

SNELLE SEMI-EXACTE MATCHING **VAN PEPTIDEN MET EEN GEHEUGENEFFICIËNTE** **INDEX VOOR UNIPROTKB**

Bram Devlaminck
Studentennummer: 01902993

Promotoren: prof. dr. Peter Dawyndt, prof. dr. ir. Bart Mesuere
Begeleiders: dr. Pieter Verschaffelt, Tibo Vande Moortele

Masterproef ingediend tot het behalen van de academische graad van
Master of Science in de Informatica

Academiejaar: 2023–2024

SAMENVATTING

Eiwitten zijn alomtegenwoordig en spelen een belangrijke rol in ons dagelijkse leven. Ze garanderen een correcte werking van belangrijke processen binnen elk organisme. Hieronder vallen de levensprocessen van ons eigen lichaam, maar ook die van dieren, planten, bacteriën en zelfs virussen. Dit verklaart het bestaan van gespecialiseerde databanken zoals UniProtKB, die voor miljoenen eiwitten bijhouden door welke organismen ze geproduceerd kunnen worden en voor welke functies ze kunnen instaan. Gebruikmakende van deze databanken probeert men voor eiwitstalen te achterhalen welke organismen aanwezig zijn in een ecosysteem en wat ze doen. Er is echter een extra stap nodig. De toestellen die eiwitten inlezen (massaspectrometers), zodat ze later omgezet kunnen worden naar een tekstueel formaat, kunnen dit slechts voor korte fragmenten, ook wel peptiden genoemd. Daarom worden eerst proteases (eiwitafbrekende enzymen) aan een staal toegevoegd. Deze zullen de eiwitten “knippen”, waarna we peptiden bekomen.

Omdat alle eiwitten in een dataset verknipt zijn in peptiden, is er geen zekerheid meer over welke eiwitten er exact aanwezig zijn. Hier komen tools zoals Unipept van pas om te zoeken uit welke mogelijke eiwitten een peptide afkomstig kan zijn. Op basis van de gevonden eiwitten kan daarna afgeleid worden welke organismen mogelijk aanwezig zijn, en wat er op dat moment gebeurt in het staal. De precisie van deze conclusie varieert van peptide tot peptide. Sommige peptiden bestaan uit een uniek patroon, waardoor ze slechts één mogelijk eiwit kunnen komen. Andere peptiden komen dan weer erg veel voor, waardoor ze deel uitmaken van duizenden eiwitten.

Zoals eerder vermeld, is Unipept één van de tools die onderzoekers kan helpen bij het vinden van de eiwitten waarvan een peptide onderdeel is. Bijkomend geeft Unipept ook informatie over welke organismen die eiwitten kunnen produceren en welke functies ze mogelijk uitvoeren. Unipept heeft op dit moment echter één grote beperking. Enkel peptiden die ontstaan zijn na het knippen van de eiwitten door de protease trypsine kunnen verwerkt worden. Dit was tot nu toe geen grote beperking omdat trypsine in de praktijk veruit de meest gebruikte protease is. Daar komt echter stilaan verandering in. Daarom is ons doel is om Unipept breder toepasbaar te maken, bijvoorbeeld voor HLA-peptiden. Deze spelen een belangrijke rol in onderzoek naar kankervaccins. Het wegwerken van deze beperking, waardoor alle soorten peptiden verwerkt kunnen worden, is de hoofdfocus van deze masterproef.

Om efficiënt alle eiwitten van UniProtKB te doorzoeken, maakt onze oplossing gebruik van een *suffix array*. Deze datastructuur bestaat uit één lijst die de startpositie van elke mogelijke suffix van alle eiwitten voorstelt. Een suffix van een woord is een deel van het woord, waarbij de eerste (0 of meerdere) tekens overgeslagen worden. Het eiwit ACGT heeft bijvoorbeeld vier verschillende suffixen: ACGT, CGT, GT en T. Deze lijstvoorstelling is een afweging tussen het bijhouden van alle nuttige informatie, en de snelheid waarmee peptiden gezocht kunnen worden. Er bestaan nog andere voorstellingen waarmee het zoeken sneller wordt, maar deze gebruiken meer geheugen. Een groot deel van deze masterproef gaat over het vinden van deze balans tussen geheugengebruik en zoeksnelheid.

Uiteindelijk hadden we 740 GB RAM nodig op de supercomputer van UGent om een suffix array voor UniProtKB op te bouwen. Om de resulterende index op een server met minder geheugen beschikbaar te maken, worden er slechts fragmenten van de volledige suffix array opgeslagen. Zo zijn we erin geslaagd om een nieuwe indexstructuur ter grootte van 335 GB publiek te stellen op de Unipept servers. Gebruikmakende van de nieuwe aanpassingen kan Unipept in minder dan 5 minuten voor 100 000 peptiden vinden in welke eiwitten ze voorkomen. Dat is erg snel: gelijkaardige *state-of-the-art* tools hebben enkele seconden tot minuten nodig om één peptide te zoeken.

Unipept 6.0: Fast Semi-exact Peptide Matching With a Memory Conservative Index for UniProtKB

Bram Devlaminck

Abstract

Unipept is an ecosystem of software tools developed for fast metaproteomics data analysis. In the past few years, Unipept has been optimized for tryptic peptide search, with an inefficient workaround for peptides with missed cleavages. However, newer fields such as immunopeptidomics work with non-trypic peptides. Unipept 6.0 introduces a new index structure that allows ultra-fast peptide matching for arbitrary peptides. This makes Unipept applicable in domains where trypsin is not the protease of choice, and also removes the performance penalty when dealing with missed cleavages. With these advancements, Unipept 6.0 is 10 to 100 times faster than Unipept 5.x when handling tryptic peptides with missed cleavages. In addition to these advancements, Unipept 6.0 maintains a superior performance compared to similar tools, ensuring efficient and accurate metaproteomics data analysis.

1 Introduction

Metaproteomics research has been invaluable in gaining a better understanding of the function of each organism in a microbiome ecosystem [1]. This started with small-scale experiments from as little as three proteins [2], but has evolved over the years to experiments with more than 100 000 peptides [3], [4]. Traditionally, trypsin is the most used protease in the (meta)proteomics field. Unipept [5]–[12], one of the first major tools to support metaproteomics data analysis, was specifically developed with this in mind. As a result, Unipept uses an index structure where the UniProtKB [13] database is preprocessed by splitting every protein according to the cleavage pattern created by trypsin. This allows Unipept to precompute the Lowest Common Ancestor (LCA) for every tryptic peptide digestion from UniProtKB. This approach has proven its efficiency in the past few years, but also has its limitations.

From time to time, trypsin will miss a cleavage site, creating a so-called missed cleavage. Since Unipept assumes there are no missed cleavages, peptides with such missed cleavages will not be present in the current Unipept index. To solve this problem, a solution was retro-fitted to handle these missed cleavages. When this option is enabled, Unipept scans the input peptides for missed cleavage positions, splits the peptides and perform separate lookups for every tryptic fragment. The resulting sets of protein matches are intersected, which delivers a set of possible protein matches. Every protein in this set is scanned left-to-right to ensure that the original pep-

tide is present, and not only its separate tryptic fragments. Finally, the LCA of the found proteins is calculated. All these extra steps, including the on-the-fly calculation of the LCA, have a significant impact on performance.

A second disadvantage of the current Unipept index, is the inability to process non-trypic peptides. Similar to peptides with missed cleavages, these are not present in the index and no efficient workaround exists for this problem.

This restriction was initially not a significant issue since Unipept was originally created for metaproteomics, where trypsin is widely used. However, other fields such as immunopeptidomics [14] require analysis tools to be able to cope with non-trypic peptides. Upstream analysis software such as MS²Rescore [15] and MS²PIP [16] have recently been adding support for immunopeptidomics research. By adding support for arbitrary peptide matching, Unipept will also automatically support these newer research fields. As an additional benefit, support for arbitrary peptide matching also removes the performance penalty associated with handling missed cleavages. This requires a new index structure with the following properties:

1. For every peptide (tryptic or non-trypic), all protein matches in UniProtKB should be found.
2. The maximum memory usage when building the index for the complete UniProtKB database should be around 1 to 2 TB due to current hardware limitations.

3. Search performance should be on par with Unipept 5.x for tryptic peptides.
4. The new index should facilitate all Unipept 5.x analyses.
5. Isoleucine and leucine should be equatable during search.

In short, we want an index that does not remove any of the features from Unipept 5.x, while eliminating the restriction that only tryptic peptides can be found efficiently.

2 Methods

At the core of our new index structure lies a suffix array. This suffix array is a data structure that can be built using the libdivsufsort [17] or libsaas [18] library. Both provide a linear time algorithm with a $5n + O(1)$ memory complexity, where n is the size of the text that needs to be indexed. Depending on the used text, one implementation is faster than the other.

2.1 Sparse Suffix Arrays

While a complete suffix array delivers the best performance, the index itself is large. The size of the suffix array can be reduced by introducing a sampling step to create a so-called sparse suffix array (SSA). This variant of a suffix array only stores every k -th suffix of the input text. This results in an SSA which is only $\frac{1}{k}$ of the original suffix array size. The disadvantage of using an SSA is that it becomes impossible to search for peptides having less than k amino acids, and that searching in general becomes slower. This trade-off does not introduce any problem, since Unipept already limits the search to peptides with 5 or more amino acids. This restriction was introduced since mass spectrometers can't read peptides with less than 5 amino acids. Furthermore, such extremely short peptides occur in a lot of proteins. Subsequently, this yields extremely generic functional and taxonomic analysis results, which is not very informative.

2.2 Equating Isoleucine and Leucine

Our solution to equate isoleucine and leucine consists of two parts. Firstly, we don't index the original text, but a slightly modified version. We replace every occurrence of leucine by isoleucine, which essentially equates them during construction of the suffix array. During search, we perform the same translation on every peptide. As a result, we find every match with I and L equated. This search is performed using the

original, unmodified text. This is important, since the original text is needed in an additional second step if we don't want to equate I and L.

In this second step, we filter away unwanted matches from the first step. This is achieved by checking every I and L location in the original, unmodified peptide with the corresponding amino acid in the original, unmodified text. A mismatch indicates that I and L were wrongfully equated. Only the I and L locations have to be checked since the suffix array search already ensures that all other characters match in the original text.

2.3 Analyses

Our solution performs the functional and taxonomic analysis at runtime. While it would be advantageous to precompute this, the suffix array and text do not contain enough information to do this. This is a trade-off we made during development between optimal memory usage and speed.

2.4 Maximum Matches

A single peptide can occur in thousands of proteins. Because of this, performing the analyses at runtime can become slow. Furthermore, the analyses for such peptides yields a generic result in most cases. More precisely, earlier research by the Unipept team [19] has shown that out the 1.3 billion tryptic peptides in UniProtKB 2023_03, only around 13 000 had more than 10 000 protein matches. From these 13 000 peptides, 95% of the peptides had `root` as LCA. This allows, with minimal loss of information, to assume that every peptide with more than 10 000 matches has `root` as LCA. Additionally, this limits the amount of information returned per query.

3 Results

Three main factors are considered during testing. The memory usage and speed during construction, the resulting index size while hosting, and search performance.

3.1 Constructing the Index

Constructing a suffix array for UniProtKB 2024_01 requires around 735 GB of RAM and 5 hours of computing time using the libdivsufsort library. Because of the high memory needs, we use the HPC of Ghent University, where the high-memory cluster has 16 nodes with each 2×64-core AMD EPYC 7773X (Milan-X @ 2.2 GHz) and 940 GiB of RAM. We use one core of a single node, since the construction phase is single threaded. The resulting suffix array is around 700 GB in size (+ an additional 88

GB for the text). Our goal is to host this new index on the UniPept servers, which have around 0.5 TB of memory available. To make this possible, we immediately perform a sampling phase with sparseness factor $k = 3$ at the end of the construction process on the HPC. This results in a reduced index size of $\frac{700}{3} + 88 \approx 322$ GB. Next to this peptide search index, UniPept needs extra information (such as the UniProt accession number, NCBI taxon ID and functional annotations per protein) to perform the provided analyses. This results in another 25 GB of RAM needed. Figure 1 visualizes the size of each component of the resulting index.

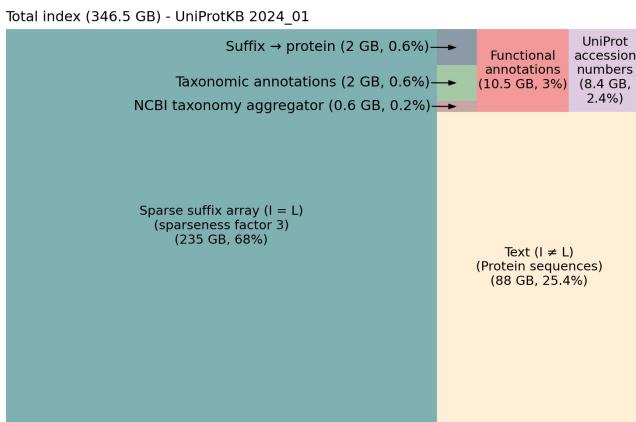


Figure 1: Visualisation of the size of each component of UniPept 6.x index for UniProtKB 2024_01.

3.2 Querying the Index

UniPept 6.x supports two types of search. Most importantly, we can efficiently search all matches and perform the taxonomic and functional aggregation at runtime. Since the analysis is executed at runtime, we already have all the information per matched protein. With UniPept 6.x we provide search without aggregation as a separate option. This was not possible with UniPept 5.x since the aggregations were pre-computed and retrieving all information from disk per protein was too slow. Both search options in UniPept 6.x perform similar. Albeit that the first option is slightly faster. This is caused by the additional information per protein that needs to be serialized, since the functional annotations are not aggregated. In this section we only discuss the search with analyses, since this is what UniPept 5.x supports and the other search option performs very similar.

Missed cleavages To evaluate the peptide search performance we use 6 files containing around 25 000 peptides each. These are samples from the SIHUMIx experiment [20], [21], where trypsin is used as a protease. This means that as well as tryptic peptides, there are also peptides present with naturally occur-

ring missed cleavages. Figure 2 shows the execution time for both UniPept 6.x and 5.x. To make the comparison as fair as possible, the *filter duplicate peptides* setting is turned off, and *advanced missed cleavage handling* is turned on. This ensures that both indices search every peptide from the input file (including duplicates and peptides with missed cleavages). UniPept 6.x is 10 to 100 times faster, which clearly removes the performance penalty that was currently associated with handling missed cleavages. Note that the UniPept 5.x index will not find peptides resulting from using a different protease, whereas this makes no difference for the UniPept 6.x index.

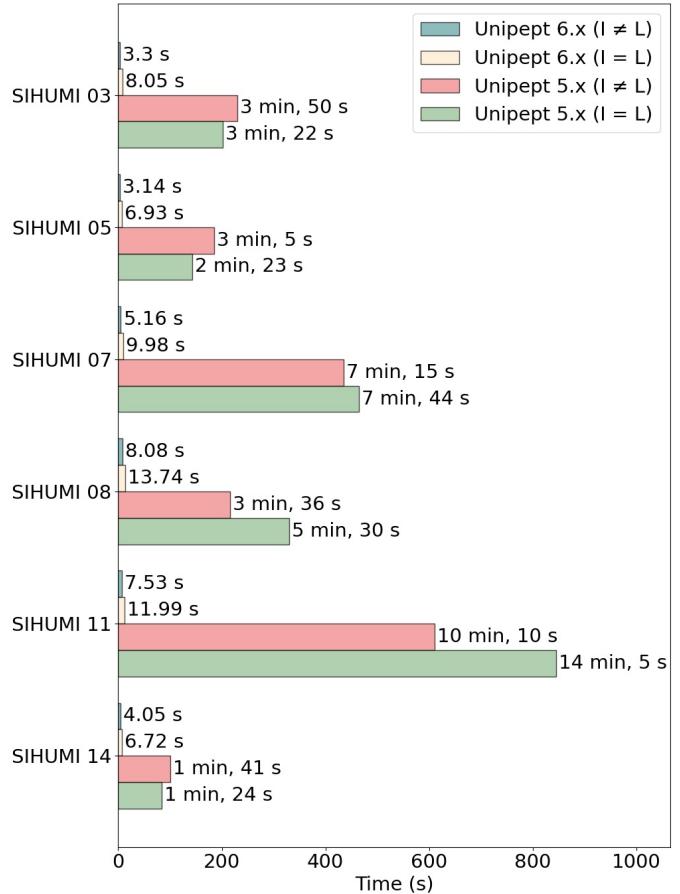


Figure 2: Execution time for UniPept 6.x and UniPept 5.x while processing different SIHUMIx samples. Each sample consists of approximately 25 000 peptides. UniPept 5.x searches with the *filter duplicate peptides* setting turned off, and *advanced missed cleavage handling* turned on. This way both indices search all peptides (included duplicates), and missed cleavages are handled by both. The used test files can be found in our GitHub repository https://github.com/BramDevlaeminck/Thesis_benchmarkdata/tree/master/SIHUMI

Tryptic peptides A second interesting case to consider is when searching strictly tryptic peptides. In that case, missed cleavage handling can be disabled in UniPept 5.x, which significantly improves the performance. Figure 3 visualizes the processing time for 100 000 tryptic peptides. It is clear that the performance is comparable when I and L are

equated, while UniPept 5.x is around a third faster than UniPept 6.x when I and L are not equated.

Furthermore, it is also clear that the extra filtering step when $I \neq L$, has a performance impact in the new index. This effect was not visible in Figure 2, where the reverse seems to apply. However, this reverse effect can be explained by the fact that searching with $I = L$ results in more matches, which requires more output to be serialized to a JSON file. When searching for larger files, there is proportionally more time spent in the search phase, which compensates for the longer serialization time.

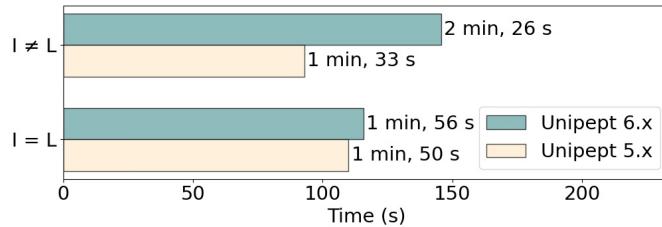


Figure 3: Processing time for 100 000 tryptic peptides for UniPept 6.x and 5.x. This is the best case scenario for UniPept 5.x where missed cleavages are **not** handled.

4 Other Tools

Tools such as the Uniprot peptide search tool [22], [23], the Expasy ScanProsite tool [24] and UniPept 5.x all have the possibility to find matches in the UniProtKB database. In this section we compare the performance and feature sets of alternative tools with UniPept 6.x. Because of the poor performance of some of these tools, we limited testing to the randomly chosen tryptic peptide ISPAVLFVIVILAVLFFISGLLHLLVR. UniPept 6.x uses above settings and finds all matches for the test peptide in less than 5 ms.

In terms of features, the UniProt peptide search tool is identical to UniPept 6.x. They both find all matches in UniProtKB, and have the option to equate I and L. The only difference is their performance. Searching the test peptide in UniPept 6.x only takes a few milliseconds, whereas the UniProt tool takes a few seconds up to multiple minutes.

The Expasy ScanProsite tool takes another approach. They provide a wide range of options for inexact matching. They call these motifs, and these are comparable to regular expressions where the user can use wildcards, character classes and even negations. The other major difference is that this tool does not use the whole UniProtKB database. Only the proteins that are part of a reference genome are indexed, which means only around one third of the complete UniProtKB database is indexed. Lastly, searching the test peptide takes around 5 minutes. As

expected, since not the whole UniProtKB database is indexed, only a subset of the matches is found.

The last major tool we compare with, is UniPept 5.x. UniPept 6.x supports all options provided by UniPept 5.x, while removing the restriction that only tryptic peptides can be searched. Searching the test peptide took less than 5 ms, and found the same matches because our test peptide is tryptic. Table 1 gives an overview of the described differences between the tools.

	UP6	UPS	ESP	UP5
Used proteins	all	all	ref. prot.	all
Approx. match	[IL]	[IL]	flexible	[IL]
Time	< 5 ms	1 s - 20 min	5 min	< 5 ms
Searchable pept.	all	all	all	~ tryptic

Table 1: Comparison of UniPept 6.x (UP6), UniProt peptide search (UPS) tool, Expasy ScanProsite (ESP) tool and UniPept 5.x (UP5). [IL] in the approximate matching row means that only I and L can be equated, while \sim tryptic in the searchable peptide row means that only tryptic peptides or tryptic peptides with missed cleavages can be found.

5 Conclusion

With a new index structure at the heart of UniPept 6.x, we have broadened UniPept’s possible use cases. The new index structure makes searching peptides with missed cleavages around 10 to 100 times faster, while adding the possibility to search arbitrary peptides, regardless of the used protease, without any performance penalty. In our benchmarks, UniPept 6.x strongly outperforms its closest competitors at finding all occurrences for an arbitrary peptide longer than 2 amino acids in UniProtKB.

Our new index is optimized for searches with leucine and isoleucine equated to each other. This corresponds to the default search configuration of UniPept. However, searching with $I \neq L$ only introduces a small performance penalty. For smaller batches of peptides, the time needed to serialize the extra matches even outweighs the extra time spent during the search phase itself.

Another advantage of the new index is that there are no extra steps required to retrieve the NCBI taxon ID for each matched protein. Extra steps are required in UniPept 5.x to retrieve all the individual taxon IDs, with a significant impact on the performance. This restriction created a bottleneck in the new Peptonizer2000 tool [25] that has now been removed.

The main disadvantage of UniPept 6.x compared to UniPept 5.x is that searching tryptic peptides is slightly slower than before, especially when searching with $I \neq L$. This slowdown is introduced by taxonomic analyses performed at runtime, whereas

these could all be precalculated with the old index that was restricted to tryptic peptides. However, this slowdown is limited and acceptable. Especially when keeping in mind that samples from experiments always have some missed cleavages, which would not have been found before, without a performance hit.

6 Future Work

The UniPept 6.x index meets the pre-established requirements, but still leaves room for improvement in several areas.

A significant part of the current computation time is invested in performing the taxonomic and functional analyses at runtime. Modifying the new index to support precalculation of these analyses, while still maintaining a peak memory usage which is manageable, would drastically improve the performance. Enhanced suffix arrays (ESA) could facilitate this, but require more memory.

Another area of improvement is the index size itself. The index size could be reduced even more by making the text and suffix array more compact. Both of these components don't utilize all the bits from the allocated bytes. The suffix array only needs 37 of the 64 bits used by every entry, while the text only needs 5 bits out of each byte per character. This could reduce the total index size from 346 GB to around 215 GB. However, this would introduce extra steps to decode every access to any of these data structures. This could introduce a non-negligible performance penalty.

Another option to reduce the index size is by switching to a completely different index structure. Both the FM-index [26] and R-index [27] have shown promising results in this respect.

Lastly, UniPept does not perform any form of inexact matching, except for equating I and L. Introducing inexact matching into UniPept could allow us to deal with small biological mutations or variations introduced during the transformation from mass spectrum to peptide. Possible interesting routes to explore are regex matching using suffix arrays [28] or inexact matching using bidirectional FM-indices [29].

References

- [1] T. Van Den Bossche, M. Ø. Arntzen, D. Becher, *et al.*, “The Metaproteomics Initiative: a coordinated approach for propelling the functional characterization of microbiomes,” *Microbiome*, vol. 9, no. 1, p. 243, Dec. 20, 2021, ISSN: 2049-2618. DOI: [10.1186/s40168-021-01176-w](https://doi.org/10.1186/s40168-021-01176-w). [Online]. Available: <https://doi.org/10.1186/s40168-021-01176-w>.
- [2] R. J. Ram, N. C. VerBerkmoes, M. P. Thelen, *et al.*, “Community Proteomics of a Natural Microbial Biofilm,” *Science*, vol. 308, no. 5730, pp. 1915–1920, 2005. DOI: [10.1126/science.1109070](https://doi.org/10.1126/science.1109070). eprint: <https://www.science.org/doi/pdf/10.1126/science.1109070>. [Online]. Available: <https://www.science.org/doi/abs/10.1126/science.1109070>.
- [3] X. Zhang and D. Figgeys, “Perspective and Guidelines for Metaproteomics in Microbiome Studies,” *Journal of Proteome Research*, vol. 18, no. 6, pp. 2370–2380, 2019, PMID: 31009573. DOI: [10.1021/acs.jproteome.9b00054](https://doi.org/10.1021/acs.jproteome.9b00054). eprint: <https://doi.org/10.1021/acs.jproteome.9b00054>. [Online]. Available: <https://doi.org/10.1021/acs.jproteome.9b00054>.
- [4] T. Van Den Bossche, B. J. Kunath, K. Schallert, *et al.*, “Critical assessment of MetaProteome investigation (CAMPI): A multi-laboratory comparison of established workflows,” *Nature Communications*, vol. 12, no. 1, p. 7305, Dec. 15, 2021, ISSN: 2041-1723. DOI: [10.1038/s41467-021-27542-8](https://doi.org/10.1038/s41467-021-27542-8). [Online]. Available: <https://doi.org/10.1038/s41467-021-27542-8>.
- [5] P. Verschaffelt, T. Van Den Bossche, L. Martens, P. Dawyndt, and B. Mesuere, “UniPept Desktop: A Faster, More Powerful Metaproteomics Results Analysis Tool,” *Journal of Proteome Research*, vol. 20, no. 4, pp. 2005–2009, Jan. 2021, ISSN: 1535-3907. DOI: [10.1021/acs.jproteome.0c00855](https://doi.org/10.1021/acs.jproteome.0c00855). [Online]. Available: <http://dx.doi.org/10.1021/acs.jproteome.0c00855>.
- [6] B. Mesuere, T. Willems, F. Van der Jeugt, B. Devreese, P. Vandamme, and P. Dawyndt, “UniPept web services for metaproteomics analysis,” *Bioinformatics*, vol. 32, no. 11, pp. 1746–1748, Jan. 2016, ISSN: 1367-4803. DOI: [10.1093/bioinformatics/btw039](https://doi.org/10.1093/bioinformatics/btw039). [Online]. Available: <http://dx.doi.org/10.1093/bioinformatics/btw039>.
- [7] R. Gurdeep Singh, A. Tanca, A. Palomba, *et al.*, “UniPept 4.0: Functional Analysis of Metaproteome Data,” *Journal of Proteome Research*, vol. 18, no. 2, pp. 606–615, Nov. 2018, ISSN: 1535-3907. DOI: [10.1021/acs.jproteome.8b00716](https://doi.org/10.1021/acs.jproteome.8b00716). [Online]. Available: <http://dx.doi.org/10.1021/acs.jproteome.8b00716>.
- [8] B. Mesuere, B. Devreese, G. Debyser, M. Aerts, P. Vandamme, and P. Dawyndt, “UniPept: Tryptic Peptide-Based Biodiversity Analysis of Metaproteome Samples,” *Journal of Proteome Research*, vol. 11, no. 12, pp. 5773–5780, Nov. 2012, ISSN: 1535-3907. DOI: [10.1021/pr300576s](https://doi.org/10.1021/pr300576s). [Online]. Available: <http://dx.doi.org/10.1021/pr300576s>.
- [9] B. Mesuere, F. Van der Jeugt, T. Willems, *et al.*, “High-throughput metaproteomics data analysis with UniPept: A tutorial,” *Journal of Proteomics*, vol. 171, pp. 11–22, Jan. 2018, ISSN: 1874-3919. DOI: [10.1016/j.jprot.2017.05.022](https://doi.org/10.1016/j.jprot.2017.05.022). [Online]. Available: <http://dx.doi.org/10.1016/j.jprot.2017.05.022>.
- [10] B. Mesuere, G. Debyser, M. Aerts, B. Devreese, P. Vandamme, and P. Dawyndt, “The UniPept metaproteomics analysis pipeline,” *PROTEOMICS*, vol. 15, no. 8, pp. 1437–1442, Feb. 2015, ISSN: 1615-9861. DOI: [10.1002/pmic.201400361](https://doi.org/10.1002/pmic.201400361). [Online]. Available: <http://dx.doi.org/10.1002/pmic.201400361>.
- [11] P. Verschaffelt, P. Van Thienen, T. Van Den Bossche, *et al.*, “UniPept CLI 2.0: adding support for visualizations and functional annotations,” *Bioinformatics*, vol. 36, no. 14, P. Luigi Martelli, Ed., pp. 4220–4221, Jun. 2020, ISSN: 1367-4811. DOI: [10.1093/bioinformatics/btaa600](https://doi.org/10.1093/bioinformatics/btaa600).

- [1] btaa553. [Online]. Available: <http://dx.doi.org/10.1093/bioinformatics/btaa553>.
- [2] P. Verschaffelt, A. Tanca, M. Abbondio, *et al.*, “UniPept Desktop 2.0: Construction of Targeted Reference Protein Databases for Metaproteogenomics Analyses,” *Journal of Proteome Research*, vol. 22, no. 8, pp. 2620–2628, Jul. 2023, ISSN: 1535-3907. DOI: [10.1021/acs.jproteome.3c00091](https://doi.org/10.1021/acs.jproteome.3c00091). [Online]. Available: <http://dx.doi.org/10.1021/acs.jproteome.3c00091>.
- [3] The UniProt Consortium, “UniProt: the Universal Protein Knowledgebase in 2023,” *Nucleic Acids Research*, vol. 51, no. D1, pp. D523–D531, Nov. 2022, ISSN: 0305-1048. DOI: [10.1093/nar/gkac1052](https://doi.org/10.1093/nar/gkac1052). eprint: <https://academic.oup.com/nar/article-pdf/51/D1/D523/48441158/gkac1052.pdf>. [Online]. Available: <https://doi.org/10.1093/nar/gkac1052>.
- [4] R. L. Mayer and K. Mechtler, “Immunopeptidomics in the Era of Single-Cell Proteomics,” *Biology*, vol. 12, no. 12, 2023, ISSN: 2079-7737. DOI: [10.3390/biology12121514](https://doi.org/10.3390/biology12121514). [Online]. Available: <https://www.mdpi.com/2079-7737/12/12/1514>.
- [5] A. Declercq, R. Bouwmeester, A. Hirschler, *et al.*, “MS²Rescore: Data-Driven Rescoring Dramatically Boosts Immunopeptide Identification Rates,” *Molecular & Cellular Proteomics*, vol. 21, no. 8, p. 100266, Aug. 2022, ISSN: 1535-9476. DOI: [10.1016/j.mcpro.2022.100266](https://doi.org/10.1016/j.mcpro.2022.100266). [Online]. Available: <http://dx.doi.org/10.1016/j.mcpro.2022.100266>.
- [6] A. Declercq, R. Bouwmeester, C. Chiva, *et al.*, “Updated MS²PIP web server supports cutting-edge proteomics applications,” *Nucleic Acids Research*, vol. 51, no. W1, W338–W342, May 2023, ISSN: 1362-4962. DOI: [10.1093/nar/gkad335](https://doi.org/10.1093/nar/gkad335). [Online]. Available: <http://dx.doi.org/10.1093/nar/gkad335>.
- [7] Y. Mori, *Libdivsufsort*, <https://github.com/y-256/libdivsufsort>.
- [8] I. Grebnov, *Libsais*, <https://github.com/IlyaGrebnov/libsais>.
- [9] *UniPept: Important statistics*, <https://github.com/unipept/unipept/wiki/important-statistics>.
- [10] T. Van Den Bossche, B. J. Kunath, K. Schallert, *et al.*, “Critical assessment of MetaProteome investigation (CAMPPI): A multi-laboratory comparison of established workflows,” en, *Nat. Commun.*, vol. 12, no. 1, p. 7305, Dec. 2021.
- [11] J. L. Krause, S. S. Schaepe, K. Fritz-Wallace, *et al.*, “Following the community development of SIHUMix – a new intestinal in vitro model for bioreactor use,” *Gut Microbes*, vol. 11, no. 4, pp. 1116–1129, 2020, PMID: 31918607. DOI: [10.1080/19490976.2019.1702431](https://doi.org/10.1080/19490976.2019.1702431). eprint: <https://doi.org/10.1080/19490976.2019.1702431>. [Online]. Available: <https://doi.org/10.1080/19490976.2019.1702431>.
- [12] UniProt peptide search tool, <https://www.uniprot.org/peptide-search>.
- [13] C. Chen, Z. Li, H. Huang, B. E. Suzek, and C. H. Wu, “A fast Peptide Match service for UniProt Knowledgebase,” *Bioinformatics*, vol. 29, no. 21, pp. 2808–2809, Aug. 2013, ISSN: 1367-4811. DOI: [10.1093/bioinformatics/btt484](https://doi.org/10.1093/bioinformatics/btt484). [Online]. Available: <http://dx.doi.org/10.1093/bioinformatics/btt484>.
- [14] Expasy ScanProsite tool, <https://prosite.expasy.org/scanprosite/>.
- [15] T. Holstein, F. Kistner, L. Martens, and T. Muth, “PepGM: a probabilistic graphical model for taxonomic inference of viral proteome samples with associated confidence scores,” *Bioinformatics*, vol. 39, no. 5, btad289, May 2023, ISSN: 1367-4811. DOI: [10.1093/bioinformatics/btad289](https://doi.org/10.1093/bioinformatics/btad289). eprint: <https://academic.oup.com/bioinformatics/article-pdf/39/5/btad289/50311697/btad289.pdf>. [Online]. Available: <https://doi.org/10.1093/bioinformatics/btad289>.
- [16] P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, Nov. 2000, pp. 390–398. DOI: [10.1109/SFCS.2000.892127](https://doi.org/10.1109/SFCS.2000.892127).
- [17] T. Gagie, G. Navarro, and N. Prezza, “Optimal-Time Text Indexing in BWT-runs Bounded Space,” in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Jan. 2018, pp. 1459–1477. DOI: [10.1137/1.9781611975031.96](https://doi.org/10.1137/1.9781611975031.96). [Online]. Available: <http://dx.doi.org/10.1137/1.9781611975031.96>.
- [18] N. Elhage, *Regular Expression Search with Suffix Arrays*, <https://blog.nelhage.com/2015/02/regular-expression-search-with-suffix-arrays/>, Feb. 2015.
- [19] T. W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S. M. Yiu, “High Throughput Short Read Alignment via Bi-directional BWT,” in *2009 IEEE International Conference on Bioinformatics and Biomedicine*, 2009, pp. 31–36. DOI: [10.1109/BIBM.2009.42](https://doi.org/10.1109/BIBM.2009.42).

DANKWOORD

Deze masterproef vormt het sluitstuk van de voorbije vijf jaar aan UGent. Jaren met extreem veel werk, maar ook met fantastische ervaringen. Deze masterproef is de ultieme combinatie hiervan. Zonder het begeleidend team van UniPept, waarmee ik wekelijks onze vaste *thesismeeting* had, zou ik nooit zo'n volmaakt resultaat kunnen afleveren. Daarom wil ik met nadruk mijn promotor prof. dr. Peter Dawyndt, copromotor prof. dr. ir. Bart Mesuere en begeleiders dr. Pieter Verschaffelt en Tibo Vande Moortele bedanken. Naast alle serieuze en leervolle gesprekken over informatica en metaproteomica, was er altijd plaats voor enkele (of laat ons eerlijk zijn, veel) ongerelateerde side-stories.

Naast deze technische ondersteuning wil ik ook graag mijn ouders en broer bedanken die me de afgelopen vijf jaar bijgestaan hebben in alle stressvolle deadline- en examenperioden. Hun steun heeft deze perioden aanzienlijk draaglijker gemaakt. Graag wil ik nog eens afzonderlijk mijn vriendin Claudia bedanken. Dankzij haar bevat deze masterproef niet enkel minder typfouten en rare zinsconstructies, maar weet ik meer dan ooit dat ik er nooit alleen voor sta, ook in individuele opdrachten zoals deze.

Tot slot wil alle mensen uit de *thesiskelder* bedanken waarmee ik ontelbare uren in de donkerste krochten van S9 gespendeerd heb. In willekeurige volgorde: Maarten, Stijn, Thomas, Clement en Jonas. Samen hebben we niet enkel gevloekt op software bugs en vage L^AT_EX errors, maar ook onvergetelijke herinneringen gemaakt. Ik ben jullie eeuwig dankbaar voor het gezelschap en jullie input bij een zoveelste dilemma tijdens het maken van deze masterproef.

TOELATING TOT BRUIKLEEN

De auteur geeft de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de bepalingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef.

Bram Devlaminck

24 mei 2024

INHOUDSOPGAVE

1 Inleiding	1
1.1 Genomica, transcriptomica & proteomica	1
1.2 Tryptische peptiden	2
1.3 Unipept	3
1.4 Probleemstelling	4
1.5 Benchmarkdatasets	5
1.5.1 Proteïnedatabanken	5
1.5.2 Peptidebestanden	7
1.6 Benchmark hardware	10
1.7 Mogelijke oplossingen	10
2 Suffixbomen	13
2.1 Wat zijn suffixbomen?	13
2.2 Het algoritme van Ukkonen	15
2.2.1 Kotlin	15
2.2.2 Rust	15
2.2.3 Performantie	18
2.3 Taxon ID aggregatie	22
2.4 Conclusie	24
3 Suffix arrays	25
3.1 Wat zijn suffix arrays?	25
3.2 Complexiteit	25
3.3 Bestaande implementaties	26
3.3.1 Enhanced suffix arrays	26
3.4 Toepassen van suffix arrays op een proteïnedatabank	27
3.4.1 Bouwen van de suffix array	27
3.4.2 Mapping van suffix naar proteïne	27
3.4.3 Berekenen van de LCA	28
3.5 Sparse en compressed suffix arrays	29
3.5.1 Zoeken met sparse suffix arrays	30
3.6 Parallelisatie	32
3.6.1 Manueel threaden vs Rayon	33
3.7 Suffix arrays vs suffixbomen	34
3.7.1 Opbouwen	34
3.7.2 Zoeken	35
4 Andere indexstructuren	39
4.1 FM-index	39
4.1.1 Verschillende implementaties	39
4.2 R-index	40
5 Een nieuwe UniProtKB-index voor Unipept	43
5.1 Opbouwen van de SA	43
5.2 Een sparseness factor kiezen	44
5.3 Isoleucine en leucine gelijkstellen	44
5.3.1 Index waarbij $I \neq L$	45

5.3.2	Index waarbij $I = L$	46
5.4	Taxonomische analyse	48
5.5	Functionele analyse	48
5.6	Vergelijking met andere tools	48
5.6.1	UniProt peptide search tool	49
5.6.2	Expasy ScanProsite tool	50
5.6.3	Unipept 5.x	50
5.7	Overzicht geheugengebruik	52
5.8	Aanbieden van de nieuwe indexstructuur	53
5.9	Productiepijplijn	53
6	Conclusie & toekomstig werk	55
6.1	Conclusie	55
6.2	Toekomstig werk	55
Referenties		57
A	Statistieken peptidebestanden	61
A.1	Human-Prot	61
A.2	Swiss-Prot	62
A.3	SIHUMI	64
B	Unipept protein counts distribution	71

1 INLEIDING

Eiwitten of proteïnen zijn alomtegenwoordig en spelen een essentiële rol in ons dagelijkse leven. Ze garanderen de correcte werking van processen binnen ons eigen lichaam, maar ook in dieren, planten, bacteriën en zelfs virussen. Om deze processen te analyseren zijn er meerdere benaderingen mogelijk.

1.1 Genomica, transcriptomica & proteomica

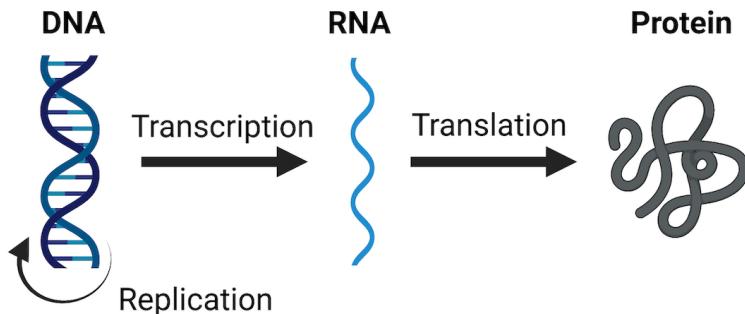
De eerste mogelijke discipline is **genomica**, het onderzoek naar het genoom. Het genoom bestaat uit al het DNA dat aanwezig is in een organisme. Het DNA bevat het “recept” voor alle proteïnen die mogelijk door het organisme geconstrueerd kunnen worden. Dit is het begin van het centraal dogma van de biologie. Dit centraal dogma beschrijft de omzetting van DNA naar proteïnen. Wanneer we gebruikmaken van een analogie, dan is het DNA een receptenboek en is een proteïne een afgewerkte gerecht (waarvoor de instructies in het receptenboek staan). Het DNA stelt dus de instructies voor van alle mogelijke proteïnen die een bepaald organisme kan maken. Het geeft dus geen informatie over de proteïnen die op dat moment in de tijd actief zijn. Het is een voorstelling van wat het organisme kan, niet wat het op *dit* moment aan het doen is. Belangrijk is dat ongeveer 98% van het menselijk genoom niet-coderend is. Dit wil zeggen dat het grootste deel van het DNA niet omgezet kan worden naar een betekenisvolle proteïne. In de plaats kunnen deze niet-coderende delen omgezet worden naar *regulatory sequences*, niet-coderende genen, of andere componenten waarvan we nog niet altijd weten of ze al dan niet een functie hebben.

De tweede discipline is de **transcriptomica**. Deze discipline onderzoekt het transcriptoom van een organisme. Dit is de verzameling van alle RNA moleculen die in het organisme aanwezig zijn. Het transcriptoom is een belangrijke indicator voor welke delen uit het DNA effectief proteïnen encoderen. Dit komt omdat RNA, meer specifiek messenger RNA (mRNA) en transfer RNA (tRNA), een belangrijk onderdeel zijn van het proces om DNA om te zetten naar proteïnen. In de restaurant analogie is dit een kopie van één recept uit het receptenboek. Dit recept beschrijft hoe het gerecht exact gemaakt moet worden.

