

# **Van metagenoom naar metaproteoom**

**Tom Naessens**

Promotor: prof. dr. Peter Dawyndt

Begeleider: Bart Mesuere

Masterproef ingediend tot het behalen van de academische graad van  
Master of Science in de ingenieurswetenschappen: computerwetenschappen

Vakgroep Toegepaste Wiskunde, Informatica en Statistiek

Voorzitter: prof. dr. Willy Govaerts

Faculteit Ingenieurswetenschappen en Architectuur

Academiejaar 2014-2015





# **Van metagenoom naar metaproteoom**

**Tom Naessens**

Promotor: prof. dr. Peter Dawyndt

Begeleider: Bart Mesuere

Masterproef ingediend tot het behalen van de academische graad van  
Master of Science in de ingenieurswetenschappen: computerwetenschappen

Vakgroep Toegepaste Wiskunde, Informatica en Statistiek

Voorzitter: prof. dr. Willy Govaerts

Faculteit Ingenieurswetenschappen en Architectuur

Academiejaar 2014-2015



# Voorwoord

Met het schrijven van dit voorwoord leg ik de laatste hand aan mijn scriptie. Ik wil graag stil staan bij de mensen die mij de afgelopen periode hebben geholpen.

In eerste plaats wil ik mijn promotor, professor Dawyndt, en begeleider, Bart Mesuere, bedanken voor alle steun en begeleiding. Jullie stonden altijd klaar om extra uitleg te geven en vragen te beantwoorden.

Daarnaast wil ik Felix en Stijn bedanken omdat ze met ijzeren hand slechte programmeerprincipes bijschaafden en een frisse blik wierpen op de problemen waar ik mee worstelde.

Ook wil ik graag Jessica en mijn vriendin, Amber, bedanken voor het doorploeteren van mijn belgicismen en het rechttrekken van kromme zinnen.

En als laatste natuurlijk mijn ouders, Jan en Brigitte, en (nogmaals) Amber, voor alle steun gedurende mijn hele opleiding. Bedankt!

Tom Naessens, mei 2015

# 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 beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef.”

Tom Naessens, mei 2015

# VAN METAGENOOM NAAR METAPROTEOOM

door

TOM NAESENS

Scriptie ingediend tot het behalen van de academische graad van  
Master of Science in de ingenieurswetenschappen: computerwetenschappen

Academiejaar 2014–2015

Promotor: prof. dr. P. DAWYNDT  
Begeleider: ir. B. MESUERE  
Faculteit Wetenschappen  
Universiteit Gent

Vakgroep Toegepaste Wiskunde, Informatica en Statistiek  
Voorzitter: prof. dr. W. GOVAERTS

## Samenvatting

Het analyseren en vergelijken van genomen wordt vandaag meestal met BLAST gedaan. Wij stellen de UniPept Metagenomics Analysis Pipeline voor, een nieuwe techniek gebaseerd op het omzetten van metagenomics naar metaproteomics en terug. Die pipeline maakt hiervoor gebruik van de UniPept Metaproteomics Analysis Pipeline. Hierbij voeren we ook een uitbreiding in op het lowest common ancestor algoritme om taxa uit hetzelfde genoom te aggregeren. Ten slotte werken we ook een proof of concept uit van een modulair en abstract visualisatieframework.

## Trefwoorden

metagenoom, metaproteoom, uniPept, benchmarking

# From Metagenome to Metaproteome

Tom Naessens

Supervisor(s): prof. dr. Peter Dawyndt, ir. Bart Mesuere

**Abstract**— Comparing and analysing genomes is usually done using BLAST [1]. We propose the UniPept Metagenomics Analysis Pipeline, a novel technique based on the conversion from metagenomics to metaproteomics and back, which makes use of the UniPept Metaproteomics Analysis Pipeline [5, 3]. To achieve this, a more specific version lowest common ancestor (LCA) method is introduced for aggregating taxa. We also work out a proof of concept to modularise and extract the current visualisations in UniPept into a standalone UniPept visualisation framework, enabling users to inspect their data without uploading their data to the UniPept web application.

**Keywords**— metagenomics, metaproteomics, uniPept, benchmarking

## I. INTRODUCTION

The UniPept platform is developed to map diversity in large and complex metaproteomic samples. The indexstructure of the underlying database has been finetuned to allow quick retrieval of all proteins containing a certain tryptic peptide. The taxon-specificity of a tryptic peptide can be determined based on the taxonomy information from UniProt[3].

Taxon-specificity of the tryptic peptide is successively derived from these occurrences using a novel lowest common ancestor approach that is robust against taxonomic misarrangements, misidentifications, and inaccuracies.

We introduce a new pipeline, the UniPept Metagenomic Analysis Pipeline or UMAP which allows the transformation of a metagenomics experiment into a metaproteomics experiment, using the functionality of UniPept described above. This pipeline is illustrated in Figure 1. For every DNA read in a metagenomics sample, genes are extracted with FragGeneScan[6] and are converted into proteins. These proteins are then split into tryptic peptides. Then, the UniPept Metaproteomics Analysis Pipeline is run to identify the consensus taxon for each tryptic peptide. The resulting taxa are afterwards aggregated using the LCA\* algorithm.

## II. LCA\*

UniPept uses, as mentioned in the introduction, a robust implementation of the lowest common ancestor algorithm which is used for the taxonomic identification of one peptide. However, when the same algorithm is used to aggregate multiple peptides from the same organism, it does not always yield the most specific result.

Therefore, we introduce an adaptation of this algorithm which chooses more specific identifications over less specific ones when the input taxa lie on the same lineage. This new approach is illustrated in Figure 2 on the next page.

P. Dawyndt and B. Mesuere are associated with the Computational Biology research group from the Department of Applied Mathematics, Computer Science and Statistics, Ghent University (UGent), Ghent, Belgium. E-mail: Peter.Dawyndt@UGent.be, Bart.Mesuere@UGent.be.

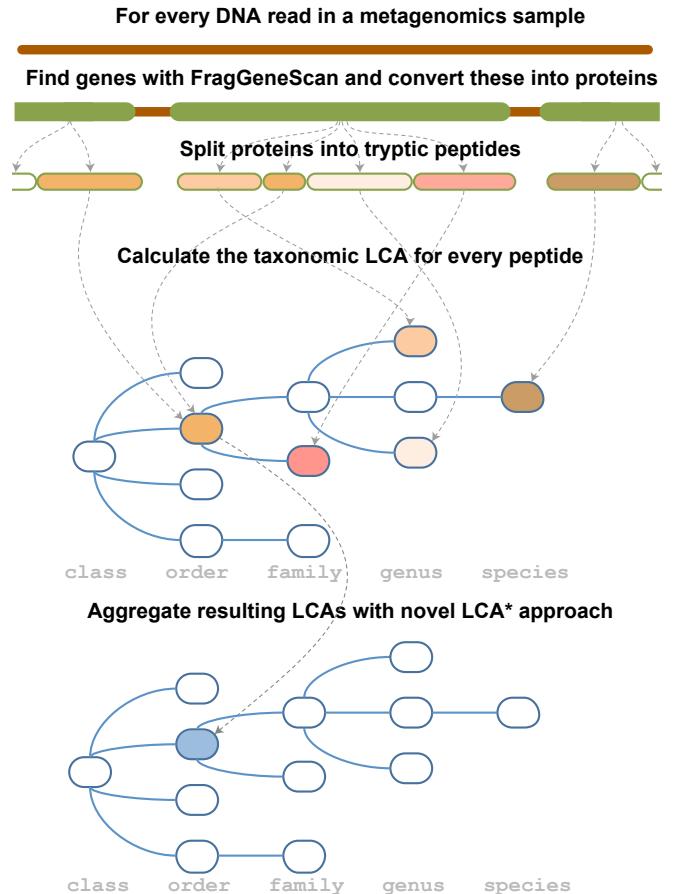


Fig. 1: Illustration of the UniPept Metagenomics Analysis Pipeline.

