart, E. Pasolli, A. Tett, C. Huttenhower, and N. Segata. *MetaPhlAn2 for enhanced metagenomic taxonomic profiling.* Nature Methods **12** (Oct. 2015), 902–903. DOI: 10.1038/nmeth.3589. URL: https://doi.org/10.1038/nmeth.3589.

[179] Bethesda(MD). *Protein [Internet].* National Library of Medicine (US), National Center for Biotechnology Information (2004). URL: https://www.ncbi.nlm.nih.gov/protein/.

[180] R. E. Ley, D. A. Peterson, and J. I. Gordon. *Ecological and Evolutionary Forces Shaping Microbial Diversity in the Human Intestine.* Cell **124** (Feb. 2006), 837–848. DOI: 10.1016/j.cell.2006.02.017. URL: https://doi.org/10.1016/j.cell.2006.02.017.

[181] G.-S. Sun, J. C. Tou, K. Liittschwager, A. M. Herrera, E. L. Hill, B. Girten, D. Reiss-Bubenheim, and M. Vasques. *Evaluation of the nutrient-upgraded rodent food bar for rodent spaceflight experiments.* Nutrition **26** (2010), 1163–1169. DOI: https://doi.org/10.1016/j.nut.2009.09.018. URL: https://www.sciencedirect.com/science/article/pii/S0899900709004109.

[182] G. D. Wu, J. Chen, C. Hoffmann, K. Bittinger, Y.-Y. Chen, S. A. Keilbaugh, M. Bewtra, D. Knights, W. A. Walters, R. Knight, R. Sinha, E. Gilroy, K. Gupta, R. Baldassano, L. Nessel, H. Li, F. D. Bushman, and J. D. Lewis. *Linking Long-Term Dietary Patterns with Gut Microbial Enterotypes.* Science **334** (2011), 105–108. DOI: 10.1126/science.1208344. eprint: https://sc

ience.sciencemag.org/content/334/6052/105.full.pdf. URL: https://science.sciencemag.org/content/334/6052/105.

[183] H. N. SHAH and D. M. COLLINS. *NOTES: Prevotella, a New Genus To Include Bacteroides melaninogenicus and Related Species Formerly Classified in the Genus Bacteroides*. International Journal of Systematic and Evolutionary Microbiology **40** (1990), 205–208. DOI: https://doi.org/10.1099/00207713-40-2-205. URL: https://www.microbiologyresearch.org/content/journal/ijsem/10.1099/00207713-40-2-205.

[184] R. Ounit and S. Lonardi. *Higher classification sensitivity of short metagenomic reads with CLARK-S*. Bioinformatics **32** (Aug. 2016), 3823–3825. DOI: 10.1093/bioinformatics/btw542. eprint: https://academic.oup.com/bioinformatics/article-pdf/32/24/3823/16920900/btw542.pdf. URL: https://doi.org/10.1093/bioinformatics/btw542.

[185] *GitHub of Kaiju*. URL: https://github.com/bioinformatics-centre/kaiju/.

[186] P. Sieber, M. Platzer, and S. Schuster. *The Definition of Open Reading Frame Revisited*. Trends in Genetics **34** (Mar. 2018), 167–170. DOI: 10.1016/j.tig.2017.12.009. URL: https://doi.org/10.1016/j.tig.2017.12.009.

[187] A. Ltd. *AArch64 architecture*. URL: https://developer.arm.com/documentation/ddi0487/latest.

[188] R. Baeza-Yates and A. Salinger. "Fast Intersection Algorithms for Sorted Sequences." *Algorithms and Applications: Essays Dedicated to Esko Ukkonen on the Occasion of His 60th Birthday*. Ed. by T. Elomaa, H. Mannila, and P. Orponen. Springer Berlin Heidelberg 2010, 45–61. DOI: 10.1007/978-3-642-12476-1_3. URL: https://doi.org/10.1007/978-3-642-12476-1_3.

[189] V. I. Levenshtein. *Binary codes capable of correcting deletions, insertions, and reversals*. Dokl. Akad. Nauk SSSR **163** (1965), 845–848. URL: http://mi.mathnet.ru/dan31411.