Tot slot bestaat ook het onderzoeksgebied van de **proteomica**. Dit is de studie van alle proteïnen die binnen een enkel organisme tot expressie kunnen komen. Hierbij probeert men te begrijpen hoe proteïnen in elkaar zitten, hoe deze binnen een bepaalde omgeving met elkaar interageren en wat hun belangrijkste functie is. Het voordeel van het bestuderen van de proteïnen is dat je weet welke proteïnen exact op een specifiek moment aanwezig zijn. Op die manier kan men afleiden wat er exact gebeurt in een organisme. Zoals eerder beschreven zijn de proteïnen het afgewerkte gerecht in de restaurant analogie. Een visualisatie van het centraal dogma van de biologie, inclusief de link met de gebruikte analogie, kan teruggevonden worden in Figuur 1.1 [61].

De focus van deze masterproef ligt vooral binnen het veld van de **metaproteomica**. Het prefix *meta* geeft aan dat de te analyseren stalen niet van één organisme afkomstig zijn, maar van **meerdere organismen** (typisch binnen hetzelfde ecosysteem). Dit maakt de analyse moeilijker aangezien proteïnen van verschillende organismen gelijkaardige aminozuursequenties kunnen hebben (al dan niet door toeval). Binnen het metaproteomica veld gaan onderzoekers op zoek naar welke organismen er in een ecosysteem aanwezig zijn en wat deze daar dan doen. Dit doet men door een verzameling van proteïnefragmenten te analyseren. Deze proteïnefragmenten noemen we peptiden wanneer ze bestaan uit twee of meer aminozuren en hun lengte beperkt blijft. In de praktijk gaat dit over sequenties van ongeveer 2 tot 50 aminozuren lang. Idealiter kunnen we deze analyses uitvoeren

met volledige proteïnen, en niet met peptiden. Dit is echter niet mogelijk vanwege beperkingen bij de huidige generatie aan machines die proteïnen inlezen. Deze kunnen slechts korte fragmenten (peptiden) inlezen. Een veelvoorkomende categorie van peptiden die we zullen analyseren zijn tryptische peptiden.

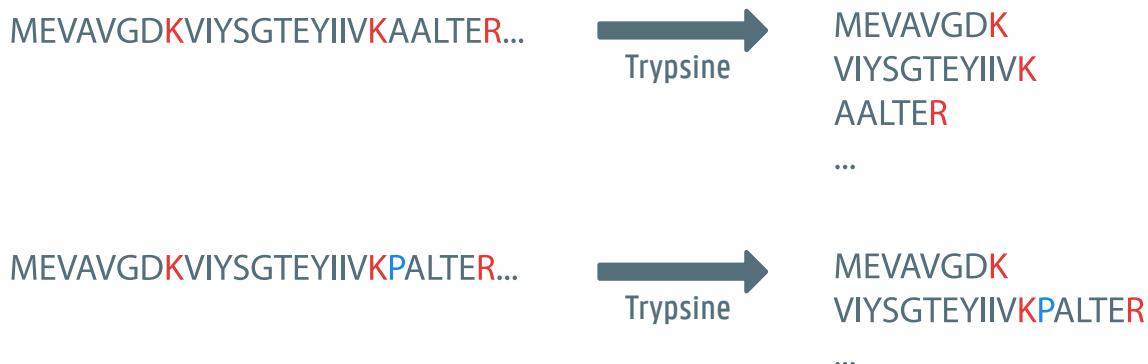


Figuur 1.1: Visualisatie van het centraal dogma van de biologie [61]. Gebruikmakende van een analogie kunnen we het DNA vergelijken met een receptenboek. Eén kopie van één recept komt overeen met RNA. Een afgewerkt gerecht komt dan weer overeen met een proteïne.

1.2 Tryptische peptiden

Tryptische peptiden ontstaan na het knippen van proteïnen aan de hand van **trypsine**. Dit is een protease (eiwitafbrekend enzym) dat proteïnen opsplitst in meerdere peptiden. Er bestaan nog andere proteases, maar trypsine is veruit de populairste door zijn eenduidig gedrag en efficiëntie.

Trypsine zal proteïnen knippen na elk voorkomen van lysine (K) of arginine (R) indien het eerstvolgende aminozuur geen proline (P) is. Deze vuistregel is echter niet perfect. Soms mist trypsine een locatie waar volgens deze regel geknipt moet worden. Dit noemen we een *missed cleavage*. Figuur 1.2 bevat een voorbeeld van de werking.



Figuur 1.2: Voorbeeld van de werking van trypsine op twee proteïnen [69]. De aminozuren in het rood zijn lysine (K) of arginine (R), waarna trypsine knipt (behalve als het eerstvolgende aminozuur proline (P) is). De tweede proteïne bevat een voorbeeld waar niet geknipt wordt na lysine, doordat het volgende aminozuur proline is.

Om de peptiden uit een experiment te kunnen gebruiken bij computeranalyses moeten deze omgezet worden naar een stringrepresentatie. Dit is echter een moeilijk en ingewikkeld proces. Eerst wordt de massa/ladingsverhouding (m/z) van peptiden aan de hand van een massaspectrometer gemeten. Daarna worden deze resultaten aan de hand van diverse zoekprocessen (via zogenaamde zoekmachines) omgezet naar de stringvoorstelling van de peptide. Deze sequenties vormen de input voor tools zoals Unipept.

Eén belangrijke consequentie van het gebruik van een massaspectrometer is dat isoleucine (I) en leucine (L) niet uit elkaar gehouden kunnen worden. Deze aminozuren bestaan uit dezelfde atomen ($C_6H_{13}NO_2$), maar hebben niet dezelfde structuur. Bijgevolg hebben ze een identieke massa, en zijn ze niet te onderscheiden van elkaar.

1.3 Unipept

Unipept [44] biedt een ecosysteem van tools aan om stalen uit het onderzoeksgebied van de metaproteomica te analyseren, maar er is ook een onderdeel (UMGAP [28]) gericht op het analyseren van stalen uit de metagenomica.

- **Unipept Web application** [44, 42, 41, 22] Dit is de originele Unipept tool en is publiek beschikbaar op <https://unipept.ugent.be>. Met de gebruiksvriendelijke *user interface* wordt het analyseren van metaproteomica data beschikbaar gesteld. De resultaten van deze analyse worden aan de hand van visualisaties en tabellen aan de gebruiker voorgesteld. Deze kunnen vervolgens makkelijk geëxporteerd worden (bijv. voor analyse in andere tools).
- **Unipept CLI¹** [66] Dit is een *power-user* tool om via de commandolijn analyses op grotere stalen uit te voeren en laat toe om de Unipept analyses in een grotere pijplijn op te nemen.
- **Unipept API²** [43, 66] Dit is een collectie van *endpoints* die andere applicaties (inclusief de Unipept CLI), toelaat om de functionaliteit van Unipept te integreren.
- **Unipept Desktop** [68, 67] Dit is de recentste toevoeging aan het Unipept ecosysteem en laat toe dat onderzoekers niet noodzakelijk met de Unipept servers moeten communiceren om analyses uit te voeren. Deze applicatie combineert de voordelen van de web app, CLI en API en laat toe om lokaal grote stalen te analyseren, gebruikmakende van een gebruiksvriendelijke UI.

Op dit moment is Unipept **exclusief gericht op de analyse van tryptische peptiden**. De huidige achterliggende indexstructuur is namelijk specifiek hiervoor ontworpen omdat tot op heden metaproteomica onderzoek bijna uitsluitend gebruikmaakt van tryptische peptiden. Het opbouwen van de huidige indexstructuur gaat in grote lijnen als volgt:

1. Haal alle proteïnen en bijbehorende taxonomische en functionele annotaties op uit de UniProt-KB databank [8].
2. Splits deze proteïnen volgens de vuistregel die trypsinen nabootst.
3. Sla alle resulterende tryptische peptiden op in een databank, samen met hun voorberekende taxonomische en functionele metadata.

Deze aanpak heeft als voordeel dat we op een efficiënte manier tryptische peptiden kunnen opzoeken (samen met de bijbehorende annotaties). Er is echter een belangrijke keurzijde aan deze manier van werken. Het zoeken van **niet-trypotische peptiden** (hieronder vallen ook peptiden met *missed cleavages*) is **problematisch**. Dit komt doordat tijdens het opbouwen van de Unipept indexstructuur de vuistregel strikt gevuld wordt, en elke peptide in de indexstructuur strikt tryptisch is. Op basis hiervan worden de taxonomische en functionele annotaties voor elke tryptische peptide voorberekend.

Op dit moment is er wel een **workaround** die toelaat om peptiden met *missed cleavages* toch op te zoeken. Hiervoor worden alle peptiden die een missed cleavage hebben opgesplitst, en worden de tryptische fragmenten apart gezocht. Daarna wordt de doorsnede genomen van de gematchte proteïnen per fragment, waarna we een verzameling van proteïnen bekomen die op zijn minst alle

¹command-line interface

²application programming interface

tryptische fragmenten bevatten. Voor deze verzameling van proteïnen moet nog gecontroleerd worden of de fragmenten weldegelijk na elkaar voorkomen (en dus verbonden zijn). Dit doen we door *brute-force* voor elke proteïne te controleren of de originele peptide hierin voorkomt. Tot slot wordt de Lowest Common Ancestor (LCA) van de gematchte proteïnen berekend. Dit gebeurt tijdens het zoeken zelf, in tegenstelling tot bij het zoeken van strikt tryptische peptiden. Indien we geen rekening houden met *missed cleavages* zijn alle LCAs namelijk al voorberekend. Het is duidelijk dat al deze extra stappen een sterke, **negatieve invloed hebben op de performantie**. Deze verminderde performantie bij *missed cleavages*, in combinatie met het compleet ontbreken van een manier om willekeurig gesplitste peptiden te zoeken, verklaart de nood aan een nieuwe indexstructuur.

Voor een gedetailleerdere beschrijving van Unipept en het onderzoeksgebied van metaproteomica is het aangeraden om de inleiding van het doctoraat van Dr. Pieter Verschaffelt te lezen [69]. Dit vormde een duidelijke en goede basis voor deze inleiding.

1.4 Probleemstelling

In deze masterproef zoeken we een oplossing voor het snel terugvinden van **willekeurige peptiden**³ in een proteïnedatabank. Bij het vinden van een match moet het daarna mogelijk zijn de informatie op te halen die hoort bij alle proteïnen waarin de peptide voorkomt. Binnen het onderzoeksgebied van de informatica kunnen we dit probleem als volgt herformuleren: “In een grote verzameling van middellange strings (alle proteïnen in onze databank), moeten we voor een verzameling van korte strings (peptiden) terugvinden in welke van deze middellange strings ze voorkomen.” We willen echter niet alleen vinden van welke proteïnen een peptide deel uitmaakt. Ook de bijbehorende taxonomische en functionele annotaties van de gematchte proteïne moeten zo snel mogelijk te vinden zijn. Deze gevonden annotaties moeten efficiënt verwerkt kunnen worden om zo vlot het eindresultaat te bekomen. Om dit te realiseren, worden waar mogelijk annotaties geaggregeerd tijdens het opbouwen van de indexstructuur.

Belangrijk hierbij is dat dit niet alleen **snel** gebeurt, maar dat we ook proberen **het vereiste geheugen tot een minimum te beperken**. Wat als acceptabel beschouwd wordt, hangt af van de omgeving waarin de analyses uitgevoerd worden. Voor stalen die geanalyseerd worden op een PC m.b.v. Unipept Desktop is dit ± 16 GB RAM. Voor grotere stalen waarvan de analyse op de Unipept servers uitgevoerd wordt mikken we op 0.5-2 TB geheugengebruik. Dit komt overeen met een realistische configuratie voor een server die gericht is op het uitvoeren van geheugenintensieve taken.

Tot slot willen we ook **semi-exacte matching** toevoegen tijdens het zoeken. Hierbij willen we de mogelijkheid aanbieden om isoleucine (I) en leucine (L) gelijk te stellen aan elkaar. Wanneer deze gelijkgesteld zijn, wil dit zeggen dat op elke plaats waar een I staat, ook een L toegelaten wordt, en omgekeerd. Door dit te doen kunnen we de beperkingen van een massaspectrometer opvangen.

Om dit allemaal te bereiken is het doel van deze thesis om meerdere datastructuren uit te werken, te implementeren in Rust, en tot slot te testen. Het gebruik van Rust laat ons toe om extreem hoge performantie te verkrijgen (vergelijkbaar met C en C++ [20]) in combinatie met *memory safety*⁴. Bovendien zijn sommige delen van Unipept al geschreven in Rust (zie UMGAP [28, 19]). Dit laat toe om waar mogelijk bestaande code te hergebruiken.

³Hiermee bedoelen we peptiden die tryptisch zouden moeten zijn, maar mogelijk één of meerdere gemiste splitsingen hebben. Daarnaast omvat dit ook peptiden die niet-tryptisch zijn, en dus op een andere manier gesplitst worden.

⁴*Memory safety* is een eigenschap die verzekert dat programma's enkel gebruik kunnen maken van geldige geheugenlocaties en geen *undefined behaviour* zoals *buffer overflows*, *dangling pointers* en andere geheugen gerelateerde fouten kunnen vertonen.

1.5 Benchmarkdatasets

Om de snelheid, het geheugengebruik en de correctheid van de onderzochte indexstructuren en zoekalgoritmen te bepalen zullen we **twee soorten benchmarkbestanden** gebruiken. De eerste soort zijn de **proteïnedatabanken** waarmee we de indexstructuur opbouwen. De grootte van deze indexstructuur is het primaire criterium aangezien deze **volledig in het werkgeheugen** moet passen, wat een harde limiet is. Indien de index niet in het geheugen past, zal het programma niet uitgevoerd kunnen worden (of met een erg grote *performance penalty* wanneer swapruimte⁵ gebruikt wordt). De tweede soort bestanden bevatten de peptiden die we gaan proberen terugvinden in de indexstructuur. We zullen hiernaar voor de rest van deze masterproef verwijzen als **peptidebestanden**. Het hoofddoel van deze peptidebestanden is om de zoekperformantie te testen. De tijd nodig om te zoeken is een zachte limiet aangezien we mikken voor hoge performantie, maar tragere code heeft enkel als gevolg dat een gebruiker langer moet wachten. Alle bestanden die in de volgende secties besproken worden, kunnen teruggevonden worden in onze GitHub repository⁶.

1.5.1 Proteïnedatabanken

Om te testen hoe goed een implementatie is en hoe deze zich verhoudt ten opzichte van bestaande implementaties, is het belangrijk om representatieve datasets te gebruiken. Deze datasets zijn allemaal proteïnedatabanken die een subset vormen van **UniProtKB** (meer specifiek UniProtKB 2023_04) [8]. UniProtKB zelf bestaat uit twee onderdelen (gegeven statistieken zijn voor release 2023_04).

1. **Swiss-Prot**: Dit is een kleinere, manueel gecureerde dataset met 570 157 proteïnen.
2. **TrEMBL**: Deze dataset bevat 251 600 768 proteïnen en is dus veel groter dan Swiss-Prot. Een bijkomend verschil is dat deze dataset **niet** manueel gecureerd is, maar geautomatiseerd geannoteerd werd.

Uiteindelijk is het doel om een indexstructuur voor UniProtKB op te bouwen waarbij de probleemstelling opgelost is. UniProtKB is echter veel te groot om mee te werken tijdens de testfase. Daarom gebruiken we tijdens het ontwikkelen twee kleinere subsets van UniProtKB. Eerst wordt een overzicht gegeven van de belangrijkste eigenschappen van UniProtKB, om daarna dieper in te gaan op de twee gebruikte subsets tijdens het testen.

UniProtKB Tabel 1.1 bevat een overzicht van de belangrijkste statistieken voor de volledige UniProtKB 2023_04 databank. Belangrijk om op te merken is dat de totale databank uit 86 805 673 041 aminozuren bestaat. Omgerekend is dit 86.81 GB aangezien een karakter opgeslagen wordt in één byte. Dit verklaart onmiddellijk de keuze om gebruik te maken van kleinere datasets tijdens het testen.

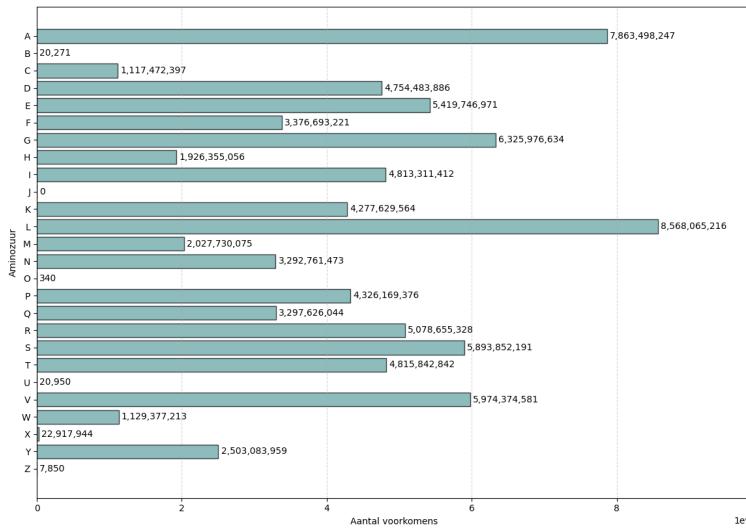
Metriek	Waarde
Totaal aantal sequenties	248 842 516
Totale lengte	86 805 673 041 AA
Minimale proteïnelengte	1 AA
Maximale proteïnelengte	45 354 AA
Gemiddelde proteïnelengte	348.84 AA
Mediaan proteïnelengte	278 AA

Tabel 1.1: Eigenschappen van de volledige UniProtKB 2023_04 databank. De afkorting *AA* staat voor *amino acids*.

In Figuur 1.3 en 1.4 is een gedetailleerdeerder overzicht van de aminozuurdistributie en verdeling van de proteïnelengten terug te vinden.

⁵Dit is wanneer een computer schijfruimte gebruikt om het tekort aan RAM-geheugen op te vangen.

⁶https://github.com/BramDevlaeminck/Thesis_benchmarkdata



Figuur 1.3: Aantal voorkomens per aminozuur voor alle proteïnen in de UniProtKB (2023_04) databank.

Swiss-Prot De Swiss-Prot databank is één van de twee standaardonderdelen van UniProtKB. Een kort overzicht van alle statistieken is terug te vinden in Tabel 1.2. Figuur 1.5 en Figuur 1.6 geven meer inzicht in de distributie van de aminozuren en lengte van de proteïnen.

Metriek	Waarde
Totaal aantal sequenties	569 619
Totale lengte	205 954 074 AA
Minimale proteïnelengte	2 AA
Maximale proteïnelengte	35 213 AA
Gemiddelde proteïnelengte	361.56 AA
Mediaan proteïnelengte	295 AA

Tabel 1.2: Eigenschappen van de Swiss-Prot databank (UniProtKB 2023_04). De afkorting *AA* staat voor *amino acids*.

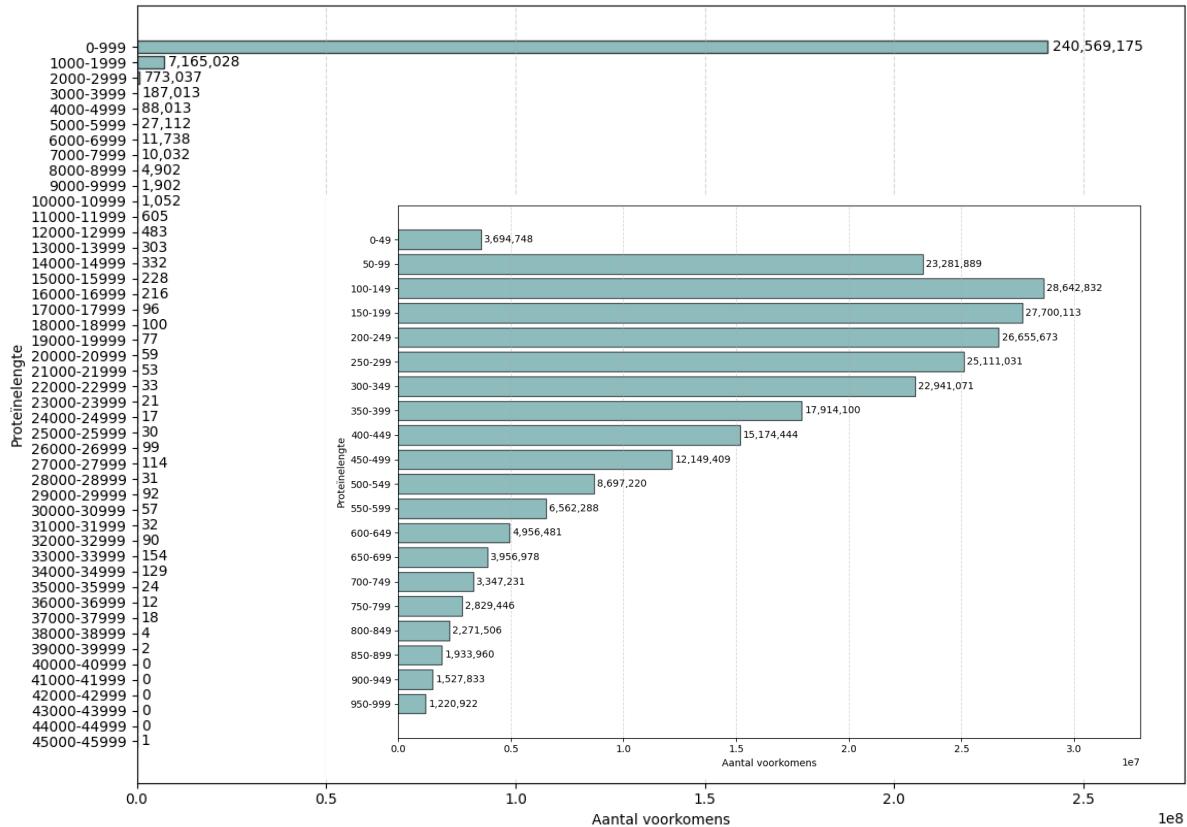
Doordat het gebruikte invoerbestand reeds verwerkt werd door een deel van de Unipept pipeline, is er een klein verschil tussen het totaal aantal sequenties in Tabel 1.2 en wat eerder aangegeven werd. Hierbij worden onder andere sequenties met een onbekend taxon ID verwijderd (538 in totaal), wat het kleine verschil verklaart.

Human-Prot De Human-Prot databank is samengesteld aan de hand van drie referentiedatabanken afkomstig uit UniProtKB. Dit zijn de Human Genome [49], Influenza B [51] en Human Papillomavirus [50] databank.

Deze Human-Prot databank is kleiner dan Swiss-Prot, waardoor het testen tijdens ontwikkeling sneller is. Tabel 1.3 somt enkele belangrijke metrieken op over deze dataset. Figuur 1.7 en 1.8 gaan dieper in op de aminozuur- en lengtedistributie van de proteïnen.

We kunnen concluderen dat zo goed als **alle letters gebruikt** worden (ook al zijn er maar 20 aminozuren). Dit komt doordat sommige letters eigenlijk een soort wildcard voorstellen. Zo staat “X” voor elk mogelijk aminozuur, “Z” voor “Q” of “E”,... [2].

Verder valt ook te zien dat de verdeling van de proteïnelengten in de UniProtKB, Swiss-Prot en Human-Prot datasets vergelijkbaar zijn. Met andere woorden: **Swiss-Prot en Human-Prot zijn**



Figuur 1.4: Lengtedistributie van de proteïnen in de UniProtKB (2023_04) databank. De kleinere grafiek bevat een gedetailleerde overzicht van de distributie in het interval $[0, 1000]$.

een representatieve kleinere voorstelling van UniProtKB. Dit laat ons toe om gebruik te maken van de Swiss-Prot en Human-Prot proteïnedatabanken en later de resultaten te veralgemenen en op te schalen naar UniProtKB.

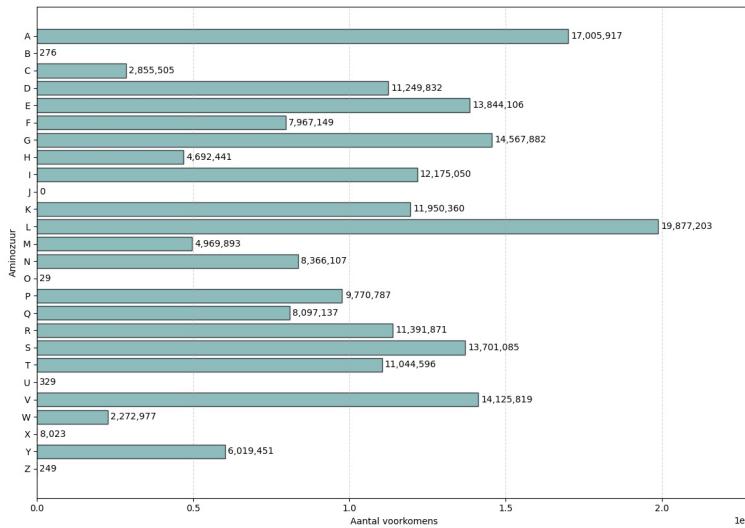
1.5.2 Peptidebestanden

De zoekperformantie van onze indexstructuur is een erg belangrijk aspect. Om dit te meten, hebben we bij elke proteïnedatabank een lijst van peptiden die we proberen te zoeken. Zowel voor de Swiss-Prot als Human-Prot databank zijn enkele datasets opgesteld.

Swiss-Prot

Voor deze proteïnedatabank hebben we enkele peptidebestanden voorzien. Twee bestanden die gesampled zijn en een reeks aan real-life stalen. De twee artificiële bestanden zijn zo gekozen dat de ene enkel tryptische peptiden bevat, terwijl de andere ook peptiden bevat met *missed cleavages*. De eerste kan dus op dit moment al efficiënt door Unipept verwerkt worden, terwijl dit voor de tweede onmogelijk is.

Artificiële stalen Tabel 1.4 bevat in kolom twee en drie een kort overzicht met statistieken voor deze gesampled bestanden.



Figuur 1.5: Aantal voorkomens per aminozuur voor alle proteïnen in de Swiss-Prot databank uit UniProtKB 2023_04. De letters B, J, O, U, X en Z stellen geen aminozuur voor, maar zijn een soort *wildcards* voor meerdere mogelijke aminozuren.

Metriek	Waarde
Totaal aantal sequenties	82 695
Totale lengte	30 293 046 AA
Minimale proteïnelengte	2 AA
Maximale proteïnelengte	35 991 AA
Gemiddelde proteïnelengte	366.32 AA
Mediaan proteïnelengte	204 AA

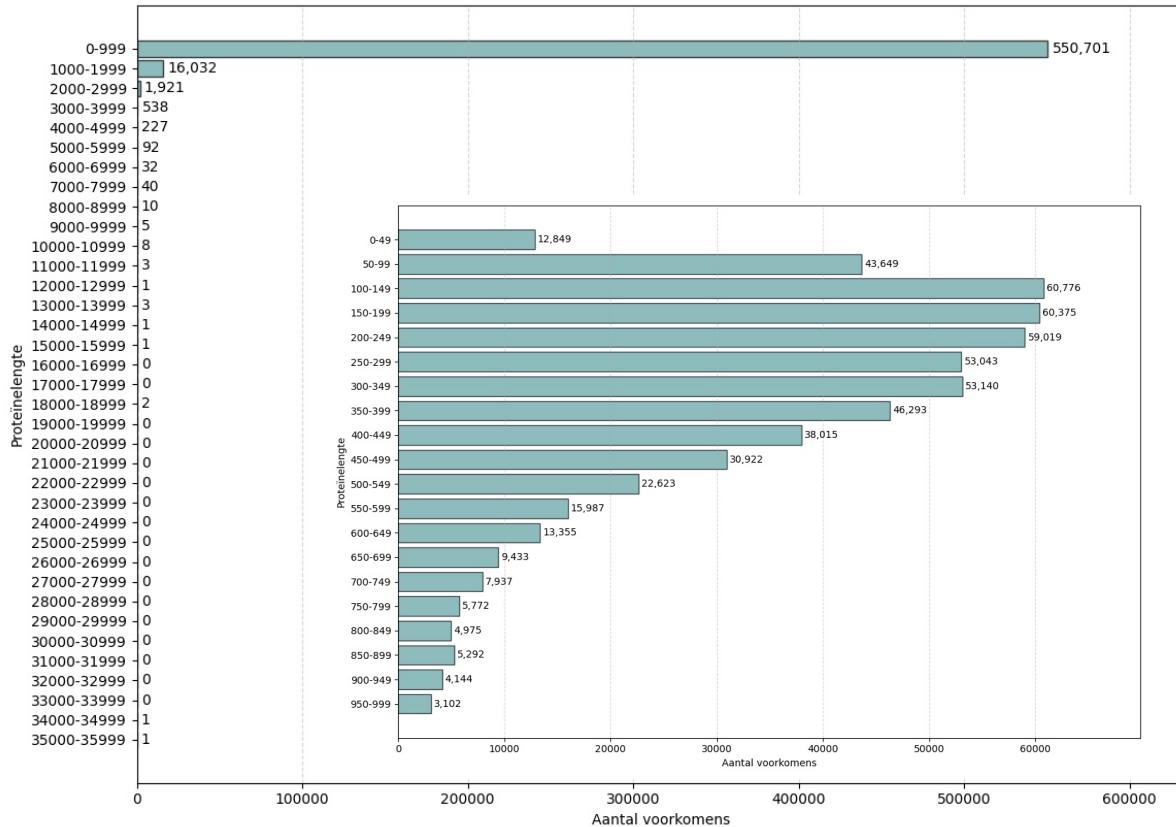
Tabel 1.3: Eigenschappen van de Human-Prot databank (UniProtKB 2023_04). De afkorting *AA* staat voor *amino acids*.

Metriek	Swiss-Prot-TRYP	Swiss-Prot-MC	Human-Prot
Totaal aantal sequenties	100 000	100 000	250 000
Totale lengte	1 605 909 AA	2 544 356 AA	2 458 834 046 AA
Minimale peptidelengte	5 AA	5 AA	1 AA
Maximale peptidelengte	50 AA	93 AA	12 AA
Gemiddelde peptidelengte	16.06 AA	25.44 AA	9.84 AA
Mediaan peptidelengte	13 AA	23 AA	10 AA
Aantal vindbare peptiden	67 375	62 581	250 000
Aantal tryptische peptiden	100 000	4107	102 659

Tabel 1.4: Eigenschappen van de verschillende peptidebestanden. *Swiss-Prot-TRYP* bevat de statistieken voor het Swiss-Prot peptidebestand met enkel tryptische peptiden. Hierbij komen er dus geen *missed cleavages* voor. *Swiss-Prot-MC* bevat net wel *missed cleavages*. De laatste kolom bevat de statistieken voor het peptidebestand dat hoort bij de Human-Prot proteïnedatabank. De afkorting *AA* staat voor *amino acids*.

Experimentele stalen Om de performantie beter te beoordelen, gebruiken we ook enkele stalen uit experimenten met een kleine micro-organisme gemeenschap, namelijk SIHUMIx⁷ [32, 4]. Aangezien deze stalen uit een experiment ontstaan zijn, bevatten deze *missed cleavages* die natuurlijk ontstaan zijn. De peptiden zijn dus effectief ontstaan uit een staal proteïnen waarop trypsin toegepast is.

⁷Simplified human intestinal microbiota



Figuur 1.6: Lengtedistributie van de proteïnen in de Swiss-Prot databank. De kleinere grafiek bevat een gedetailleerdeerder overzicht van de distributie in het interval [0, 1000[.

Tabel 1.5 bevat de belangrijkste statistieken voor elk peptidebestand. Deze peptidebestanden worden in combinatie met de Swiss-Prot proteïnedatabank gebruikt tijdens het testen.

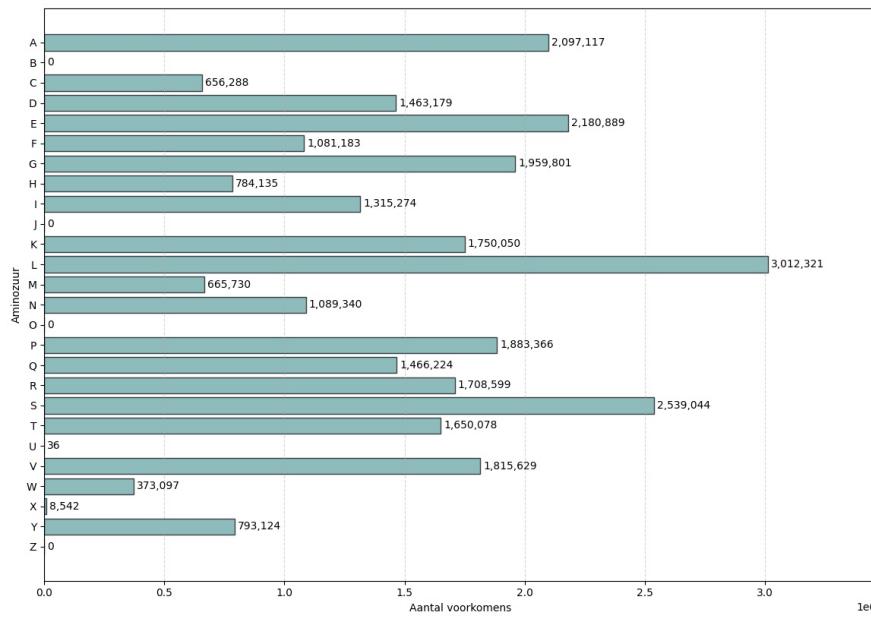
Metriek	SIHUMI 03	SIHUMI 05	SIHUMI 05	SIHUMI 08	SIHUMI 11	SIHUMI 14
Totaal aantal sequenties	25 000	25 000	24 424	25 000	24 998	25 000
Totale lengte	420 544 AA	420 423 AA	373 633 AA	316 114 AA	366 894 AA	430 674 AA
Minimale peptidelengte	6 AA					
Maximale peptidelengte	50 AA	50 AA	47 AA	43 AA	50 AA	50 AA
Gemiddelde peptidelengte	16.82 AA	16.82 AA	15.30 AA	12.64 AA	14.68 AA	17.23 AA
Mediaan peptidelengte	15 AA	16 AA	14 AA	12 AA	14 AA	16 AA
Aantal vindbare peptiden	2570	2698	3652	4135	3792	2761
Aantal tryptische peptiden	17 263	162	152	207	153	242

Tabel 1.5: Eigenschappen van de SIHUMIx peptidebestanden. Elke kolom stelt een staal voor met als bestandsnaam S<XX>.txt. Deze stalen kunnen teruggevonden worden in onze GitHub repository^a onder de SIHUMI-folder. De afkorting AA staat voor *amino acids*.

^ahttps://github.com/BramDevlaeminck/Thesis_benchmarkdata

Human-Prot

Voor deze databank hebben we één peptidebestand waarin we HLA-peptiden simuleren. Dit zijn **korte, niet-tryptische peptiden** waarin men typisch geïnteresseerd is in het immunopeptidomics onderzoeksgebied [39]. Dit betekent dat UniPept op dit moment niet gebruikt kan worden om dit soort stalen te analyseren. Elke peptide in dit peptidebestand is een sample van een proteïne uit de Human-Prot databank. Hierdoor zijn alle peptiden die we zoeken effectief vindbaar in de dataset. Een kort overzicht van de eigenschappen is terug te vinden in de laatste kolom van Tabel 1.4.



Figuur 1.7: Aantal voorkomens per aminozuur voor alle proteïnen in de Human-Prot databank.

Appendix A bevat aanvullende grafieken over bovenstaande peptidebestanden. Deze tonen voor elk bestand de distributie van de aminozuren en peptidelengte.

1.6 Benchmark hardware

Alle benchmarks werden uitgevoerd op een virtuele machine van team UniPept tenzij anders vermeld. Tabel 1.6 bevat een overzicht van de hardware van de fysieke machine en hoeveel toegekend is aan de VM. Alle informatie omtrent UniPept infrastructuur kan teruggevonden worden op de UniPept GitHub wiki [65].

Een deel van de kleinere testen werd ook lokaal uitgevoerd op een laptop. Dit zal elke keer expliciet vermeld worden indien dit het geval was. De gebruikte laptop is een M1 Pro MacBook Pro (14 inch) uit 2021. Tabel 1.7 bevat de exacte specificaties van deze laptop.

Naast een VM en eigen laptop hebben we ook toegang tot enkele andere machines indien hier nood toe zou zijn. Een eerste mogelijke optie is via een andere, krachtigere server van UniPept. Het is mogelijk om applicaties die op dit moment op verschillend servers draaien, tijdelijk te herverdelen zodat er een machine met 768 GiB ram ter beschikking komt. Een andere optie is de HPC van UGent⁸. Hierop zijn nodes⁹ beschikbaar tot 940 GiB RAM. Tot slot heeft de CompOmics¹⁰ onderzoeksgrond aan UGent ook enkele machines staan die tot 2 TiB aan geheugen hebben. Deze onderzoeksgrond werkt aan softwaretoepassingen voor het verwerken van proteomica-data, waar het onderzoeksgebied van de metaproteomica onder valt.

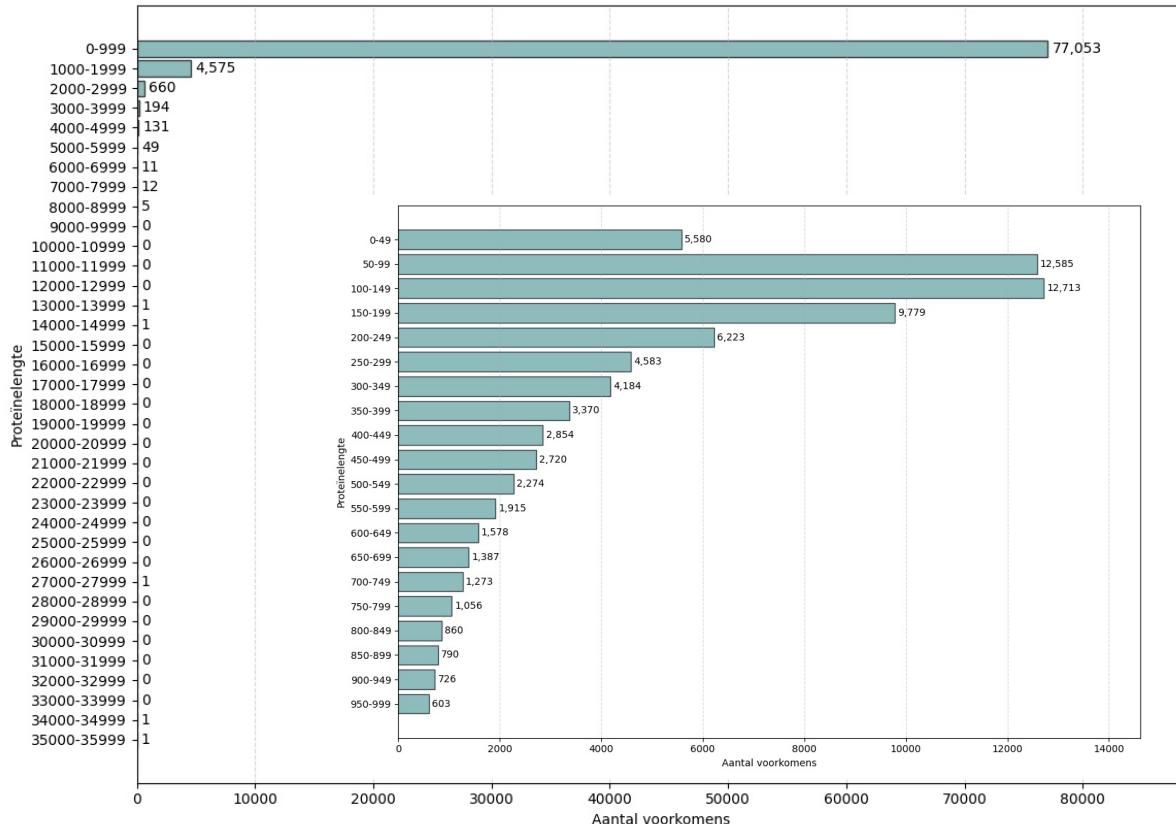
1.7 Mogelijke oplossingen

De essentie van onze probleemstelling is het snel vinden van een grote hoeveelheid korte strings in één erg lange string. Dit is een vorm van stringmatching. Hiervoor bestaan twee verschillende strategieën (die ook gecombineerd kunnen worden).

⁸<https://docs.hpc.ugent.be/>

⁹Elke node bestaat is een fysieke server. Deze nodes vormen samen een cluster op de HPC.

¹⁰Computational Omics



Figuur 1.8: Lengtedistributie van de proteïnen in de Human-Prot databank. De kleinere grafiek bevat een gedetailleerdeerder overzicht van de distributie in het interval $[0, 1000]$.

1. **Verwerk de korte zoekstring op voorhand** (van lengte n) zoals in het algoritme van Knuth-Morris-Pratt [30], Boyer-Moore-Horspool [26] en het shift-AND-algoritme [10].
2. **Verwerk de lange tekst op voorhand** (van lengte m) zoals bij suffixbomen [40], suffix arrays [38] en (bidirectionele) FM-indices [14, 34].

Voor beide strategieën bestaan er algoritmen die in lineaire tijd ten opzichte van de stringlengte een indexstructuur bouwen, en deze kunnen doorzoeken. Er is echter een belangrijk verschil.

Bij de strategie waar we de korte zoekstring op voorhand verwerken kan het opbouwen in $O(n)$ tijd en geheugen, en het zoeken in $O(m)$ tijd (met n de lengte van de zoekstring, en m de lengte van de tekst). Hierbij is het **zoeken dus lineair in de tijd ten opzichte van de totale tekst**. Dit is nadelig wanneer er veel korte strings zijn, waarvoor elke keer extra werk moet gebeuren. Daarna moeten we bovendien nog voor elke korte string de zoekoperatie uitvoeren, waarvoor de uitvoeringstijd lineair is in de lengte van de volledige tekst.