## III. BENCHMARKING THE UNIPEPT METAGENOMICS ANALYSIS PIPELINE

To test the accuracy of the UMAP, we run two toolchains on three sorts of genomes: *i*) fully sequenced genomes, to be able to measure the accuracy of the taxonomic identification, *ii*) simulated unknown genomes (using a kind of leave-one-out strategy), to measure the accuracy of the taxonomic identification on genomes that have not been sequenced yet, and *iii*) genomes simulated with wgsim[4] based on fully sequenced genomes of read length 250 with different read error percentages of 0, 1, 2 and 5, to measure the effect of read errors on the identification. The two toolchains that were developed can be seen in Figure 3 on the following page. The left toolchain is the UMAP itself, while the right toolchain introduces a filtering step to simulate the unknown genomes.

The UMAP has been found to identify fully sequenced genomes (*i*) very accurately with an average of 97% of the pro-

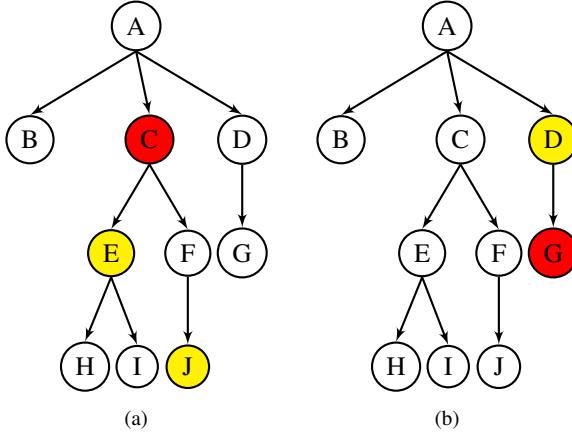


Fig. 2: Examples of the novel LCA\* calculation. On the left, LCA\*(E, J) is calculated, yielding node C as a result. On the right, the LCA of node D and G is calculated, yielding the more specific result G instead of the regular result D.

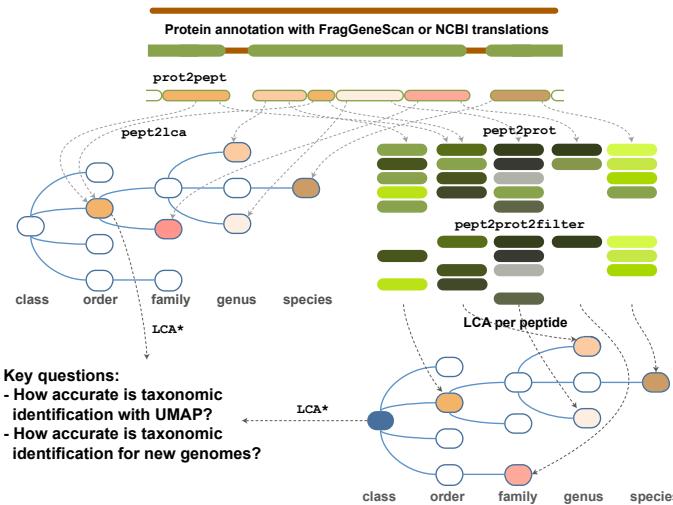


Fig. 3: The two toolchains used in the benchmarking process. The left toolchain is the UMAP itself, while the right toolchain introduces a filtering step to simulate the unknown genomes.

teins mapped to the species level. When the toolchain is being run on simulated unknown genomes (*ii*), a shift in specificity to the lesser accurate levels is observed. 38% of them could still be mapped to the species level. When reads were introduced into the genomes with wgsim (*iii*), we notice less specific results as the error percentage rises. The amount of proteins mapped to the species level drops just below 50% when the read error percentage crosses the level of 2%.

The full results of the benchmarking process were published as a poster, shown on the next page, which was presented during the annual BIG N2N symposium, edition 2015[2].

#### IV. UNIPEPT VISUALISATION FRAMEWORK

We also set the first steps to create a modular and abstract visualisation framework based on the current visualisations in the UniPept web application. This visualisation framework allows users to inspect their results visually and interactively, without

having to enter their data in UniPept. As a proof of concept, the treeview visualisation (as can be seen in Figure 4) was extracted from the UniPept code and put into a separate modular JavaScript plugin.

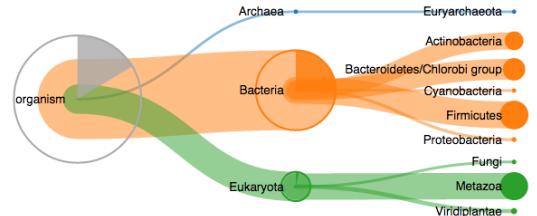


Fig. 4: Example of the treeview visualisation from UniPept

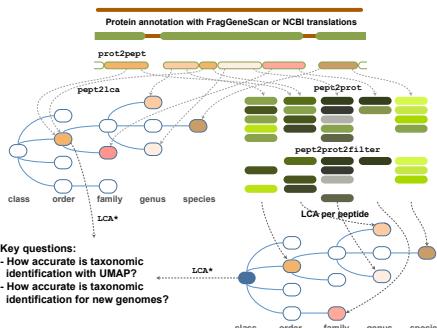
#### REFERENCES

- [1] Stephen F Altschul et al. “Basic local alignment search tool”. In: *Journal of molecular biology* 215.3 (1990), pp. 403–410.
- [2] *BIG N2N annual symposium | From Nucleotides to Networks*. <http://www.bign2n.ugent.be/node/239>. (Visited on 05/22/2015).
- [3] UniProt Consortium et al. “UniProt: a hub for protein information”. In: *Nucleic Acids Research* (2014), gku989.
- [4] H Li. *lh3/wgsim*. <https://github.com/lh3/wgsim>.
- [5] Bart Mesuere et al. “UniPept: tryptic peptide-based biodiversity analysis of metaproteome samples”. In: *Journal of proteome research* 11.12 (2012), pp. 5773–5780.
- [6] Mina Rho, Haixu Tang, and Yuzhen Ye. “FragGeneScan: predicting genes in short and error-prone reads”. In: *Nucleic acids research* 38.20 (2010), e191–e191.

# Benchmarking the UniPept Metagenomics Analysis Pipeline

T. Naessens, B.T. Habtemariam, R. Deklerck, S. Houbraken, M. Niklaus, I. Melckenbeeck  
Promoter: Prof. Dr. Peter Dawyndt

## Abstract



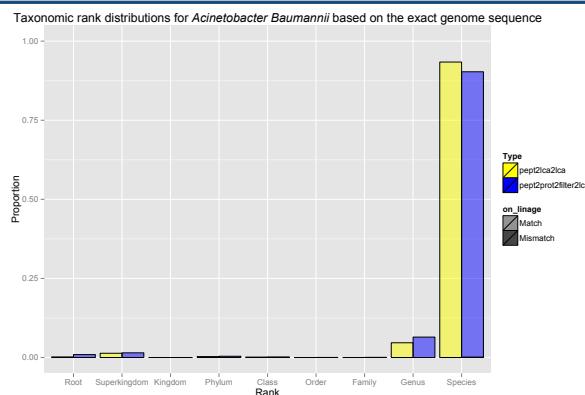
**Context** The UniPept Metagenomics Analysis Pipeline (UMAP) is an approach to solve the problem of taxonomic identification with metaproteomics. This is achieved by predicting all proteins on each DNA strand of a metagenomics sample, running the UniPept metaproteomics pipeline on these proteins, as indicated in the left hand side of the picture on the left, and aggregating them back to one resulting taxon. This last step is done using a novel LCA\* algorithm, which exploits the fact that the proteins all originate from one DNA strand.