Definitions and descriptions

A^*	Kleene star, set of all strings which can be formed with the alphabet $\mathcal{A}$
M	Set of all sequences/reads from (meta-) genomic data
R	Set of IDs describing the order of reads
M_R	Set of tuples each containing a read and its ID
S_R	A relation between all possible non-empty substrings of sequences from M and the IDs from R
DB	Set of all genomes in the form of strings derived from a database
T	Set of all taxonomic IDs of the genomes in DB
DB_T	Set of tuples each containing a genome with its respective taxonomic ID
G_T	A relation between all possible non-empty substrings of genomes from DB and the IDs from T
k_L	The lowest or smallest k in the range of k's $\in [k_L, k_H]$
k_H	The highest or largest k in the range of k's $\in [k_L, k_H]$
T_M	Container of matching taxonomic IDs, saved into a bit array set
R_M	Container of matching read IDs saved into an `std::vector`

Abundance	The proportion of a taxon's DNA in a dataset, expressed as a percentage
Accuracy	Refers here to the degree to which software can correctly identify a taxon
Database	(Fasta) files containing the reference
DNA sequence / Read	A string of digitized DNA gained by DNA sequencing
DNA sequencing	The process of determining the order of nucleotides in DNA
Frames	Translated words generated by shifting the starting position of the translation by up to two DNA bases
Hits	Events where matches happen
HPCC	High performance computing cluster
Identification	Synonym for taxonomic profiling
Index	Data structure holding the database
LCA	Lowest common ancestor, a node in the taxonomic tree that is one level higher up
MAGs	Metagenome-assembled genomes, (often novel) genomes assembled from metagenomic data
Matches	DNA or protein sequences from new data which are identical to sequences in the reference
Platform	A certain configuration of hardware like a Desktop PC or a Laptop
Primary memory	Memory from RAM and CPU cache combined and characterized by fast access and volatility
Profile	A table of all organisms or taxa that have been identified in a dataset together with their respective abundance
Reference	Genomic or protein data from known species
r-Identification	List of read IDs together with their identified taxa and scores
Robustness	Ability to ignore errors without influencing accuracy
Secondary memory	Non-volatile memory, usually hard drives (HDD) or solid-state drives (SSD) as well as flash drives
Taxon / Taxa	Taxonomic unit, usually a species
Taxonomic profiling	Process of creating an r-identification or/and a profile by searching for similarities between a reference and sequenced data

Eidesstattliche Erklärung / *Declaration under Oath*

Ich erkläre an Eides statt, dass ich die Arbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

I declare under penalty of perjury that this thesis is my own work entirely and has been written without any help from other people. I used only the sources mentioned and included all the citations correctly both in word or content.

....

Ort / *City*, Datum / *Date* Unterschrift / *Signature*

Silvio Weging

March 1, 2023

Email: silvio.weging@gmail.com **Google Scholar**: Silvio Weging
GitHub: SilvioWeging **ORCID**: 0000-0002-8484-4352

Research interests

Taxonomic Profiling, Algorithm Engineering, Mathematical Optimization, Bioinformatics, Big Data

Education

Martin-Luther University Halle-Wittenberg
Dr. rer. nat. Computer Science / Bioinformatics
Mentors: Ivo Grosse; Andreas Gogol-Döring
2016-05 – 2022-10

Otto-von-Guericke Universität Magdeburg
M. Sc. Mathematics / Computermathematics
2011-10 – 2014-05

Otto-von-Guericke Universität Magdeburg
B. Sc. Computermathematics
2008-10 – 2011-12

Dissertation

Novel resource-efficient methods for robust and accurate taxonomic profiling of metagenomic data
Mentors: Ivo Große; Andreas Gogol-Döring

Master thesis

Zone models on the basis of Kriging approximation for use in adaptive substitute model-based optimization procedures
Mentors: Gennadiy Averkov; IAV GmbH Chemnitz

Bachelor thesis

Efficient conversion of number representations based on floating point numbers
Mentors: Stefan Schirra; Marc Mörig

Publications

Taxonomic Analysis of Metagenomic Data with kASA
Silvio Weging, Andreas Gogol-Döring, Ivo Grosse
Nucleic Acids Research, 2021
`https://doi.org/10.1093/nar/gkab200`