Indien we de lange tekst indexeren, kan het opbouwen in $O(m)$ tijd en het zoeken in $O(n)$ tijd en geheugen (opnieuw met n de lengte van de zoekstring, en m de lengte van de tekst). Hierbij is het mogelijk om één keer de indexstructuur te bouwen voor de lange tekst, waarna elke korte string **in lineaire tijd ten opzichte van zijn eigen lengte gezocht** kan worden. Het nadeel is echter dat het opbouwen van de indexstructuur voor een grote tekst traag kan worden, en bovendien veel geheugen kan innemen.

Onderdeel	Fysieke server	Virtuele Machine
CPU	2×Intel Xeon 4410Y (12 cores / 24 threads, 2 - 3.9 GhZ, 30 MiB cache)	12 threads
RAM	768 GiB	128 GiB
Opslag	6×16 TiB HDD (3.5 inch, 7.2K RPM SATA), 4×3.84 TiB SSD (2.5 inch, SATA)	1 TiB SSD, 4 TiB HDD
Besturingssysteem	Debian 12 (met Proxmox)	Ubuntu 22.04 LTS

Tabel 1.6: Hardwarespecificaties van de fysieke server en virtuele machine die gebruikt worden tijdens het testen. Deze virtuele machine draait samen met enkele andere VMs op de server.

Onderdeel	hardware
Model	MacBook Pro (14 inch, 2021)
CPU	8-core M1 Pro, 6 performance cores, 2 efficiency cores
RAM	16 GB (LPDDR5)
Opslag	512 GB SSD
Besturingssysteem	MacOS (14) Sonoma

Tabel 1.7: Hardwarespecificaties van de gebruikte laptop voor kleinere testen.

Bij ons probleem hebben een grote databank met erg veel proteïnen (een lange tekst) waarin we erg veel peptiden (korte strings) zoeken, wat overeenkomt met de tweede aanpak. Daarom verkennen we in deze masterproef die aanpak. In de eerste instantie willen we **exacte matches** kunnen zoeken, maar later willen we ook **inexacte matching** uitvoeren.

Aangezien de indexstructuur slechts eenmalig voor een bepaalde proteïnedatabank opgebouwd moet worden, ligt de **primaire restrictie bij het geheugengebruik** tijdens het opbouwen. Het blijft echter steeds belangrijk dat de indexstructuur in een redelijke tijd opgebouwd kan worden en performant genoeg is om snel een groot aantal peptiden te zoeken. De richttijd die we ons vooropstellen om de indexstructuur op te bouwen voor UniProtKB is maximaal één à twee dagen. Dit is een acceptabele tijdsduur aangezien de indexstructuur slechts om de acht weken opnieuw opgebouwd moet worden, overeenkomstig met de frequentie waarbij een nieuwe versie van de UniProtKB databank uitgebracht wordt. In de volgende hoofdstukken verkennen we verschillende indexstructuren om een proteïnedatabank te indexeren.

2 SUFFIXBOMEN

Suffixbomen zijn een eerste datastructuur die het mogelijk maken om efficiënt te controleren waar een korte string voorkomt in een andere, grotere string. Meer specifiek gaat het om een variant hiervan, de zogenaamde veralgemeende suffixboom. Deze laat toe om efficiënt te **controleren of een string deel is van één of meerdere strings uit een verzameling**.

We behandelen deze datastructuur als eerste omdat hij vrij intuïtief en het minst complex is. Bovendien kan een goede tijdscomplexiteit bereikt worden aangezien de zoektijd naar exacte matches in een suffixboom lineair is in de lengte van de zoekstring/peptide. Het opbouwen van de suffixboom kan ook in lineaire tijd gebeuren, maar dan lineair in de totale lengte van alle proteïnen in de databank. Hiervoor hebben we een **eigen Rust implementatie** geschreven. Aan de hand hiervan was het mogelijk om vertrouwd te raken met de programmeertaal Rust. Daar komt bij dat het ons ook de mogelijkheid geeft om enkele interessante implementatiedetails, specifiek voor deze programmeertaal, te bespreken naast de bereikte performantie.

2.1 Wat zijn suffixbomen?

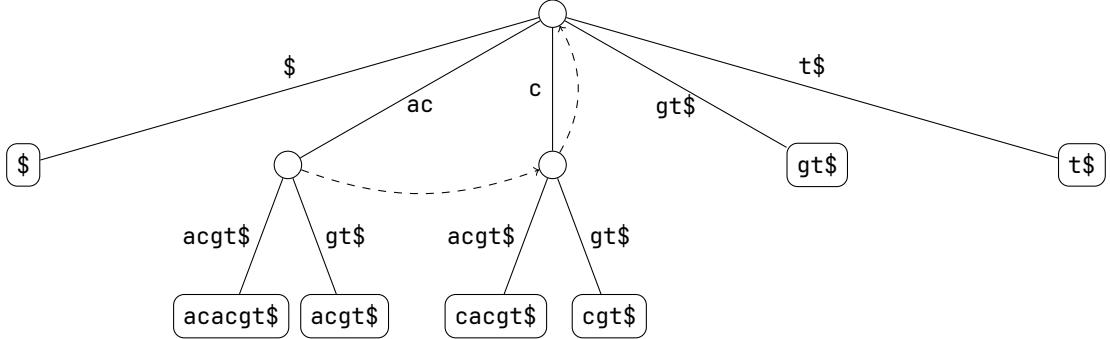
Een suffixboom van een tekst T wordt formeel gedefinieerd als volgt [9]:

1. Het is een boomstructuur met exact n bladeren waarbij alle takken samenkommen in de wortel van de boom. De bladeren krijgen elk een uniek nummer in het interval $[0, n[$.
2. Elke interne top (behalve de wortel) heeft op zijn minst 2 kinderen.
3. Elke boog heeft een label die een niet-lege substring van de tekst T voorstelt.
4. Voor alle bogen die uit een bepaalde top vertrekken geldt dat het eerste karakter van hun label verschillend is.
5. Voor elk blad $i \in [0, n[$ geldt dat de aaneenschakeling van alle labels op het pad van de wortel naar blad i overeenkomt met suffix S_i . Dit is de suffix die op index i in tekst T begint.

Merk op dat niet voor elke tekst T een suffixboom zal bestaan. Deze bestaat alleen wanneer geen enkele suffix een prefix is van een andere suffix. In de praktijk wordt daarom een **uniek eindteken**, meestal $\$$, toegevoegd aan de tekst. Op deze manier zal er gegarandeerd een suffixboom voor de (licht aangepaste) tekst T bestaan.

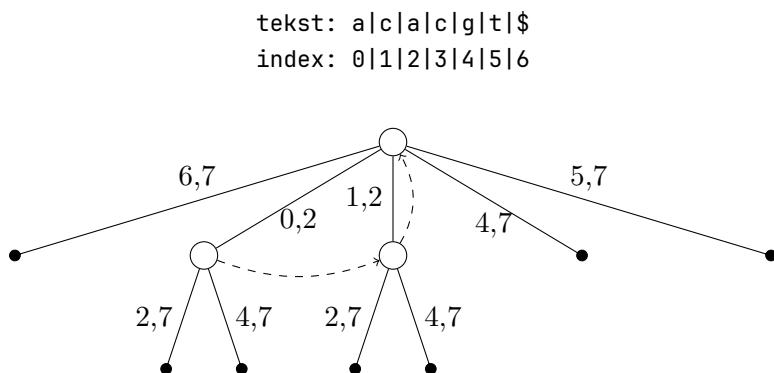
Het navigeren in een suffixboom kunnen we efficiënter maken door ook de **suffix links** bij te houden. Dit is een pointer van een interne top v naar een andere interne top $s(v)$. Deze suffix link bestaat als het pad uit de wortel naar top v voorgesteld kan worden door de label $c\alpha$, waar c een karakter is en α een (mogelijk lege) string. De top $s(v)$ zal dan de top zijn, waarvoor geldt dat het pad van de wortel tot aan de top de substring α voorstelt. Indien α leeg is, zal $s(v)$ de wortel van de suffixboom zijn.

Figuur 2.1 stelt de suffixboom voor van de string `acacgt$`. Hierbij stellen de stippenlijnen de suffix links voor. Merk op dat we $\$$ als uniek eindteken gebruiken.



Figuur 2.1: Suffixboom voor de string `acacgt$`. De stippellijnen stellen de suffix links voor.

Natuurlijk is het niet efficiënt om de structuur op deze manier op te slaan. Als de tekst lengte n heeft, heeft de suffixboom ten hoogste $2n - 1$ toppen en $2n - 2$ bogen. Het aantal toppen en bogen is dus $\Theta(n)$. Jammer genoeg vraagt het opslaan van alle prefixen in de bladeren $\Theta(n^2)$ geheugen [5]. We kunnen dit oplossen aan de hand van pointers naar het begin en einde van een substring in de originele string. Hierdoor moeten we **geen kopie** meer opslaan **van de originele string in elk blad**. We kunnen bij elke boog tussen de toppen labels bijhouden. Het label van het blad kunnen we daarna reconstrueren door de labels van de bogen op weg naar dit blad achter elkaar te plaatsen. Op deze manier wordt de nodige opslag per top een constante en krijgen we **lineair geheugengebruik** ($O(n)$). Figuur 2.2 toont hoe dit er in de praktijk uitziet. Merk op dat indexering start vanaf nul en dat de eindindex exclusief is. Een boog met waarde 1,3 stelt dus de substring `ca` voor uit het voorbeeld.



Figuur 2.2: Suffixboom voor de string `acacgt$`, gebruikmakende van indices. De stippellijnen stellen de suffix links voor.

Deze suffixbomen laten toe om in $O(n)$ tijd substrings te zoeken binnen één string. Hierbij is n de lengte van de gezochte substring. Indien we ook willen weten hoeveel voorkomens er exact zijn, en waar deze zich bevinden neemt dit $O(k)$ extra tijd. Hierbij is k het aantal voorkomens van de substring in de tekst. Zoals eerder vermeld willen we echter peptiden zoeken in een **verzameling van proteïnen**. Daarom wordt gebruikgemaakt van een **veralgemeende suffixboom**. Hierbij wordt de verzameling van proteïnen die geïndexeerd moet worden behandeld alsof het één grote tekst is. De manier waarop deze tekst gevormd wordt is echter erg belangrijk. Alle proteïnen moeten aan elkaar gehangen worden met ertussen een teken dat niet in de proteïnen en peptiden zelf voorkomt. Als eindteken moet er een ander uniek teken gebruikt worden. Wanneer niet aan de eerste voorwaarde voldaan wordt, zullen we ongeldige matches vinden. Het zou namelijk mogelijk zijn om een match te vinden die over twee of meer proteïnen gespreid is. Wanneer niet aan de tweede voorwaarde voldaan wordt, dan breken we de eigenschap van suffixbomen waarbij elke suffix nooit een prefix van een andere suffix is. Dit zou als gevolg hebben dat niet elke suffix een eigen blad in de boom krijgt.

2.2 Het algoritme van Ukkonen

Het algoritme van Ukkonen [62] is een complexe, maar efficiënte manier om suffixbomen op te bouwen in lineaire tijd met lineair geheugengebruik. De beschrijving in de originele paper is vrij theoretisch, wat het algoritme minder toegankelijk maakt. Het komt echter uitgebreid aan bod in een aantal andere publicaties en boeken [23, 5, 9, 15]. Deze vormden een grote hulp bij het maken van een eigen implementatie.

2.2.1 Kotlin

Een eerste implementatie¹ van Ukkonen's algoritme heb ik met behulp van de programmeertaal Kotlin gemaakt. Hierdoor kon er gefocust worden op het algoritme zonder belemmerd te worden door restricties opgelegd door de *borrow checker*² van Rust. Tijdens het implementeren was de referentiecode van prof. Jan Fostier [15] een groot hulpmiddel, omdat we hierdoor tijdens het debuggen het verloop van het programma konden opvolgen.

Het belangrijkste verschil tussen de Kotlin-implementatie en de referentie-implementatie is de manier waarop de kinderen voorgesteld worden. Bij de eerste is dit aan de hand van een `HashMap` terwijl de laatste gebruikmaakt van een `pointer array`. De reden voor de eerste aanpak is om alles zo simpel mogelijk te houden. Hierdoor kan een karakter rechtstreeks als sleutel gebruikt worden, en was een omzetting naar een index niet nodig. Om dit prototype te maken, heb ik gekozen voor Kotlin boven Python aangezien Kotlin performanter is en ook een aangename ontwikkelingservaring biedt. Hierdoor is het mogelijk om de testdatasets op te bouwen binnen de 10 minuten.

Het grootste struikelblok tijdens het implementeren van het algoritme van Ukkonen waren enkele off-by-one fouten. Aangezien je tijdens het algoritme werkt met substrings, maar deze opgeslagen worden aan de hand van hun begin- en eindindex, wordt het debuggen veel omslachtiger. Tot slot had ik op het einde ook enkele bugs die niet voorkwamen in kleinere voorbeelden die met de hand uit te werken waren. Dit maakte het lokaliseren en oplossen van de laatste problemen vrij tijdsintensief.

2.2.2 Rust

Door het *ownership* systeem van Rust zijn er enkele belangrijke nuances in het implementeren van het algoritme van Ukkonen in Rust. In de volgende secties bespreken we eerst enkele van deze nuances en hoe deze zich vertalen in onze eigen implementatie³. Daarna bespreken we de performantie op vlak van snelheid en geheugengebruik.

Boomstructuren

Deze quote komt rechtstreeks uit een Medium artikel [1] en toont direct aan waarom het maken van een suffixboom in Rust niet-triviaal is.

Rust is known to be notorious difficult when it comes to certain data structures like linked lists, trees, etc.

De oorzaak hiervoor ligt bij het *ownership* systeem van Rust. Dit systeem zorgt ervoor dat elk stukje data slechts één eigenaar kan hebben. In dit geval kan dus **slechts één top een andere top opslaan**, of er een *mutable reference* naar hebben. Concreet wil dit dus zeggen dat slechts één top een *pointer* kan hebben naar een andere top, met de toelating om die top aan te passen. Dit is net wat nodig is tijdens het opbouwen van de boom, want er worden nog kinderen toegevoegd en toppen gesplitst. Dit is een probleem aangezien ouders pointers naar kinderen moeten hebben, de kinderen een verwijzing naar hun ouder, en er dan ook nog eens pointers zijn voor de suffix links.

¹<https://github.com/BramDevlaeminck/SuffixTree>

²<https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html>

³<https://github.com/BramDevlaeminck/FastPeptideMatching/tree/master/suffixtree>

Als oplossing hiervoor introduceert Rust het `Rc`<T> datatype. Hierbij stapt Rust af van zijn standaard *ownership* systeem en wordt gebruikgemaakt van **Reference Counting** [55]. Pas wanneer alle referenties weg zijn, zal het geheugen automatisch vrijgegeven worden. De beperking hierbij is echter dat deze referenties *immutable* zijn. Dit volstaat niet tijdens het opbouwen van de boom aangezien wij wijzigingen moeten kunnen maken aan de toppen waar naar verwezen wordt.

Om dit toch mogelijk te maken, introduceert Rust het *interior mutability* patroon [57]. Hier voor wordt gebruikgemaakt van het `Refcell` datatype. Dit laat toe om data toch aan te passen, ook al is de referentie *immutable*. Aangezien dit de standaard Rust regels doorbreekt, is dit `unsafe`⁴ en kan Rust *at compile-time* geen *memory safety* meer garanderen. `Refcell` zal gelukkig de nodige code invoegen zodat runtime memory safety wel gegarandeerd is. Mogelijke foutieve geheugenoperaties zullen dus tijdens het uitvoeren van het programma gedetecteerd worden, ten koste van performantie.

Maar zelfs dan blijft er nog altijd een probleem. Geheugen dat beheerd wordt aan de hand van *reference counting* zal enkel vrijgegeven kunnen worden indien de *reference counter* op 0 staat. Er zijn echter scenario's waar dit nooit zal gebeuren. Namelijk bij **cyclische verwijzingen**; een patroon dat jammer genoeg erg vaak voorkomt. In ons geval is dit een ouder die een pointer heeft naar een kind, dat zelf een pointer heeft naar die ouder. Als oplossing hiervoor introduceert Rust dan weer het `Weak`<T> datatype.

De manier waarop Rust memory safety garandeert, introduceert duidelijk extra complexiteit en *performance overhead* die ongewenst is en we willen vermijden. Een optie zou kunnen zijn om expliciet het `unsafe` keyword te gebruiken, wat meer vrijheid geeft. Het nadeel hiervan is natuurlijk dat we dan de garanties van memory safety kwijt zijn, wat net één van de hoofdredenen is om Rust te gebruiken. Dit was dus geen optie. Gelukkig zijn we op een alternatief gestoten: een **arena-based implementatie** [36]. Het idee hierbij is dat er één arena gemaakt wordt waarbij ownership erg simpel is. In mijn implementatie is dit bijvoorbeeld een `Vector`. Alle toppen worden hierbij in deze ene vector opgeslagen. In plaats van pointers naar elkaar bij te houden, zullen de toppen getallen bijhouden. Elk getal stelt de index van een top in de arena voor, waarnaar anders een pointer wordt bijgehouden.

Aangezien we alles in één vector opslaan, is het mogelijk om de nodige hoeveelheid geheugen onmiddellijk aan te vragen bij het opstarten van het programma. Zoals in sectie 2.1 beschreven staat zijn er ten hoogste $2n - 1$ toppen voor een tekst van lengte n . Voor de Swiss-Prot databank die een inputstring van 206 523 693 karakters vormt, wil dit zeggen dat er in het slechtste geval 413 047 385 toppen zijn. In de praktijk worden er echter slechts 328 922 516 toppen gevormd. Indien we het geheugen op voorhand aanvragen zou het geheugengebruik dus nog met een factor $\frac{413\,047\,385}{328\,922\,516} \approx 1.26$ hoger liggen. Daarom hebben we die aanpak niet gekozen.

Na het maken van deze ontwerpbeslissingen bleef slechts één moeilijkheid over. Uitzoeken hoe de cursor (die bijhoudt waar we zijn in de boom tijdens het bouwen), de input string en de boom zich tot elkaar moeten verhouden in het systeem van eigenaarschap. Uiteindelijk viel dit vrij makkelijk uit te zoeken, gebruikmakende van de foutmeldingen gegeven door `rustc`⁵. Het omzetten van de resterende Kotlin-code naar Rust was erg simpel en bijna een één-op-één vertaling. Hier heb ik er echter voor gekozen om kinderen voor te stellen op dezelfde manier als in de C++-referentiecode. Er wordt dus een array bijgehouden per top waarin plaats voorzien wordt voor alle mogelijke kinderen van die top. Dit gebeurt ook indien een top geen, of slechts een klein aantal kinderen bevat. Dit heeft als gevolg dat de **voorstelling van elke top even groot** is.

⁴Dit is code waarvan de compiler niet kan nagaan of die aan alle voorwaarden voldoet die nodig zijn om *memory safety* te kunnen garanderen. Dit sleutelwoord bestaat zodat de programmeur meer vrijheid zou kunnen krijgen om bepaalde patronen toch te kunnen toepassen. De verantwoordelijkheid om correct het geheugen te gebruiken wordt hier bij de programmeur gelegd. Een andere reden om `unsafe` te gebruiken is om bepaalde interacties met hardware uit te voeren. Deze zijn inherent onveilig en zouden anders onmogelijk zijn.

⁵De compiler voor de Rust programmeertaal

Geheugenefficiëntie

Null pointers worden ook wel *the billion-dollar mistake* genoemd vanwege het grote aantal bugs dat ze veroorzaken.

And then I went and invented a null pointer. And if you use a null pointer you either have to check every reference or you risk disaster. [24]

Daarom voorziet Rust een andere manier om de waarde *null* voor te stellen. Dit kan aan de hand van de `Option<T>` enum.

```
enum Option<T> {
    None,
    Some(T),
}
```

Deze enum heeft twee mogelijke waarden: `None` en `Some(T)`. `None` is het equivalent van *null*, terwijl `Some(T)` wil zeggen dat de waarde verschillend is van *null*. Meer specifiek heeft de waarde type `T`. Aangezien het grootste deel van wat bijgehouden wordt per top pointers zijn, maakte ik veelvoudig gebruik van deze `Option` enum. Alle pointers in een top kunnen namelijk *null* zijn. De *parent pointer* moet nullable zijn aangezien de wortel geen ouder heeft. Al zou dit wel elegant opgelost kunnen worden door de *parent pointer* van de wortel naar zichzelf te laten wijzen. De *child pointers* moeten allemaal nullable zijn omdat bladeren geen kinderen hebben en in de interne toppen zijn niet alle kinderen altijd nodig. Tot slot moeten de suffix links nullable zijn aangezien niet elke top een suffix link heeft naar een andere top.

Dit werkt perfect en kon mooi afgehandeld worden op de idiomatische manier die overeenkomt met goede Rust-code. Na de eerste benchmarks bleek het **geheugengebruik** echter **problematisch**. Bijna exact **2× zo hoog als de equivalente C++ implementatie**. Om zo'n drastisch verschil in geheugengebruik te kunnen verklaren, moest er wel iets fundamenteel verschillen aan de manier waarop toppen hun data bijhouden. Al snel bleek dat het gebruik van `Option<usize>`⁶ de boosdoener was. Het gebruik van `Option<T>` zorgt namelijk voor **8 bytes aan overhead**. Aangezien een `usize` 8 bytes groot is op een 64-bit machine, verklaart dit inderdaad de verdubbeling van het geheugengebruik. Dit valt makkelijk te controleren aan de hand van de `std::mem::size_of` functie, deel van de Rust standaardbibliotheek.

```
assert_eq!(mem::size_of::<Option<usize>>(), 16);
assert_eq!(mem::size_of::<usize>(), 8);
```

Als oplossing heb ik uiteindelijk mijn **eigen *null* waarde gedefinieerd** die gebruikmaakt van een *trait*⁷. Deze oplossing doet volledig het doel van de `Option<T>` enum teniet, maar is jammer genoeg nodig omdat het gewoonweg niet acceptabel is om het geheugengebruik hiervoor te verdubbelen. Bovendien blijft **memory safety gegarandeerd**, aangezien het foutief indexeren van de *null* waarde (`usize::MAX` in dit geval) een index-out-of-bounds error genereert. Dergelijke indexeringsfouten worden tijdens het uitvoeren gedetecteerd en geven dus geen verdere problemen (afgezien van een mogelijke crash van het programma).

⁶**usize**: The pointer-sized unsigned integer type [58]. De grootte van dit datatype is het aantal bytes nodig om een referentie naar elk mogelijke locatie in het geheugen bij te kunnen houden. Voor 32- en 64-bit machines zijn dit resp. 4 en 8 bytes.

⁷Een trait in Rust definieert een functionaliteit die een bepaald type heeft, en kan delen met andere types.

```

/// Custom trait implemented by types that have a value that represents NULL
pub trait Nullable<T> {
    const NULL: T;

    fn is_null(&self) -> bool;
}

/// Type that represents the index of a node in the arena part of the tree
pub type NodeIndex = usize;

impl Nullable<NodeIndex> for NodeIndex {
    /// Use usize::MAX as NULL value since this will in practice never be reached.
    /// It is not possible to create 2^64-1 nodes (on a 64-bit machine).
    /// This would simply never fit in memory
    const NULL: NodeIndex = usize::MAX;

    fn is_null(&self) -> bool {
        *self == Self::NULL
    }
}

```

2.2.3 Performantie

Natuurlijk is het belangrijk dat de implementatie performant en correct is. Aangezien we ook over een bestaande C++ implementatie van Ukkonen's algoritme beschikken, was dit een perfecte maatstaf om mee te vergelijken. Uiteindelijk heb ik één aanpassing moeten maken in deze C++ code om een eerlijke vergelijking uit te voeren. Oorspronkelijk werd er in elke top ruimte voorzien voor 256 mogelijke kinderen. Dit was veel te hoog voor onze usecase. Er zijn namelijk slechts 20 aminozuren en enkele *wildcard characters*. Dit verklaart onmiddellijk waarom het geheugengebruik initieel ongeveer tien keer hoger hoger was dan nodig. Uiteindelijk ben ik gegaan voor een implementatie (zowel in Rust als C++) waar plaats is voor **28 kinderen**. De 26 letters van het alfabet en de karakters # en \$. # en \$ die gebruikt worden als resp. scheidingstekens en eindtekens in veralgemeende suffixbomen. Dit is ook wat al gebeurde in de bestaande C++ implementatie. De exacte waarden van het scheidings- en eindteken zijn niet belangrijk, de enige eigenschappen die ze moeten hebben is dat ze niet voorkomen als teken in de proteïnen of peptiden, en verschillend zijn van elkaar.

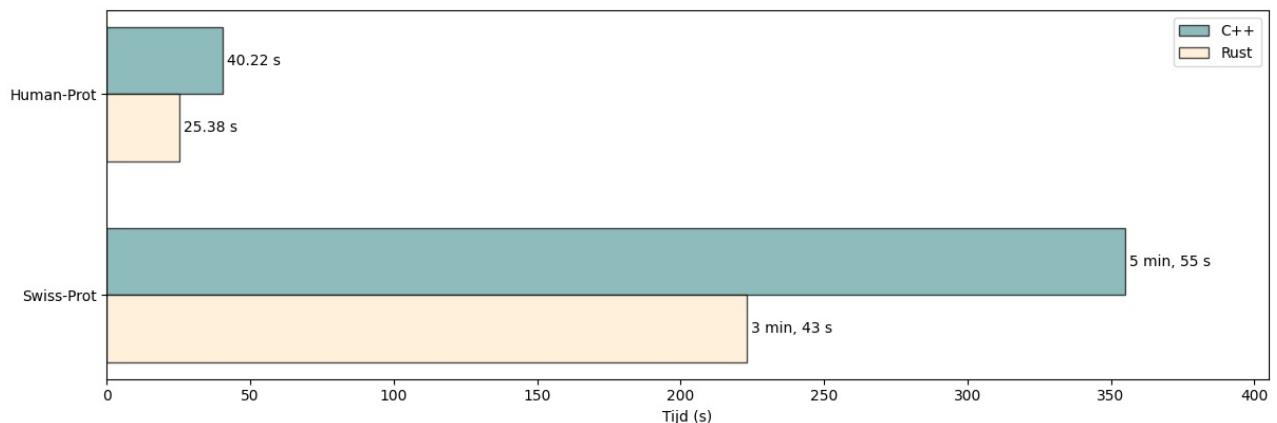
Een andere aanpak zou kunnen zijn om HashMaps te gebruiken. Het totale geheugengebruik zal hierdoor afnemen naar ongeveer 60% van het huidige verbruik. Dit gaat echter ten koste van performantie tijdens het zoeken, aangezien voor elke opvraging van een kind een hash berekend moet worden. Hoe dan ook blijft het **geheugengebruik extreem hoog**, welke implementatie ook gekozen wordt.

Het vergelijken van de implementaties heb ik opgesplitst in twee stukken:

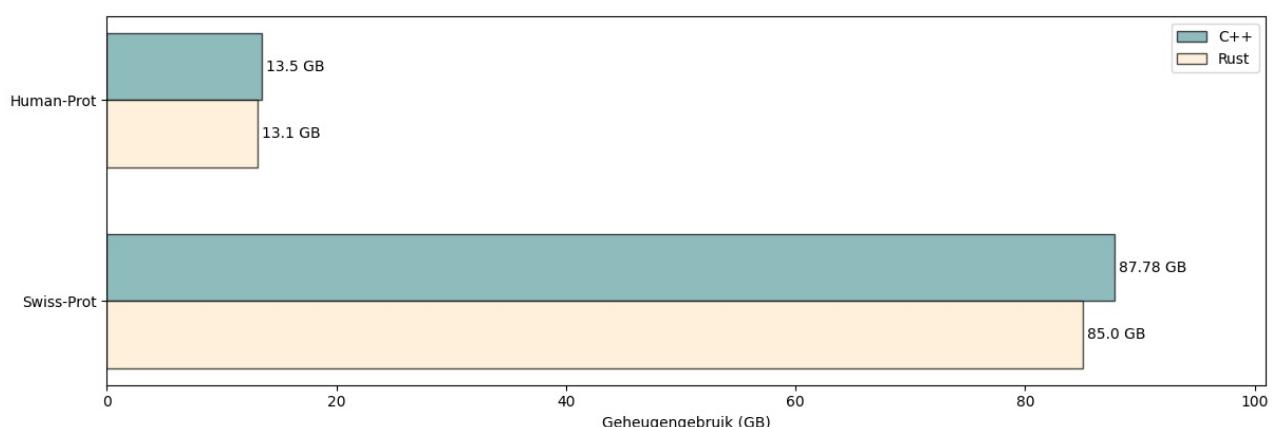
1. Het opbouwen van de indexstructuur. Hierbij ligt de hoofdfocus op de indexgrootte aangezien deze de primaire beperkende factor is.
2. Zoeken in de indexstructuur. Hierbij ligt de focus op de snelheid van het zoeken.

Opbouwen

Om een representatief resultaat te krijgen, kijken we steeds naar het gemiddelde van 10 uitvoeringen. De uitvoeringsijd en het geheugengebruik zijn gemeten aan de hand van het Unix `time` commando. De resultaten hiervan terug te vinden in Figuur 2.3.



(a) Tijd nodig om de suffixboom op te bouwen.



(b) Maximaal gebruikt geheugen tijdens het opbouwen van de suffixboom.

Figuur 2.3: Vergelijking tussen C++ en Rust implementaties voor het opbouwen van de suffixboom a.d.h.v. het algoritme van Ukkonen. De tijd en het geheugengebruik zijn gemeten met het Unix `time` commando. Als invoerbestand gebruiken we hier de Swiss-Prot en Human-Prot proteïnedatabank.

Uit deze grafieken vallen twee duidelijke conclusies te trekken.

1. De implementatie in Rust is $\pm 33\%$ sneller.
2. Het geheugengebruik is erg vergelijkbaar. Dit valt te verwachten aangezien beide implementaties 8 bytes nodig hebben per *pointer* en evenveel plaats voorzien voor de kinderen. Het kleine verschil valt te verklaren vanwege één veld uit de C++ implementatie dat niet bijgehouden wordt in de Rust implementatie. Dit veld is de diepte van de top in de boom. Op de enkele plaatsen waar dit nodig is, kan gebruikgemaakt worden van andere variabelen om tot een equivalent resultaat te komen.

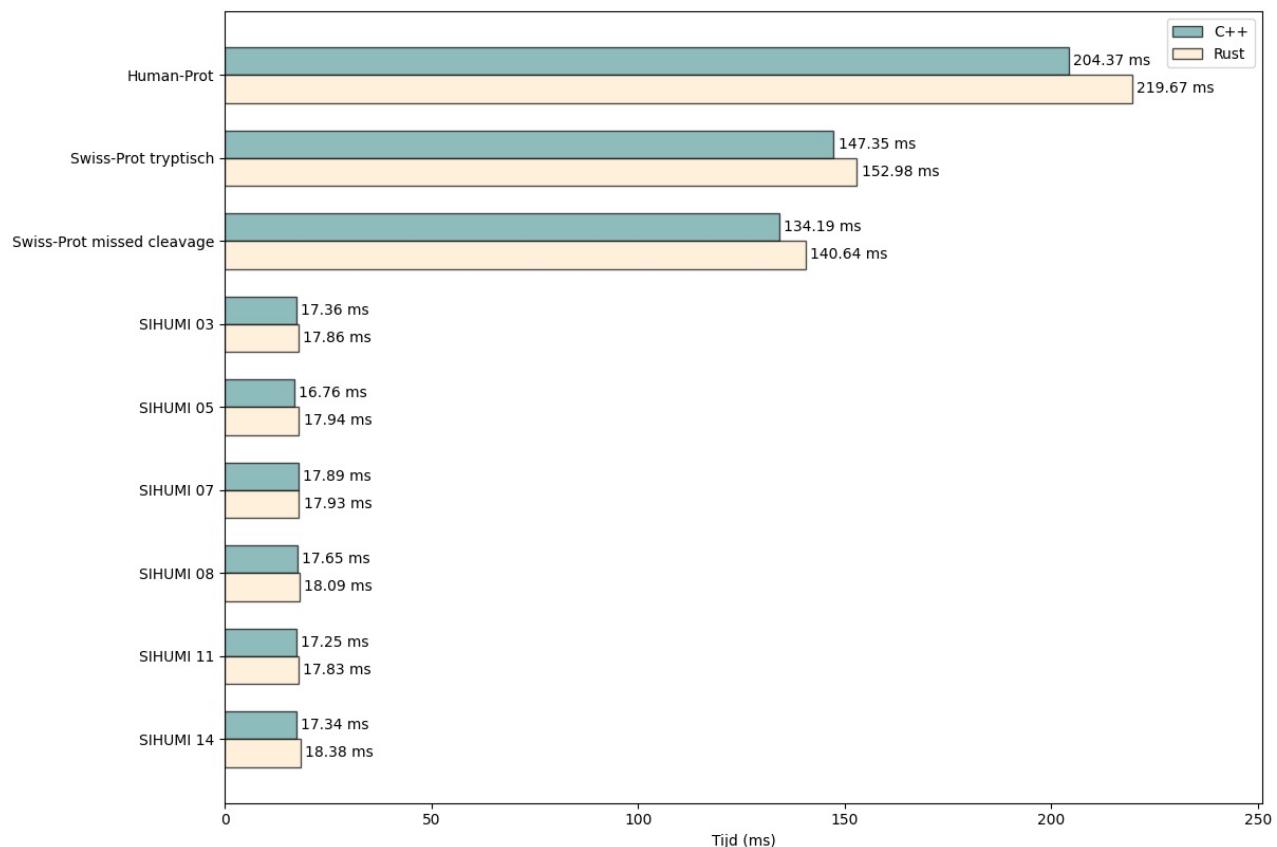
Zoeken

Bij het zoeken zijn er twee belangrijke toepassingen om de snelheid te meten.

1. Zoek totdat we weten of er een match bestaat voor de peptide.
2. Zoek totdat er een match is, en doorzoek daarna de volledige subboom om alle informatie van de bladeren op te halen.

Zoek een match Het meten van de zoektijd tot een match is een belangrijke indicatie van performantie, omdat bepaalde stukken informatie voorberekend kunnen worden voor elke top in de boom. Hier voor wordt informatie uit de bladeren tot aan de top van de boom gepropageerd. In ons geval is dit bijvoorbeeld de LCA⁸ van de taxon IDs. Dit laat toe om het zoekproces te stoppen zodra er een match is. De top waarin het zoeken stopt, zal de voorberekende LCA waarin we geïnteresseerd zijn bevatten. Dit is de LCA van de taxa die horen bij de proteïnen waarvan de gezochte peptide een substring is.

Figuur 2.4 toont de nodige tijd om alle peptiden van de gebruikte peptidebestanden éénmalig te zoeken totdat er een (mis)match was voor de peptide. De grafiek bevat de gemiddelde resultaten van 5000 uitvoeringen, maar zelfs dan bleven de resultaten wat schommelen. Doordat de te meten tijd zo klein is, kan de kleinste invloed van omgevingsfactoren al voor een zichtbaar verschil zorgen. Dit kan bv. een achtergrondproces zijn, maar ook invloed van een andere VM die op de fysieke machine bezig is. Dit was ook merkbaar tijdens het testen, waar de verschillen tussen twee opeenvolgende uitvoeringen vaak groter waren dan het verschil tussen de C++ en Rust implementatie. Toch kunnen we besluiten dat de **C++ implementatie een beetje performanter** is.



Figuur 2.4: Uitvoeringstijd in milliseconden voor het zoeken tot een match voor alle peptidebestanden. Deze resultaten zijn het gemiddelde van 5000 uitvoeringen. Eén iteratie wordt gezien als eenmaal elke peptide die deel is van het peptidebestanden te zoeken in de suffixboom, en te stoppen wanneer er een (mis)match gevonden is. Het meten van de tijd is gebeurd in de code zelf.

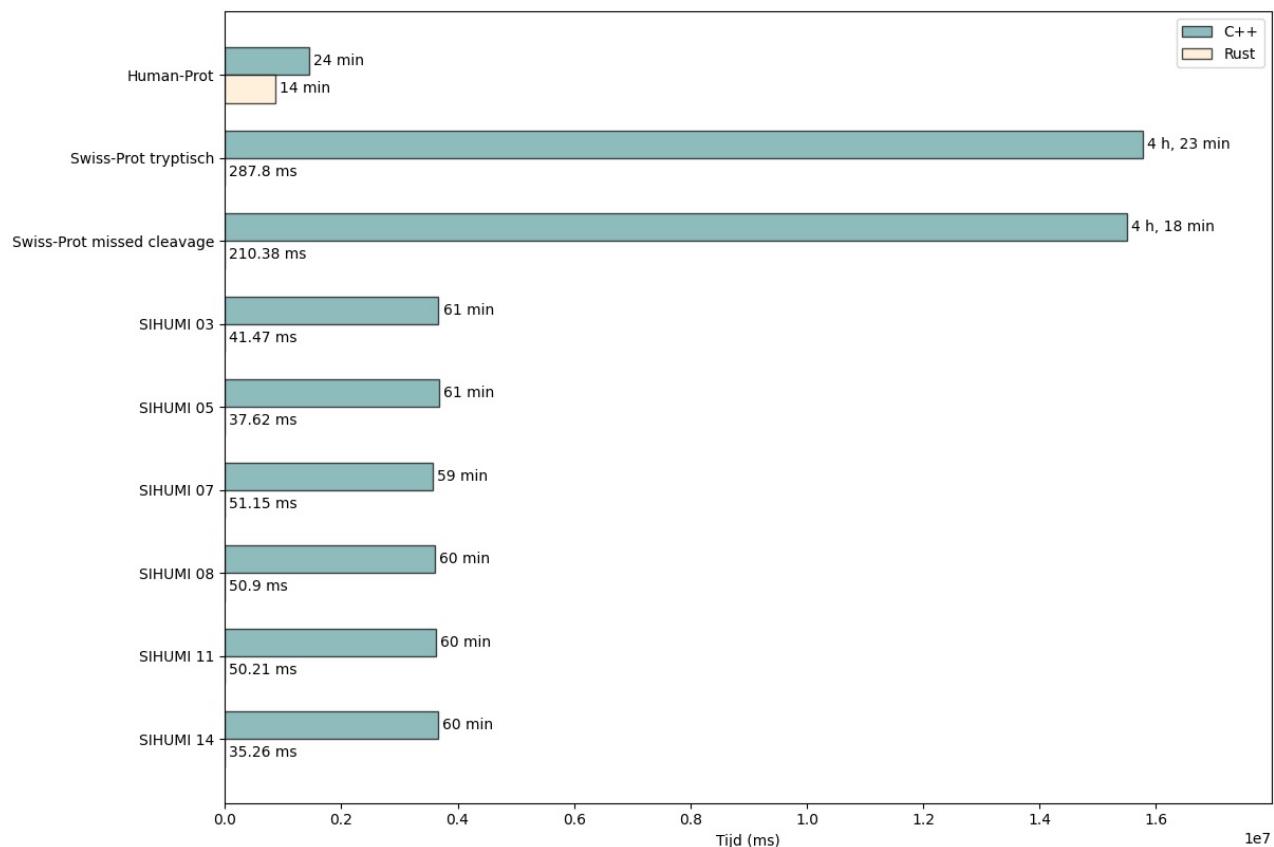
Het verschil met de huidige implementatie van UniPept is aanzienlijk. Daar duurt het op dit moment 2 minuten en 12 seconden om alle peptiden van het strikt tryptische Swiss-Prot peptidebestand te zoeken, en maar liefst 30 minuten en 37 seconden voor het peptidebestand met *missed cleavages*.

⁸Lowest Common Ancestor, Laagste Gemeenschappelijke Voorouder. Dit is de meest specifieke top in een boomstructuur (d.w.z. de top zo diep mogelijk in de boom) die een ouder is van een elke top in de gegeven set van toppen.

Dit is resp. $\frac{132\,000}{152\,98} = 857$ en $\frac{1\,837\,000}{140\,64} = 13\,000$ keer trager. Het gebruik van de suffixboom heeft echter ook nadelen ten opzichte van de huidige implementatie. Deze laatste gebruikt slechts 6.7 GiB geheugen, en dit kan zelfs nog naar beneden. Het geheugengebruik tijdens het opbouwen van de huidige UniPept index wordt namelijk gedomineerd door een merge sort stap. Het is mogelijk deze stap minder geheugen te geven, maar dan zal het sorteren wel langer duren. De belangrijkste conclusie is echter dat de implementatie aan de hand van suffixbomen ongeveer 13 keer meer geheugen in neemt dan de huidige index.

Zoek match en haal informatie over kinderen op Indien we alle proteïnen willen vinden waar een peptide mee matcht, dan moeten we de volledige subboom doorzoeken, startende van de top waar de match eindigt. De reden hiervoor is dat alle bladeren in deze subboom de gematchte proteïnen en de bijbehorende taxonomische informatie bevatten.

Figuur 2.5 bevat een overzicht van de nodige zoektijd voor beide implementaties op alle peptidebestanden. We zien duidelijk dat er hier een **significant verschil is tussen de C++ en Rust implementatie**. Vermoedelijk komt dit door de **andere memory layout** die ontstaat doordat de Rust implementatie één grote vector gebruikt, terwijl de C++ implementatie losse toppen gebruikt die verspreid liggen in het geheugen. Deze verbeterde lokaliteit zal het aantal cache-hits verhogen, wat op zijn beurt de performantie ten goede komt.

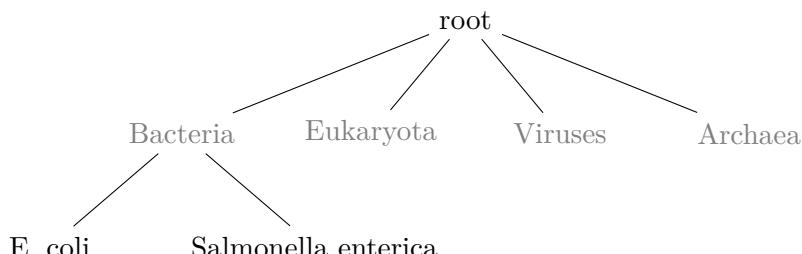


Figuur 2.5: Uitvoeringstijd inclusief het doorzoeken van de volledige subboom na match voor alle peptidebestanden. Deze resultaten zijn het gemiddelde van 10 uitvoeringen. Eén uitvoering wordt gezien als één keer elke peptide die deel is van het peptidebestand te zoeken in de suffixboom, en bij een match de volledige resterende subboom te doorzoeken. Dit toont de tijd die nodig is om informatie uit de bladeren op te halen voor alle proteïnen waar een peptide substring van is. Het meten van de tijd is gebeurd in de code zelf.