**Approach** To benchmark, we run the UMAP on both completely sequenced genomes and simulated reads from those genomes and

compare the results with a separate analysis on those genomes, but where proteins that were found in Uniprot to be originating from that genome, are filtered out. This allows us to simulate what would happen if the UMAP is being run on unknown genomes while still producing comparable results.

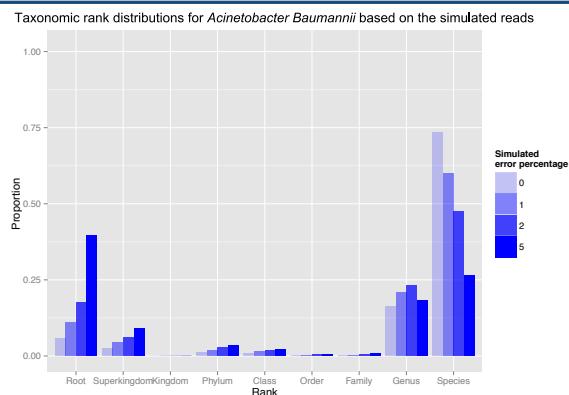
**Results** To summarise obtained results, we have found that the UMAP performs very well for known genomes, where on average 97% is mapped on the species level. For simulated reads where no error was introduced, this number is reduced to 74% and lowers the more error is introduced. On the simulated unknown genomes, 38% is mapped to the species level.

## Benchmarking results for *Acinetobacter baumannii*



The barplot above shows the taxonomic rank distribution of the results of both toolchains for the taxonomic identification of the peptides from the *Acinetobacter Baumannii* organism. The yellow bars show the result with the default UMAP toolchain, where the blue bars correspond with the different approach where the proteins from the

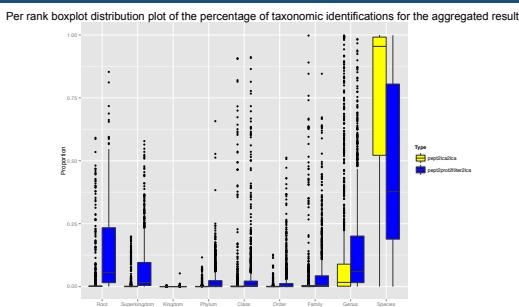
original sequence have been filtered out in the identification process. A specific level of identification is obtained for both toolchains. We also see a shift to the less specific ranks when we filter out the proteins already occurring in the originating genome. This shift is expected as this filter step causes a loss of specific information.



The above barplot shows the distribution of identifications found at the different levels in simulated reads on one genome with read lengths of 500 with 0%, 1%, 2% and 5% error rates. As can be seen in the plot, the identification of the peptides is about 20% less accurate when using reads with 0% than when using the exact genome

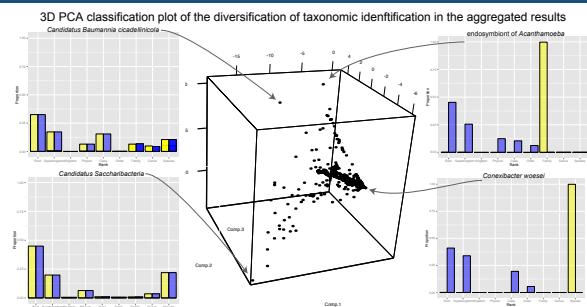
sequence. When using reads without errors, almost 75% of the proteins are mapped to the species rank and 20% to genus. Introducing errors in the reads predictably hampers the identification of the peptides, resulting in a worse specific identification when the error rate increases.

## Total aggregated results for 1145 genomes



Identification of proteins from 1145 genomes with (blue) and without (yellow) filtering out proteins that were found to be originating from the initial genomes. Without filtering, the identifications are on average for 97%

mapped to species rank, with the remaining being mapped at genus rank. Enabling the filter step results in a general shift to the less specific ranks, which is expected as specific information is being filtered out.



Above is the PCA plot of the identification results for 1145 completely sequenced genomes. This plot clusters the genomes by the way their proteins are classified. Most are clustered along one line, but some outliers can be seen. The most extreme outliers have been

accompanied by their corresponding bar plots. The reason for these outliers can vary from wrongly identified or classified genomes in the source database, genomes with very less or very much specific peptides or proteins, etc.

# Inhoudsopgave

<b>1 Inleiding</b>	<b>1</b>
1.1 Genomics en proteomics . . . . .	1
1.2 Metagenomics en metaproteomics . . . . .	2
1.3 Van metagenoom naar metaproteoom . . . . .	3
1.4 Unipept . . . . .	5
1.5 Geschreven code . . . . .	5
<b>2 LCA*</b>	<b>8</b>
2.1 Inleiding . . . . .	8
2.2 Huidige implementatie in Unipept . . . . .	11
2.2.1 Tabelgebaseerde LCA . . . . .	11
2.3 Beperkingen van de tabelmethode . . . . .	12
2.4 Boomgebaseerde LCA(*) . . . . .	13
2.4.1 Implementatie . . . . .	13
2.4.2 Van LCA naar LCA* . . . . .	20
2.5 Toekomstig werk . . . . .	25
<b>3 Case Study: Benchmarking the Unipept Metagenomics Analysis Pipeline</b>	<b>27</b>
3.1 Bepalen van een benchmarkstrategie . . . . .	28
3.2 Implementatie . . . . .	29
3.2.1 Gebruikte tools en bibliotheken . . . . .	29
3.2.2 Implementatie van de benchmark . . . . .	31
3.2.3 Snelheidsoptimalisatie . . . . .	34
3.3 Resultaten . . . . .	36
3.4 Toekomstig werk . . . . .	39
<b>4 Uitbreidingen van de Unipept command-line interface</b>	<b>41</b>
4.1 Inleiding tot de Unipept CLI . . . . .	41
4.2 Verwerken van FASTA files . . . . .	43
4.3 Bepalen van de optimale batch size voor de Unipept CLI . . . . .	48
4.4 Memory leak . . . . .	49
4.5 Retry-strategie bij fouten . . . . .	52

<b>5 Visualisatieframework voor Unipept</b>	<b>55</b>
5.1 Overzicht van visualisaties in Unipept . . . . .	55
5.2 Modularisatie en abstractie . . . . .	57
5.3 Implementatie . . . . .	59
5.3.1 Modularisatie . . . . .	59
5.3.2 Abstractie . . . . .	65
5.4 Case studies . . . . .	67
5.4.1 Modularisatie: vergelijken van resultaten van toolchains . . . . .	67
5.4.2 Abstractie: Overschrijven van standaardmethodes . . . . .	70
5.4.3 Niet-biologische invoerdata . . . . .	72
5.5 Optimalisaties aan de bestaande treeview-visualisatie . . . . .	72
5.6 Benchmarking . . . . .	75
5.7 Conclusie . . . . .	76
5.8 Toekomstig werk . . . . .	76