NASA GeneLab RNA-Seq Consensus Pipeline: Standardized Processing of Short-Read RNA-Seq Data
Eliah G. Overbey, Amanda M. Saravia-Butler, Zhe Zhang, Komal S. Rathi, Homer Fogle, Willian A. da Silveira, Richard J. Barker, Joseph J. Bass, Afshin Beheshti, Daniel C. Berrios, Elizabeth A. Blaber, Egle Cekanaviciute, Helio A. Costa, Laurence B. Davin, Kathleen M. Fisch, Samrawit G. Gebre, Matthew Geniza, Rachel Gilbert, Simon Gilroy, Gary Hardiman, Raúl Herranz, Yared H. Kidane, Colin P.S. Kruse, Michael D. Lee, Ted Liefeld, Norman G. Lewis, J. Tyson McDonald, Robert Meller, Tejaswini Mishra, Imara Y. Perera, Shayoni Ray, Sigrid S. Reinsch, Sara Brin Rosenthal, Michael Strong, Nathaniel J Szewczyk, Candice G.T. Tahimic, Deanne M. Taylor, Joshua P. Vandenbrink, Alicia Villacampa, Silvio Weging, Chris Wolverton, Sarah E. Wyatt, Luis Zea, Sylvain V. Costes, Jonathan M. Galazka
iScience, 2021
https://doi.org/10.1016/j.isci.2021.102361

Revamping Spaceomics in Europe
Pedro Madrigal, Alexander Gabel, Alicia Villacampa, Aránzazu Manzano, Colleen S Deane, Daniela Bezdan, Eugénie Carnero-Diaz, F. Javier Medina, Gary Hardiman, Ivo Grosse, Nathaniel Szewczyk, Silvio Weging, Stefania Giacomello, Stephen Harridge, Tessa Morris-Paterson, Thomas Cahill, Willian A. da Silveira, Raúl Herranz
Cell Systems, 2020
https://doi.org/10.1016/j.cels.2020.10.006

Elevated virulence of an emerging viral genotype as a driver of honeybee loss
McMahon Dino P., Natsopoulou Myrsini E., Doublet Vincent, Fürst Matthias, Weging Silvio, Brown Mark J. F., Gogol-Döring Andreas and Paxton Robert J.
Proceedings of the Royal Society B, 2016
https://doi.org/10.1098/rspb.2016.0811

Research experience

Consortium Member
Space omics ESA topical team
2019 – Present
`https://issop.space/space-omics-topical-team/`

Consortium Member
NASA GeneLab AWG microbes group for the analysis of space-related data
2018 – Present
`https://genelab.nasa.gov/`

Research assistant
Martin-Luther University Halle-Wittenberg
Faculty of Science III - Department of Computer Science
2016-05 – 2021-09
`https://www.informatik.uni-halle.de/arbeitsgruppen/bioinformatik/mitarbeiterinnen/weging/`

Research assistant
German Centre for Integrative Biodiversity Research (iDiv) Leipzig
Bioinformatics Group
08-2014 – 06-2016
`https://www.idiv.de/de/index.html`

Research assistant
Leipzig University of Applied Sciences (HTWK)
Laboratory for Biosignal Processing
07-2015 – 12-2015
`https://labp.github.io/`

Teaching experience

Teaching during PhD 2016-2021
Basics of Bioinformatics (BA)
Data Structures and Efficient Algorithms (BA)
Mathematical Foundations of Computer Science and Concepts of Modeling (BA)
Object-Oriented Programming (BA)
Statistical Data Analysis (BA)
Algorithms on Sequences II (MA)
Biological Networks (MA)
Median student rating: 1.3, Average: 1.35

Talks given	**KoBIS TH Mittelhessen** March 2019 Topic: Taxonomic Analysis of Metagenomic Data on a Notebook
	Mittelerde Meeting HS Mittweida June 2018 Topic: kASA - k-Mer Analysis of Sequences based on Amino acids
Skills	**Programming** Proficient in: C++, Python, Shell Familiar with: C#, Java
	Languages German (mother tongue), English (fluent)
Other interests and hobbies	Pen & Paper, Bouldering, Computer Games, Cycling, Concerts