2.3 Taxon ID aggregatie

Eén van de belangrijkste analyses die UniPept aanbiedt, is de **taxonomische analyse** waarbij uitgezocht wordt met welke organismen de peptiden uit een staal overeenkomen. Aangezien peptiden kunnen matchen met proteïnen uit verschillende organismen, moet er een manier gekozen worden om deze **informatie te aggregeren**. Een optie is om een schatting te maken en het organisme met de grootst mogelijke kans te nemen. UniPept kiest echter voor een andere aanpak waarbij de informatie **conservatief** veralgemeend wordt. Er is namelijk geen manier om met zekerheid te zeggen uit welke proteïne de peptide effectief komt. Dit komt omdat gebruikers enkel een verzameling peptiden als input geven. Geen extra info die toelaat om te snoeien zonder mogelijk correcte informatie te verliezen. Anders gezegd: UniPept zal enkel info geven die geldt voor alle gematchte proteïnen. Eén van deze stukjes informatie is het taxon ID. UniPept zal niet de lijst van alle mogelijke taxon IDs teruggeven omdat dit twee nadelen heeft. Ten eerste kan dit een erg grote lijst worden indien de peptide met erg veel proteïnen matcht. Ten tweede zou dit ook vereisen om altijd de volledige subboom na een match te overlopen. In plaats daarvan gaan we **via voorberekeningen de taxon IDs aggregeren gebruikmakende van de NCBI taxonomy⁹ database [13, 60]**. Met andere woorden, we gaan op zoek naar de laagste gemeenschappelijke voorouder van alle taxon IDs die in de bladeren van de subboom zitten van een bepaalde top. Hiervoor bestaan verschillende strategieën die al uitgewerkt zijn in UMGAP [28, 19], en die hier herbruikbaar waren.

Initieel hebben we gekeken naar het LCA*¹⁰ algoritme. Dit is een heuristiek gebaseerd op de LCA⁸. Bij LCA* zoeken we het meest specifieke taxon in de boom die ofwel een ouder of kind is van elke taxon in de boom. Anders gezegd is dit de LCA van een verzameling taxa, nadat we alle taxa verwijderd hebben die ouder zijn van minstens één taxon in die verzameling [28]. Figuur 2.6 legt dit uit aan de hand van een klein voorbeeld. Het voordeel van LCA* ten opzichte van LCA is dat de resulterende data langer exact blijft. De LCA van een verzameling van taxon IDs zal namelijk altijd als waarde 1 hebben, dus de root zijn, zodra één element in de verzameling als root geannoteerd is. Dit gedrag willen we zo veel mogelijk vermijden.



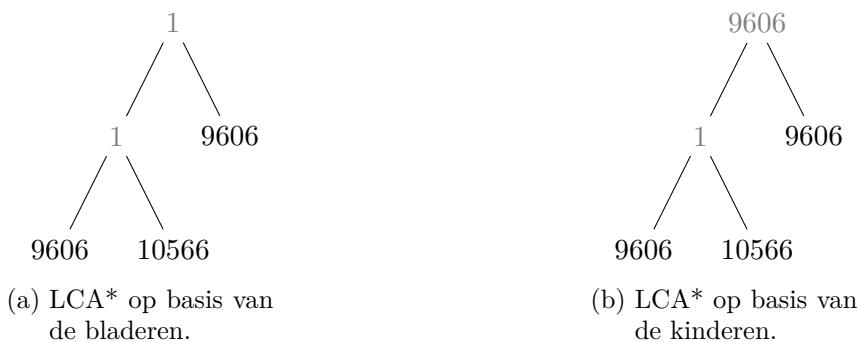
Figuur 2.6: Sterk versimpelde voorstelling van de NCBI taxonomy. We berekenen de LCA en LCA* van de verzameling van zwarte toppen. De grijze toppen zijn *placeholders*. De LCA van de verzameling {root, E. coli, Salmonella enterica} is vanzelfsprekend root, aangezien dit de wortel van de boomstructuur is. Bij het berekenen van de LCA* verwijderen we echter alle taxa die ouder zijn van minstens één taxon in de verzameling. Root is een ouder van zowel E. coli als Salmonella enterica, en wordt daarom niet gebruikt tijdens het berekenen van de LCA*. De LCA* wordt dus berekend door de LCA van de verzameling {E. coli, Salmonella enterica} te zoeken. Het resultaat hiervan is de top Bacteria. Dit illustreert hoe LCA* beter overweg kan met “te algemene” taxonomische annotaties.

⁹Dit is een boomstructuur die de evolutieaire relaties tussen organismen representeert. De laagste gemeenschappelijke voorouder van twee bladeren in deze boom representeert het evolutionair punt waarop deze twee organismen van elkaar afgesplitst zijn. Elke top in deze taxonomische boom heeft een uniek taxon ID, waarbij de wortel van de boom ID met waarde 1 heeft.

¹⁰LCA = Lowest Common Ancestor, Laagste Gemeenschappelijke Voorouder. De * wijst erop dat dit algoritme een variant is.

Om het geaggregeerde taxon ID van elke interne top in de boomstructuur efficiënt voor te berekenen, was het idee om dit te doen op basis van de directe kinderen van die top. Hiermee bedoelen we de eerstegraads kinderen. Deze liggen één niveau lager in de boomstructuur, en hebben als ouder allemaal dezelfde top waarvan we het taxon ID willen berekenen. De andere, tragere optie is om dit te doen op basis van de bladeren van de subboom van de top. Door het te doen aan de hand van de directe kinderen van de top kunnen we in één bottom-up sweep van de boom alle taxon IDs berekenen. Dit bleek echter niet mogelijk in combinatie met LCA* omdat gebruikmaken van de directe kinderen een ander resultaat geeft dan gebruikmaken van de bladeren van de subboom. Figuur 2.7 toont een minimaal voorbeeld uitgewerkt voor beide strategieën. De lichtgrijze toppen zijn ingevuld aan de hand van aggregatie, terwijl de zwarte toppen gegeven zijn. Onderstaande uitleg behandelt de werkwijze bij beide opties.

- Het toepassen van LCA* voor het berekenen van de top op basis van de bladeren van de boom ($\{9606, 10566, 9606\}$) heeft als resultaat 1 voor de wortel van de boom. 9606 en 10566 zijn geen ouder of kind van elkaar, dus zal LCA* hetzelfde doen als LCA. De kleinste gemeenschappelijke ouder van deze twee taxa is 1.
- Het toepassen van LCA* op basis van de directe kinderen geeft als resultaat 9606. Dit valt simpel te verklaren aangezien de LCA* van de linker subboom 1 is. Als we daarna dan de LCA* van $\{1, 9606\}$ nemen, wordt 1 verwijderd. Aangezien dit een ouder is van 9606. De LCA van 9606 is gewoon zichzelf.



Figuur 2.7: Minimaal voorbeeld van de 2 aggregatiestrategieën gebruikmakende van LCA*. De grijze toppen zijn berekend aan de hand van een LCA*, terwijl de zwarte toppen gegeven zijn. In de NCBI databank stelt het organisme met NCBI taxon ID 10566 het *Human papillomavirus* voor, organisme met ID 9606 de *Homo sapiens* en ID 1 is een representatie van de wortel van de volledige NCBI taxonomy boomstructuur.

Het berekenen van de **LCA* op de eerste manier is echter niet schaalbaar voor de volledige suffixboom**. Om een idee van grootorde te geven: de suffixboom voor de Swiss-Prot dataset bevat in totaal 328 922 516 toppen, waarvan 206 523 693 bladeren.

Daarom hebben we **voorlopig toch voor de standaard LCA aggregatie manier** gekozen. Deze laat wel toe de toppen op deze efficiëntere manier te aggregeren. UMGAP biedt twee implementaties aan om de LCA te berekenen voor een gegeven verzameling van taxa. Gebruikmakende van RMQs¹¹ of een boomstructuur. Mijn implementatie maakt gebruik van de RMQ implementatie aangezien deze significant sneller is (8 min 58 sec vs 20 min en 25 sec voor de Swiss-Prot databank) in mijn toepassing. Tot slot heb ik voor Swiss-Prot ook eens vergeleken hoe groot de behaalde tijdsvermindering is als we de LCAs aggregeren aan de hand van de directe kinderen, vergeleken met aggregatie op basis van de bladeren. Bij het aggregeren op basis van de bladeren met de RMQ implementatie was de uitvoeringstijd maar liefst 12 uur, 19 minuten en 16 seconden. Dit is dus een extreem groot verschil.

¹¹Range Minimum Queries

2.4 Conclusie

Het is duidelijk dat suffixbomen erg **performant** zijn voor het zoeken van willekeurige peptiden in een grote verzameling van proteïnen. Zelfs als we alle bladeren onder een interne knoop voor een match willen afgaan, valt dit mee. Bovendien is ook het opbouwen van de indexstructuur iets wat relatief snel gaat.

Door de eigen **implementatie in Rust**, kunnen we ook wat tijd besparen ten opzichte van een equivalente C++ implementatie. Een deel van de **tijdwinst** zit in het **opbouwen van de boom**, maar vooral tijdens het zoeken, wanneer **informatie uit de bladeren** gehaald moet worden. Vermoedelijk ligt de andere geheugenstructuur hiervoor aan de basis.

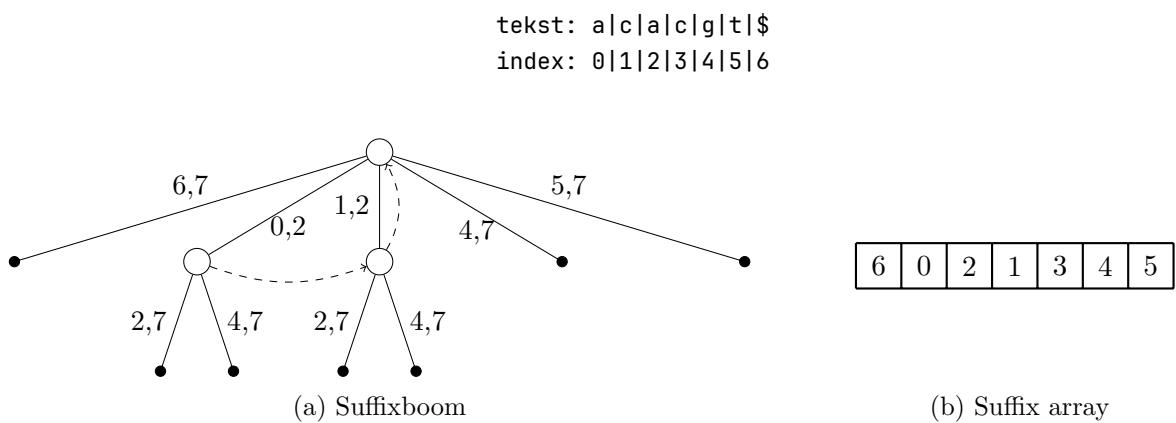
Ondanks de veelbelovende resultaten op vlak van snelheid is er een keerzijde aan de medaille. Het **geheugengebruik is zo groot dat we op zoek moeten naar een andere indexstructuur**. Voor de Swiss-Prot databank gaat het geheugengebruik al boven 85 GB als we ook de taxon IDs voorberekenen, terwijl ons einddoel is om de indexstructuur op te bouwen voor de volledige UniProt-KB databank. Rekening houdende met het lineaire gedrag van het geheugengebruik wil dit zeggen dat alles nog \pm 500 maal opgeschaald moet worden. Dit zou echter vereisen dat we een server nodig hebben met ongeveer 45 TB aan RAM. We hebben echter geen dergelijke machine ter beschikking. Daarom moeten we op zoek naar andere indexstructuren die minder geheugen vereisen, eventueel met een impact op de zoeksnelheid. In de praktijk hebben we namelijk wat marge om aan zoeksnelheid in te boeten, aangezien suffixbomen meerdere grootteordes sneller zijn dan de huidige UniPept index. Een belangrijke voorwaarde hierbij is natuurlijk dat dit enkel acceptabel is als we hierdoor volledig UniProtKB kunnen indexeren.

3 SUFFIX ARRAYS

Een tweede datastructuur die we in meer detail bekijken, is de suffix array. Het voordeel van deze datastructuur is het lagere geheugengebruik in vergelijking met suffixbomen.

3.1 Wat zijn suffix arrays?

Suffix arrays (SAs) zijn een geheugenefficiëntere voorstelling voor de bladeren van een suffixboom. In plaats van een boomstructuur met een veel pointers maken ze gebruik van **een array die de volgnummers van elke suffix in de originele string bevat**. Deze volgnummers worden lexicografisch gesorteerd op basis van de overeenkomstige suffix. Figuur 3.1 geeft een voorbeeld van een suffixboom en suffix array opgebouwd over de tekst `acacgt$`.



Figuur 3.1: Suffixboom en suffix array voor de string `acacgt$`.

Wanneer we de bladeren van de suffixboom van links naar rechts doorlopen, dan zien we dat dit overeen komt met de suffix array. Dit is de link tussen deze twee datastructuren. Merk op dat **de suffix array minder data bevat ten opzichte van de suffixboom**. De interne knopen en suffix links uit de suffixboom ontbreken. Indien deze informatie ook nodig is, kan gebruik gemaakt worden van zogenaamde Enhanced Suffix Arrays (ESAs). Hierbij worden naast de suffix array nog drie extra tabellen bijgehouden: de Longest Common Prefix (LCP), child en suffix link tabellen.

3.2 Complexiteit

Aangezien de suffix array bestaat uit de volgnummers van de lexicografisch gesorteerde suffixen, kunnen we deze suffixen aan de hand van traditionele sorteeralgoritmen zoals merge sort [46] in $O(n^2 \log n)$ tijd en $O(n^2)$ geheugen sorteren, wat onmiddellijk de bijhorende suffix array als resultaat oplevert. Hierbij is n de lengte van de tekst. De totale tijdscomplexiteit hiervan is $O(n^2 \log n)$ en niet $O(n \log n)$ aangezien elke vergelijking van twee suffixen in het slechtste geval n karaktervergelijkingen vereist. Ondertussen bestaan er echter verschillende algoritmen die een tijdscomplexiteit van $O(n)$ bereiken voor het opbouwen van de suffix array van een tekst [47, 31, 52, 33, 45]. Bovendien vereisen deze veel minder geheugen dan een equivalente suffixboom. Sommige implementaties vereisen slechts $5n + O(1)$ geheugen [33, 45, 21].

Zoeken in een suffix array kan in $O(m \log n)$. Hierbij is n opnieuw de lengte van de tekst, en m de lengte van de zoekstring. Het resultaat van deze zoekopdracht is een interval in de suffix array waarbinnen de matches liggen, of een leeg interval indien er geen matches zijn. Indien we effectief alle matches willen ophalen moeten we deze allemaal overlopen en komt de totale complexiteit uit op $O(m \log n + |Occ|)$ met $|Occ|$ het aantal gevonden matches in de tekst.

3.3 Bestaande implementaties

Aangezien er meerdere sterk geoptimaliseerde implementaties bestaan voor het opbouwen van een suffix array, vergelijken we eerst de performantie van deze implementaties. Hieruit kunnen we nadien een snelle en geheugenefficiënte implementatie selecteren. Tabel 3.1 bevat een overzicht van verschillende algoritmen. Voor sommige werden verschillende implementaties getest.

Algoritme	Programmeertaal	Tijd (in s)		Geheugen (in GB)	
		32-bit	64-bit	32-bit	64-bit
libdivsufsort ^a	C	15.01	15.97	1.03	1.86
libdivsufsort ^b	Rust + bindings to C	16.00	15.52	1.03	1.86
libdivsufsort ^c	Rust	20.23	-	1.03	-
dark archon a4 ^d	C	39.34	-	1.09	-
libsais ^e	C	6.38	6.46	1.03	1.86
SA-IS ^f	C++	24.39	-	3.80	-
SA-IS ^g	C++	18.73	-	1.46	-
radixSA ^h	C++	9.74	11.26	2.11	3.52

Tabel 3.1: Uitvoeringstijd en maximaal geheugengebruik voor het opbouwen van een suffix array aan de hand van verschillende algoritmen voor de Swiss-Prot proteïnedatabank. Indien er een 32-bit en 64-bit integer implementatie beschikbaar was, werden deze allebei getest. Een - staat voor niet getest. Deze testen werden lokaal uitgevoerd op een M1 Pro MacBook Pro. De specificaties hiervan zijn terug te vinden in tabel 1.7.

^a<https://github.com/y-256/libdivsufsort>

^b<https://github.com/baku4/libdivsufsort-rs>

^c<https://github.com/fasterthanlime/stringsearch/tree/master/crates/divsufsort>

^d<https://github.com/kvark/dark-archon>

^e<https://github.com/IlyaGrebnov/libsais>

^f<https://github.com/Tascate/Suffix-Arrays-in-CPP>

^g<https://github.com/sile/sais>

^h<https://github.com/mariusmmni/radixSA64>

We kunnen concluderen dat **libsais duidelijk de snelste implementatie is om Swiss-Prot te indexeren. Samen met libdivsufsort gebruikt deze de laagste hoeveelheid geheugen**, wat libdivsufsort ook interessant maakt. Een ander voordeel dat libsais en libdivsufsort gemeen hebben, naast hun minimale geheugengebruik, is dat ze allebei een 64-bit integer implementatie hebben. Dit is belangrijk voor het indexeren van UniprotKB omdat de totale tekst langer is dan het grootste 32-bit integer. Dit zorgt ervoor dat alle 32-bit integer implementaties onbruikbaar zijn voor dit einddoel. Tot slot valt ook te zien dat het verschil tussen de C en Rust versie die bindings heeft naar de C-code klein is. De overhead van het oproepen van de C-code uit Rust is dus minimaal.

3.3.1 Enhanced suffix arrays

Om een indicatie te hebben van de extra hoeveelheid geheugen die nodig is om gebruik te maken van Enhanced Suffix Arrays kunnen we via de libsais bibliotheek de LCP array berekenen. Deze LCP array is slechts één van de drie extra tabellen die deel zijn van ESAs, maar het is voor ons wel de meest interessante. De baseline waarmee we willen vergelijken is de hoeveelheid geheugen die we nodig hebben om enkel de SA op te bouwen. Voor de Swiss-Prot proteïnedatabank was dit

2.3 GB. Als we ook de LCP array berekenen, loopt het maximale geheugengebruik op naar 5.72 GB. **Het berekenen van deze extra array vraagt dus meer dan dubbel zoveel geheugen, waardoor we deze optie niet verder verkennen.** Bovendien berekenen we deze LCP array om het voorberekenen van de geaggregeerde taxon ID voor elke interne top van de overeenkomstige suffixboom mogelijk te maken. Daarna moet er op basis van deze SA en LCP array nog een compacte representatie gevonden worden van de boomstructuur. Dit is niet vanzelfsprekend.

3.4 Toepassen van suffix arrays op een proteïnedatabank

Het moeilijkste deel van onze probleemstelling is het opbouwen van de suffix array. Dit kunnen we oplossen aan de hand van de algoritmen uit sectie 3.3. Eens we die suffix array opgebouwd hebben, blijft er echter nog een stuk van ons probleem over. Eerst moeten we nog een mapping maken van de gevonden suffixen naar de bijbehorende proteïne. Op basis van deze proteïne wordt daarna de LCA gezocht.

3.4.1 Bouwen van de suffix array

Zoals in de inleiding (sectie 1.4) beschreven wordt, willen we Rust gebruiken vanwege de combinatie tussen *memory safety* en hoge performantie. We willen echter gebruikmaken van de al bestaande geoptimaliseerde algoritmen om een suffix array op te bouwen. Om beide doelen te bereiken, maken we gebruik van de interoperabiliteit tussen Rust en C/C++. Zo bestaan er al bindings¹ van Rust naar de originele implementatie van libdivsufsort[45] (in C). Ook al blijkt uit het testen dat dit algoritme voor het opbouwen van de indexstructuur over een kleinere proteïnedatabank niet het snelste is, is het geheugengebruik wel minimaal. Dit laat toe om te experimenteren met het opbouwen van een SA zonder al te veel extra werk, en al onmiddellijk te zien hoe het geheugengebruik evolueert.

Later hebben we zelf nog een simple Rust wrapper geschreven rond de libsais C-code. Hiervoor hebben we de `bindgen`² crate gebruikt. Op deze manier was het ook mogelijk om gebruik te maken van de snellere libsais algoritme eens we wisten dat SAs een efficiënte en schaalbare oplossing waren voor de probleemstelling.

Het nadeel van het gebruiken van deze bindings naar C-code is dat het oproepen van de effectieve C-code gebeurt in een `unsafe` blok. Hierdoor kunnen er geheugenfouten in het programma sluipen. Dit risico is echter miniem aangezien dit bestaande, geteste bibliotheken zijn. Bovendien zijn we ook zeker dat eventuele geheugenfouten enkel hierdoor kunnen ontstaan en is dit de verantwoordelijkheid van de ontwikkelaar van de bibliotheek. Dit is dus een afweging tussen optimale performantie (waarbij we het wiel niet hoeven heruit te vinden), en de garantie van *memory safety*.

3.4.2 Mapping van suffix naar proteïne

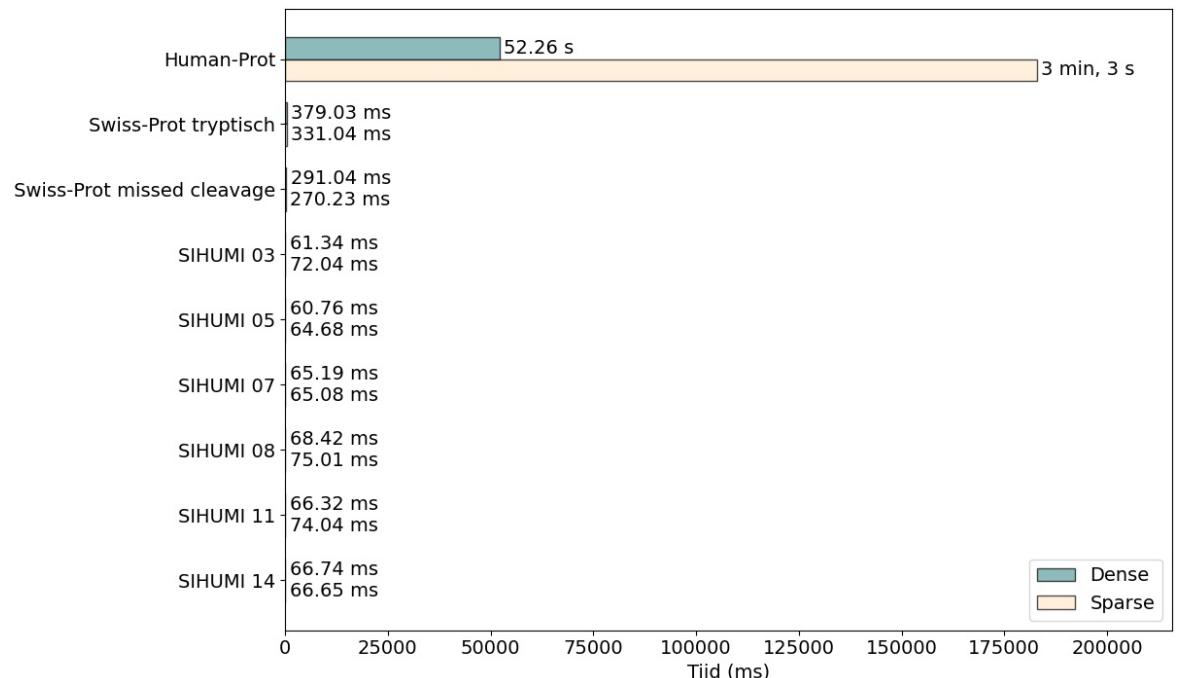
Bepalen welke proteïne hoort bij een bepaalde suffix kan op twee manieren. **Een eerste optie, laten we deze een *dense mapping* noemen, is om expliciet voor elke suffix bij te houden bij welke proteïne die hoort.** Dit kan aan de hand van een array die even lang is als het aantal suffixen. Het voordeel van deze aanpak is dat het vinden van de bijbehorende proteïne in $O(1)$ tijd kan, hiervoor is echter wel $O(m)$ geheugen nodig, met m de lengte van de totale tekst.

De tweede optie, aan de hand van een *sparse mapping*, is om enkel de eerste of laatste suffix per proteïne bij te houden. Het voordeel van deze aanpak is dat er minder geheugen nodig is, meer precies $O(p)$ geheugen met p het aantal proteïnen. Het nadeel is dan weer dat het vinden van de bijbehorende proteïne trager is. Dit neemt $O(\log p)$ tijd in beslag aan de hand van binair zoeken.

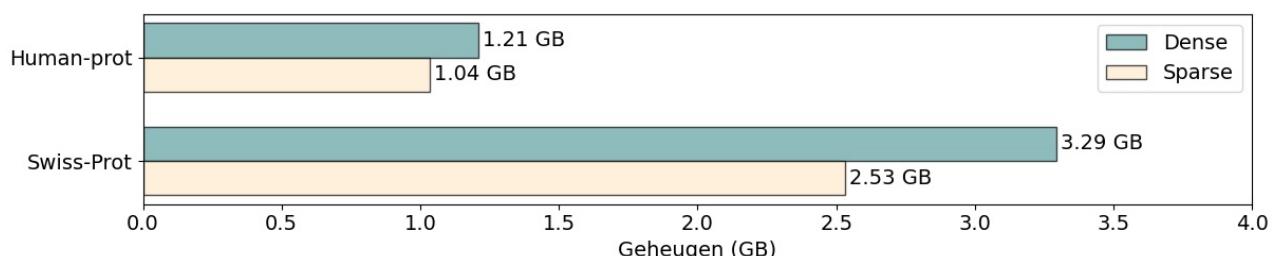
¹<https://crates.io/crates/libdivsufsort-rs>

²<https://crates.io/crates/bindgen>

In Figuur 3.2 wordt de uitvoeringstijd voor beide implementaties vergeleken. **Standaard** (en in alle komende testen) zullen wij gebruikmaken van de *sparse mapping* aangezien de performantie impact in de praktijk beperkt is, en er een significante hoeveelheid geheugengebruik uitgespaard kan worden. Zo loopt het verschil in geheugengebruik op tot 0.8 GB voor Swiss-Prot. Dit is maar liefst 25% van de indexgrootte bij de *dense mapping*. Het grote verschil in uitvoeringstijd bij het Human-Prot peptidebestand valt te verklaren vanwege het extreem hoog aantal matches dat daar te vinden is voor de korte peptiden. In de praktijk zullen we dit maximaal aantal matches echter limiteren (zie sectie 3.7.2). Dit zal op zijn beurt ook de overhead van de sparse mapping beperkt houden.



(a) Tijd nodig om alle matches te zoeken.



(b) Maximaal gebruikt geheugen tijdens het zoeken naar alle matches.

Figuur 3.2: Zoektijd en geheugengebruik bij het gebruik bij een *dense* of *sparse mapping* van de suffixen naar de proteïnen.

3.4.3 Berekenen van de LCA

Zoals eerder vermeld, bevat een suffix array geen informatie over de interne toppen die voorkomen bij een suffixboom. Dit zorgt ervoor dat het niet mogelijk is om op basis hiervan de LCA van de organismen voor te berekenen voor al deze interne toppen. Dit moet nu *on the fly* gebeuren tijdens het zoekproces zelf. Hierdoor is er ook **geen reden meer om LCA te gebruiken in de plaats van LCA***. LCA* was namelijk onze eerste keuze aangezien het resultaat hierbij minder onderhevig is aan proteïnen die een te algemene annotatie hebben. Bij suffixbomen zijn we daar echter van af moeten stappen om het voorberekenen efficiënter te maken.

3.5 Sparse en compressed suffix arrays

Om het geheugengebruik van suffix arrays verder te verkleinen, kan er gebruikgemaakt worden van sparse of compressed suffix arrays. In principe doen ze allebei hetzelfde. Er wordt namelijk slechts een stuk van de originele suffix array bijgehouden. Het verschil zit in welk stuk bijgehouden wordt. Zoals het gezegde *there is not such thing as free lunch* ons vertelt, heeft alles zijn voor- en nadelen. Het verkleinen van de SA heeft een negatieve impact op de zoekperformantie. We zullen namelijk slechts een deel van de peptide kunnen zoeken in de verkleinde SA. Dit deel van de peptide is natuurlijk korter dan de volledige peptide, en zal in het algemeen meer matches opleveren. Daarna moet voor elk van deze matches het niet-gematchte deel van de peptide nog gecontroleerd worden.

Sparse suffix arrays (SSAs) bouwen een suffix array op basis van elke k -de suffix van de input tekst. Indien we slechts de helft van de suffixen bijhouden, komt dit erop neer dat enkel suffixen die beginnen op een **even index in de inputtekst** bijgehouden worden. Bij **compressed suffix arrays (CSAs)** wordt daarentegen slechts elke k -de waarde van de SA bijgehouden. Indien we in dit geval de helft van de suffixen bijhouden komt dit erop neer dat enkel suffixen die op een **even index staan in de opgebouwde suffix array** over blijven.

Voor beide opties is de populairste manier om ze op te bouwen aan de hand van **sampling op de volledige SA**. Hierdoor blijft het maximale geheugengebruik tijdens het opbouwen identiek aan het gebruik van de volledige SA. De resulterende index zal wel kleiner zijn, waardoor de server die deze index zal hosten lagere geheugenvereisten heeft.

Het samplen van een opgebouwde suffix array is de meest gebruikte methode tot op vandaag, vanwege de bestaande sterk geoptimaliseerde implementaties van de klassieke SA constructiealgoritmen. Bij sparse suffix arrays is het beste algoritme qua tijdscomplexiteit tot nu toe een Monte Carlo algoritme dat $O(n)$ tijd en $O(b)$ geheugen nodig heeft en een Las Vegas algoritme dat $O(n\sqrt{\log b})$ tijd en $O(b)$ geheugen gebruikt. Hierbij is n de lengte van de tekst, en b het aantal effectief gebruikte suffixen in de SSA [18]. Van het Monte-Carlo algoritme is er een bestaande implementatie³. Wanneer we aan de hand hiervan een SSA met sparseness factor 3 voor Swiss-Prot opbouwen, blijkt dit niet alleen trager te zijn (± 10 minuten). Ook het geheugengebruik ligt merkelijk hoger (± 10 GB). Terwijl we slechts een tiental seconden nodig hebben om de standaard SA op te bouwen in combinatie met 1.8 GB RAM. Voor compressed suffix arrays bestaat er een algoritme dat een tijdscomplexiteit van $O(n)$ heeft in combinatie met $O(n \log \sigma)$ bits aan geheugen [29]. Hierbij is n de tekstlengte en σ de alfabetgrootte.

Het grootste nadeel aan deze algoritmen in de context van deze thesis is dat er nog geen sterk geoptimaliseerde implementaties bestaan. Bovendien zal de **factor van ingevoegde sparseness in ons geval altijd vrij klein** zijn om de zoektijden beperkt te houden (aangezien we werken met een erg grote dataset en vrij korte strings). Indien we dus een implementatie hebben om rechtstreeks een CSA of SSA te bouwen, maar met een grotere constante qua geheugengebruik, zal vanwege de kleine sparseness factor de winst snel verloren gaan. **In ons geval is een SSA interessanter dan een CSA** omwille van de verschillende restricties tijdens het zoeken die een SSA en CSA hebben.

Bij het gebruik van sampling factor k kunnen we bij een SSA alle sequenties van lengte k en groter zoeken. Bij een CSA is het mogelijk dat ook sommige sequenties die groter zijn dan k niet te vinden zijn, en sommige kortere wel. Dit omdat het mogelijk is dat er in het slechtste geval een gat van $n - \frac{n}{k}$ suffixen zit tussen twee opeenvolgende bijgehouden suffixen in de CSA. Hierbij is n de lengte van de invoertekst. Het zoeken van extreem korte sequenties is echter niet interessant omdat deze erg weinig informatie bevatten, terwijl het verliezen van enkele langere sequenties (zonder te kunnen voorspellen welke) net extra informatieverlies met zich meebrengt.

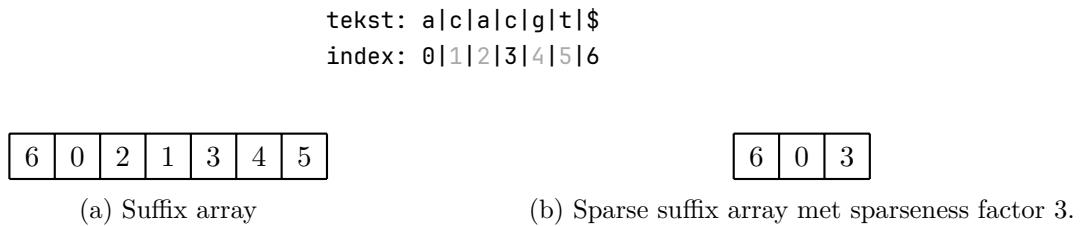
³<https://github.com/lorrainea/SSA/tree/main/MA>

Bovenstaand probleem kan opgelost worden door gebruik te maken van extra datastructuren. Zo kan een FM-index [14] gebruikmaken van een CSA, en kunnen we hierbij wel alle sequenties zoeken. Om dit te doen moeten we echter tijdens het opbouwen van de indexstructuur nog bijkomende onderdelen berekenen (zoals de BWT⁴ van de tekst). Dit verhoogt opnieuw het geheugengebruik en de uitvoeringstijd tijdens het opbouwen.

3.5.1 Zoeken met sparse suffix arrays

Het zoeken met een SSA is erg gelijkaardig aan het zoeken met een volledige SA. Een belangrijke **restrictie** is echter dat met de SSA **geen strings gezocht kunnen worden die kleiner zijn dan de sparseness factor k** . Bovendien heeft deze sparseness factor ook impact op de zoekperformantie. Bij het zoeken met sparseness factor k moeten we k verschillende zoekopdrachten uitvoeren. In zoekopdracht i worden de eerste $i - 1$ tekens van de zoekstring overgeslagen. In plaats van de zoekstring, zoeken we dus een suffix van de zoekstring in de sparse indexstructuur.

Voor elke matchende suffix (stel dat dit suffix s is) moet daarna gecontroleerd worden of de overgeslagen prefix van i tekens matcht met de i tekens die op posities $[s - i, s]$ in de geïndexeerde tekst staan. Naast de SSA moet dus ook de volledige tekst in het werkgeheugen bijgehouden om te zoeken in de (sparse) SA. Als dit zo is, dan matcht suffix $s - i$ met de gezochte string. Het eindresultaat is de unie van de resultaten van elke iteratie. Figuur 3.3 geeft een voorbeeld van het opzoeken van de peptide **acg** in de tekst **acacgt\$**. Hierbij wordt gebruikgemaakt van een SSA met sparseness factor $k = 3$.



Figuur 3.3: SA en SSA voor de tekst **acacgt\$**.

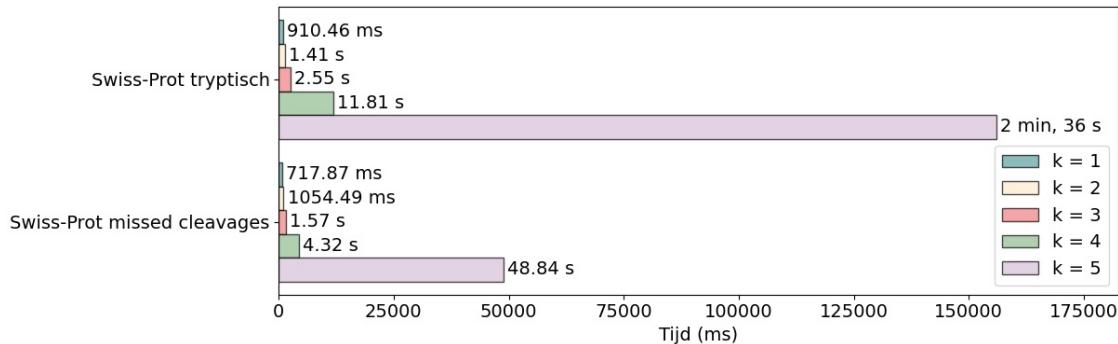
1. Sla 0 tekens over. Zoek **acg** met de SSA. We vinden geen matches.
2. Sla 1 teken over. Dit wil zeggen dat we **cg** zoeken met de SSA. Dit levert één match op met suffix 3 (op index 2 in de SSA).
 - a) Controleer of de suffix die deels gematcht is ook volledig matcht met onze zoekstring. Dit wil zeggen dat we de overgeslagen prefix van $i = 1$ tekens (in dit geval de letter **a**) moeten kunnen matchen met de eerste i tekens van suffix $3 - 1 = 2$.
 - b) We zien dat het eerste teken van suffix 2 inderdaad een **a** is. Dit kan rechtstreeks via de tekst gecontroleerd worden aan de hand van i karakters die vergeleken moeten worden. Suffix 2 is dus een match voor de zoekstring **acg**.
3. Sla 2 tekens over. Zoek **g** in de SSA. We vinden geen matches.
4. We concluderen dat enkel suffix 2 matcht met de gezochte string **acg**.

Performantie en indexgrootte

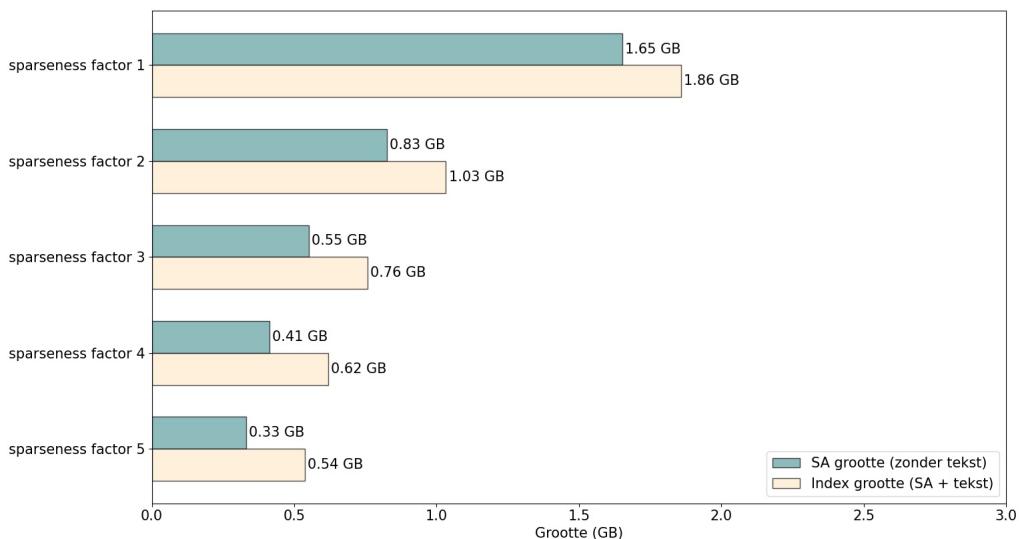
Aangezien de sparseness factor k impact heeft op de zoekperformantie is het belangrijk te zien hoe groot deze impact is, zodat we een goede keuze kunnen maken. Op het eerste gezicht lijkt het vergroten van deze factor enkel meer iteraties toe te voegen, maar dit heeft zware gevolgen. Deze extra iteraties zorgen ervoor dat er **kortere strings** in de SSA gezocht worden, die in het algemeen

⁴Burrows-Wheeler transformatie [6]

meer matches opleveren. Zeker wanneer de zoekstrings zelf al vrij kort zijn, wat voor UniPept typisch het geval is. In het tryptische Swiss-Prot peptidebestand zitten er bijvoorbeeld veel peptiden met lengte 5. Voor al deze matches moeten we controleren of de tekens die ervoor komen matchen met de overgeslagen prefix. Figuur 3.4 (a) visualiseert de impact op de zoektijd van het Swiss-Prot peptidebestand met en zonder *missed cleavages* in de Swiss-Prot databank. Deze bestanden worden hiervoor gebruikt omdat de kortste sequentie die ze bevatten 5 aminozuren lang is. Hierdoor blijft nog steeds elke peptide zoekbaar. Bij het Human-Prot zoekbestand zijn er ook kortere peptiden die we moeten overslaan, waardoor dit een slechte representatie is voor de evolutie van de zoektijd.



(a) Zoektijd voor het Swiss-Prot peptidebestand met en zonder *missed cleavages*. De zoekoperaties zijn uitgevoerd op een SSA gebouwd op basis van de Swiss-Prot proteïnedatabank, met $k = 1, 2, 3, 4, 5$.



(b) Grootte van de volledige index en SSA voor de Swiss-Prot databank met sparseness factor $k = 1, 2, 3, 4, 5$.

Figuur 3.4: Zoektijd en indexgrootte van een SSA voor Swiss-Prot met $k = 1, 2, 3, 4, 5$.

De zoektijd explodeert wanneer we de sparseness factor k verhogen, omdat een deel van de gezochtpeptiden bestaat uit 5 aminozuren. Wanneer we deze proberen zoeken in een SSA met sparseness factor 5, zoeken we eigenlijk enkel één letter in de SSA (de laatste van de peptide). Dit zal erg veel matches opleveren, en op zijn beurt erg veel werk vereisen om alle prefixen te controleren.

Anderzijds zal het verhogen van de sparseness factor de indexgrootte verkleinen. Hierbij is het belangrijk dat enkel de SA mee verkleint. Er blijft **altijd een vaste hoeveelheid geheugen nodig om de tekst zelf op te slaan**. In Figuur 3.4 (b) is duidelijk te zien dat, voor dit geval, de overgang van sparseness factor 4 naar 5 erg weinig winst heeft qua nodige opslagruimte, maar een grote impact heeft op de performantie.

Er zijn dus twee erg belangrijke bevindingen over sparse suffix arrays, en de manier waarop wij ze opbouwen.