# Hoofdstuk 1

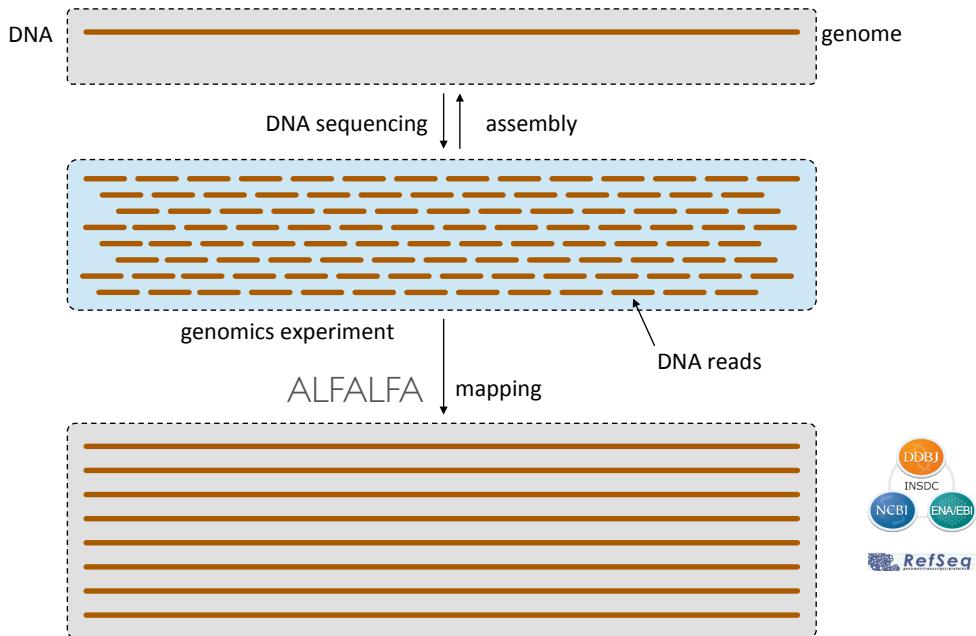
## Inleiding

In deze thesis, die kadert binnen het UniPept-project, wordt de mogelijkheid onderzocht om een metagenomics probleem om te zetten in een metaproteomics probleem. Voor we iets meer vertellen over de structuur en de opbouw van deze thesis is het belangrijk de titel van de thesis te verduidelijken. Metagenomics en metaproteomics zijn namelijk twee termen uit de biologie die wel wat uitleg vergen. In deze inleiding leggen we uit wat beide termen inhouden, bekijken we de onderliggende concepten, lichten we toe wat UniPept precies is en welke rol UniPept zal spelen in dit onderzoek.

### 1.1 Genomics en proteomics

Genomics is de studie waarbij genomen worden onderzocht. Een genoom is het geheel van de genetische samenstelling van het materiaal in een organisme en wordt uitgedrukt als een DNA-sequentie, bestaande uit nucleotiden. Er kunnen verschillende analyses uitgevoerd worden op een genoom, maar voor alle experimenten zal het genoom eerst moeten worden gesequenereerd om de ordering van de DNA-baseparen te bepalen. Dat proces heeft een heel aantal korte DNA-fragmenten (DNA-reads genaamd) als resultaat, zoals in Figuur 1.1 op de volgende pagina wordt geïllustreerd. Die verzameling van korte reads dient als basis voor een aantal processen: *i*) assembly, waarbij de korte reads allemaal samen tot één consensusgenoom worden samengevoegd en *ii*) mapping, waarbij de reads op andere genomen worden gemapt om zo de afstand (verwantschap) van het genoom in kwestie tot andere genomen te bepalen. Bij die laatste stap wordt gebruikt gemaakt van genoomdatabanken waarin alle gekende genomen zijn opgeslagen. Voorbeelden van genoomdatabanken zijn INSDC en RefSeq. Een recente read mapper is bijvoorbeeld ALFALFA[24].

Verwant aan het genoom is het proteoom. De studie van proteomen wordt proteomics genoemd. Een proteoom bestaat uit de volledige set aan eiwitten die geëncodeerd in door een genoom, een organisme of een deel ervan. Net als een genoom kan een proteoom ook worden gesequenereerd in protein reads, peptiden genaamd. Die protein reads kunnen dan net zoals DNA reads gemapt worden aan de hand van tools, zoals UniPept[13, 12], worden gemapt op eiwitdatabanken, zoals geïllustreerd in Figuur 1.2 op pagina 3. UniProt[6] is een



Figuur 1.1: Illustratie van een genomics experiment. Een genoom, voorgesteld door een bruine lijn, wordt gesequenereerd in verschillende DNA reads. Deze reads kunnen dan opnieuw worden geassembleerd naar een volledig consensusgenoom of kunnen worden gemapt op meerdere genomen aan de hand van DNA read mappers en genoomdatabanken.

voorbeeld van zo'n soort eiwitdatabank.

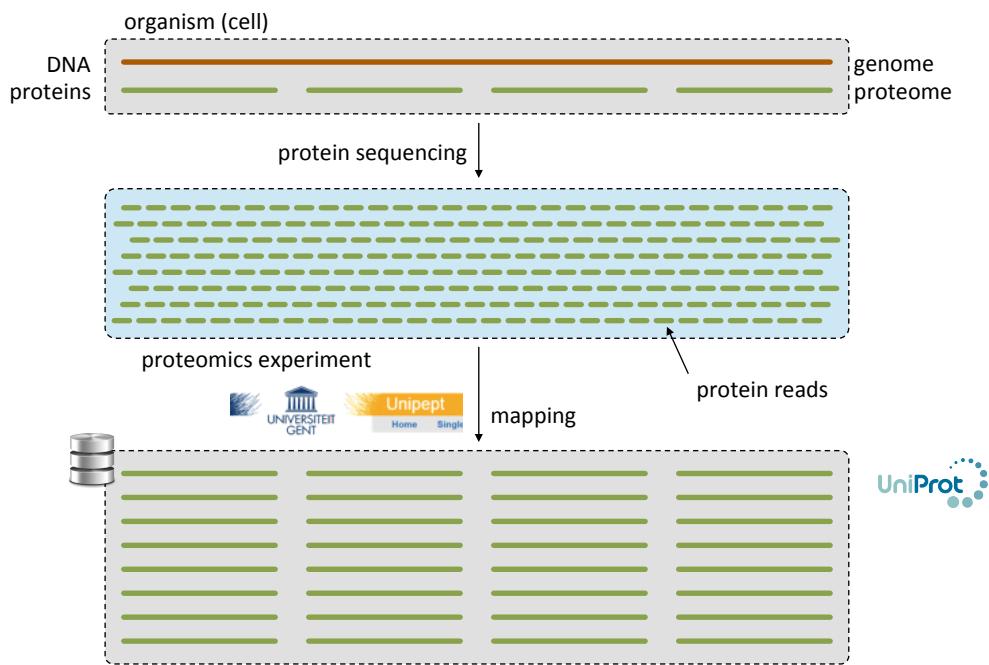
## 1.2 Metagenomics en metaproteomics

Wanneer stalen worden genomen uit de natuur, bijvoorbeeld bij vijveronderzoek, of in de medische wereld, bijvoorbeeld bij de analyse van iemands stoelgang, krijgen onderzoekers te maken met gemeenschappen in plaats van geïsoleerde specimen. Ze hebben met andere woorden te maken met stalen van meerdere genomen of metagenomen en meerdere proteomen of metaproteomen. Daarbij komt ook nog eens dat slechts één procent van de organismen die in de natuur wordt gevonden, in een lab kan worden gekweekt. Het is dus belangrijk om rechtstreeks met stalen uit de natuur te kunnen werken en voor een vlotte analyse van die stalen is het noodzakelijk dat er technieken bestaan die meerdere genomen en proteomen tegelijkertijd kunnen verwerken en analyseren. Wanneer we het hebben over het analyseren van metagenomen en metaproteomen bevinden we ons respectievelijk in het onderzoeksgebied van de metagenomics en de metaproteomics.

Bij dit soort onderzoek zijn er drie grote vragen: *i)* Welke organismen bevinden zich in een staal? *ii)* Wat doen ze precies? en *iii)* Hoe doen ze het? In deze thesis houden we ons vooral bezig met de eerste vraag.

Onderzoek in metagenomics gebeurt typisch op twee manieren. De eerste manier is *targeted metagenomics* waarbij een specifiek stuk, meestal een deel van het 16S ribosomaal

### 1.3. VAN METAGENOOM NAAR METAPROTEOOM HOOFDSTUK 1. INLEIDING



Figuur 1.2: Illustratie van een proteomics experiment. Eiwitten, geannoteerd op een genoom, worden gesequeneerd in protein reads. Die reads kunnen dan worden gemapt op overeenkomstige proteomen uit eiwitdatabanken.

RNA, uit de verschillende organismen in het staal worden gesequeneerd. Een tweede manier is *shotgun metagenomics* waarbij de volledige DNA strands uit het metagenoom worden gesequeneerd. Opnieuw aan de hand van verschillende genoomdatabanken kan de mapping van die reads naar gesequeneerde genomen gebeuren. Beide aanpakken zijn geïllustreerd in Figuur 1.3 op de volgende pagina.

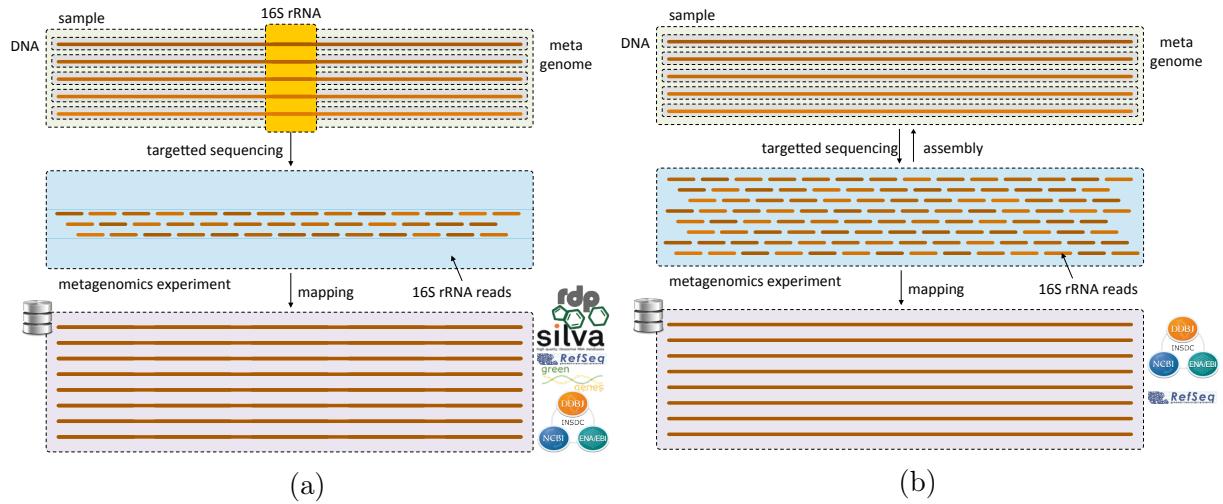
Het proces van metagenomics kan, analoog aan de processen van genomics en proteomics, toegepast worden op metaproteomics. Dat wordt geïllustreerd in Figuur 1.4 op de pagina hierna. Hierbij worden alle eiwitten op de DNA strands uit het sample geannoteerd en gesequeneerd in protein reads of peptiden. Die peptiden kunnen vervolgens opnieuw, bijvoorbeeld door Unipept, worden gemapt op eiwitdatabanken.

## 1.3 Van metagenoom naar metaproteoom

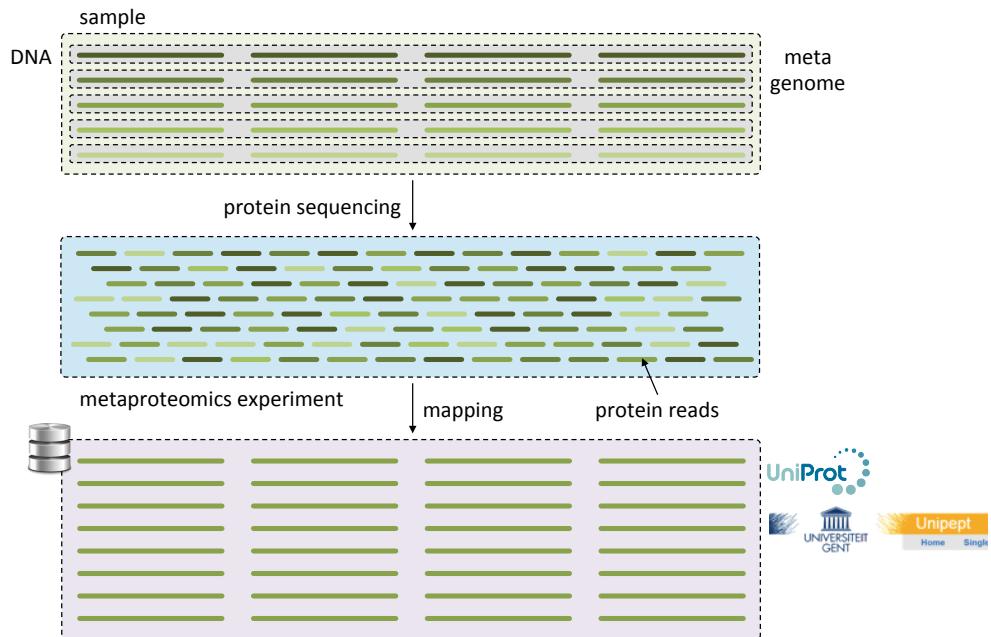
De klassieke oplossing voor de mapping van een reeks DNA reads op genomen in een genoomdatabank is het gebruik van BLAST, of verwante tools. Die tools zijn vaak zeer traag en vereisen zeer veel rekencapaciteit. Aangezien Unipept al in staat is om zeer snel metaproteomen te analyseren zou het erg voordelig zijn om dat metagenomics probleem om te zetten naar een metaproteomics probleem. Dat is dan ook wat we proberen te doen. Het grootste voordeel van die omzetting is dat we voor de meeste stappen integraal gebruik kunnen maken van de Unipept Metaproteomics Analyse Pipeline. Die pipeline is al

### 1.3. VAN METAGENOOM NAAR METAPROTEOON HOOFDSTUK 1. INLEIDING

---



Figuur 1.3: Illustratie van een metagenomics experiment. Links de 16S rRNA variant, rechts de shotgun variant. Bij een metagenomics experiment wordt (een gedeelte van) de DNA strands uit een metagenoom gesequenereerd en daarna op genoomdatabanken gemapt.



Figuur 1.4: Illustratie van een metaproteomics experiment. Alle eiwitten, geannoteerd op de verschillende DNA strands, worden gesequenereerd in protein reads, peptiden genaamd. Die peptiden kunnen dan worden gemapt op overeenkomstige proteomen aan de hand van eiwitdatabanken.

geïmplementeerd en geoptimaliseerd om alles in zo weinig mogelijk tijd te berekenen. Wat dan nog rest is de conversie van en naar metagenomics.