1. Probeer de sparseness factor zo laag mogelijk te houden, zo blijft ook de zoektijd voor kortere peptiden beperkt.
2. Het maximale geheugengebruik voor het opbouwen van de SSA blijft constant onafhankelijk van de sparseness factor. Het introduceren van een sparseness factor verkleint enkel de resulterende SSA zodat er minder geheugen nodig is tijdens het hosten van de index.

Dit impliceert dat we de sparseness factor k enkel moeten verhogen om het geheugengebruik te beperken bij een al opgebouwde indexstructuur. Dit is vooral van toepassing voor UniProtKB waar het nuttig is om kort een krachtige machine te gebruiken die de SA bouwt, waarna een minder krachtige machine de SSA host. In het beste geval wordt **de sparseness factor zo gekozen zodat de SSA samen met de tekst net in het geheugen past** van deze minder krachtige machine.

3.6 Parallelisatie

Om het zoeken nog verder te versnellen, kan gebruikgemaakt worden van parallelisatie. We hebben namelijk een **groot aantal peptiden** waarvoor we telkens dezelfde **statische indexstructuur** moeten doorzoeken. Rust maakt dit proces vrij simpel omdat het *ownership* systeem dataraces voorkomt (behalve wanneer gebruikgemaakt wordt van *unsafe* code of het *interior mutability* patroon)[59]. Om een datatype te gebruiken in combinatie met multithreading moet deze de `Sync` en `Send` trait implementeren. Deze traits worden door het typesysteem automatisch afgeleid. Namelijk, wanneer alle componenten van een type aan de `Sync` en `Send` trait voldoen, dan voldoet je nieuwe type ook automatisch.

Uiteindelijk hebben we twee geparalleliseerde implementaties gemaakt. In de eerste wordt alles volledig zelf beheerd. Hierbij verdelen we zelf welke data naar welke thread gaat, worden de threads manueel opgestart en sluiten we ze ook zelf af. Op deze manier hebben we ook volledig controle over het aantal threads dat gebruikt wordt. In de tweede implementatie wordt gebruikgemaakt van de Rayon crate [53]. Deze laat toe om op een simpele manier een sequentiële lus over een variabele te paralleliseren. In ons geval was dit (nadat alle types voldeden aan de `Sync` en `Send` trait) zo simpel als het vervangen van `.iter()` door `.par_iter()`. Dit valt te zien in Codefragment 1. Ook in hier is het mogelijk om manueel een specifiek aantal threads te kiezen. Standaard gebruikt Rayon het aantal beschikbare logische cores op de machine, maar het instellen van een ander aantal kan aan de hand van één lijntje code.

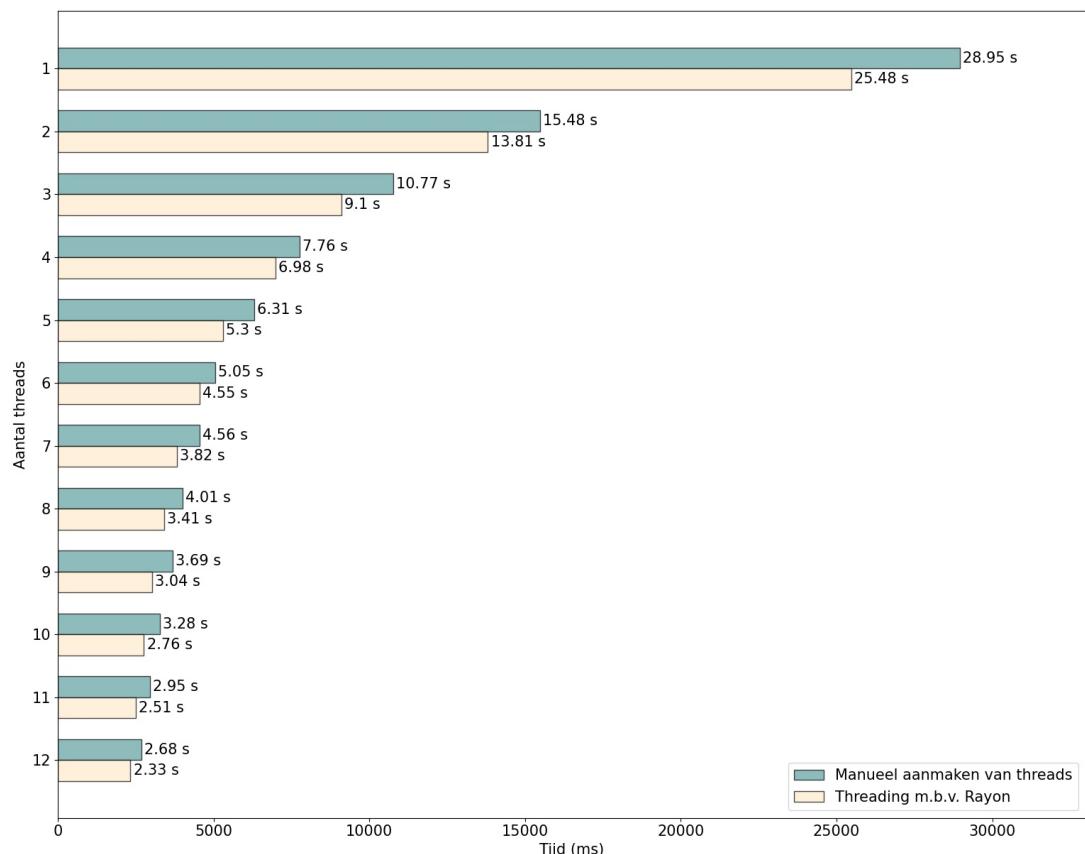
```
1 // Sequentieel
2 let results = peptiden
3     .iter()
4     .map(|peptide| search_peptide(peptide))
5     .collect();
6
7 // Parallel
8 let results = peptiden
9     .par_iter()
10    .map(|peptide| search_peptide(peptide))
11    .collect();
```

Codefragment 1: Vergelijking van de sequentiële en geparalleliseerde implementatie a.d.h.v. Rayon.

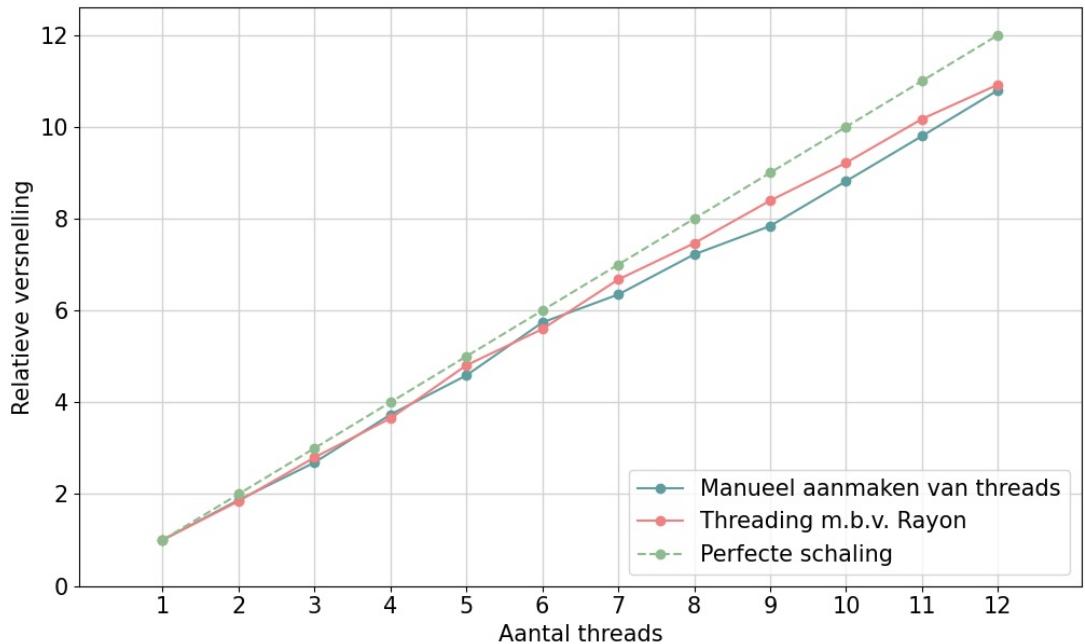
3.6.1 Manueel threaden vs Rayon

Aangezien we twee verschillende implementaties hebben, is het interessant om na te gaan hoe deze ten opzichte van elkaar presteren. Figuur 3.5 toont de evolutie van de zoektijden voor een verschillend aantal threads. We zien duidelijk dat de versie die gebruikmaakt van **Rayon net iets sneller** is en dat beide implementaties ongeveer **lineair schalen**. De schaling is niet perfect één-op-één ten opzichte van het aantal threads omdat het inlezen en uitschrijven van de output sequentieel blijft. Het verschil in uitvoeringstijd valt mogelijk te verklaren aan de manier waarop de data verdeeld wordt over de threads, in combinatie met een efficiëntere manier om de resultaten uit de threads te verwerken. In de eigen implementatie kreeg elke thread simpelweg $\frac{1}{x}$ van alle peptiden toegekend, met x het aantal threads. De resultaten moeten daarna aan de hand van enkele stappen uit de thread-scope gehaald worden zodat deze terug beschikbaar zijn voor de rest van het programma. Rayon maakt echter gebruik van een ingewikkelder systeem aan de hand van *work stealing* [54] om de data over de threads te verdelen. Nadien worden de resultaten onmiddellijk teruggegeven, alsof er geen parallelisatie gebeurt. Indien Rayon gebruikt wordt, zijn dus geen extra tussenstappen nodig om de resultaten uit de thread-scope te halen.

Naast het verschil in performantie zijn er nog andere voordelen verbonden aan het gebruik van Rayon. De **code is namelijk veel simpeler en daarom ook beter te onderhouden**. Dit motiveert onze keuze om finaal gebruik te maken van Rayon.



(a) Absolute uitvoeringstijd voor 1–12 threads.



(b) Relatieve versnelling ten opzichte van uitvoering op 1 thread.

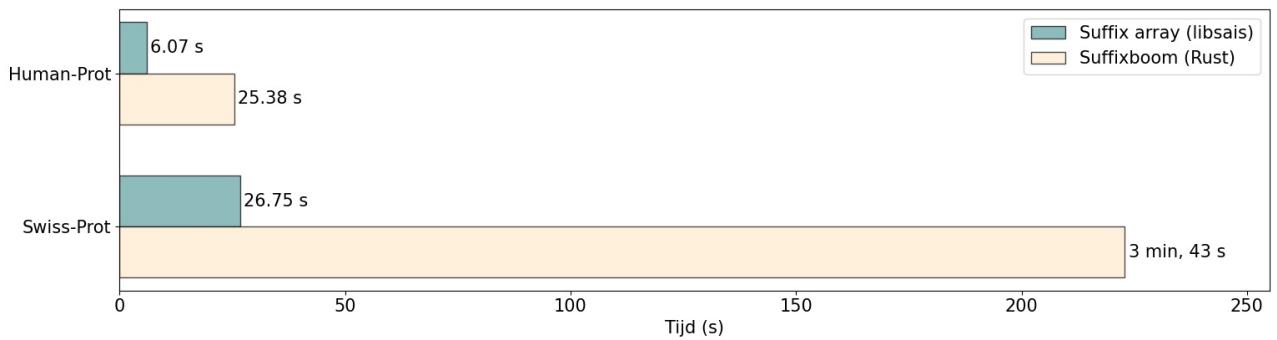
Figuur 3.5: Tijdsmeting om het strikt tryptische Swiss-Prot peptidebestand te zoeken in een index met sparseness factor 3 voor 5% van UniProtKB.

3.7 Suffix arrays vs suffixbomen

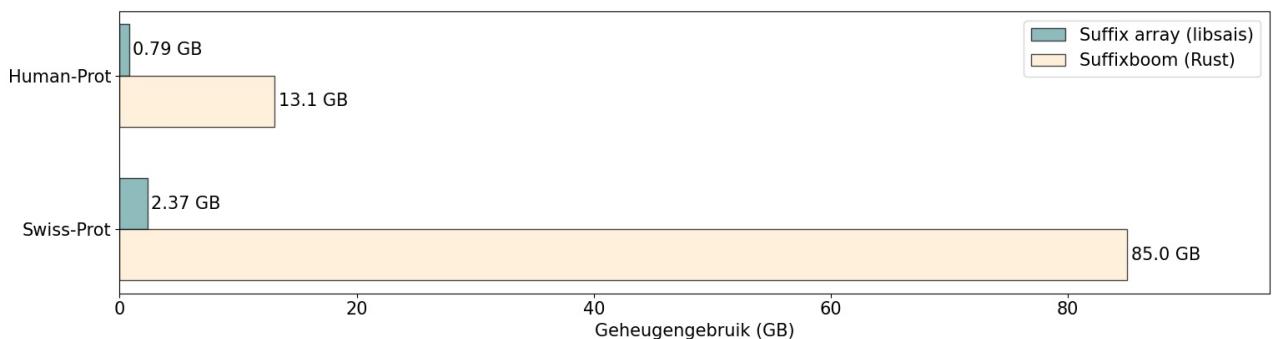
Nu we verschillende manier verkend hebben om suffix arrays op te bouwen, is het interessant om suffix arrays te vergelijken met suffixbomen. Op basis hiervan kunnen we vaststellen welke verbeteringen we gemaakt hebben, of deze indexstructuur goed genoeg schaalt om toepasbaar te zijn op de volledige UniProtKB databank en in welke tijd de peptiden gezocht kunnen worden.

3.7.1 Opbouwen

Figuur 3.6 visualiseert de tijd nodig om de indexstructuur op te bouwen in combinatie met het geheugengebruik. Er is duidelijk een **mooie tijdsinst** verkregen, wat een bonus is voor het lokaal opbouwen van indices. Voor het opbouwen van de index op UniProtKB is dit echter minder van belang aangezien dit proces slechts om de 8 weken moet gebeuren. Zolang de nodige CPU-tijd hiervoor niet langer dan één à twee dagen is, is dit acceptabel. Een veel belangrijkere vaststelling is het **maximale geheugengebruik tijdens opbouwen**. Ook dit is **drastisch gedaald**. Exact omwille van deze reden webben we voor het gebruik van een suffix array gekozen. Wanneer we het resultaat voor Swiss-Prot extrapoleren naar UniProtKB, gebruikmakende van de assumptie dat UniProtKB ongeveer 500 keer groter is, dan is de verwachting dat er ongeveer 1.2 TB RAM nodig zal zijn. Hierbij heeft de sparseness factor k geen invloed op het maximale geheugengebruik tijdens het opbouwen van de suffix array. We moeten namelijk eerst de volledige suffix array bouwen om daarna te samplen hieruit. Deze 1.2 TB blijft een grote hoeveelheid aan geheugen, maar is wel al beschikbaar op de huidige generatie aan servers. Zo bieden zowel Amazon via AWS, google via GCP en Microsoft via Azure instanties aan met enkele TB aan RAM.



(a) Tijd nodig om de indexstructuur op te bouwen.



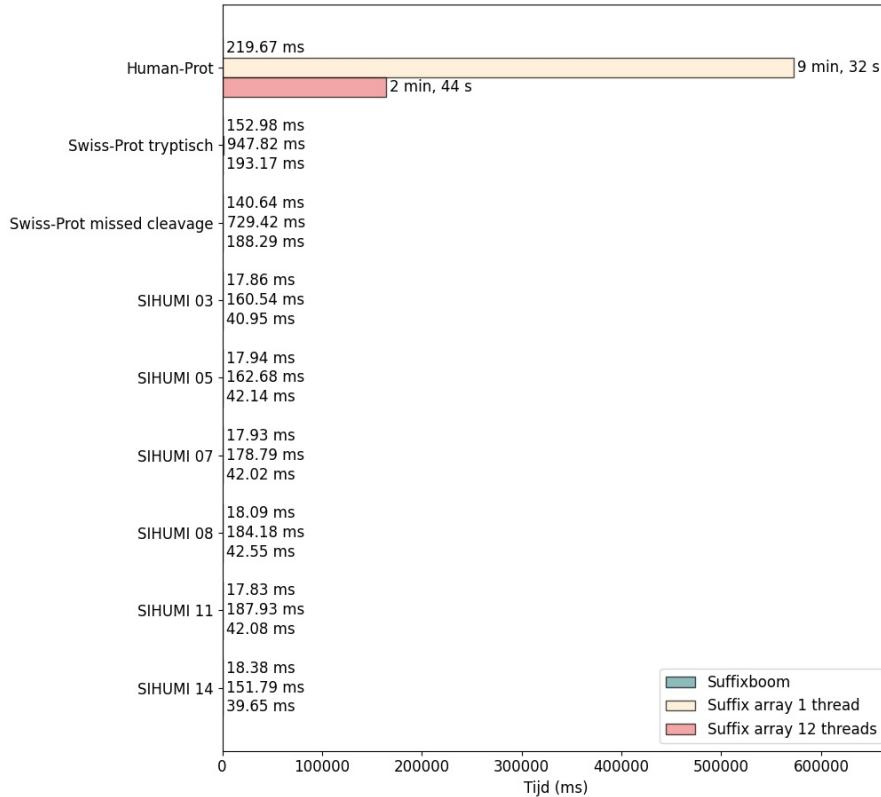
(b) Maximaal geheugengebruik om de indexstructuur op te bouwen.

Figuur 3.6: Vergelijking tussen de nodige tijd en hoeveelheid geheugen om een suffix array met libsais of een suffixboom in onze eigen Rust implementatie op te bouwen. De tijd en het geheugengebruik zijn gemeten met het Unix `time` commando. Als invoerbestand gebruiken we hier de Swiss-Prot of Human-Prot proteïnedatabank.

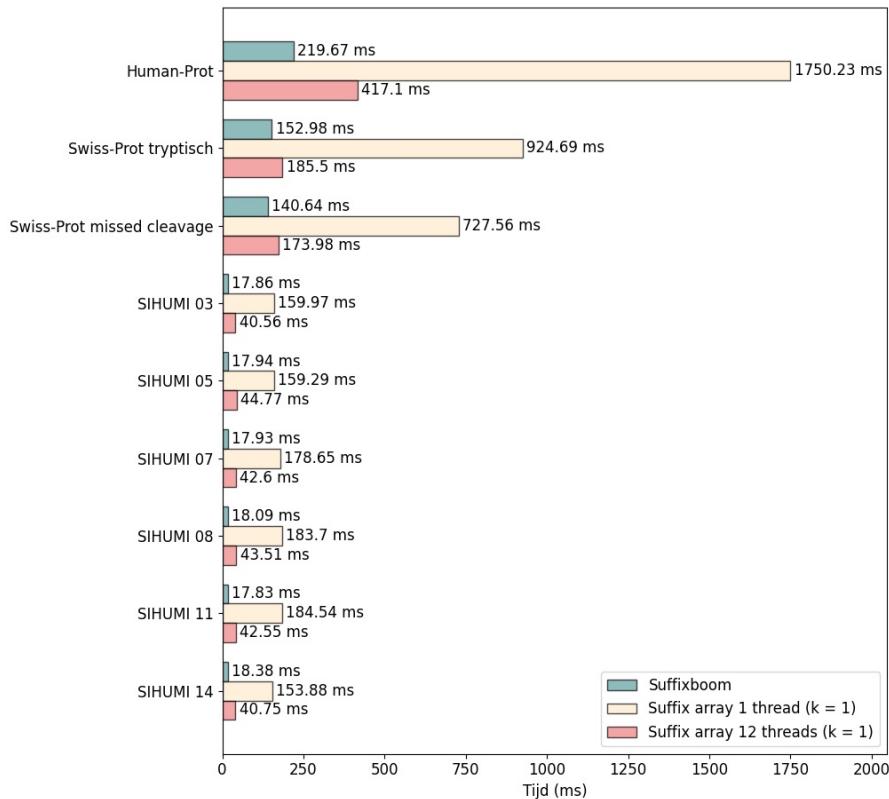
3.7.2 Zoeken

Aangezien er met de SA-indexstructuur geen voorberekening gebeurt van LCA's, is het enkel nuttig om de zoektijd inclusief het berekenen van de LCA te bekijken. De tijd tot een match levert ons in dat geval nog niet alle nodige informatie op, zoals dit wel het geval was bij de suffixboom.

Figuur 3.7 (a) toont de zoekperformantie met suffix arrays. Hier is onmiddellijk zichtbaar de performantie voor het Human-Prot zoekbestand significant slechter is. Na wat onderzoek bleek dat het berekenen van de LCA* erg traag werd indien er een groot aantal matches was. Daarom is er besloten om een **drempelwaarde** (laten we deze B , van bovengrens, noemen) in te stellen. Indien een peptide meer dan B matches heeft, wordt verondersteld dat de root van de NCBI taxonomy de laagste gemeenschappelijke voorouder is van alle matches. Uit eerder onderzoek uitgevoerd door het Unipept-team [63] op de Unipept index die gebouwd was op basis van UniProt 2023_03 bleek dat dit in de praktijk in de overgrote meerderheid van de gevallen ook effectief het geval is. De resultaten hiervan kunnen teruggevonden worden in de tabellen in appendix B. Zo matchten slechts $\pm 13\,000$ van de 1.3 miljard tryptische peptiden met meer dan 10 000 proteïnen. Hierbij was voor 95% van de peptiden de LCA gelijk aan de wortel. Slechts voor 200 peptiden was er een resultaat op soortniveau.



(a) Bereken de LCA* voor alle matches. Voor de suffix arrays is gebruik gemaakt van sparseness factor $k = 1$.



(b) Bereken de LCA* enkel als er minder dan 10 000 matches zijn.

Figuur 3.7: Berekenen van de LCA* (inclusief zoeken) voor alle peptiden (a) zonder en (b) met het gebruik van de drempelwaarde ($B=10\,000$) op het Human-Prot en Swiss-Prot proteïnenbestand. De suffix array maakt gebruik van sparseness factor $k = 1$.

In Figuur 3.7 is duidelijk te zien dat de **uitvoeringstijd drastisch daalt** wanneer de drempelwaarde op 10 000 matches geplaatst wordt. Als we dit specifiek bekijken voor de Human-Prot peptidebestanden en proteïnedatabank is de uitvoeringstijd maar liefst 300 keer kleiner. Bovendien is ook hier het **informatieverlies minimaal**. Van de 250 000 peptiden zijn er 12 peptiden die een ander resultaat krijgen. Deze 12 peptiden zijn echter slechts twee unieke peptiden (die gewoon meerdere keren voorkomen in het peptidebestand). Dit zijn de peptiden EKP en SKE. Indien we de LCA* effectief berekenen is het resultaat in beide gevallen 9606, terwijl we gebruikmakende van een drempelwaarde de root (1) terug geven.

Nu we het zoeken in de suffix array afgetopt hebben aan de hand van een drempelwaarde, is het interessant om te kijken hoe de zoekperformantie zich verhoudt ten opzichte van het zoeken in een suffixboom. In de praktijk is het zoeken in een suffix array 5 tot 10 maal trager voor onze testbestanden. Een deel van deze extra zoektijd kan opgevangen worden door gebruik te maken van parallel zoeken. Indien we hier gebruik van maken, duurt het zoeken van onze testbestanden op de Human-Prot en Swiss-Prot databank opnieuw enkele tientallen tot honderden milliseconden. Natuurlijk zouden we het zoeken in een suffixboom ook kunnen paralleliseren, maar dit hebben we niet geprobeerd omwille van de eerder vermelde geheugenproblemen bij suffixbomen.

4 ANDERE INDEXSTRUCTUREN

Naast suffixbomen en suffix arrays bestaan er ook nog enkele andere interessante indexstructuren. We behandelen in dit hoofdstuk de FM-index en de R-index.

4.1 FM-index

De FM-index [14] werd als eerste beschreven door Ferragina en Manzini. Officieel staat de naam voor *Full-text index in Minute space*. Een FM-index bestaat uit **3 essentiële componenten**. De **Borrows-wheeler transformatie** [6], een (**sparse/compressed**) **suffix array (SA)** en **bitvectoren** die de rank-operatie ondersteunen. Het opbouwen van de index voor tekst T kan gebeuren in $O(n)$ tijd, met n de lengte van T . Het zoeken of er een match is, kan in $O(m)$ tijd, met m de lengte van patroon P . Het zoeken inclusief het vinden van waar alle matches in de originele string zitten, kan in $O(m + |\text{occ}(P, T)| \log n)$. Hierbij is $\text{occ}(P, T)$ de set van alle indices waar patroon P matcht in de tekst T . In plaats van de originele tekst volledig bij te houden, wordt de BWT van de tekst bijgehouden. Door de extra informatie die in de BWT verwerkt zit (ten opzichte van de tekst zelf), is het mogelijk om een **grotere sparseness factor** k toe te passen op de SA, zonder dat dit impact heeft op de minimale lengte van zoekbare peptiden. Zo is het mogelijk om sparseness factor $k = 32$ te gebruiken, waarbij het nog steeds mogelijk is om peptiden die bestaan uit één aminozuur te zoeken, maar dit heeft natuurlijk een negatieve impact op de performantie. Als laatste onderdeel wordt er per karakter uit het alfabet één bitvector met ondersteuning voor de rank-operatie bijgehouden. Hierbij is deze bitvector even lang als de tekst T . De rank-operatie kan efficiënt ondersteund worden aan de hand van het rank9 algoritme [70], wat ongeveer 25% extra geheugen vraagt ten opzichte van een normale bitvector [9]. Op deze manier laat de FM-index zonder verlies van functionaliteit toe om een **kleinere index** te bekomen door een hogere sparseness factor k op de SA toe te passen. Het is echter belangrijk ook de overhead van de bitvector per karakter uit het alfabet mee te rekenen. Voor een groot alfabet zullen deze bitvectoren een significante bijdrage leveren aan de totale indexgrootte. Meer precies: voor een tekst T van lengte n over een alfabet Σ , zullen deze bitvectoren samen ongeveer $\frac{(\Sigma|-1)-n}{8} \cdot 1.25$ bytes groot zijn. Merk op dat we gebruik maken van $|\Sigma| - 1$ omdat we het unieke eindteken niet mee rekenen, aangezien deze slechts één maal voorkomt in de tekst. Omwille hiervan is het efficiënter de locatie van dit eindteken op te slaan in één integer. Om aan te tonen dat deze bitvectoren niet verwaarloosbaar zijn, werken we dit uit voor UniProtKB 2023_04. De totale tekst, na het concateneren van alle proteïnen met het scheidingsteken, voor deze databank is ongeveer $88 \cdot 10^9$ tekens lang. Dit vertaalt zich naar een totale grootte van de bitvectoren van $\frac{27 \cdot 88 \cdot 10^9}{8} \cdot 1.25 \approx 375$ GB. Tot slot kan men via een variant van de FM-index, de bidirectionele FM-index [34], en door gebruik te maken van zogenaamde **zoekschema's**, ook **inexacte matching** ondersteunen waarbij maximaal x mismatches toegelaten zijn.

4.1.1 Verschillende implementaties

Om een goed beeld te krijgen over het geheugengebruik tijdens het opbouwen, hebben we verschillende FM-index implementaties getest. Opnieuw focussen we ons vooral op het geheugengebruik aangezien dit de primaire restrictie is tijdens het opbouwen. Tabel 4.1 geef een overzicht van de geteste implementaties. Omdat ons einddoel bestaat uit het indexeren van UniProtKB, kunnen we ons opnieuw enkel focussen op de 64-bit implementaties. UniProtKB bevat namelijk meer tekens dan door de maximale 32-bit integer voorgesteld kan worden. Als referentie maken we gebruik van de resultaten uit Tabel 3.1. Hieruit konden we concluderen dat er 1.86 GB RAM nodig is om een suffix array op te bouwen (met 64-bit integers). Wanneer we dit vergelijken met de resultaten voor

de geteste FM-index implementaties, zien we dat deze **bijna twee maal zo veel geheugen nodig hebben**. Zelfs wanneer een sparseness factor geïntroduceerd wordt, zal dit steeds in een hoger geheugengebruik resulteren tijdens het opbouwen dan bij een suffix array. De volledige suffix array is namelijk nodig om efficiënt de BWT af te leiden. Om die reden hebben we in het kader van deze masterproef deze optie niet verder verkend.

Implementatie	Programmeertaal	Tijd (in s)		Geheugen (in GB)	
		32-bit	64-bit	32-bit	64-bit
https://crates.io/crates/fm-index	Rust	-	33.92	-	3.22
https://github.com/simongog/sdsl-lite	C++	-	27.74	-	3.70
https://github.com/ocfnash/FM-Index	Cython met C++ bindings	57.94	-	1.96	-

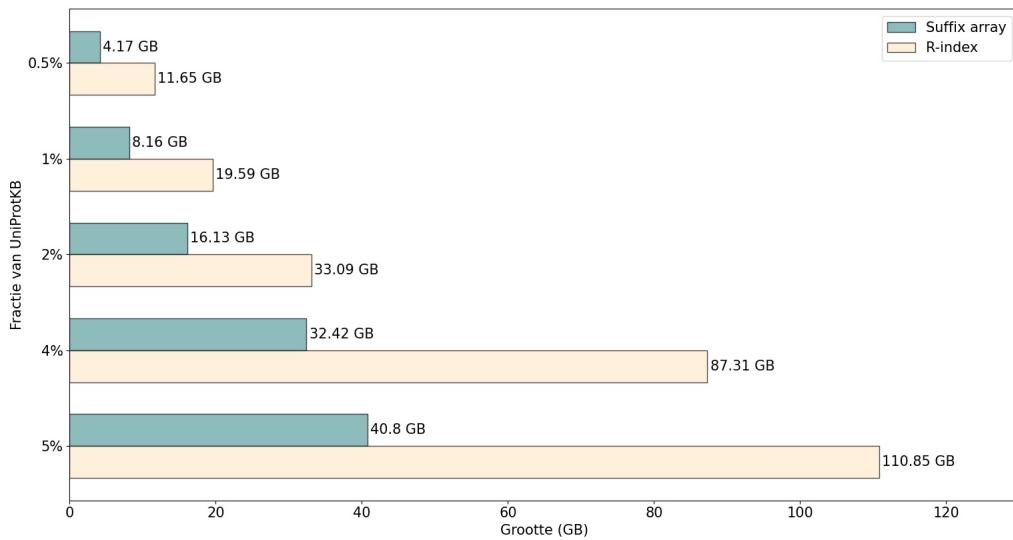
Tabel 4.1: Uitvoeringstijd en maximaal geheugengebruik voor het opbouwen van een FM-index voor de Swiss-Prot proteïnedatabank aan de hand van verschillende implementaties. Afhankelijk van de gebruikte implementatie (32- of 64-bit) is een andere kolom ingevuld. Een - staat voor niet getest. Deze testen werden lokaal uitgevoerd op een M1 Pro MacBook Pro. De specificaties hiervan zijn terug te vinden in tabel 1.7.

4.2 R-index

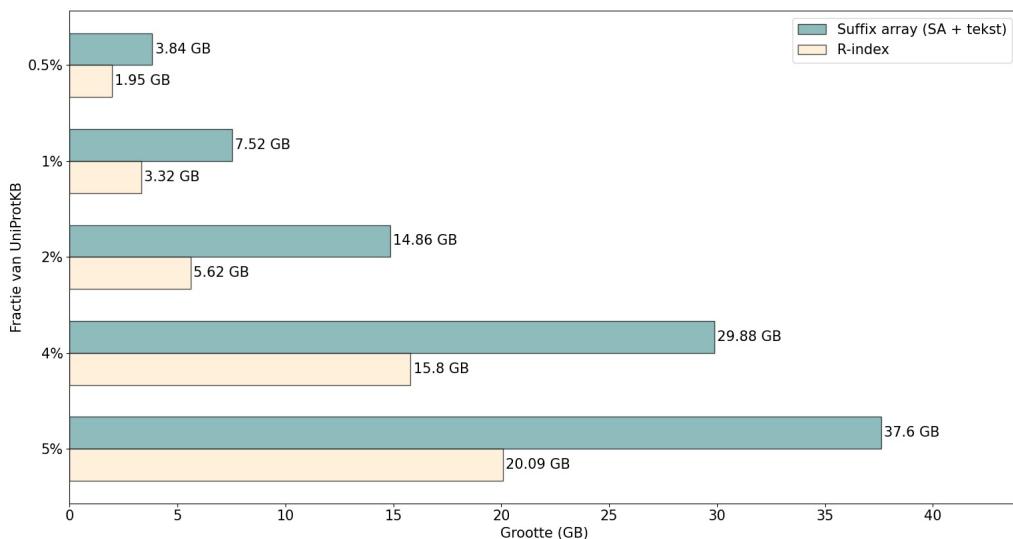
R-indices [16, 17] zijn een verdere evolutie van de FM-index waarbij **run-length encoding (RLE)** toegepast wordt op de **BWT van de tekst**. De index heeft grootte $O(r)$ met r het aantal BWT runs van de tekst van lengte n . Vanwege het gebruik van RLE op de BWT, zal de **resulterende index kleiner** zijn naarmate er **meer herhaling** voorkomt in de geïndexeerde tekst. Om na te gaan wat het effect hiervan is op onze proteïnedatabanken, hebben we de R-index getest op de eerste 0.5%, 1%,... van de volledige UniProtKB databank. Figuur 4.1 visualiseert het geheugengebruik en de resulterende indexgrootte in vergelijking met een suffix array. Merk op dat het geheugengebruik tijdens het opbouwen en de grootte van de resulterende R-index meer dan verdubbelt bij het overgaan van 2% (1 651 521 046 tekens) naar 4% (3 319 904 170 tekens) van UniProtKB. De oorzaak hiervan is het overschakelen van 32-bit naar 64-bit integers binnen de R-index. Bij de indices tot en met 2% van UniProtKB maakt de R-index implementatie gebruik van 32-bit integers omdat er minder tekens in de geïndexeerde tekst staan dan de maximale 32-bit integer waarde. Bij grotere databanken wordt deze 32-bit integer limiet overschreden, waardoor de R-index implementatie moet overschakelen naar het gebruik van 64-bit integers.

Enerzijds valt duidelijk te zien dat het opbouwen van een **suffix array minder geheugen vraagt**. Anderzijds is de **resulterende R-index kleiner**, en wordt deze bovendien procentueel kleiner voor grotere databanken. Dit is de winst die verkregen wordt door het gebruik van de run-length encoding op de BWT van de tekst. Natuurlijk zouden we bij de suffix array ook vrij simpel de resulterende index kunnen verkleinen door de SA sparse te maken. Hierbij zal op een gegeven punt de winst in indexgrootte erg klein zijn omdat de volledige tekst altijd opgeslagen moet worden. Bij het gebruik van sparseness factor $k = 3$ voor de grootste geteste index zou de totale index voor de suffix array teruggedrongen worden tot 15.31 GB. Van deze 15.31 GB is 4.17 GB de tekst zelf. Merk op dat de grootte van de resulterende index voor suffix arrays dan nog steeds lineair zal stijgen als de databank groeit, maar met een kleinere constante. Aangezien de **resulterende indexgrootte voor R-indices sublineair stijgt**, zullen we de sparseness factor k van suffix arrays groter en groter moeten maken om een even kleine resulterende index te hebben. Dit maakt een R-index op dit vlak een interessante indexstructuur voor grote proteïnedatabanken.

Opnieuw is het geheugengebruik tijdens het opbouwen een sterk beperkende factor. Dit geheugengebruik is meer dan twee keer zo hoog als bij het bouwen van een suffix array, waardoor we deze optie niet verder verkennen. Bovendien kunnen we door het gebruik van een hogere sparseness factor de resulterende suffix array makkelijk drie maal kleiner maken, zonder al te veel performantieverlies.



(a) Maximaal geheugengebruik tijdens het opbouwen van een suffix array en R-index.



(b) Resulterende indexgrootte voor een suffix array (sparseness factor $k = 1$) en R-index.

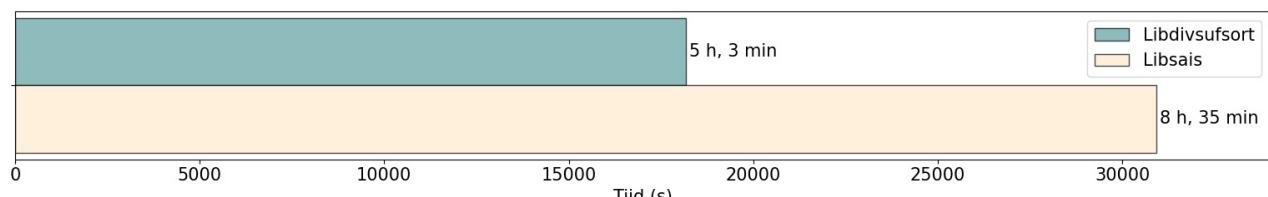
Figuur 4.1: Vergelijking van de suffix array en R-index op vlak van geheugengebruik en de resulterende indexgrootte voor verschillende deelverzamelingen van UniProtKB. Bij elke test gaat het om de eerst x procent van de databank. Bij de suffix array wordt sparseness factor $k = 1$ gebruikt. Het gebruik van een andere sparseness factor heeft enkel invloed op de grootte van de resulterende index.

5 EEN NIEUWE UNIPROTKB-INDEX VOOR UNIPEPT

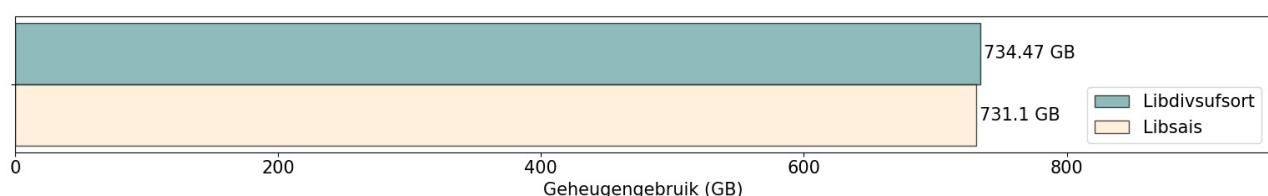
In de vorige hoofdstukken hebben we suffixbomen, suffix arrays, FM-indices en R-indices verkend als indexstructuren die toegepast kunnen worden om een proteïnedatabank te indexeren. Hieruit bleek dat een suffix array de laagste geheugenvereisten heeft tijdens het opbouwen. In dit hoofdstuk gaan we dieper in op het opbouwen van een index voor UniProtKB gebruikmakende van een suffix array. Aangezien we de huidige UniPept index willen vervangen door deze nieuwe indexstructuur, behandelen we bovendien ook nog enkele extra gewenste features. Het resultaat kan in onze GitHub repository¹ teruggevonden worden.

5.1 Opbouwen van de SA

Zoals vermeld in sectie 3.7.1 levert een ruwe extrapolatie op dat we 1.2 TB RAM nodig hebben om een suffix array op te bouwen voor UniProtKB. Hierbij wordt er echter van uitgegaan dat UniProtKB 500 keer meer proteïnen dan Swiss-Prot bevat. Dit is een overschatting van de realiteit waar UniProtKB op dit moment *slechts* ± 440 keer groter is. Dit zorgt ervoor dat het opbouwen mogelijk al **haalbaar is op de HPC van UGent** waar nodes beschikbaar zijn tot 940 GiB RAM. Na dit te proberen bleek *slechts* 740 GB RAM nodig. Dit is minder dan verwacht omdat onder andere de NCBI taxonomy in het geheugen gehouden wordt. Deze blijft even groot, onafhankelijk van de grootte van de proteïnedatabank. Figuur 5.1 toont de nodige tijd en hoeveelheid geheugen om dit te realiseren. **Opvallend hierbij is dat de libsaas implementatie hier trager is dan libdivsufsort**, terwijl dit voor alle kleinere datasets net omgekeerd was. Afhankelijk van de dataset is het ene algoritme dus sneller dan het andere. Het geheugengebruik van beide algoritmen blijft echter erg gelijkaardig, wat voor ons het belangrijkste is. Voor het opbouwen van de UniPept index voor UniProtKB zullen we dus gebruikmaken van libdivsufsort.



(a) Tijd nodig om een SA-index voor UniProtKB op te bouwen.



(b) Maximale hoeveelheid geheugen gebruikt tijdens het opbouwen van een SA-index voor UniProtKB.

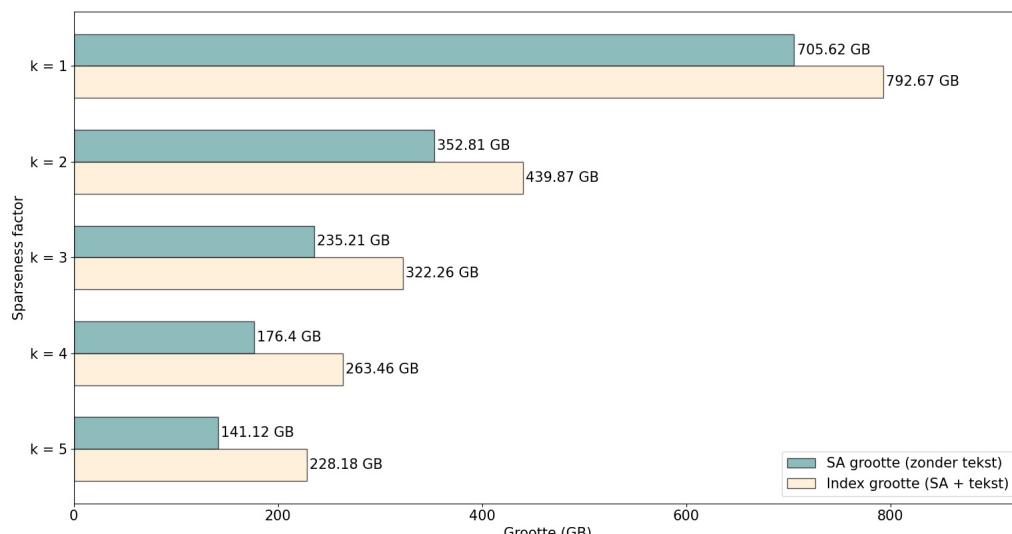
Figuur 5.1: Statistieken voor het opbouwen van een SA voor UniProtKB.