De omzetting zelf gebeurt volgens de werkwijze aangegeven in Figuur 1.5 op pagina 7. Voor elke DNA read in een metagenomics dataset kunnen we een proteomics experiment doen. Hierbij zoeken we, bijvoorbeeld aan de hand van FragGeneScan, alle (fragmenten van) eiwitten in de DNA read in kwestie. De bekomen eiwitten worden opgedeeld in tryptische peptiden, waarna elke peptide afzonderlijk op zijn taxonomische identificatie wordt gemapped door middel van het lowest common ancestor (LCA) algoritme. Aan de hand van een uitbreiding van dit lowest common ancestor algoritme kunnen we de bekomen taxa aggregeren naar een consensusclassificatie. Dat proces zullen we vanaf nu aanduiden met de UniPept Metagenomics Analysis Pipeline, of ook wel UMAP.

In Hoofdstuk 2 op pagina 8 bespreken we de motivatie en implementatie van het nieuwe LCA\* algoritme gebruikt in de laatste stap, waarna we in Hoofdstuk 3 op pagina 27 de performantie benchmarken van de nieuwe UniPept Metagenomics Analysis Pipeline.

## 1.4 UniPept

In dit hoofdstuk is de naam UniPept een aantal keer gevallen. Aangezien deze thesis kadert binnen het UniPept-project wordt in deze sectie een korte beschrijving van UniPept gegeven en wordt uitgelegd waar UniPept zich momenteel bevindt in de bovenstaande processen.

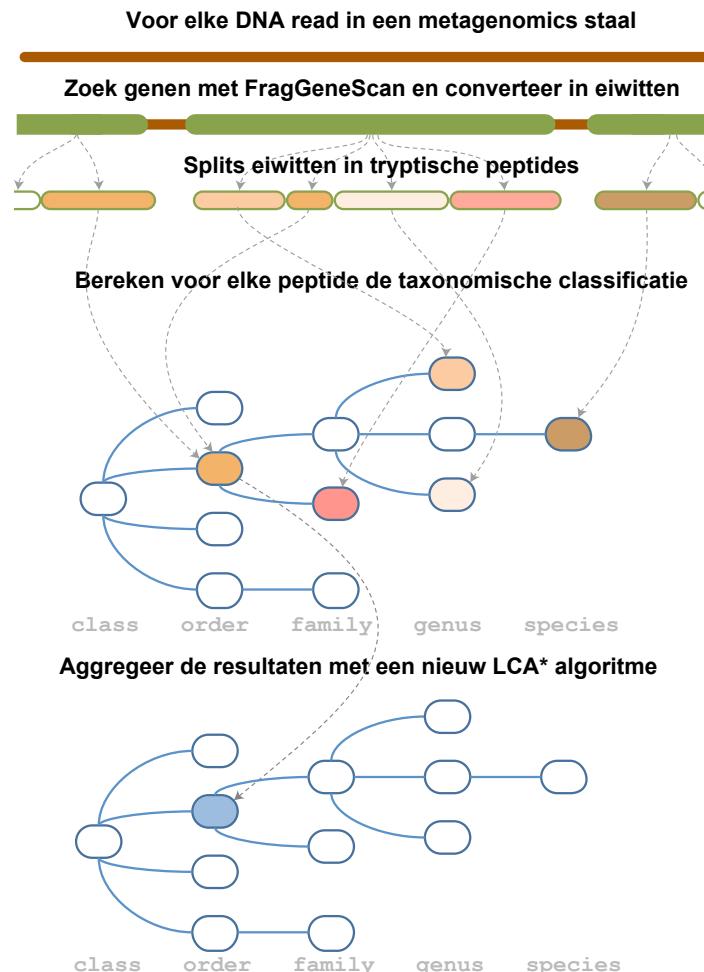
Kort samengevat is UniPept een framework met een gebruiksvriendelijke webinterface om de biodiversiteit van grote en complexe metaproteomics samples te onderzoeken aan de hand van informatie uit tryptische peptiden. Achter de webinterface bevindt zich een indexstructuur die ervoor gemaakt is om snel alle eiwitten uit de UniProtKB op te vragen waar de tryptische peptide in voorkomt. Door het gebruik van een speciale lowest common ancestor aanpak kan ook de taxonomische identificatie aan een peptide gekoppeld worden die bestand is tegen verkeerde identificaties, classificaties en onjuistheden in de taxonomieboom. Die achterliggende indexstructuur is naast de webinterface ook aanspreekbaar via de UniPept command-line interface (CLI) die een interface aanbiedt voor de UniPept web services. De CLI laat de gebruiker toe grote datasets te verwerken zonder dat via de browser te moeten doen.

Hoofdstuk 4 op pagina 41 geeft na een inleiding tot de UniPept command-line interface een reeks van problemen met bijhorende oplossingen die naar boven zijn gekomen tijdens het intensief gebruik van de tools. In Hoofdstuk 5 op pagina 55 geven we een overzicht van de al bestaande visualisaties in UniPept. Daarnaast motiveren we de abstractie en modularisatie van die visualisaties in een apart UniPept visualisation framework.

## 1.5 Geschreven code

Alle code geschreven binnen de context van deze thesis is verzameld op de UGent GitHub onder de UniPept organisatie en voornamelijk onder de repositories `unipept-visualizations`

en `unipept-metagenomics-scripts`. Toegang tot deze repositories kan gegeven worden door de promotor of begeleider van deze thesis.



Figuur 1.5: Illustratie van de werkwijze om een metagenomics probleem om te zetten naar een metaproteomics probleem. Voor elke DNA read in het metagenomics sample kunnen we een proteomics experiment doen. Hierbij zoeken we, bijvoorbeeld aan de hand van FragGeneScan, alle eiwitten op de DNA read in kwestie. De bekomen eiwitten worden opgedeeld in tryptische peptiden waarna elke peptide afzonderlijk op zijn taxonomische identificatie wordt gemapped door middel van het lowest common ancestor (LCA) algoritme. Aan de hand van een uitbreiding van dit lowest common ancestor algoritme kunnen we de bekomen taxa aggregeren naar een consensusclassificatie.

# Hoofdstuk 2

## LCA\*

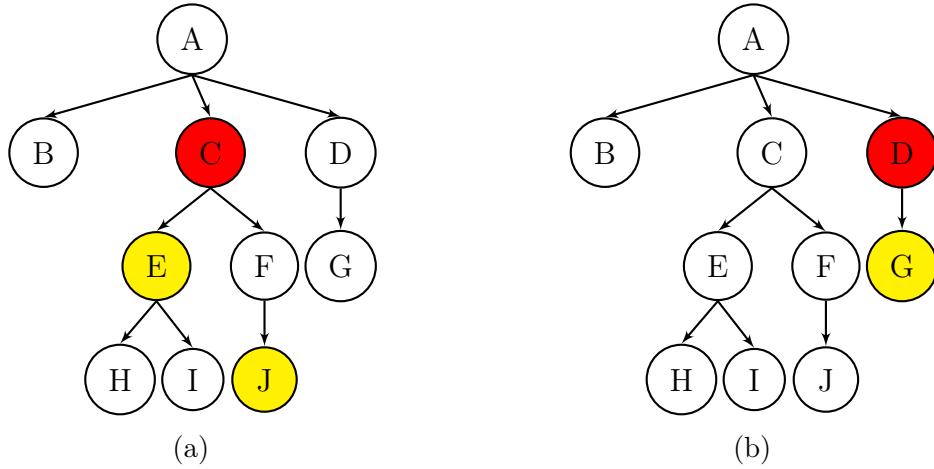
In dit hoofdstuk leggen we eerst het algemene concept van de lowest common ancestor (LCA) uit. We bekijken de verschillende toepassingen van dit concept in de biologie en lichten toe hoe dit LCA algoritme wordt gebruikt in UniPept. Na deze inleiding stellen we een variant van dat algoritme voor, LCA\*, en leggen we uit waarom LCA\* in sommige gevallen een specifieker resultaat oplevert dan algemene LCA. Hierna diepen we een zeer snelle implementatie van het originele LCA algoritme uit, tonen we aan hoe we dit algoritme kunnen uitbreiden tot een algoritme voor LCA\* en lossen we enkele problemen op die die uitbreiding met zich meebrengt. We sluiten af met een voorbeeld van hoe LCA\* zou kunnen worden ingebouwd in de UniPept webservices.

### 2.1 Inleiding

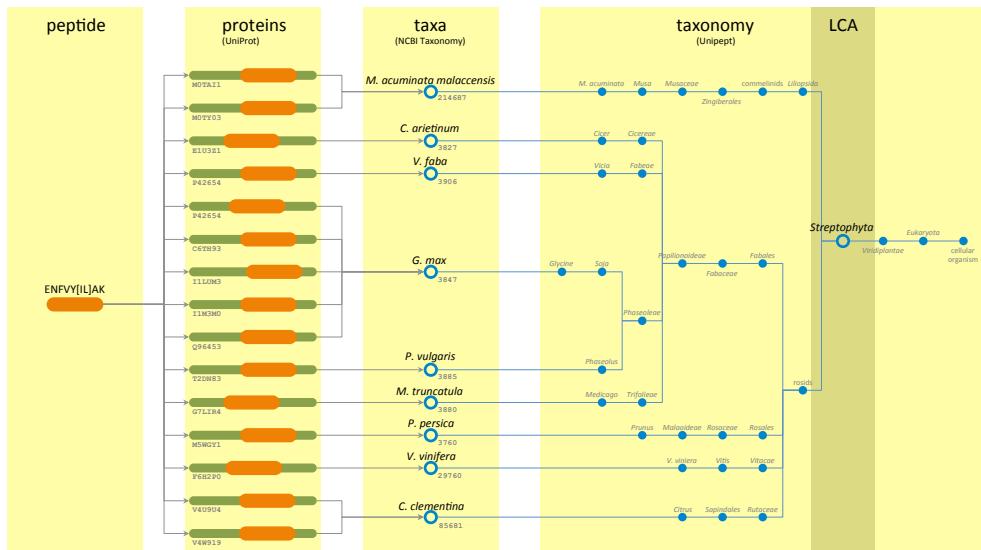
Het lowest common ancestor (LCA) probleem is een klassiek probleem in de grafentheorie. De LCA in een boomstructuur wordt gedefinieerd als de gemeenschappelijke ouder van twee (of meer) nodes die zich het verste van de wortel bevindt. Toegepast op het voorbeeld in Figuur 2.1 op de pagina hierna, kunnen we bijvoorbeeld de LCA berekenen van node E en J. Nodes waarvan we de LCA berekenen, zullen we in het vervolg “querynodes” noemen. Volgens de definitie hierboven is de LCA van E en J gelijk aan C. De notatie die we hier in het vervolg voor zullen gebruiken is:  $\text{LCA}(E, J) = C$ . Wanneer we de LCA berekenen van twee nodes op hetzelfde pad, resulteert dit in de node dichtst bij de wortel, zoals te zien is op Figuur 2.1b op de volgende pagina.

Binnen de (meta)proteomics wordt de berekening van de LCA ook vaak gebruikt. Dit wordt geïllustreerd in Figuur 2.2 op de pagina hierna waar in de laatste stap de LCA genomen wordt van alle gevonden taxa in de UniPept taxonomy.

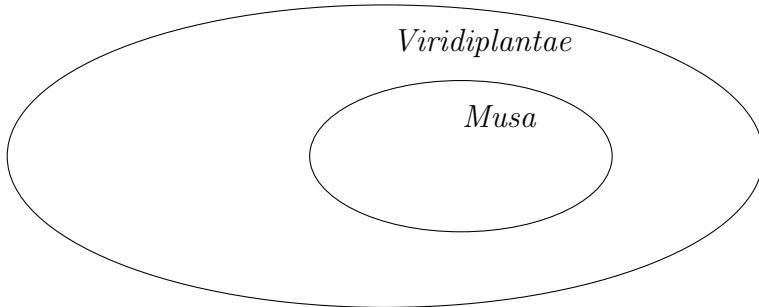
De metagenomics toolchain, geïllustreerd in Figuur 1.5 op pagina 7, voert op het resultaat van de LCA nogmaals een LCA stap uit waarbij alle gevonden taxa geaggregeerd worden tot één consensustaxon. Het toepassen van het LCA algoritme zoals hierboven beschreven levert echter niet altijd het gewenste resultaat op. Onderstaand voorbeeld illustreert waarom.



Figuur 2.1: Bovenstaande figuren worden twee voorbeelden getoond van het algemene LCA concept. Links nemen we de LCA van node E en J, met C als resultaat. Rechts berekenen we de LCA van twee nodes op hetzelfde pad, D en G. Dit geeft de node dichtst bij de wortel als resultaat, namelijk D. Nodes in het geel zijn de nodes waarvan de LCA wordt berekend, de rode nodes zijn het resultaat van de LCA bewerking.



Figuur 2.2: Illustratie van het gebruik van lowest common ancestor binnen de (meta)proteomics. Het lowest common ancestor algoritme wordt hier gebruikt om in de laatste stap de gevonden taxa in de UniPept taxonomie te aggregeren tot één consensus-taxon.



Figuur 2.3: Illustratie van de verzameling van de *Viridiplantae* en de *Musa*

Stel dat we in een tropisch oerwoud een onbekende plant tegenkomen en willen weten welke (soort) plant dit is, dan kunnen we een experiment doen. We nemen van deze plant een sample en verwerken het via de technieken beschreven in de inleiding, zodat we uiteindelijk een reeks van peptiden bekomen. Voor elke peptide uit deze reeks kunnen we dan alle taxa opzoeken waar de peptiden in voorkomen. Stel nu dat dit resulteert in volgende resultaat: 95% van de taxa behoort tot het rijk (*kingdom*) van de groene planten, de *Viridiplantae* en voor 5% behoort tot het geslacht (*genus*) van de bananenplanten, de *Musa*. We kunnen dan beide groepen als verzamelingen voorstellen: de planten = {baardgras, distel, banaan, maanvaren, ...}. De verzameling van de bananenplanten wordt dan: {*callimusa*, *ingentimusa*, ...}. De bananenplanten zijn dus een deelverzameling van alle planten. Een visuele representatie van dit voorbeeld is te vinden in Figuur 2.3.

Wanneer beide verzamelingen op een boomstructuur worden weergegeven, zou dit opnieuw voorgesteld kunnen worden zoals op fig. 2.1b, waarbij D de klasse van alle planten voorstelt en G die van de bananen.

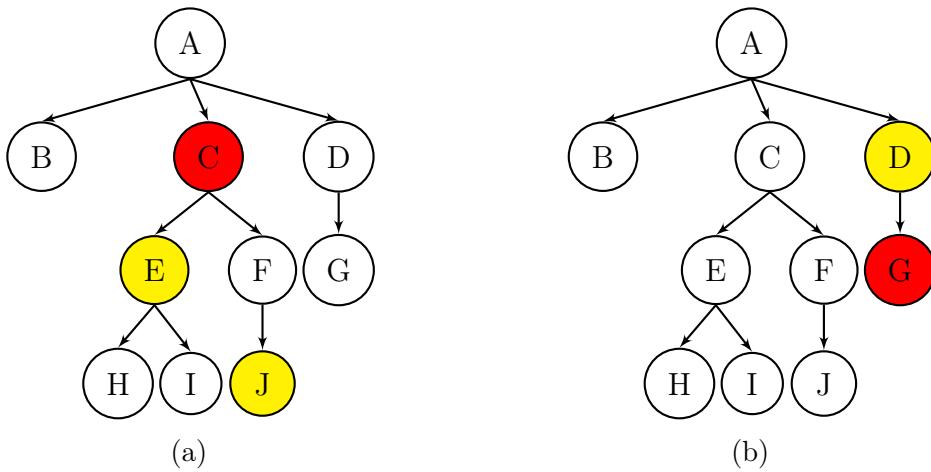
Als het LCA algoritme wordt uitgevoerd op deze twee nodes zouden we als resultaat op node D, de planten, uitkomen. Ook al is dit correct, is er toch een specifieker identificatie mogelijk. We weten namelijk dat alle peptiden uit de sample die we in oerwoud hebben genomen uit één enkele plant afkomstig zijn.