¹<https://github.com/BramDevlaeminck/FastPeptideMatching>

5.2 Een sparseness factor kiezen

Een volgende stap na het opbouwen van de volledige SA bestaat uit het kiezen van de optimale sparseness factor. Zoals eerder aangegeven in de conclusie van sectie 3.5.1 willen we deze sparseness factor zo laag mogelijk houden, met als restrictie dat de sparse suffix array (SSA) nog steeds in het werkgeheugen moet passen. Uit Figuur 3.4 (a) bleek namelijk dat het kiezen van een hogere sparseness factor een dramatische impact op de zoektijd van korte peptiden heeft. Dit effect zal nog versterkt worden bij het gebruik van een grotere databank, omdat er dan meer matches gevonden worden.

De servers die UniPept hosten hebben elk \pm 0.5 TB RAM ter beschikking. Dit wil zeggen dat we de resulterende index hierin moeten krijgen, en ook nog genoeg ruimte moeten overlaten zodat de server zeker niet crasht tijdens het hosten van de index en het verwerken van meerdere requests tegelijk. Figuur 5.2 toont een overzicht van de indexgrootte voor verschillende sparseness factors. Aangezien we ongeveer 0.5 TB RAM ter beschikking hebben, zal **sparseness factor $k = 3$** de kleinste factor zijn die comfortabel in het geheugen past. We hebben hiervoor namelijk al 322 GB RAM nodig. Hierbij moet natuurlijk nog wat overhead gerekend worden voor de mapping van suffix naar proteïne, en alle annotaties die bij een proteïne horen bij te houden. Bovendien bevatten deze servers ook nog andere databanken om verdere aggregaties te berekenen. Indien we sparseness factor 2 zouden gebruiken, komt de totale indexgrootte uit op \pm 440 GB. Dit op zich past nog in het geheugen, maar laat niet genoeg ruimte voor de andere processen.



Figuur 5.2: Grootte van de SA en totale index opgebouwd op UniProtKB voor sparseness factor $k = 1, 2, 3, 4, 5$.

5.3 Isoleucine en leucine gelijkstellen

Naast het vinden van exacte matches is het vinden van inexakte matches ook interessant. Vooral het vinden van matches waarbij we de aminozuren I (isoleucine) en L (leucine) aan elkaar gelijkstellen is een belangrijke optie in de huidige UniPept index. **Deze twee aminozuren kunnen niet gedifferentieerd worden door een massaspectrometer omdat ze een identieke massa hebben.** Door deze restrictie is het voor onderzoekers erg nuttig om alle matches te vinden waar een I ook een L kan zijn, of omgekeerd. Om deze vorm van inexakte matching toe te voegen aan de index gebruikmakende van een suffix array zijn er meerdere opties:

1. **Twee indices:** Bouw een extra index waarbij in de tekst elke L door een I vervangen wordt, of vice versa. Bij het verwerken van een zoekopdracht wordt deze afgehandeld door de juiste index

afhankelijk van de gekozen configuratie. Indien I en L gelijkgesteld worden, hoeft er slechts één bijkomende operatie uitgevoerd te worden. Dit is dezelfde vervangoperatie die uitgevoerd is op de originele tekst bij het opstellen van de index. Deze optie wordt niet verder uitgewerkt omdat we willen vermijden dat we twee volledig aparte indices moeten hosten, waardoor het geheugengebruik zou verdubbelen.

2. **Index waarbij $I \neq L$:** Genereer de varianten van de gezochte peptide *on the fly* tijdens het zoekproces. Hierbij kan gebruikgemaakt worden van het feit dat twee peptiden die identiek zijn, behalve dat elke locatie waar een I een L kan staan (of vice versa), een gelijkaardig zoekpad afleggen in de SA. Deze zoekpaden splitsen enkel wanneer het teken dat verwerkt wordt (uit de proteïnedatabank) tijdens de binary search een I, J, K of L is. Dit zorgt ervoor dat voor een willekeurig patroon een groot stuk van de zoekboom gemeenschappelijk zal zijn. Door dit gemeenschappelijke stuk kan veel dubbel werk vermeden worden, we zullen echter steeds 2^q opties moeten verkennen. Hierbij is q de som van het aantal I's en L'en in de peptide.
3. **Index waarbij $I = L$:** Bouw de index voor een variant van de UniProtKB databank. Hierbij vervangen we elke I door een L, of vice versa, en bouwen we dus een index op waarbij I gelijkgesteld is aan L. Tijdens het zoeken van een peptide doen we dezelfde vervanging, waarna we alle matches hebben waarbij I gelijkgesteld is aan L. Indien we I niet gelijk willen stellen aan L, kunnen we achteraf de foute matches wegfilteren aan de hand van de originele proteïnedatabank.

In de volgende secties worden de tweede en derde aanpak verder uitgewerkt.

5.3.1 Index waarbij $I \neq L$

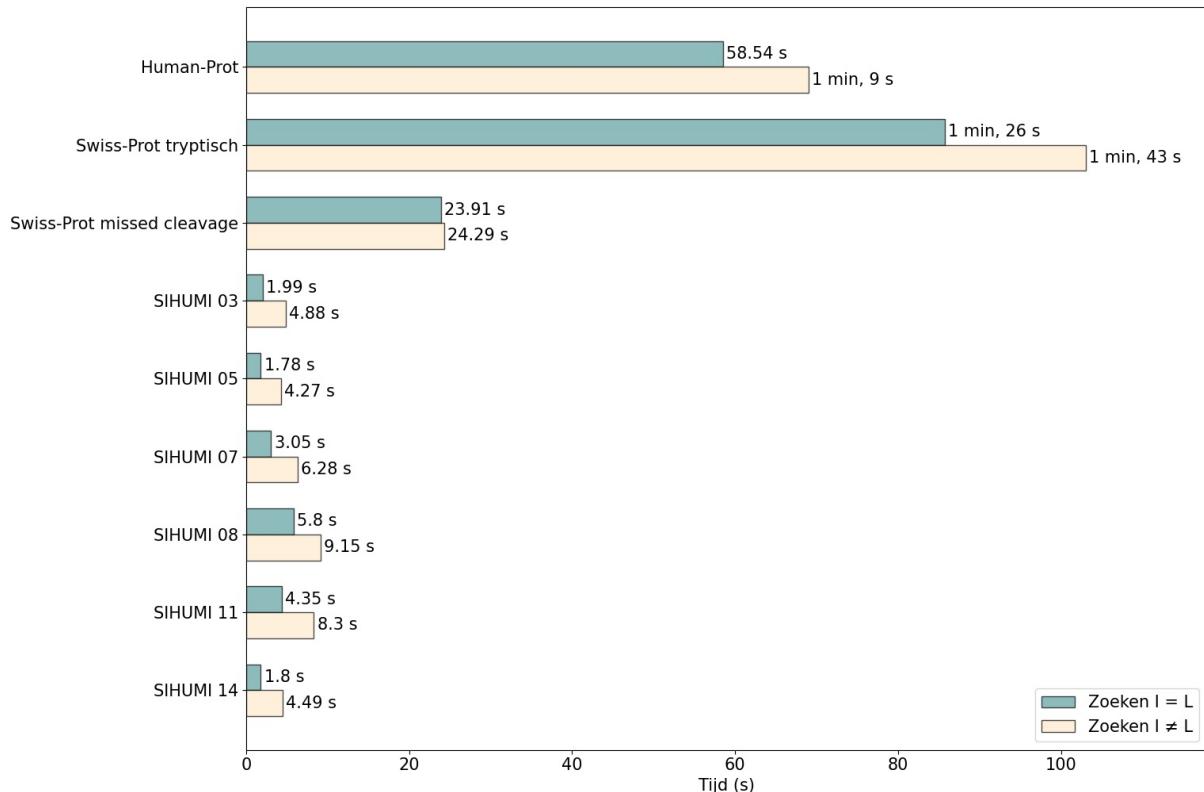
Wanneer de indexstructuur zelf gebouwd is met I niet gelijkgesteld aan L, moeten we alle verschillende IL-combinaties tijdens het zoekproces verkennen. Hierbij is het echter belangrijk om te weten dat er ook bepaalde extreem slechte gevallen bestaan waarbij de zoekruimte erg groot wordt. De sequenties waarop we zoeken zijn namelijk niet random verdeeld over alle karakters van het alfabet. Biologisch gezien komen bepaalde aminozuursequenties veel vaker voor. Eén van deze patronen zijn de zogenaamde *leucine rich repeats* [37]. Dit zijn sequenties waarin een reeks L'en na elkaar voorkomt. In UniProtKB (2024_01) bestaat er een **sequentie waar maar liefst 2397 L'en na elkaar voorkomen**. Dit is de proteïne met *accession number A0A1Q9EZQ0*. Wanneer we nu ook weten dat in UniProtKB ook reeksen aan I's voorkomen, dan zorgt dit voor bepaalde extreem slechte gevallen. Zo is er **ook een proteïne met 641 opeenvolgende I's** (*accession number: A0A5J4P3H7*). In het slechtste geval zou een gebruiker dus een sequentie van 641 I's of L'en kunnen proberen te matchen. Dit zorgt ervoor dat we in de zoekboom $2^{641} \approx 9.12 \cdot 10^{192}$ opties moeten proberen. Het grootste deel van deze opties zal niet voorkomen, waardoor de takken van deze boom allemaal extreem kort zullen zijn. Dit zal echter verwarring kunnen veroorzaken ten opzichte van het gigantisch aantal opties. Om dit in perspectief te plaatsen: Men schat dat er in het totaal 10^{79} atomen in het universum zijn [48] en dat het universum ongeveer $4.36^{20} \approx 6.16 \cdot 10^{12}$ milliseconden oud is [27]. Zelfs als het controleren van één optie minder dan een milliseconde duurt, dan zou dit dus nog onmogelijk zijn. **Om de zoektijd en het geheugengebruik te beperken, hebben we ervoor gekozen om twee vormen van restricties op te leggen wanneer I en L gelijkgesteld worden.**

1. Laat maximaal 5 seconden aan zoektijd per peptide toe.
2. Laat per peptide in totaal maximaal 34 I's en L'en toe.

Deze twee restricties samen moeten de servers deels helpen beschermen tegen Denial of Service attacks. In de praktijk wordt normaal de tijdslimiet van 5 seconden eerst bereikt. Dit vertaalt zich naar een sequentie van ongeveer 25 I's of L'en. Elke keer we één extra teken zouden willen toelaten, verdubbelt de zoektijd. Zo duurt het al één minuut om een sequentie met 30 opeenvolgende I's of L'en te zoeken. Wanneer we echter nog veel meer I's of L'en toelaten, stuiten we vrij snel op een geheugenlimiet. Om te voorkomen dat het programma meer geheugen probeert te vragen dan dat

de server heeft, waarna het crasht, hebben we beslist maximaal 34 I's of L'en toe te laten. In het slechtste geval gebruikt één enkele thread hierbij iets meer dan 2 GB RAM. Aangezien het zoeken multithreaded is, moeten we dus zeker 20 GB aan vrij geheugen voorzien wanneer de index ingeladen is. **Wanneer we deze limieten testen door alle testpeptidebestanden te zoeken in de volledige UniProtKB databank worden deze limieten geen enkele keer bereikt.**

In de praktijk vertaalt het gelijkstellen van I en L op deze manier zich naar een **vrij kleine overhead**. Deze beperkte overhead valt te zien in Figuur 5.3. Deze figuur toont ook onmiddellijk de zoekperformantie waarbij $I \neq L$ op UniProtKB. Zoals verwacht op basis van de resultaten voor de kleinere Swiss-Prot en Human-Prot proteïnedatabanken, is dit zoeken inderdaad extreem snel.



Figuur 5.3: Zoektijd in UniProtKB (Swiss-Prot + TrEMBL) met en zonder het gelijkstellen van I en L met een SA met sparseness factor $k = 3$. De index zelf is opgebouwd op basis van een tekst waar $I \neq L$.

5.3.2 Index waarbij $I = L$

Een andere optie om zoeken waarbij I en L gelijkgesteld zijn mogelijk te maken, is door de index specifiek voor dit geval op te bouwen. Hierbij wordt in de originele tekst elke L vervangen door een I, of vice versa, om op deze aangepaste tekst een index te bouwen. Wanneer we in de gezochte peptiden hetzelfde doen, vinden we alle matches, waarbij een I en L gelijkgesteld zijn. Dit is duidelijk erg simpel, en bovendien efficiënt.

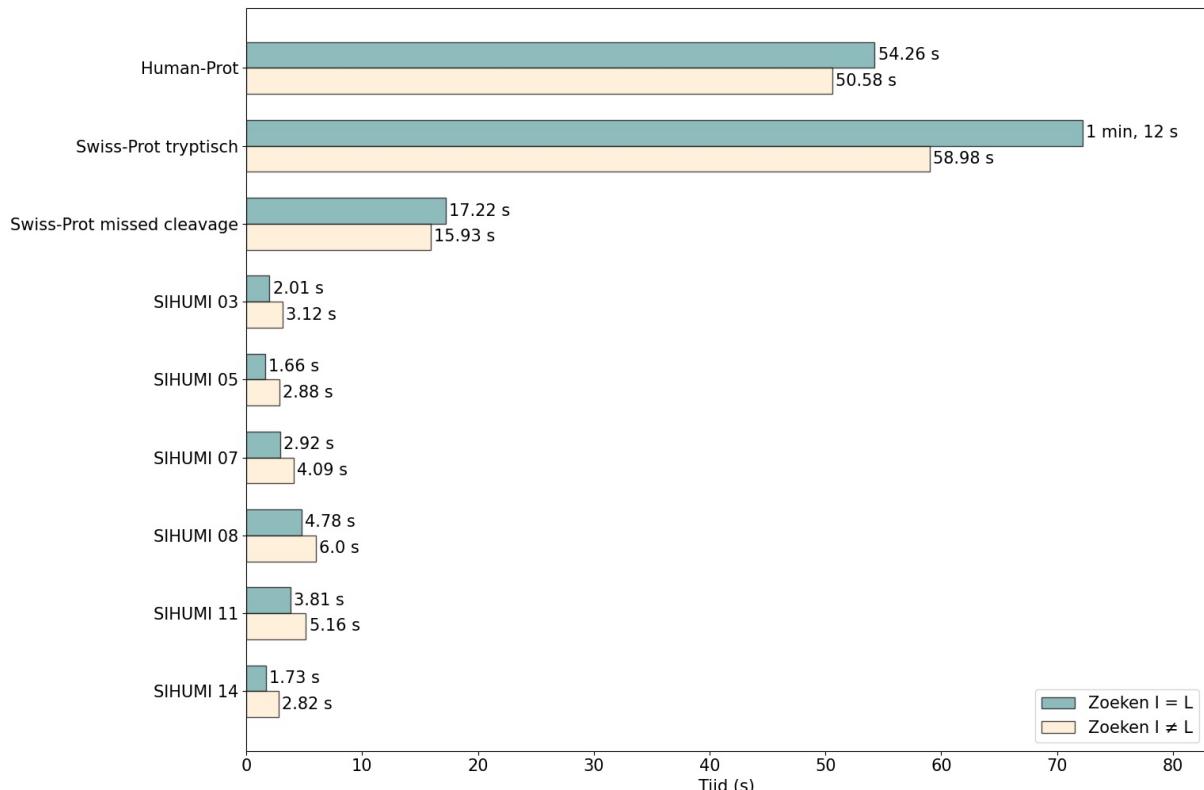
Door het zoekproces licht aan te passen, kunnen we bovendien nog steeds de matches vinden waarbij $I \neq L$. Dit kan via volgend stappenplan:

- 1. Zoek alle matches** in de SA, waarbij $I = L$. Hiervoor moeten we in elke peptide dezelfde vervangoperatie uitvoeren als tijdens het bouwen van de suffix array.
- 2. Filter de ongeldige matches weg.** Dit kunnen we doen door voor alle matches te controleren op elke positie waar een I of L is, als daar hetzelfde teken staat als in de tekst. Wanneer in de

peptide een L staat, en de tekst een I (of omgekeerd), wil dit zeggen dat deze match ongeldig is, want ze is enkel gevonden vanwege het gelijkstellen van I aan L tijdens het zoeken. Merk op dat we hiervoor de originele tekst nodig hebben, en niet de aangepaste tekst die gebruikt is bij het bouwen van de suffix array. Gelukkig moeten we niet allebei deze teksten in het geheugen houden. Via enkele kleine aanpassingen tijdens het zoeken kan de originele tekst perfect gebruikt worden om te zoeken in de suffix array van de gemodificeerde tekst.

Dit stappenplan is niet alleen **simpel**, maar het bevat ook **geen extreem slechte gevallen** zoals bij de index waarbij we $I \neq L$ stellen, en dan alle opties proberen verkennen. Bovendien is ook de impact van de filterstap beperkt, aangezien enkel locaties waar een I of L staat in de peptide gecontroleerd moeten worden. Andere tekens moeten we nooit controleren, aangezien het zoeken in de SA ons al garandeert dat deze overeenkomen. Er is dus geen nood aan een combinatie van systemen die de server moeten beschermen tegen de slechtste gevallen, zoals bij sectie 5.3.1.

Naast deze voordelen blijkt ook dat deze manier van zoeken in het algemeen iets sneller is. Dit valt te zien wanneer we Figuur 5.4 vergelijken met Figuur 5.3. Een laatste voordeel is dat het gelijkstellen van I en L de standaard optie is in de huidige UniPept gebruikersinterface. Dit is net ook wat hier het minste werk vereist tijdens het zoeken. Vanwege al deze voordelen kiezen we ervoor om van deze zoekstrategie gebruik te maken.



Figuur 5.4: Zoektijd in UniProtKB (Swiss-Prot + TrEMBL) met en zonder het gelijkstellen van I en L met een SA met sparseness factor $k = 3$. De index zelf is opgebouwd op basis van een tekst waar $I = L$.

Een merkwaardige vaststelling is dat voor onze kleinere SIHUMI testbestanden het zoeken met $I \neq L$ sneller is, ondanks dat zoeken met $I \neq L$ net meer werk vraagt. Dit komt omdat er meer resultaten zijn zonder de filterstap. Deze extra resultaten moeten allemaal meegenomen worden in het berekenen van de LCA*, maar ook tijdens het serialiseren van de resultaten naar een JSON-bestand. Bovendien zorgt een grotere hoeveelheid data ook voor een tragere doorstroming van de index naar de UniPept API. Bij de grotere testbestanden wordt er in verhouding meer tijd aan het zoeken gespendeerd, waardoor daar de verwachte tijdwinst wel zichtbaar is.

5.4 Taxonomische analyse

Door het gebruik van suffix arrays moet de taxonomische analyse volledig uitgevoerd worden tijdens het zoekproces zelf. Dit heeft zowel voor- als nadelen.

Het grootste nadeel is de **extra overhead per zoekopdracht**. In de oude index was het enkel nodig alle matches te zoeken, waarna de voorberekende LCA onmiddellijk opgehaald kon worden. Nu moeten we deze LCA ook nog berekenen. Indien we dit zouden willen oplossen, is er nood aan een manier om de boomstructuur op een geheugenefficiënte manier te reconstrueren. Hiervoor zijn de extra tabellen van Enhanced Suffix Arrays nodig.

Anderzijds zijn er ook enkele voordelen verbonden met het berekenen tijdens het zoekproces zelf. Zo hebben we **minder geheugen** nodig, wat erg belangrijk is aangezien het geheugengebruik al erg hoog is. Bovendien laat dit ook toe dat we vrij kunnen kiezen welke aggregatiemethode we gebruiken. De oude Unipept index maakte gebruik van een (minder interessante) variant van **LCA***. Dit werd gedaan om het voorberekenen efficiënter te maken. Bij suffixbomen hebben we zelf gebruik moeten maken van de standaard LCA methode om dit op te lossen. Bij suffix arrays kunnen we echter vrij gebruik maken van LCA*, en blijft de performantie bovendien erg goed. Een laatste belangrijk voordeel is dat er **geen extra werk nodig is om de taxon IDs van alle matches op te halen**. In de huidige Unipept index is hier een significante hoeveelheid extra werk voor nodig, wat resulteert in een bottleneck bij de Peptonizer2000 tool [25].

5.5 Functionele analyse

Een laatste feature die de nieuwe index moet ondersteunen, is de functionele analyse uit Unipept. Deze analyse wordt uitgevoerd door het aantal voorkomens van de verschillende GO, EC en INTERPRO termen die horen bij de gevonden matches te tellen. Elk van deze termen is een korte string van 5 à 10 karakters lang, en kan vrij eenvoudig mee ingelezen worden en per proteïne bijgehouden worden. Al deze annotaties samen, maken het databankbestand dat ingelezen wordt om de index mee op te bouwen ongeveer 24 GB groter. Wanneer we deze informatie allemaal ingeladen hadden, bleek het programma echter ongeveer 105 GB extra RAM te gebruiken. De verklaring hiervoor is dat Rust 24 bytes overhead heeft per vector, en 24 bytes overhead per string. Om deze annotaties bij te houden, hadden we net één vector per proteïne nodig, en die vector bevatte een reeks aan korte strings (de annotaties zelf). Om dit op te lossen heeft Tibo Vande Moortele een simpele compressiebibliotheek² ontwikkeld voor deze annotaties. Hierbij worden 2 karakters uit een annotatie gecomprimeerd naar 1 byte, en kunnen bovendien enkele tekens weggelaten worden. Na het gebruik van deze bibliotheek resulteert dit in **10.5 GB extra geheugengebruik**, wat in de context van de al 320 GB grote index (wanneer de tekst ook meegerekend wordt) bijna verwaarloosbaar is. Aangezien ook deze analyse *on the fly* uitgevoerd wordt, heeft ook dit een negatieve impact op de performantie. De volgende sectie gaat dieper in op de performantie van de finale indexstructuur.

5.6 Vergelijking met andere tools

Naast Unipept zijn er nog andere tools die toelaten om UniProtKB te doorzoeken. In deze sectie vergelijken we de performantie en eigenschappen van Unipept met de **Uniprot Peptide search tool** en de **Exasy ScanProsite tool**. We zullen de versie van Unipept die de nieuwe index gebruikt Unipept 6.x noemen, aangezien de nieuwe index gebruikt zal worden vanaf Unipept 6.0, terwijl de oude versie Unipept 5.x genoemd wordt. Vanwege de beperkte performantie van een aantal van deze tools zullen we ons beperken tot het opzoeken van één willekeurige (tryptische) peptide **ISPAVLFVIVILAVLFFISGLLHLLVR**. Als referentie gebruiken we de zoektijd met Unipept 6.x waarbij, zoals eerder vermeld, sparseness factor $k = 3$ gebruikt wordt. Hierbij duurt het zoeken van de peptide

²<https://github.com/unipept/unipept-index/tree/main/fa-compression>

ongeveer **3 milliseconden** en levert deze 50 matches op. Wanneer we I en L gelijkstellen duurt dit opnieuw ongeveer 3 milliseconden, maar levert dit wel 52 matches op. Hierbij wordt natuurlijk geen extra tijd geïntroduceerd door een API call over het internet aangezien deze index (nog) niet publiek beschikbaar is.

5.6.1 UniProt peptide search tool

De UniProt peptide search tool [7, 64] is een onderdeel van de UniProt site en laat toe om alle voor-komens van een bepaalde peptide te vinden in de volledige UniProtKB databank. Als opties kunnen we kiezen om enkel in Swiss-Prot te zoeken, of in Swiss-Prot en TrEMBL. Voor beide databanken kunnen we kiezen of we I en L aan elkaar willen gelijkstellen. Deze features komen exact overeen met de eigenschappen van onze nieuwe indexstructuur, met de uitzondering dan Unipept extra analyses aanbiedt. Qua performantie is deze tool echter extreem onstabiel. Zo duurt het zoeken van **ISPAVLVFVILAVLFFISGLLHLLVR** in de volledige UniProtKB databank **enkele seconden tot zelfs 20 minuten**. Bovendien vermoeden we ook dat er een vorm van caching toegepast wordt, aangezien het zoeken voor een tweede keer zo goed als onmiddellijk het resultaat geeft. Dit maakt het benchmarken extreem moeilijk. Zoals verwacht zijn de gevonden matches identiek aan de matches die onze nieuwe indexstructuur vindt. Naast de variabele performantie, faalt deze tool ook regelmatig tijdens het zoeken en ophalen van de resultaten.

Vanwege de extreem gelijkaardige mogelijkheden van deze tool, hebben we contact opgenomen met UniProt om de vergelijking meer compleet te maken. Hun eigen index werkt op basis van Apache Lucene³, waarbij alle proteïnen opgesplitst worden in 3-grammen. Volgende sectie bevat de gestelde vragen, met de gegeven antwoorden.

How much memory is needed to build the index structure?

Answer: There is no specific requirement for the RAM. One of our users had tested that it works with 32 GB RAM.

How long does it take to build the index?

Answer: It depends on the hardware. We usually run it in a machine with 256 GB RAM to index the UniProtKB sequences plus the related information. It can take 1 to 2 days depending on how busy the machine is.

How much memory is needed to host the index?

Answer: We had been running the PeptideSearch services in a server with 256 GB RAM for many years and upgraded the RAM to 512 GB to improve the performance.

How many requests do you handle per day?

Answer: There are around 350 searches per day in April 2024.

How many peptides does the average request contain?

Answer: Around 80% of the cases search one peptide.

What percentage of requests treat isoleucine and leucine as equivalent?

Answer: Around 6% of the cases treat isoleucine=leucine.

What is the average time needed to handle 1 request? During our own testing we noticed that the performance can vary a lot from request to request. Is there an explanation for this?

Answer: If you are using the tool on our website, response times can vary due to several reasons:

1. Load of the server running PeptideSearch (currently running at PIR in Delaware, US)
2. Load/availability of the dispatching server which is located at the EBI, UK)

³<https://lucene.apache.org/>

Uit deze antwoorden kunnen we concluderen dat hun index lagere geheugenvereisten heeft. Vermoedelijk komt dit doordat Apache Lucene geen harde geheugenlimiet heeft om de index te kunnen gebruiken. Het zal echter zoveel geheugen proberen gebruiken als het toegewezen krijgt, om zo onder andere caching toe te passen. Dit verklaart waarschijnlijk ook deels de variabele performantie. Afhankelijk van eerdere zoekopdrachten zijn verschillende stukken van de index waarschijnlijk sneller doorzoekbaar.

Interessant is ook dat 80% van de zoekopdrachten uit één peptide bestaan, en dat slechts in 6% van de gevallen I en L gelijkgesteld worden. Dit is net het tegenovergestelde van de geregistreerde requests bij UniPept. Vermoedelijk komt het verschil bij het gelijkstellen van I en L doordat dit standaard aan staat bij UniPept, terwijl dit bij UniProt net standaard uit staat.

5.6.2 Expasy ScanProsite tool

De Expasy ScanProsite tool [12] laat toe om aan de hand van motieven die lijken op reguliere expressies allerlei patronen te zoeken. De beschrijving van motieven laat toe om onder andere wildcards, karakterklassen⁴, inverse klassen en herhalingen te specificeren. De flexibiliteit die hierdoor aangeboden wordt dus een sterk voordeel. Een nadeel van deze tool is dat niet de volledige UniProtKB databank doorzocht wordt. Er wordt enkel rekening gehouden met proteïnen die deel uit maken van een *reference proteome*⁵. Dit komt erop neer dat ze bij UniProt 2024_01 **slechts rekening houden met 85 152 388 van de 249 751 891 proteïnen** (34%). Wanneer we de zoekperformantie testen, duurt het zoeken van de peptide ISPAVLFVIVILAVLFFISGLLHLLVR zo'n **5.5 minuten** zonder gelijkstellen van I en L en 10 minuten met het gelijkstellen van I en L. Hierbij zijn er slechts 30 resp. 32 matches gevonden, wegens de deelverzameling van proteïnen die gebruikt wordt.

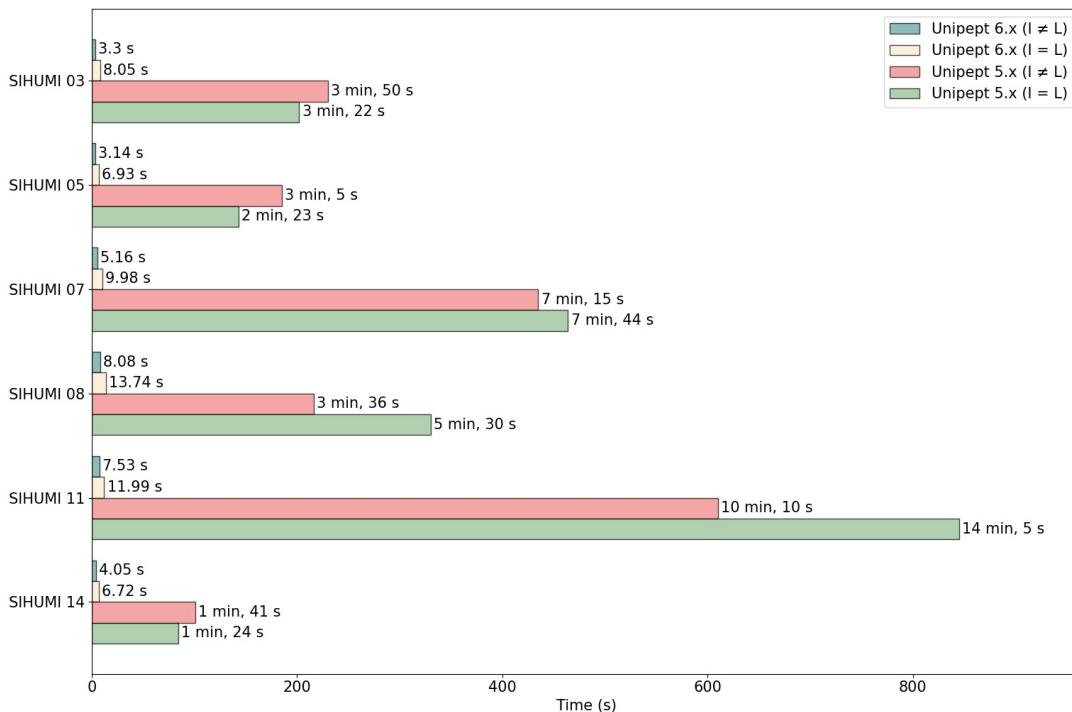
5.6.3 UniPept 5.x

UniPept 5.x kan, **op voorwaarde dat een peptide tryptisch is**, alle matches vinden in UniProtKB met de optie om I en L gelijk te stellen. Onze willekeurig gekozen testpeptide is tryptisch, dus vindt deze index zoals verwacht alle matches. Bovendien gebeurt dit gebruikmakende van een API call in **± 4 milliseconden**, wat duidelijk veel sneller is dan de andere bestaande opties. Opnieuw is de zoektijd hier al zo klein dat de invloed van het gelijkstellen van I en L niet te meten valt. Het is echter een grote beperking dat dit enkel werkt voor tryptische peptiden, of peptiden met zogenaamde *missed cleavages*. Volgend voorbeeld illustreert dit. Stel dat we nu de peptide ILAKLFIS zoeken. Dit is geen tryptische peptide en bijgevolg kunnen er geen matches gevonden worden. De nieuwe indexstructuur vindt echter 14 matches zonder I en L gelijk te stellen, en 207 matches bij het gelijkstellen van I en L.

Tot slot vergelijken we de zoektijd van UniPept 6.x en 5.x in meer detail, aangezien één peptide hier geen goed zicht op geeft. Hierbij berekenen we met beide indices ook de functionele en taxonomische analyses. Het is belangrijk dat dit meegenomen wordt tijdens het meten van de zoektijd, aangezien dit fundamentele functies van UniPept zijn. We bekijken twee scenario's om zo een goed zicht op de performantie te krijgen. In het eerste geval houden we rekening met *missed cleavages*. Dit is het meest realistische geval, aangezien stalen in de praktijk altijd *missed cleavages* bevatten. Figuur 5.5 geeft een overzicht van de zoektijd op de SIHUMI peptidebestanden. Hieruit blijkt dat UniPept 6.x 10 tot 100 keer sneller is dan UniPept 5.x.

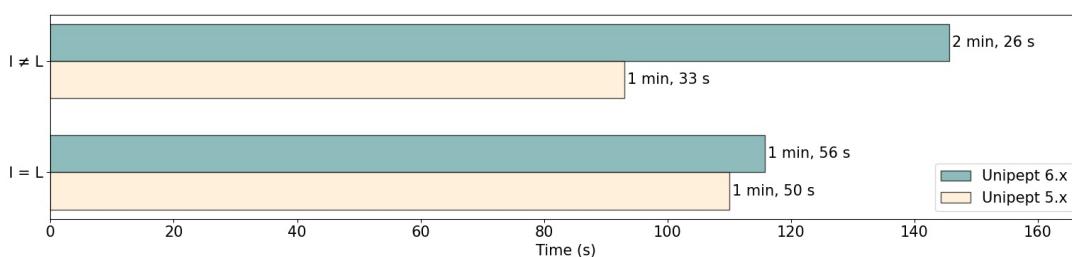
⁴Hiermee bedoelen we het toelaten van een reeks aan tekens op een bepaalde plaats. Dit is gelijkaardig aan de [ABC] syntax uit reguliere expressies, waar we een A, B of C toelaten.

⁵Deze zogenaamde *reference proteomes* zijn een collectie van proteïnen die door een bepaald organisme gemaakt kunnen worden, en die bovendien taxonomisch belangrijk bevonden worden. Deze laatste voorwaarde wil dus zeggen dat dit niet zomaar van elk organisme is, enkel van een selecte groep organismen die op basis van verschillende factoren belangrijk bevonden wordt. Een voorbeeld hiervan is de *human reference proteome*. Dit zijn alle proteïnen die door een mens aangemaakt kunnen worden.



Figuur 5.5: Taxonomische en functionele analyse met UniPept 6.x vs UniPept 5.x voor de SIHUMI peptidebestanden. UniPept 5.x wordt uitgevoerd met *filter duplicate peptides* uitgeschakeld, en *advanced missed cleavage handling* ingeschakeld. Op deze manier zullen beide versies alle peptiden zoeken, en ook *missed cleavages* afhandelen. Merk op dat UniPept 6.x altijd *missed cleavages* zoekt, en bovendien zelfs peptiden zal vinden die op een arbitraire manier gesplitst zijn.

Het andere geval is wanneer we een peptidebestand verwerken waarin enkel tryptische peptiden voorkomen. Dit is niet realistisch, maar is het beste scenario voor UniPept 5.x aangezien de *advanced missed cleavage handling* dan niet gebruikt moet worden. Merk op dat UniPept 6.x wél *missed cleavages* zal verwerken, aangezien dit geen negatieve invloed heeft bij de nieuwe indexstructuur. Figuur 5.6 toont het verschil in zoektijd tussen UniPept 6.x en 5.x voor het tryptische Swiss-Prot peptidebestand. Ter herinnering: dit bestand bestaat uit 100 000 tryptische peptiden. Hieruit blijkt dat UniPept 6.x 33% trager is dan UniPept 5.x wanneer $I \neq L$, maar dat de zoektijd erg vergelijkbaar is wanneer $I = L$. Dit verschil is acceptabel omdat we eigenlijk altijd rekening willen houden met *missed cleavages*. De enige reden dat dit amper gedaan werd in UniPept 5.x, is vanwege de grote impact op de performantie. Bovendien kan UniPept 6.x ook ingezet worden voor niet-triptische peptiden, wat niet mogelijk was voor UniPept 5.x en eerder.

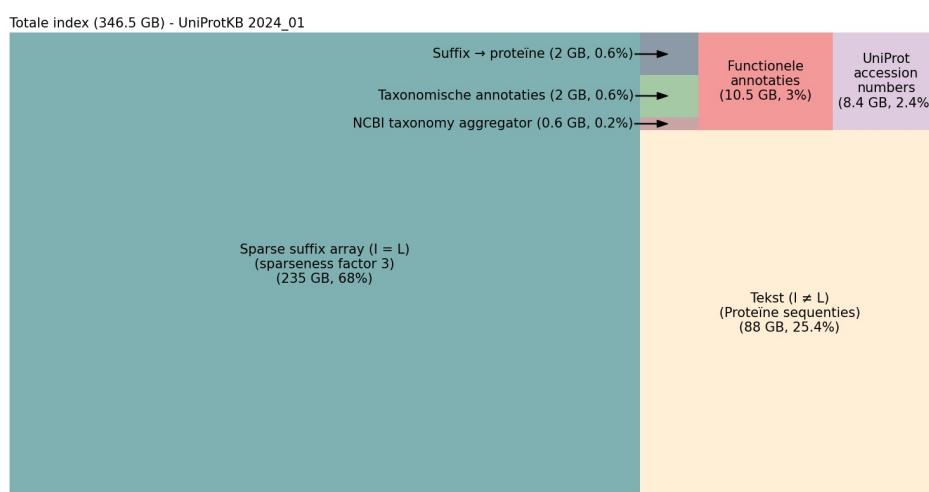


Figuur 5.6: Taxonomische en functionele analyse met UniPept 6.x vs UniPept 5.x voor het tryptische Swiss-Prot peptidebestand. UniPept 5.x wordt uitgevoerd met zowel *filter duplicate peptides*, als *advanced missed cleavage handling* uitgeschakeld. Dit is het beste scenario voor UniPept 5.x, aangezien er geen *missed cleavages* aanwezig zijn, en deze dus ook niet moeten gezocht worden. Merk op dat UniPept 6.x wél *missed cleavages* verwerkt, aangezien dit geen invloed heeft op de performantie bij de nieuwe indexstructuur.

5.7 Overzicht geheugengebruik

In de vorige secties zijn extra onderdelen besproken om de functionaliteiten van UniPept toe te voegen aan de nieuwe index. Deze extra onderdelen vragen extra geheugen, wat het totale geheugengebruik voor UniProtKB 2024_01 laat oplopen tot net geen 350 GB. Figuur 5.7 geeft een overzicht van het geheugengebruik per onderdeel. De volgende opsomming herhaalt kort welke rol elk onderdeel speelt in het geheel.

- Sparse suffix array: De sparse suffix array die opgebouwd wordt op basis van UniProtKB 2024_01 met $k = 3$ en $I = L$. Op basis hiervan zoeken we efficiënt alle suffixen waarmee een peptide matcht.
- Tekst: Dit is de UniProtKB databank met $I \neq L$, waarbij alle proteïnen geconcateneerd worden tot één lange string. Tussen elke proteïne hangt hetzelfde, unieke scheidingsteken (-), en op het einde van de tekst een ander uniek teken (\$).
- Suffix → proteïne: De sparse mapping die beschreven is in sectie 3.4.2. Deze mapping wordt gebruikt om de bijhorende proteïne te vinden van een gematchte suffix.
- NCBI taxonomy aggregator: Een object dat de gebruikte NCBI taxonomy bevat om de LCA* aggregaties uit te voeren. Aangezien deze opgebouwd wordt op basis van de NCBI taxonomy, is de grootte hiervan onafhankelijk van de gebruikte proteïnedatabank.
- UniProt accession numbers: Elke proteïne in UniProtKB heeft een uniek ID (accession number). Voor elke proteïne die matcht met de gezochte peptide wordt dit ID als onderdeel van de output teruggegeven.
- Taxonomische annotaties: Het taxon ID dat per proteïne bijgehouden wordt. Op basis van deze taxon IDs wordt de LCA* van alle gevonden matches berekend.
- Functionele annotaties: De gecomprimeerde functionele annotaties die beschreven zijn in sectie 5.5. Op basis van deze annotaties wordt de functionele analyse van UniPept uitgevoerd.



Figuur 5.7: Visualisatie van het totale geheugengebruik voor de index gebouwd op basis van UniProtKB 2024_01. In totaal vraagt deze index 346.5 GB RAM, waarvan het grootste deel gebruikt wordt door de SSA (sparseness factor $k = 3$, $I = L$) en de proteïne sequenties (de tekst met $I \neq L$) zelf. Naast de twee onderdelen die nodig zijn voor het matchen van de suffixen, is er nog een kleine 25 GB extra RAM nodig om de UniPept analyses aan te kunnen bieden, en om de ID's van de gevonden matches terug te kunnen geven.

5.8 Aanbieden van de nieuwe indexstructuur

In alle eerder vermelde benchmarks bespraken we enkel de zoektijd van een peptidebestand. Bij het opstarten moeten we echter **eerst de indexstructuur inladen in RAM**. Dit alleen duurt 20 tot 25 minuten. We willen dit inladen slechts eenmalig doen om dan alle requests onmiddellijk te kunnen afhandelen. Dit doen we aan de hand van een simpele **webserver** met de Axum crate [3]. Deze webserver laadt de indexstructuur in bij het opstarten, en blijft daarna wachten op HTTP-requests die een JSON-object bevatten met daarin de peptiden die we willen zoeken. Op deze manier is dit probleem elegant opgelost, en kunnen we bovendien aan de hand van **JSON-bestanden** erg makkelijk de input verwerken, en de resultaten terugsturen. Codefragment 2 en 3 geven een voorbeeld input en output.

```
1  {
2      "peptides": [
3          "LAKLFISAV",
4          "ACCISDJLSAAC"
5      ],
6      "equalize_I_and_L": true,
7      "cutoff": 10000
8  }
```

Codefragment 2: Voorbeeld JSON-input voor de webserver waarbij de peptiden LAKLFISAV en ACCISDJLSAAC gezocht worden. Tijdens het zoeken worden I en L gelijkgesteld, en wordt de drempelwaarde B=10 000 gebruikt. Indien er meer matches zijn dan de drempelwaarde B, worden er slechts B matches teruggegeven en wordt de resulterende LCA* op 1 gezet. Dit laatste argument zou ook weggeleggen kunnen worden aangezien de standaardwaarde gebruikt wordt.

5.9 Productiepijplijn

Om een nieuwe versie van de index voor UniProtKB in productie te brengen, zijn verschillende stappen nodig. Een groot deel hiervan valt buiten het domein van deze masterproef. Eerst moet de nieuwe databank gedownload worden, waarna alle data die UniPept nodig heeft hieruit geëxtraheerd moeten worden. De masterproef van medestudent Stijn De Clercq gaat hier dieper op in, en welke verbeteringen hierop aangebracht zijn het afgelopen jaar. Dankzij zijn werk is het veel gemakkelijker geworden om deze nieuwe index te integreren in de UniPept pijplijn. De output van deze pijplijn levert een reeks **tsv**⁶ bestanden op, die gebruikt worden als input om de indexstructuur op te bouwen. Aangezien we de index enkel kunnen bouwen op de HPC vanwege het hoge geheugengebruik, moeten de nodige **tsv** bestanden verplaatst worden naar de HPC. Dit gaat vrij snel aangezien alle communicatie binnen het UGent-netwerk gebeurt aan een snelheid van 1 Gbps. Daarna wordt er een job op de HPC in de wachtrij gezet om de effectieve nieuwe SSA op te bouwen. Hierbij moet eerst wat tijd gerekend worden waarbij de job in de wachtrij staat. Meestal duurt dit niet langer dan enkele uren, waarna het opbouwen van de index zelf een 5-tal uur in beslag neemt. Eens de HPC klaar is met het bouwen van de index, moet de resulterende SSA van ongeveer 235 GB verplaatst worden naar alle UniPept servers die de index zullen hosten. Opnieuw is dit communicatie binnen het UGent-netwerk, waardoor dit ongeveer een half uur in beslag neemt. Tot slot moet de index op alle servers opgestart worden. Zoals vermeld in sectie 5.8, duurt dit ongeveer 20–25 minuten, waarna de index binnenvkomende requests kan verwerken.

⁶Tab-Separated Values

```

1   {
2     "result": [
3       {
4         "sequence": "LAKLFISAV",
5         "lca": 2,
6         "taxa": [
7           2249812,
8           1871048,
9           1262816,
10          1869212
11        ],
12        "uniprot_accession_numbers": [
13          "A0A365TSP2",
14          "A0A7C5MYD2",
15          "R6CCR3",
16          "A0A7V8W1P5"
17        ],
18        "fa": {
19          "counts": {
20            "EC": 0,
21            "IPR": 4,
22            "GO": 3,
23            "all": 4
24          },
25          "data": {
26            "GO:0005886": 2,
27            "IPR:IPR025970": 1,
28            "IPR:IPR010656": 2,
29            "GO:0000271": 1,
30            "IPR:IPR007267": 1,
31            "IPR:IPR004681": 2,
32            "GO:0016020": 1,
33            "GO:0022857": 2
34          }
35        },
36        "cutoff_used": false
37      }
38    ]
39  }

```

Codefragment 3: Output van de input gebruikt in Codefragment 2. Hierbij bevat de sleutel `result` één element voor elke peptide die minstens één match opleverde. Peptiden zonder match worden dus simpelweg weggelaten in de output. Elk element bevat de berekende LCA* voor alle matches, het taxon ID dat overeenkomt met elke match en het UniProt accession number voor elke match. Verder bevat de sleutel `fa` de functionele analyse zoals deze op dit moment door de Unipept API teruggegeven wordt. Tot slot is er ook nog een extra sleutel `cutoff_used` die aanduidt of de bovengrens voor maximaal aantal matches bereikt werd. In dit geval stond deze bovengrens B op 10 000 matches. Indien deze wel bereikt wordt, zal de LCA automatisch op 1 gezet worden en zullen er exact B matches teruggegeven worden (en dus ook matches weggelaten worden).

6 CONCLUSIE & TOEKOMSTIG WERK

In deze masterproef hebben we verschillende opties verkend om de huidige UniPept index voor UniProtKB te vervangen. Hierbij lag de hoofdfocus op het vinden van een nieuwe indexstructuur die aan volgende opties voldoet:

1. De index moet het mogelijk maken om arbitraire peptiden te zoeken.
2. Het geheugengebruik van de index moet beperkt zijn, zodat het mogelijk is niet enkel kleinere proteïnedatabanken te indexeren, maar ook de volledige UniProtKB databank.
3. De indexstructuur moet semi-exacte matching ondersteunen, zodat I en L aan elkaar gelijkgesteld kunnen worden.
4. De indexstructuur moet de taxonomische en functionele analyse van UniPept ondersteunen.

6.1 Conclusie

Een eerste indexstructuur die we bekijken hebben waren **suffixbomen**. Hierbij hebben we een eigen Rust implementatie gemaakt van het algoritme van Ukkonen. Suffixbomen bieden een extreem grote vrijheid, en het zoeken gaat bovendien extreem snel. Ze zijn echter **onbruikbaar** om grote proteïnedatabanken te indexeren vanwege hun **geheugengebruik**.

Een volgende optie die we verkend hebben zijn **suffix arrays**. Na testen bleek dat deze datastructuur ons een goede **balans gaf tussen snelheid en geheugengebruik**. Aan de hand hiervan zijn we erin geslaagd een nieuwe indexstructuur voor UniProtKB op te bouwen die de huidige UniPept-index volledig kan vervangen, en alle bovenstaande doelstellingen haalt. Aan de hand van een aangepast zoeksysteem ondersteunen we zowel exacte als semi-exacte matching, waarbij efficiënt alle informatie van de gematchte peptiden teruggegeven kan worden. Het grootste nadeel van deze manier van werken is dat de functionele en taxonomische analyses van UniPept *on the fly* moeten gebeuren. Ondanks dit nadeel, is de nieuwe versie van UniPept 6.0 10 tot 100 maal sneller dan UniPept 5.x wanneer beide indices *missed cleavages* verwerken. Wanneer *advanced missed cleavages handling* uitgezet wordt op UniPept 5.x, is UniPept 6.x slechts een fractie trager. Dit wil zeggen dat UniPept 6.x willekeurige peptiden (al dan niet tryptisch of met *missed cleavages*) kan verwerken in ongeveer dezelfde tijd die UniPept 5.x nodig heeft om enkel strikt tryptische peptiden te verwerken.

Als derde en vierde indexstructuur hebben we kort de **FM-index en R-index** getest. Na enkele testen bleek al snel dat het **geheugengebruik** tijdens het bouwen **dubbel zo hoog lag** als bij suffix arrays. Daarom hebben we deze opties niet verder uitgewerkt, ook al hebben beide indices interessante eigenschappen voor ons probleem. Zo laten bidirectionele FM-indices toe om algemenere inexakte matching uit te voeren, en is de resulterende index bij R-indices kleiner door het gebruik van run-length encoding.

6.2 Toekomstig werk

Ondanks dat we een index gebouwd hebben die de huidige UniPept index kan vervangen, en willekeurige peptiden kan vinden in plaats van enkel tryptische peptiden, zijn er nog op meerdere vlakken verbeteringen mogelijk.

De gemakkelijkste, en meest directe verbetering is het verder **verkleinen van de resulterende index**. UniProt 2023_04 bevat ongeveer 86.8 miljard aminozuren en 250 miljoen sequenties, waardoor we iets meer dan 87 miljard verschillende suffixen moeten kunnen voorstellen in de suffix array. Aangezien $87 \cdot 10^9 > 2^{32}$ maken we gebruik van 64-bit integers. Hiermee kunnen we echter veel te veel verschillende getallen voorstellen. In de praktijk hebben we slechts nood aan $1 + \lfloor \log_2(87 \cdot 10^9) \rfloor = 37$ bits per waarde in de suffix array. Aangezien de huidige sparse suffix array (met sparseness factor $k = 3$) ongeveer 235 GB groot is, zou de SSA na deze aanpassing slechts 136 GB groot zijn. Bij de tekst kunnen we dezelfde aanpassing maken. Daar hebben we $26 + 2 = 28$ verschillende tekens, wat aan de hand van 5 bits voorgesteld kan worden. Aangezien de huidige tekst ongeveer 88 GB groot is, zou dit resulteren in een tekst die slechts $88 \cdot \frac{5}{8} \approx 55$ GB groot is. Hierbij komt nog taxonomische en functionele informatie in combinatie met de UniProt accession numbers en de suffix naar proteïne mapping. Hierdoor zou de totale resulterende index van $235 + 88 + 23 = 346$ GB naar $136 + 55 + 23 = 214$ GB gaan, wat iets meer dan **33% winst** oplevert. Merk op dat dit de nodige hoeveelheid geheugen tijdens het opbouwen niet verkleint, aangezien we in die fase wel gebruik moeten maken van 64-bit integers en 8-bit karakters. Door deze manier van compressie te gebruiken zou het ook mogelijk worden om sparseness factor $k = 2$ te gebruiken op de Unipept servers. In het totaal zou de index dan $353 \cdot \frac{37}{64} + 55 + 23 \approx 282$ GB groot zijn, wat comfortabel in 0.5 TB RAM past. Ondanks de overhead die de compressie introduceert, kan de performantie op deze manier mogelijk toch nog verbeterd worden. Bovendien zullen hierdoor ook peptiden die slechts uit twee aminozuren bestaan gematcht kunnen worden aan de hand van de SSA.

Ook tijdens het opbouwen van de SA zijn er nog interessante pistes om te verkennen. Zo zou het extreem voordelig zijn om een manier te vinden om de suffix array onmiddellijk **sparse te maken tijdens het opbouwen**. Hierbij zal het bovendien belangrijk zijn dat deze implementatie een **lage constante factor heeft op vlak van geheugengebruik**. Indien deze constante vrij groot is, zal de winst verloren gaan ten opzichte van de huidige implementatie, aangezien we altijd een kleine sparseness factor k gebruiken. Als alternatief kunnen we mogelijk het maximale geheugengebruik beperken via een online algoritme. Zo kan tijdens het opbouwen één proteïne per keer toegevoegd worden aan de suffix array, maar kunnen we nieuwere UniProtKB releases ook efficiënt afhandelen door te trekken van de index uit de vorige release.

Een derde punt van verbetering zit in het **berekenen van de LCA***. Indien deze voorberekend wordt tijdens het opbouwen van de index, zal dit niet enkel de performantie ten goede komen, het zal ook de nood voor de drempelwaarde B (die standaard op 10 000 staat) sterk verminderen. De meest duidelijke piste hiervoor is aan de hand van Enhanced Suffix Arrays (ESAs). In deze masterproef hebben we deze piste niet verder verkend vanwege de eerste indicatie dat het berekenen van de extra tabellen het geheugengebruik zou verdubbelen, en dat het *on the fly* berekenen van de analyses een acceptabele overhead met zich meebrengt.

Een vierde mogelijke route is het **uitbreiden van de inexakte matching**. Op dit moment is dit slechts in een extreem beperkte vorm aanwezig. We hebben namelijk enkel de optie om I en L aan elkaar gelijk te stellen. Tools zoals de eerder vermelde Expasy ScanProsite tool ondersteunen verschillende manieren om inexakte matching uit te voeren. Zo zouden we ondersteuning voor karakterklassen, een reeks van herhalingen (al dan niet met een minimum en maximum bereik) en wildcards kunnen toevoegen. Gelijkaardig aan hoe *livegrep* [35] gebruikmaakt van Google's RE2 [56] regex engine en meerdere suffix arrays [11]. Een andere manier van inexakte matching kan via het toelaten van maximaal x mismatches. Op deze manier kan men ook omgaan met kleine mutaties die ontstaan in proteïnen, of kleine variaties die ontstaan bij het omzetten van spectra naar aminozurequenties. Over deze laatste vorm van inexakte matching is door het team van Unipept een vervolgmasterproef uitgeschreven om dit volgend jaar te verkennen. Hierbij zal verder ingegaan worden op de FM-index en de R-index.

REFERENTIES

- [1] *A Tree Structure implemented in Rust.* <https://applied-math-coding.medium.com/a-tree-structure-implemented-in-rust-8344783abd75>.
- [2] *Amino Acid Codes.* <https://www.ddbj.nig.ac.jp/ddbj/code-e.html#amino-1>. Geraadpleegd: 16 april 2024.
- [3] *Axum.* <https://docs.rs/axum/latest/axum/>. Geraadpleegd: 26 maart 2024.
- [4] Tim Van Den Bossche e.a. „Critical Assessment of MetaProteome Investigation (CAMPI): a multi-laboratory comparison of established workflows”. In: *Nature Communications* 12.1 (dec 2021). DOI: [10.1038/s41467-021-27542-8](https://doi.org/10.1038/s41467-021-27542-8). URL: <https://doi.org/10.1038/s41467-021-27542-8>.
- [5] Gunnar Brinkmann. *Algoritmen en datastructuren 3, 2021.* https://twicaagt.ugent.be/~gbrinkma/DA3_2021/lesnotas_AD3_2021.pdf. Jul 2021.
- [6] Michael Burrows en David John Wheeler. *A Block-sorting Lossless Data Compression Algorithm.* Tech. rap. Digital Equipment Corporation, 1994.
- [7] Chuming Chen e.a. „A fast Peptide Match service for UniProt Knowledgebase”. In: *Bioinformatics* 29.21 (aug 2013), p. 2808–2809. ISSN: 1367-4811. DOI: [10.1093/bioinformatics/btt484](https://doi.org/10.1093/bioinformatics/btt484). URL: <http://dx.doi.org/10.1093/bioinformatics/btt484>.
- [8] The UniProt Consortium. „UniProt: the Universal Protein Knowledgebase in 2023”. In: *Nucleic Acids Research* 51.D1 (nov 2022), p. D523–D531. ISSN: 0305-1048. DOI: [10.1093/nar/gkac1052](https://doi.org/10.1093/nar/gkac1052). eprint: <https://academic.oup.com/nar/article-pdf/51/D1/D523/48441158/gkac1052.pdf>. URL: <https://doi.org/10.1093/nar/gkac1052>.
- [9] Peter Dawyndt en Jan Fostier. *Computational Challenges in Bioinformatics [Lecture notes, Ghent University].* 2022.
- [10] Bálint Dömölki. „An algorithm for syntactical analysis”. In: *Computational Linguistics* 3 (1964), p. 29–46.
- [11] Nelson Elhage. *Regular Expression Search with Suffix Arrays.* <https://blog.nelhage.com/2015/02/regular-expression-search-with-suffix-arrays/>. Geraadpleegd: 28 april 2024. Feb 2015.
- [12] *Expasy ScanProsite tool.* <https://prosite.expasy.org/scanprosite/>. Geraadpleegd: 26 maart 2024.
- [13] S. Federhen. „The NCBI Taxonomy database”. In: *Nucleic Acids Research* 40.D1 (dec 2011), p. D136–D143. DOI: [10.1093/nar/gkr1178](https://doi.org/10.1093/nar/gkr1178). URL: <https://doi.org/10.1093/nar/gkr1178>.
- [14] P. Ferragina en G. Manzini. „Opportunistic data structures with applications”. In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. Nov 2000, p. 390–398. DOI: [10.1109/SFCS.2000.892127](https://doi.org/10.1109/SFCS.2000.892127).
- [15] Jan Fostier. *Ukkonen in C++.* <https://github.ugent.be/jfostier/CCB/tree/master/suffixtree>. Geraadpleegd: 10 oktober 2023.
- [16] Travis Gagie, Gonzalo Navarro en Nicola Prezza. „Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space”. In: *Journal of the ACM* 67.1 (jan 2020), p. 1–54. ISSN: 1557-735X. DOI: [10.1145/3375890](https://doi.org/10.1145/3375890). URL: [http://dx.doi.org/10.1145/3375890](https://doi.org/10.1145/3375890).
- [17] Travis Gagie, Gonzalo Navarro en Nicola Prezza. „Optimal-Time Text Indexing in BWT-runs Bounded Space”. In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial en Applied Mathematics, jan 2018, p. 1459–1477. DOI: [10.1137/1.9781611975031.96](https://doi.org/10.1137/1.9781611975031.96). URL: [http://dx.doi.org/10.1137/1.9781611975031.96](https://doi.org/10.1137/1.9781611975031.96).

- [18] Paweł Gawrychowski en Tomasz Kociumaka. *Sparse Suffix Tree Construction in Optimal Time and Space*. 2016. arXiv: [1608.00865 \[cs.DS\]](https://arxiv.org/abs/1608.00865).
- [19] *Github UMGAP*. <https://github.com/unipept/umgap>.
- [20] Isaac Gouy. *The Computer Language 23.03 Benchmarks Game*. <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html>. Geraadpleegd: 8 november 2023.
- [21] Ilya Grebnov. *libsais*. <https://github.com/IlyaGrebnev/libsais>. Geraadpleegd: 2 mei 2023.
- [22] Robbert Gurdeep Singh e.a. „Unipept 4.0: Functional Analysis of Metaproteome Data”. In: *Journal of Proteome Research* 18.2 (nov 2018), p. 606–615. ISSN: 1535-3907. DOI: [10.1021/acs.jproteome.8b00716](https://doi.org/10.1021/acs.jproteome.8b00716). URL: [http://dx.doi.org/10.1021/acs.jproteome.8b00716](https://dx.doi.org/10.1021/acs.jproteome.8b00716).
- [23] Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, mei 1997. DOI: [10.1017/cbo9780511574931](https://doi.org/10.1017/cbo9780511574931). URL: <https://doi.org/10.1017/cbo9780511574931>.
- [24] Tony Hoare. *Null References: The Billion Dollar Mistake*. <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/>. Aug 2009.
- [25] Tanja Holstein e.a. „PepGM: a probabilistic graphical model for taxonomic inference of viral proteome samples with associated confidence scores”. In: *Bioinformatics* 39.5 (mei 2023), btad289. ISSN: 1367-4811. DOI: [10.1093/bioinformatics/btad289](https://doi.org/10.1093/bioinformatics/btad289). eprint: <https://academic.oup.com/bioinformatics/article-pdf/39/5/btad289/50311697/btad289.pdf>. URL: <https://doi.org/10.1093/bioinformatics/btad289>.
- [26] R. Nigel Horspool. „Practical fast searching in strings”. In: *Software: Practice and Experience* 10.6 (1980), p. 501–506. DOI: <https://doi.org/10.1002/spe.4380100608>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380100608>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380100608>.
- [27] *How Old is the Universe*. https://map.gsfc.nasa.gov/universe/uni_age.html. Geraadpleegd: 1 april 2024.
- [28] Felix Van der Jeugt e.a. „UMGAP: the Unipept MetaGenomics Analysis Pipeline”. In: *BMC Genomics* 23.1 (jun 2022). DOI: [10.1186/s12864-022-08542-4](https://doi.org/10.1186/s12864-022-08542-4). URL: <https://doi.org/10.1186/s12864-022-08542-4>.
- [29] Dominik Kempa en Tomasz Kociumaka. „Breaking the $O(n)$ -Barrier in the Construction of Compressed Suffix Arrays and Suffix Trees”. In: *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial en Applied Mathematics, jan 2023, p. 5122–5202. ISBN: 9781611977554. DOI: [10.1137/1.9781611977554.ch187](https://doi.org/10.1137/1.9781611977554.ch187). URL: [http://dx.doi.org/10.1137/1.9781611977554.ch187](https://dx.doi.org/10.1137/1.9781611977554.ch187).
- [30] Donald E. Knuth, James H. Morris Jr. en Vaughan R. Pratt. „Fast Pattern Matching in Strings”. In: *SIAM Journal on Computing* 6.2 (1977), p. 323–350. DOI: [10.1137/0206024](https://doi.org/10.1137/0206024). eprint: <https://doi.org/10.1137/0206024>. URL: <https://doi.org/10.1137/0206024>.
- [31] Pang Ko en Srinivas Aluru. „Space efficient linear time construction of suffix arrays”. In: *Journal of Discrete Algorithms* 3.2 (2005). Combinatorial Pattern Matching (CPM) Special Issue, p. 143–156. ISSN: 1570-8667. DOI: <https://doi.org/10.1016/j.jda.2004.08.002>. URL: <https://www.sciencedirect.com/science/article/pii/S1570866704000498>.
- [32] Jannike Lea Krause e.a. „Following the community development of SIHUMIx – a new intestinal in vitro model for bioreactor use”. In: *Gut Microbes* 11.4 (2020). PMID: 31918607, p. 1116–1129. DOI: [10.1080/19490976.2019.1702431](https://doi.org/10.1080/19490976.2019.1702431). eprint: <https://doi.org/10.1080/19490976.2019.1702431>. URL: <https://doi.org/10.1080/19490976.2019.1702431>.
- [33] Kvark. *Archon*. <https://github.com/kvark/dark-archon>. Geraadpleegd: 22 november 2023.
- [34] T. W. Lam e.a. „High Throughput Short Read Alignment via Bi-directional BWT”. In: *2009 IEEE International Conference on Bioinformatics and Biomedicine*. 2009, p. 31–36. DOI: [10.1109/BIBM.2009.42](https://doi.org/10.1109/BIBM.2009.42).
- [35] livegrep. <https://github.com/livegrep/livegrep>. Geraadpleegd: 28 april 2024.

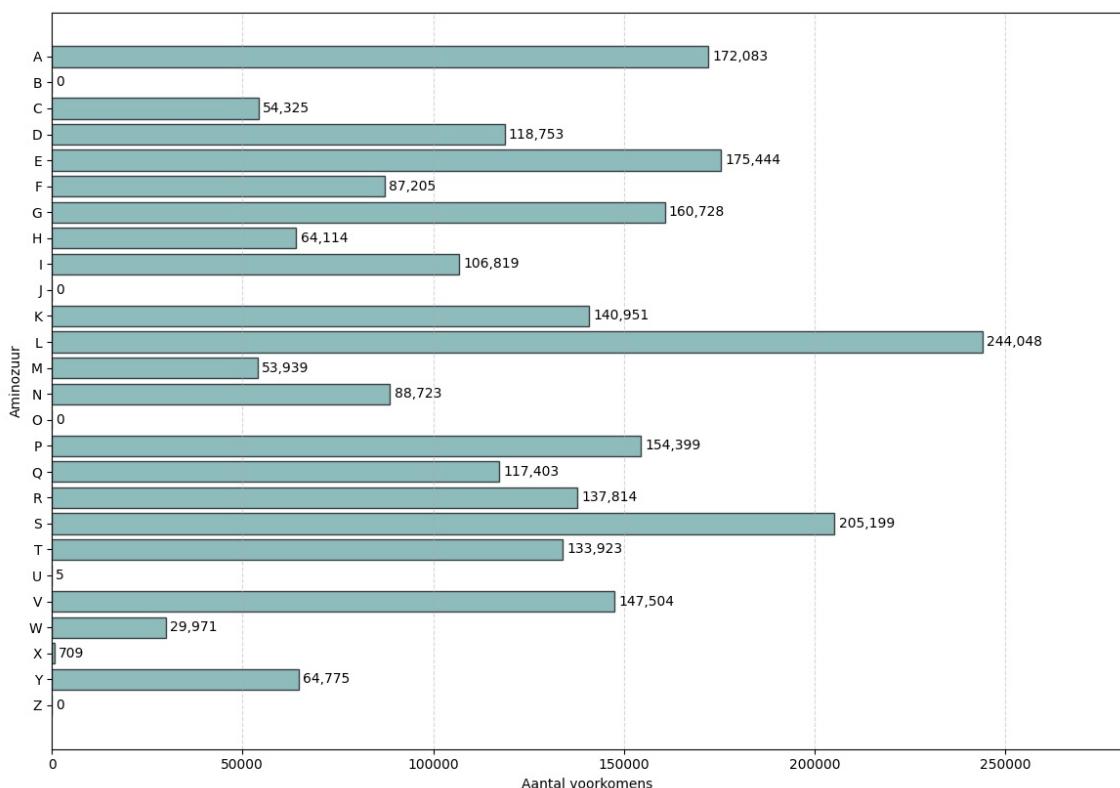
- [36] Ben Lovy. *No More Tears, No More Knots: Arena-Allocated Trees in Rust*. <https://dev.to/deciduously/no-more-tears-no-more-knots-arena-allocated-trees-in-rust-44k6>. Dec 2019.
- [37] H. Luo en H. Nijveen. „Understanding and identifying amino acid repeats”. In: *Briefings in Bioinformatics* 15.4 (feb 2013), p. 582–591. ISSN: 1477-4054. DOI: [10.1093/bib/bbt003](https://doi.org/10.1093/bib/bbt003). URL: <http://dx.doi.org/10.1093/bib/bbt003>.
- [38] Udi Manber en Gene Myers. „Suffix Arrays: A New Method for on-Line String Searches”. In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '90. San Francisco, California, USA: Society for Industrial en Applied Mathematics, 1990, p. 319–327. ISBN: 0898712513.
- [39] Rupert L. Mayer en Karl Mechtler. „Immunopeptidomics in the Era of Single-Cell Proteomics”. In: *Biology* 12.12 (2023). ISSN: 2079-7737. DOI: [10.3390/biology12121514](https://doi.org/10.3390/biology12121514). URL: <https://www.mdpi.com/2079-7737/12/12/1514>.
- [40] Edward M. McCreight. „A Space-Economical Suffix Tree Construction Algorithm”. In: *J. ACM* 23.2 (apr 1976), p. 262–272. ISSN: 0004-5411. DOI: [10.1145/321941.321946](https://doi.org/10.1145/321941.321946). URL: <https://doi.org/10.1145/321941.321946>.
- [41] Bart Mesuere e.a. „High-throughput metaproteomics data analysis with UniPept: A tutorial”. In: *Journal of Proteomics* 171 (jan 2018), p. 11–22. ISSN: 1874-3919. DOI: [10.1016/j.jprot.2017.05.022](https://doi.org/10.1016/j.jprot.2017.05.022). URL: <http://dx.doi.org/10.1016/j.jprot.2017.05.022>.
- [42] Bart Mesuere e.a. „The UniPept metaproteomics analysis pipeline”. In: *PROTEOMICS* 15.8 (feb 2015), p. 1437–1442. ISSN: 1615-9861. DOI: [10.1002/pmic.201400361](https://doi.org/10.1002/pmic.201400361). URL: <http://dx.doi.org/10.1002/pmic.201400361>.
- [43] Bart Mesuere e.a. „UniPept web services for metaproteomics analysis”. In: *Bioinformatics* 32.11 (jan 2016), p. 1746–1748. ISSN: 1367-4803. DOI: [10.1093/bioinformatics/btw039](https://doi.org/10.1093/bioinformatics/btw039). URL: <http://dx.doi.org/10.1093/bioinformatics/btw039>.
- [44] Bart Mesuere e.a. „UniPept: Tryptic Peptide-Based Biodiversity Analysis of Metaproteome Samples”. In: *Journal of Proteome Research* 11.12 (nov 2012), p. 5773–5780. ISSN: 1535-3907. DOI: [10.1021/pr300576s](https://doi.org/10.1021/pr300576s). URL: <http://dx.doi.org/10.1021/pr300576s>.
- [45] Yuta Mori. *Libdivsufsort*. <https://github.com/y-256/libdivsufsort>. Geraadpleegd: 22 november 2023.
- [46] John Von Neumann. *First Draft of a Report on the EDVAC*. <https://web.mit.edu/STS.035/www/PDFs/edvac.pdf>. 1945.
- [47] Ge Nong, Sen Zhang en Daricks Wai Hong Chan. „Linear Suffix Array Construction by Almost Pure Induced-Sorting”. In: mrt 2009, p. 193–202. DOI: [10.1109/DCC.2009.42](https://doi.org/10.1109/DCC.2009.42).
- [48] Michael A. Persinger. „Support for Eddington's Number and his Approach to Astronomy: Recent Developments in the Physics and Chemistry of the Human Brain”. In: *International Letters of Chemistry, Physics and Astronomy* 13 (mei 2013), p. 8–19. ISSN: 2299-3843. DOI: [10.56431/p-n56z0k](https://doi.org/10.56431/p-n56z0k). URL: <http://dx.doi.org/10.56431/p-n56z0k>.
- [49] *Proteomes · Homo sapiens*. <https://www.uniprot.org/proteomes/UP000005640>. Geraadpleegd: 5 oktober 2023.
- [50] *Proteomes · Human papillomavirus*. <https://www.uniprot.org/proteomes/UP000126093>. Geraadpleegd: 5 oktober 2023.
- [51] *Proteomes · Influenza B virus (B/Bangkok/141/1994)*. <https://www.uniprot.org/proteomes/UP000119054>. Geraadpleegd: 5 oktober 2023.
- [52] Sanguthevar Rajasekaran en Marius Nicolae. „An elegant algorithm for the construction of suffix arrays”. In: *Journal of Discrete Algorithms* 27 (2014), p. 21–28. ISSN: 1570-8667. DOI: <https://doi.org/10.1016/j.jda.2014.03.001>. URL: <https://www.sciencedirect.com/science/article/pii/S1570866714000173>.
- [53] *Rayon*. <https://docs.rs/rayon/latest/rayon/>.

- [54] *Rayon FAQ: How does Rayon balance work between threads?* <https://github.com/rayon-rs/rayon/blob/main/FAQ.md>. Geraadpleegd: 24 januari 2024.
- [55] *Rc<T>, the Reference Counted Smart Pointer.* <https://doc.rust-lang.org/book/ch15-04-rc.html#rct-the-reference-counted-smart-pointer>.
- [56] *RE2.* <https://github.com/google/re2>. Geraadpleegd: 28 april 2024.
- [57] *RefCell<T> and the Interior Mutability Pattern.* <https://doc.rust-lang.org/book/ch15-05-interior-mutability.html#refcellt-and-the-interior-mutability-pattern>.
- [58] *Rust documentation: usize.* <https://doc.rust-lang.org/std/primitive.usize.html>.
- [59] *Rust: Data Races and Race Conditions.* <https://doc.rust-lang.org/nomicon/races.html#data-races-and-race-conditions>. Geraadpleegd: 24 januari 2024.
- [60] Conrad L Schoch e.a. „NCBI Taxonomy: a comprehensive update on curation, resources and tools”. In: *Database* 2020 (jan 2020). DOI: [10.1093/database/baaa062](https://doi.org/10.1093/database/baaa062). URL: <https://doi.org/10.1093/database/baaa062>.
- [61] *The inTelligence And Machine lEarning (TAME) Toolkit for Introductory Data Science, Chemical-Biological Analyses, Predictive Modeling, and Database Mining for Environmental Health Research.* <https://uncsrp.github.io/Data-Analysis-Training-Modules/>. Geraadpleegd: 12 mei 2024.
- [62] E. Ukkonen. „On-line construction of suffix trees”. In: *Algorithmica* 14.3 (sep 1995). Vrij leesbaar op: <https://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf>, p. 249–260. DOI: [10.1007/bf01206331](https://doi.org/10.1007/bf01206331). URL: <https://doi.org/10.1007/bf01206331>.
- [63] *Unipept: Important statistics.* <https://github.com/unipept/unipept/wiki/important-statistics>. Geraadpleegd: 3 maart 2024.
- [64] *UniProt peptide search tool.* <https://www.uniprot.org/peptide-search>. Geraadpleegd: 26 maart 2024.
- [65] Pieter Verschaffelt. *Unipept Infrastructure Wiki.* <https://github.com/unipept/unipept/wiki#infrastructure>. Geraadpleegd: 10 november 2023.
- [66] Pieter Verschaffelt e.a. „Unipept CLI 2.0: adding support for visualizations and functional annotations”. In: *Bioinformatics* 36.14 (jun 2020). Red. door Pier Luigi Martelli, p. 4220–4221. ISSN: 1367-4811. DOI: [10.1093/bioinformatics/btaa553](https://doi.org/10.1093/bioinformatics/btaa553). URL: <http://dx.doi.org/10.1093/bioinformatics/btaa553>.
- [67] Pieter Verschaffelt e.a. „Unipept Desktop 2.0: Construction of Targeted Reference Protein Databases for Metaproteogenomics Analyses”. In: *Journal of Proteome Research* 22.8 (jul 2023), p. 2620–2628. ISSN: 1535-3907. DOI: [10.1021/acs.jproteome.3c00091](https://doi.org/10.1021/acs.jproteome.3c00091). URL: <http://dx.doi.org/10.1021/acs.jproteome.3c00091>.
- [68] Pieter Verschaffelt e.a. „Unipept Desktop: A Faster, More Powerful Metaproteomics Results Analysis Tool”. In: *Journal of Proteome Research* 20.4 (jan 2021), p. 2005–2009. ISSN: 1535-3907. DOI: [10.1021/acs.jproteome.0c00855](https://doi.org/10.1021/acs.jproteome.0c00855). URL: <http://dx.doi.org/10.1021/acs.jproteome.0c00855>.
- [69] Verschaffelt, Pieter. „Building the Unipept Ecosystem : empowering high-throughput metaproteomics data analysis for characterizing complex microbial communities”. eng. Proefschrift. Ghent University, 2023, XXXVI, 158.
- [70] Sebastiano Vigna. „Broadword Implementation of Rank/Select Queries”. In: *Experimental Algorithms*. Red. door Catherine C. McGeoch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 154–168. ISBN: 978-3-540-68552-4.

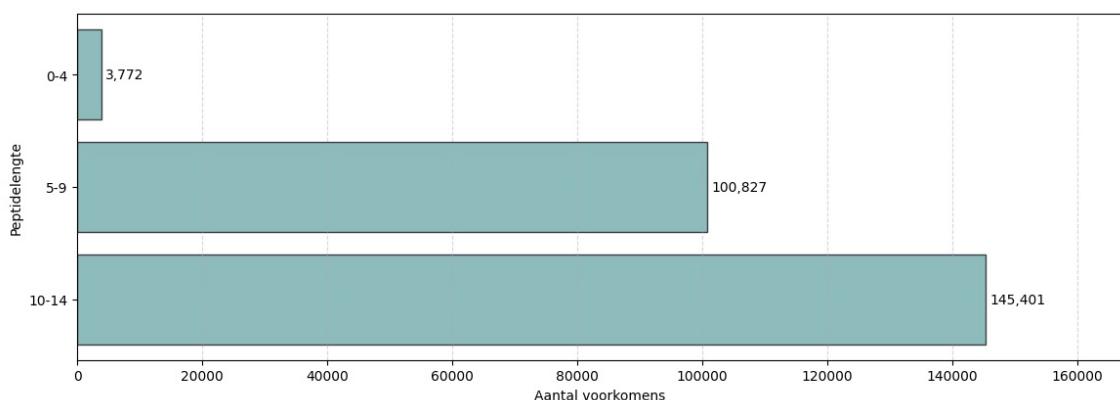
A STATISTIEKEN PEPTIDEBESTANDEN

Deze grafieken visualiseren de verdeling van aminozuren en de lengte van de peptiden voor de verschillende peptidebestanden.

A.1 Human-Prot

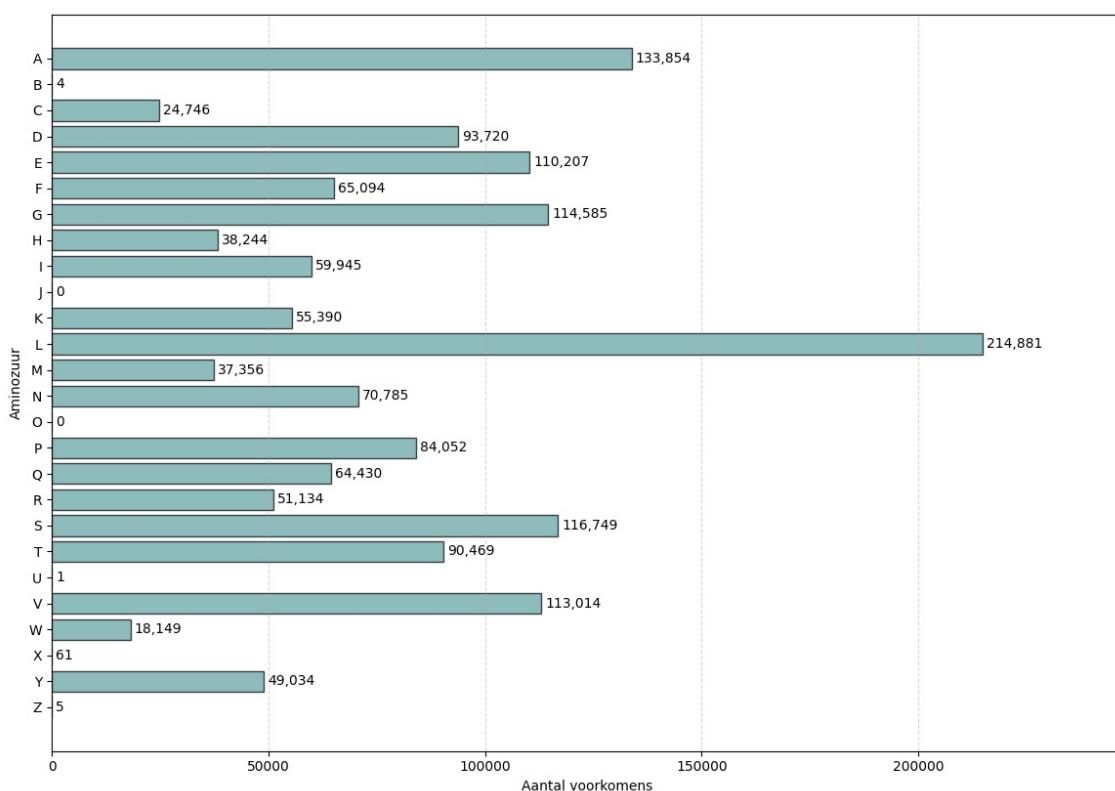


Figuur A.1: Distributie van de aminozuren in het Human-Prot peptidebestand.

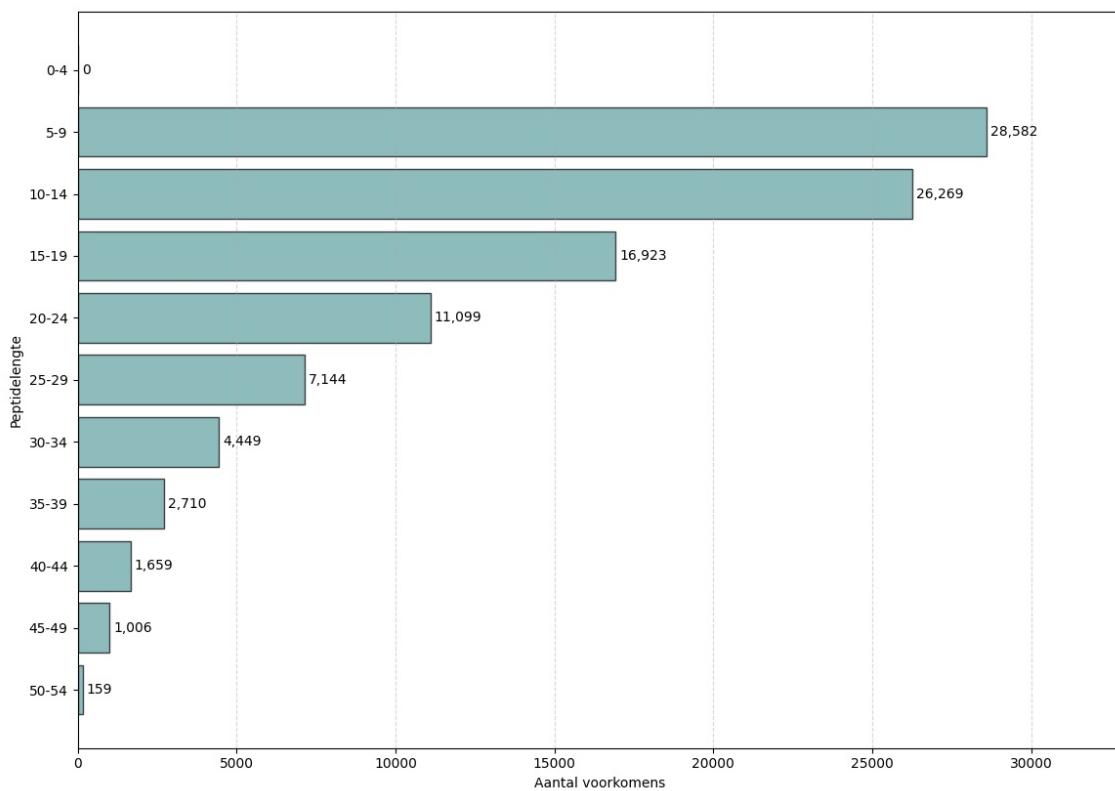


Figuur A.2: Lengtedistributie van de peptiden in het Human-Prot peptidebestand.

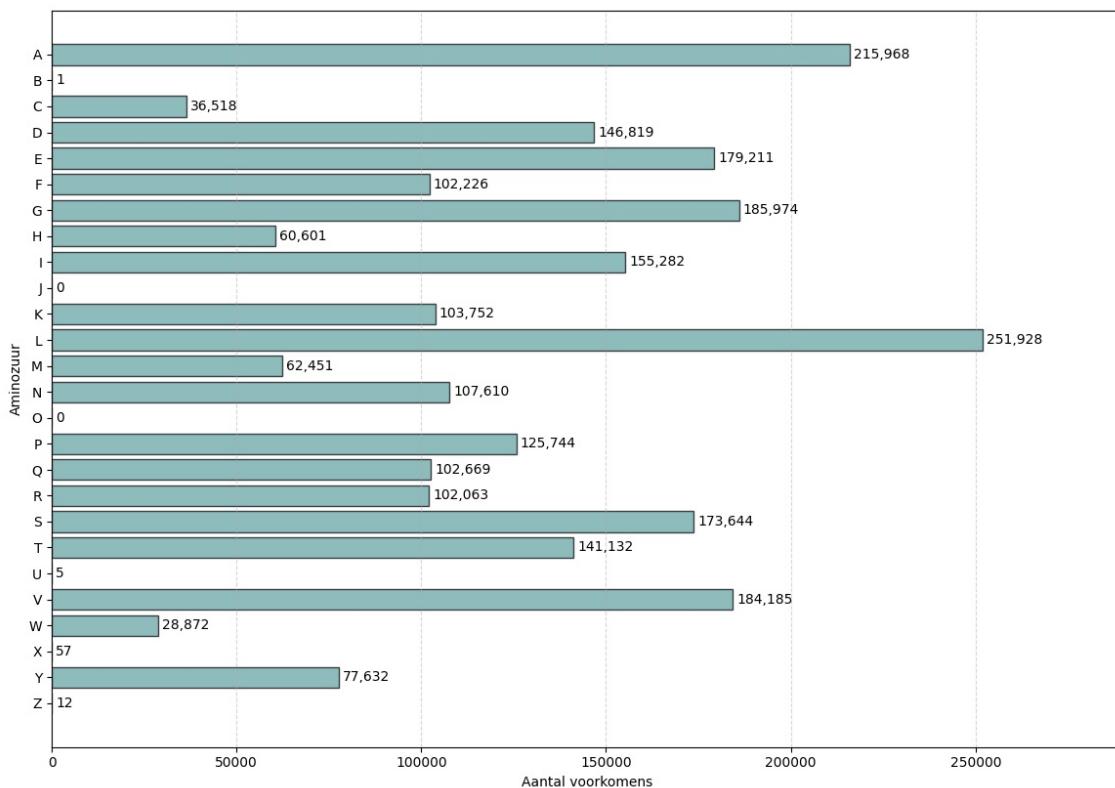
A.2 Swiss-Prot



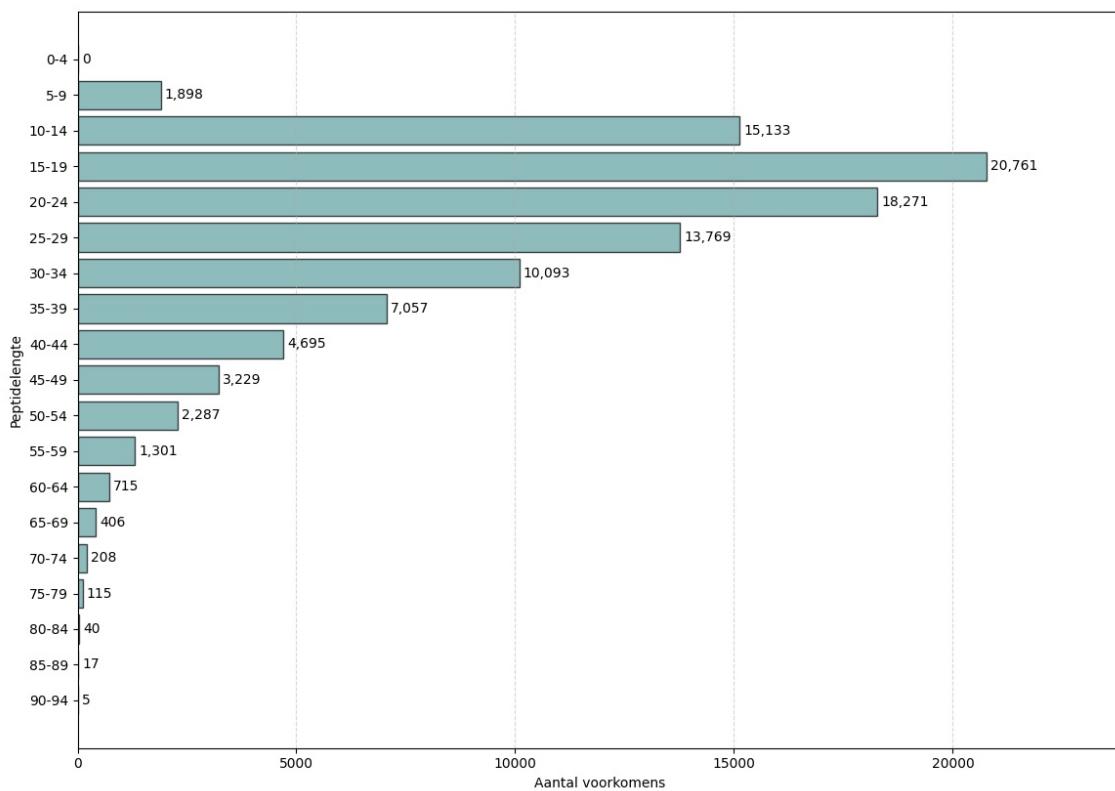
Figuur A.3: Distributie van de aminozuren in het Swiss-Prot peptidebestand met enkel tryptische peptiden



Figuur A.4: Lengtedistributie van de peptiden in het Swiss-Prot peptidebestand met enkel tryptische peptiden

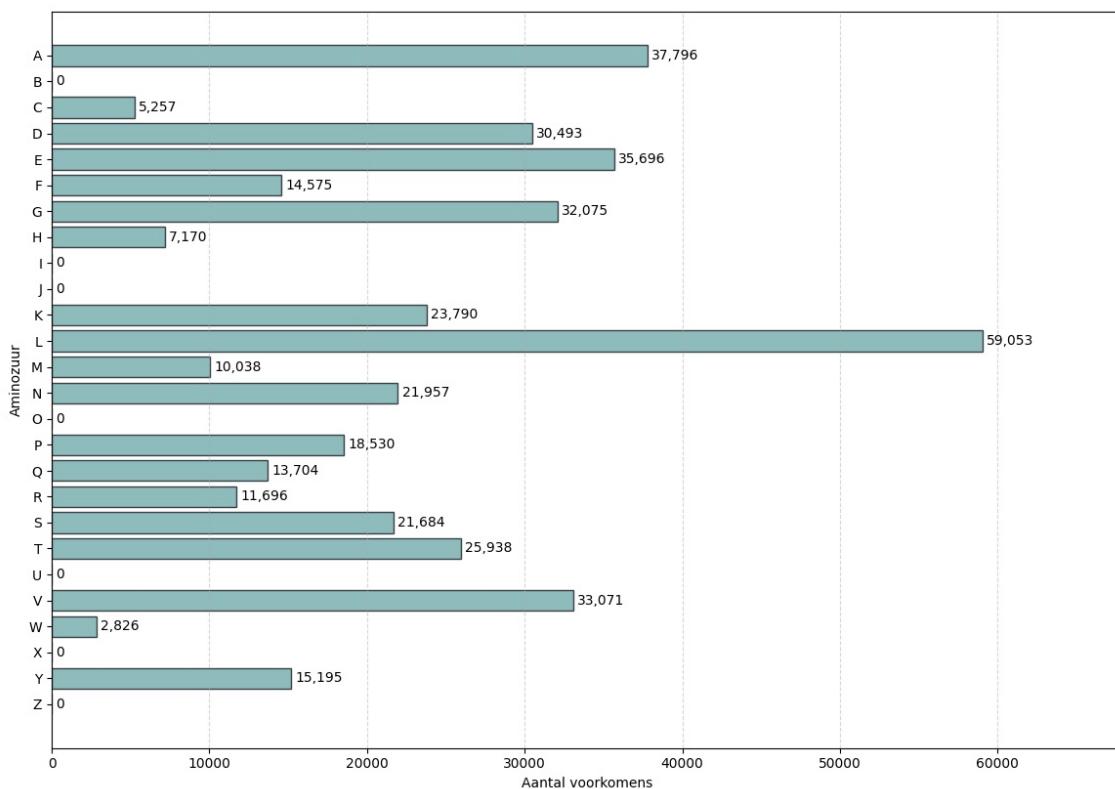


Figuur A.5: Distributie van de aminozuren in het Swiss-Prot peptidebestand met *missed cleavages*.

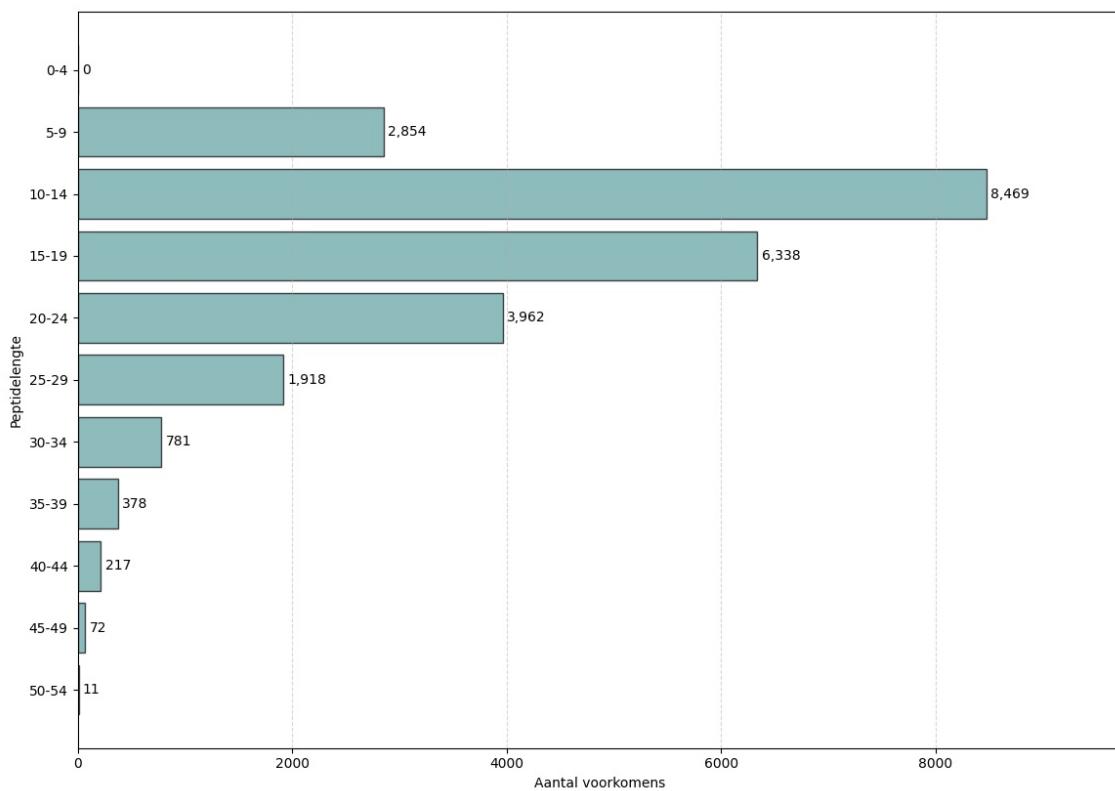


Figuur A.6: Lengtedistributie van de peptiden in het Swiss-Prot peptidebestand met *missed cleavages*.

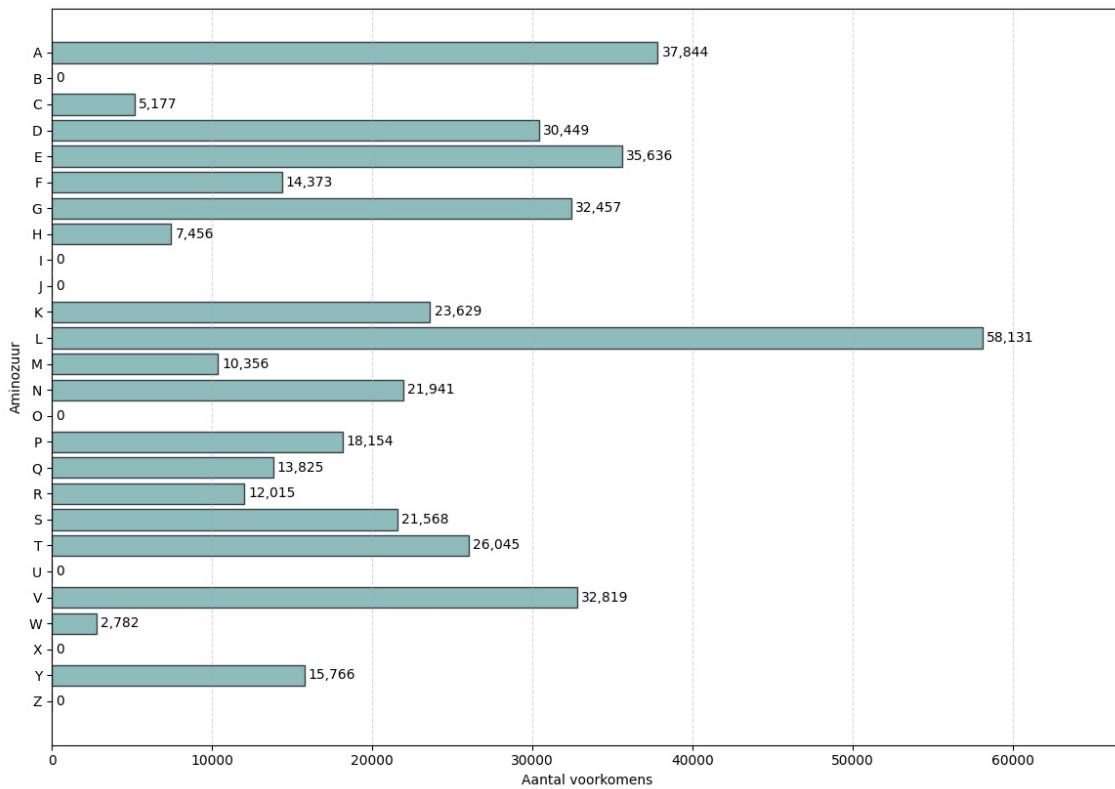
A.3 SIHUMI



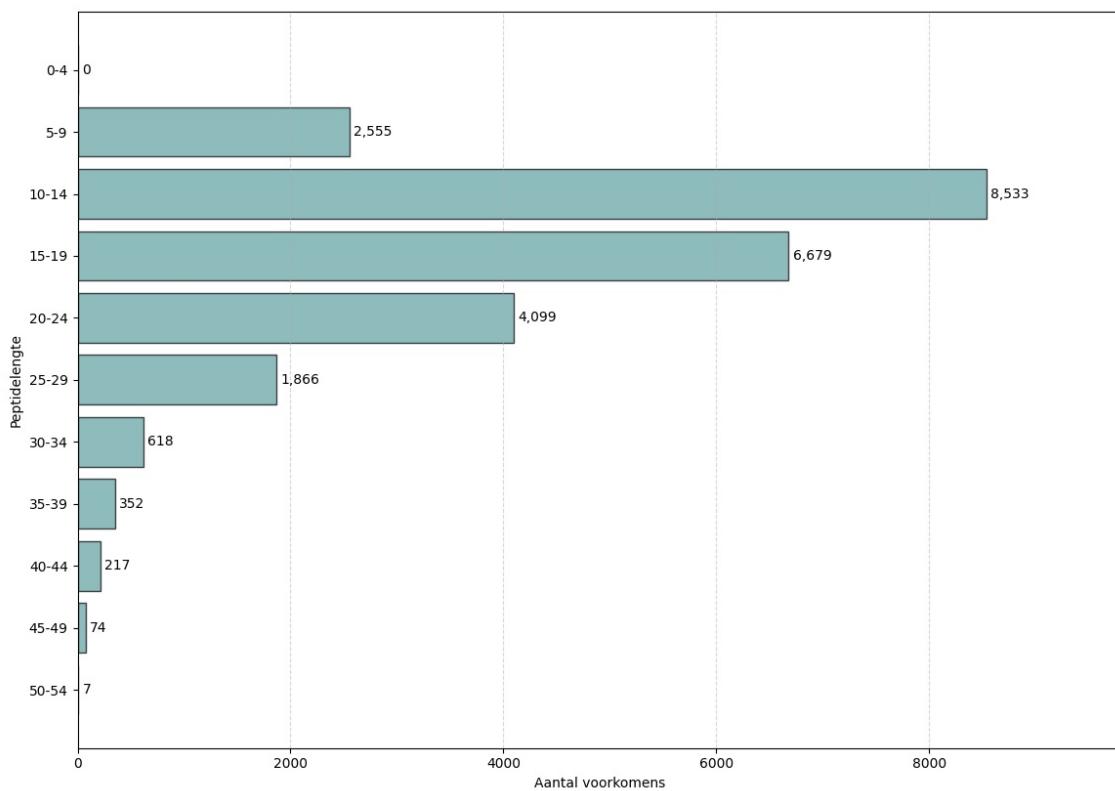
Figuur A.7: Distributie van de aminozuren in het SIHUMI 03 peptidebestand.



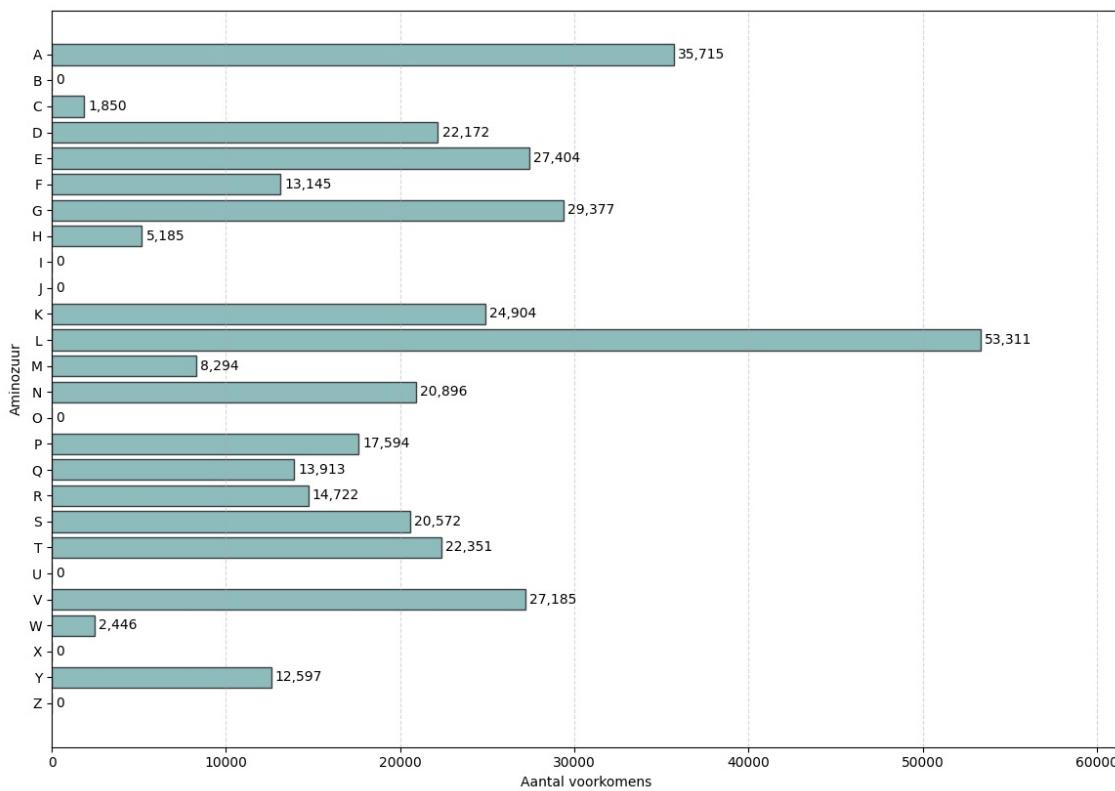
Figuur A.8: Lengtedistributie van de peptiden in het in het SIHUMI 03 peptidebestand.



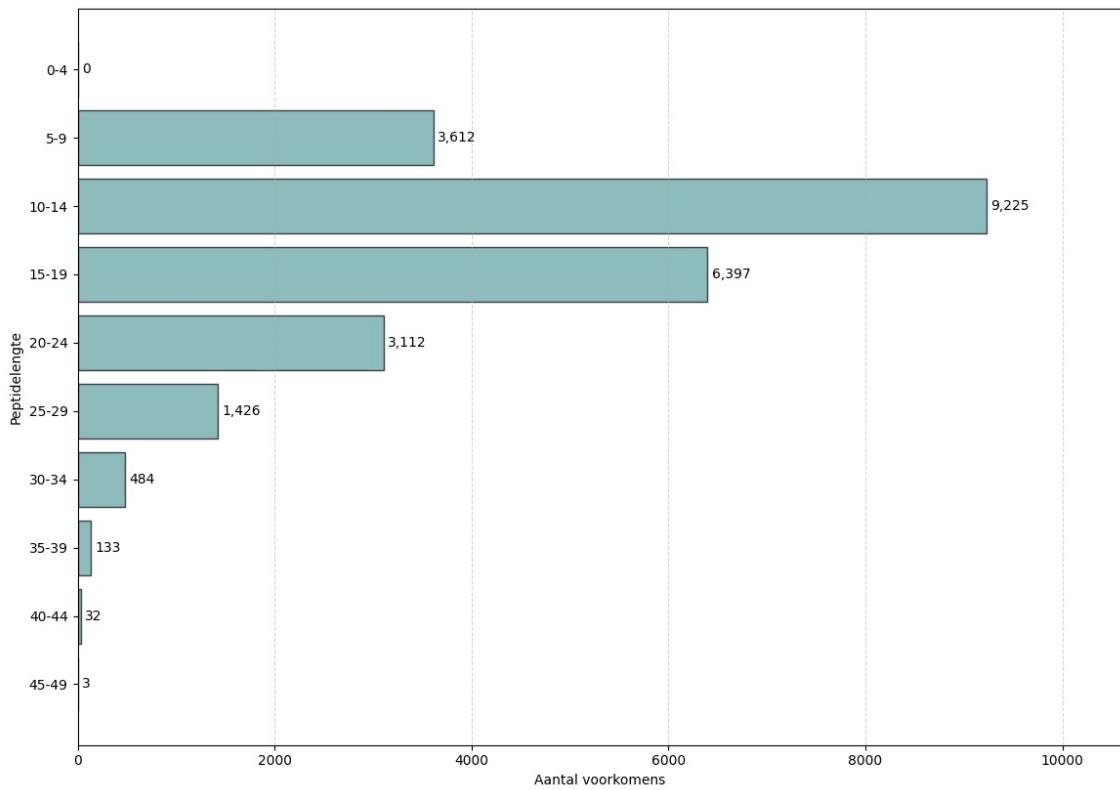
Figuur A.9: Distributie van de aminozuren in het SIHUMI 05 peptidebestand.



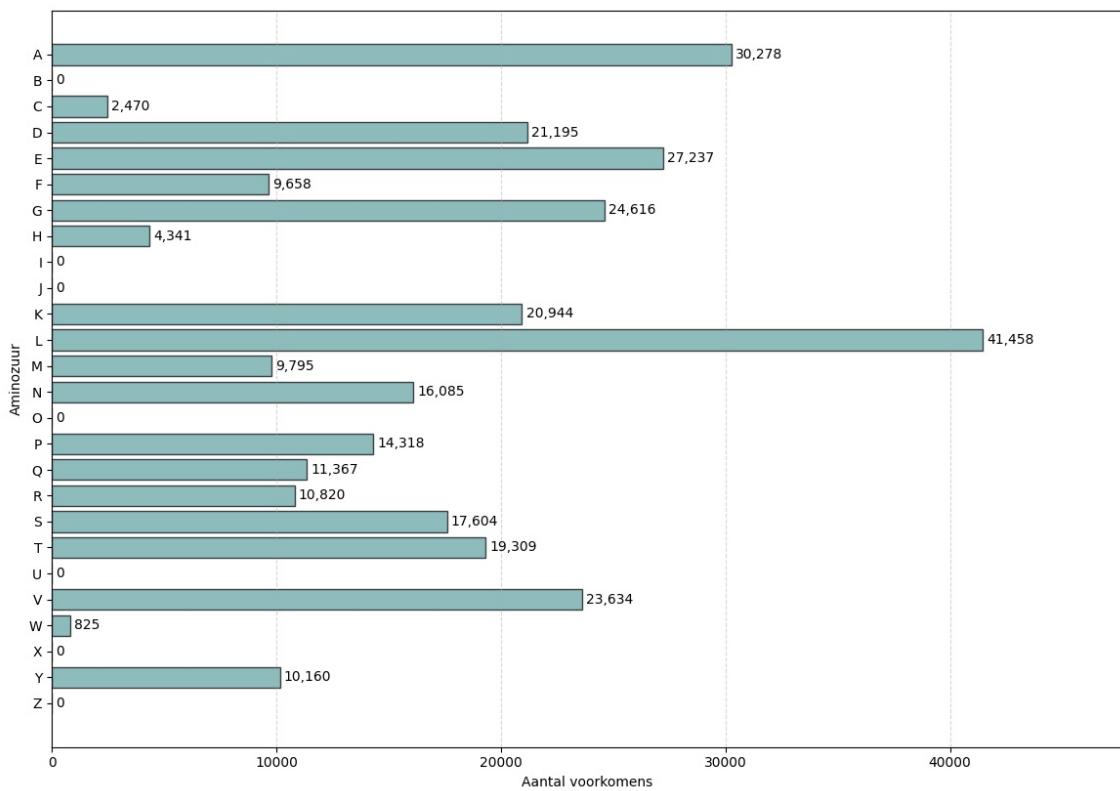
Figuur A.10: Lengtedistributie van de peptiden in het in het SIHUMI 05 peptidebestand.



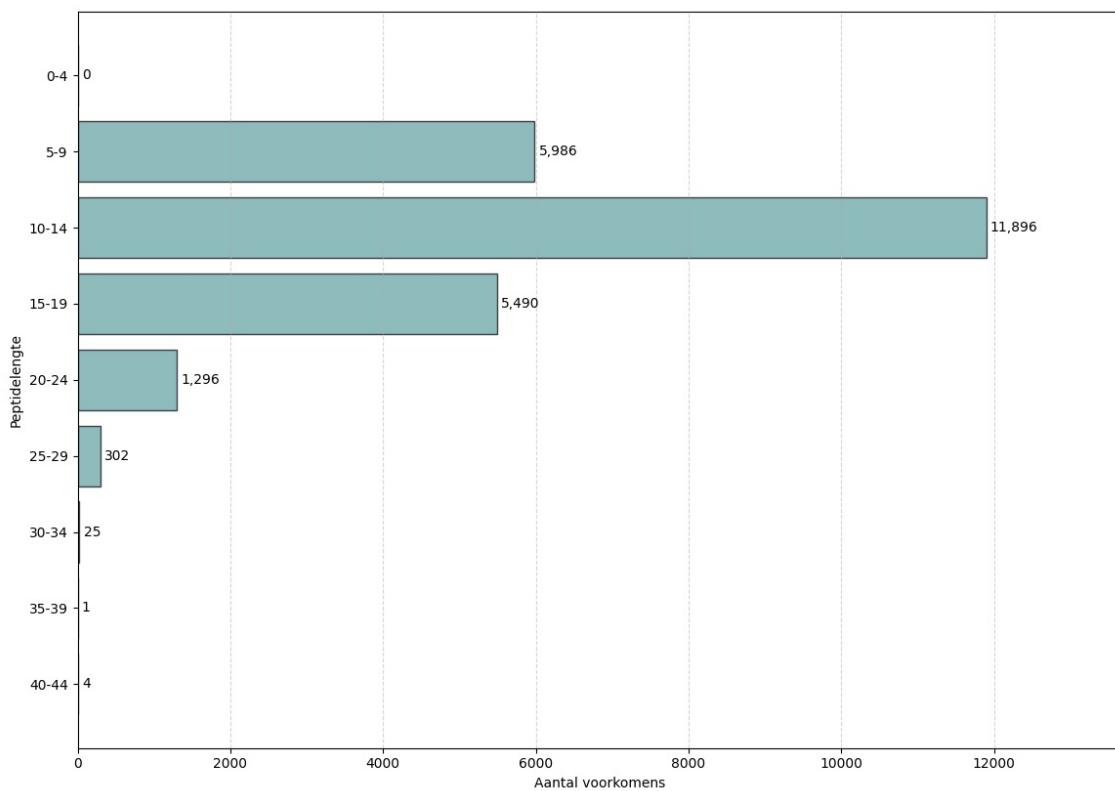
Figuur A.11: Distributie van de aminozuren in het SIHUMI 07 peptidebestand.



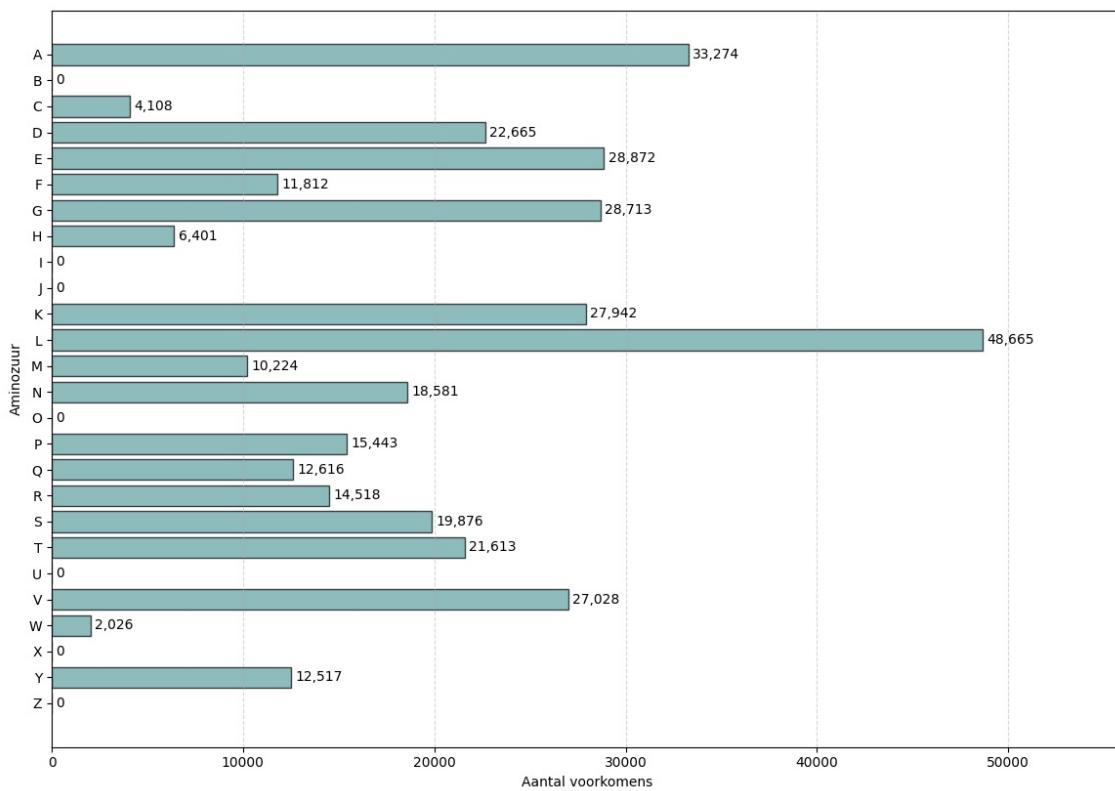
Figuur A.12: Lengtedistributie van de peptiden in het in het SIHUMI 07 peptidebestand.



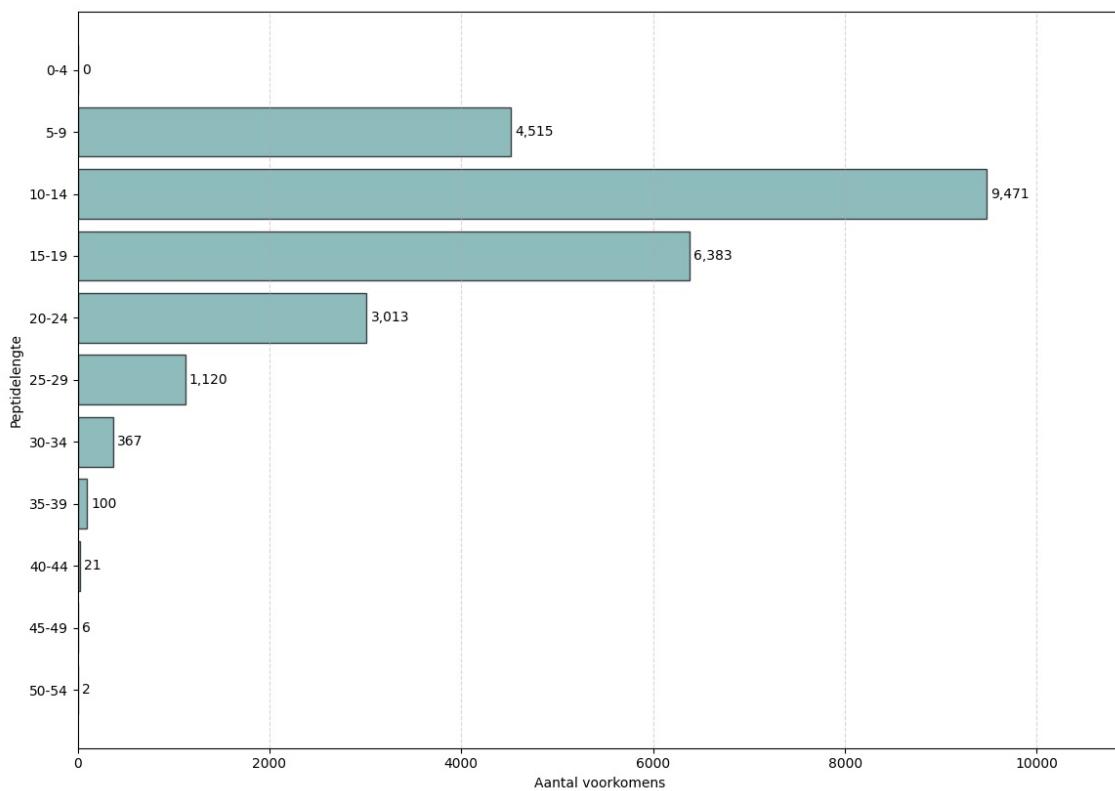
Figuur A.13: Distributie van de aminozuren in het SIHUMI 08 peptidebestand.



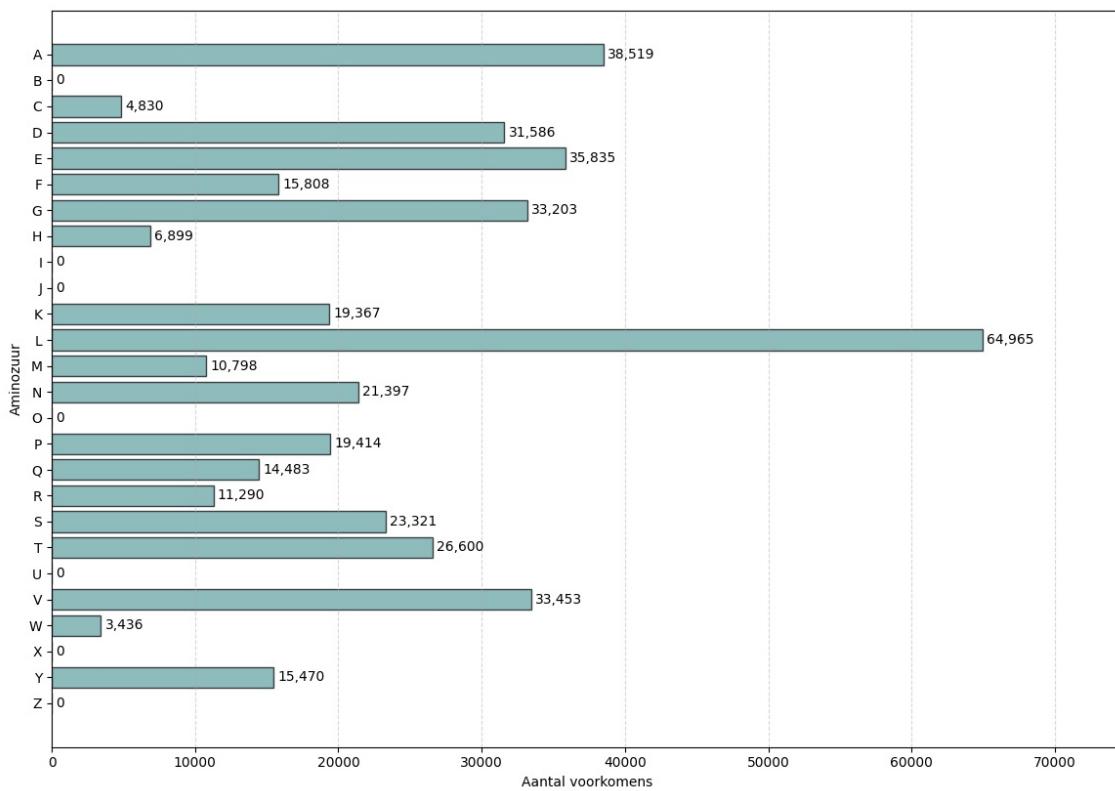
Figuur A.14: Lengtedistributie van de peptiden in het in het SIHUMI 08 peptidebestand.



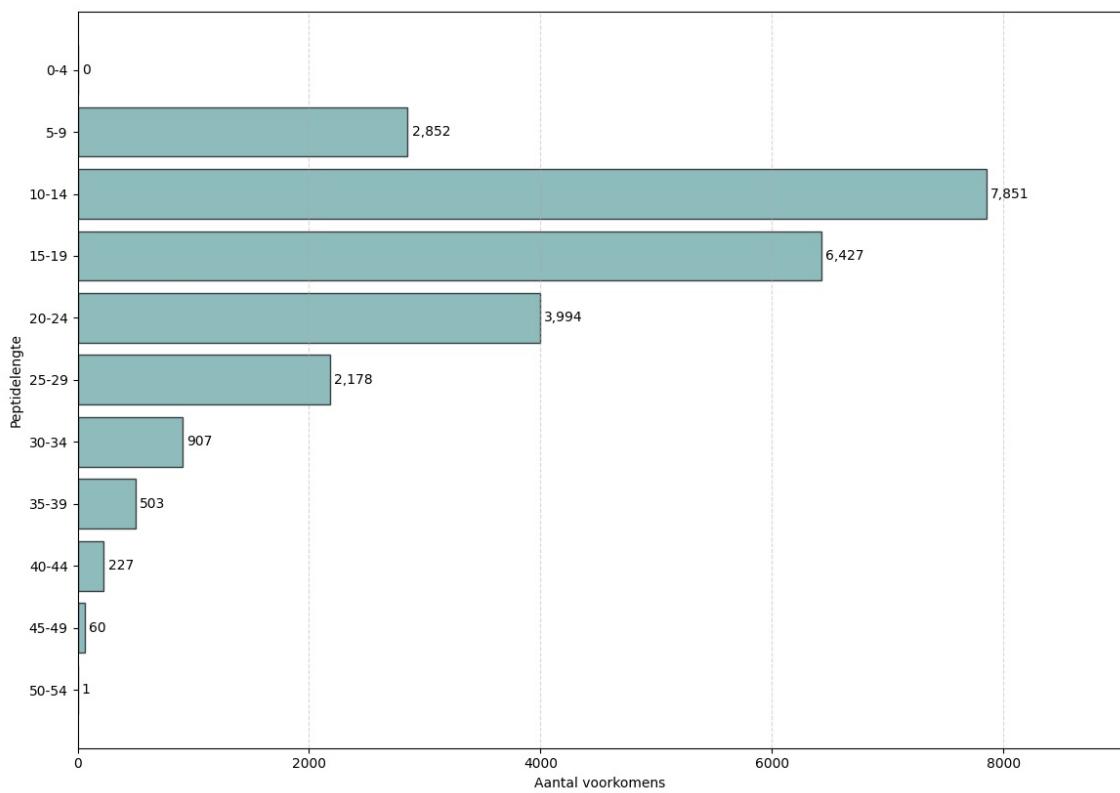
Figuur A.15: Distributie van de aminozuren in het SIHUMI 11 peptidebestand.



Figuur A.16: Lengtedistributie van de peptiden in het in het SIHUMI 11 peptidebestand.



Figuur A.17: Distributie van de aminozuren in het SIHUMI 14 peptidebestand.



Figuur A.18: Lengtedistributie van de peptiden in het in het SIHUMI 14 peptidebestand.

B UNIPEPT PROTEIN COUNTS DISTRIBUTION

# Proteïnen	# Peptiden
≥ 1	1 342 470 764
≥ 2	355 979 324
≥ 10	38 697 210
$\geq 10^2$	2 921 879
$\geq 10^3$	217 922
$\geq 10^4$	13 008
$\geq 10^5$	118
$\geq 10^6$	0

Tabel B.1: Het aantal verschillende tryptische peptiden die matchen met op zijn minst x proteïnen uit UniProtKB 2023_03. Een klein voorbeeld: Stel dat we de tryptische peptide ACACA zoeken. Deze heeft 24 694 matches in UniProtKB. Dit wil zeggen dat dit één van de 13 008 peptiden is met meer dan 10^4 matches.

# NCBI taxonomy rank	# Peptiden
root	12 369
superkingdom	43
kingdom	16
subkingdom	0
superphylum	0
phylum	8
subphylum	7
superclass	1
class	18
subclass	1
superorder	0
order	0
infraorder	1
superfamily	0
family	2
subfamily	0
tribe	1
subtribe	0
genus	55
subgenus	0
species_group	0
species_subgroup	0
species	200
subspecies	0
strain	1
varietas	0
forma	0

Tabel B.2: Overzicht van de resulterende LCA voor de 13 008 verschillende peptiden uit tabel B.1 die met meer dan 10^4 proteïnen matchen. Voor de overgrote meerderheid is de resulterende LCA de root. Slechts voor 200 peptiden is het resultaat op soortniveau.

# Peptiden	LCA
119	<i>Alphainfluenzavirus influenzae</i>
32	<i>Human immunodeficiency virus</i>
14	<i>Hepatitis B virus</i>
9	<i>Betainfluenzavirus influenzae</i>
4	<i>Orthoflavivivirus denguei</i>
3	<i>Simian immunodeficiency virus</i>
1	<i>Alcidodes juglans</i>
1	<i>Bacillus subtilis</i>
1	<i>Bacteroides thetaiotaomicron</i>
1	<i>Cannabis sativa</i>
1	<i>Capsicum baccatum</i>
1	<i>Echinocucumis hispida</i>
1	<i>Geissoloma marginatum</i>
1	<i>Homo sapiens</i>
1	<i>Human immunodeficiency virus</i>
1	<i>Kalanchoe fedtschenkoi</i>
1	<i>Leucosceptrum canum</i>
1	<i>Loxia curvirostra</i>
1	<i>Marinilactibacillus piezotolerans</i>
1	<i>Melanocenchrus jacquemontii</i>
1	<i>Merops nubicus</i>
1	<i>Morbillivirus hominis</i>
1	<i>Phalaenopsis pulcherrima</i>
1	<i>Phormidesmis priestleyi</i>
1	<i>Rhodobacter maris</i>

Tabel B.3: Overzicht van de geassocieerde soort voor de 200 peptiden uit tabel B.2 die op soortniveau eindigen. Van de 200 peptiden eindigt de meerderheid in een LCA die geassocieerd is met een virus (zoals HIV of influenza). Dit komt doordat er veel onderzoek gedaan wordt naar de verschillende bestaande rassen. Deze zitten allemaal in de UniProt Knowledgebase.