Tijdens ons onderzoek hebben we gevonden dat er 5% van de peptiden enkel en alleen voorkomen bij bananenplanten. Het zou zonde zijn dit bewijs zomaar te negeren en te besluiten dat de onderzochte plant tot het rijk (*kingdom*) van de planten behoort, terwijl we ook specieker informatie kunnen geven; namelijk dat de onderzochte plant tot de klasse van de bananenplanten behoort. Aangezien er peptiden van de bananenplant in het sample voorkomen, kunnen we daar namelijk uit afleiden dat we specifiek met een bananenplant te maken hebben, en niet algemeen met een plant.

Als we terugkeren naar onze verzamelingen in Figuur 2.3, willen we nu niet de unie nemen van beide verzamelingen. Dat zou immers de minder specifieke klasse van de planten opleveren. Wat we willen, is de specieker identificatie van de plantensoort (namelijk de bananenplant) en dat bekomen we door de doorsnede van beide verzamelingen te nemen.

Wanneer de doorsnede van beide verzamelingen leeg is, is er geen specieker identificatie mogelijk en nemen we alsnog de unie, zoals in fig. 2.4a wordt geïllustreerd.

Deze variant van de LCA zullen we om verwarring te vermijden, aanduiden met LCA\*.



Figuur 2.4: Voorbeelden van het LCA\* algoritme. Links wordt  $\text{LCA}^*(E, J) = C$  uitgewerkt, rechts  $\text{LCA}^*(D, G) = G$ . Nodes in het geel zijn de nodes waarvan de LCA wordt berekend, de rode nodes zijn de resulterende LCA\*.

## 2.2 Huidige implementatie in UniPept

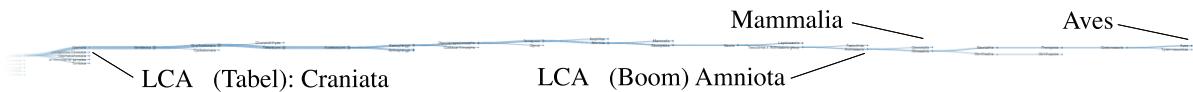
Zoals geillustreerd wordt in Figuur 2.2 op pagina 9 worden uit de UniProtKB alle eiwitten opgezocht waar een gegeven peptide in gevonden wordt. Voor die lijst van eiwitten wordt hun overeenkomstig taxon uit de NCBI taxonomie opgezocht. Die taxa worden dan afgebeeld op de UniPept taxonomie, die een opgekruiste versie is van de NCBI taxonomie. Onder andere uncultured, unspecified, mixed, ... culturen zijn weggefiterd in de UniPept taxonomie. Nu wordt de resulterende lijst van taxa in de UniPept taxonomie geaggregeerd tot één consensustaxon voor het originele peptide via de LCA methode. In deze sectie bespreken we de huidige implementatie van deze methode en welke beperkingen die met zich meebrengt.

### 2.2.1 Tabelgebaseerde LCA

De implementatie van de LCA in UniPept is het eenvoudigst te begrijpen aan de hand van een voorbeeld. In Tabel 2.1 op de volgende pagina worden alle gevonden taxa waar de peptide MFNWMVTR in voorkomt opgeliist, met daarnaast hun lineage naar de root. Om de LCA te vinden van deze wordt de tabel van links naar rechts overlopen, startend bij de minst specifieke rang. In het voorbeeld is dit dus de rang *superkingdom*. Aangezien alle organismes zich binnen hetzelfde superkingdom bevinden, namelijk de Eukaryota, schuiven we een rang op naar rechts. Ook bij de rang *kingdom* zien we dat alle organismes zich binnen hetzelfde kingdom bevinden. Op deze manier schuiven we steeds naar rechts – dus naar een specifiekere rang op – tot we bij de rang *class* komen. Tot op dit niveau is het eerste organisme niet gespecificeerd en verschillen ook de groepen van de overige peptides. Peptide 2 tot en met 5 bevinden zich binnen de rang van de *Aves* en die daarna bevinden zich binnen de rang van de *Mammalia*. Dit is dus een splitsing van de takken in de taxonomie en nemen we de vorige rang als LCA resultaat. In het voorbeeld is dat dus de *Craniata*

Tabel 2.1: (Ingekorte) tryptische peptide analyse van peptide MFNWMVTR

Organism	Superkingdom	Kingdom	Phylum	Subphylum	Class	Superorder	...
Pelodiscus sinensis	Eukaryota	Metazoa	Chordata	Craniata			...
Columba livia	Eukaryota	Metazoa	Chordata	Craniata	Aves	Neognathae	...
Meleagris gallopavo	Eukaryota	Metazoa	Chordata	Craniata	Aves	Neognathae	...
Ficedula albicollis	Eukaryota	Metazoa	Chordata	Craniata	Aves	Neognathae	...
Taeniopygia guttata	Eukaryota	Metazoa	Chordata	Craniata	Aves	Neognathae	...
Ornithorhynchus anatinus	Eukaryota	Metazoa	Chordata	Craniata	Mammalia	Neognathae	...
Monodelphis domestica	Eukaryota	Metazoa	Chordata	Craniata	Mammalia		...
Sarcophilus harrisii	Eukaryota	Metazoa	Chordata	Craniata	Mammalia		...
...	...	...	...	...	...	...	...



Figuur 2.5: Weergave van *Mammalia* en *Aves* op de tak van de *Craniata*. We zien dat de LCA op de tabel resulteert in *Craniata*, terwijl de boomgebaseerde LCA de *Amniota* vindt. Tussen de *Craniata* en de *Amniota* staan echter een tiental rangloze organismen. De boomgebaseerde taxonomische identificatie is dus een stuk specieker.

binnen de rang *subphylum*.

## 2.3 Beperkingen van de tabelmethode

De huidige implementatie in UniPept met de tabelgebaseerde methode is zeer snel aangezien alle mogelijke LCAs kunnen worden voorberekend, maar kent wel enkele beperkingen. In de NCBI taxonomy zijn er een heel aantal organismen aanwezig die niet geklassificeerd zijn onder een rang zoals rijk, familie, genus, etc. Deze organismen worden wel in de taxonomie opgenomen waar ze als rangloos geklassificeerd worden.

In Tabel 2.1 zien we het ingekorte resultaat van de tryptische peptide analyse op de peptide MFNWMVTR. Als LCA wordt hiervoor door UniPept het subphylum *Craniata* gevonden. Wanneer we echter de *Craniata*, de *Aves* en de *Mammalia* op een boomstructuur visualiseren krijgen we het resultaat uit Figuur 2.5. Als we op de boom de LCA zoeken van de *Mammalia* en de *Aves* komen we uit bij de *Amniota*, en niet bij de *Craniata*. De tiental onderverdelingen tussen de *Craniata* en de *Amniota* zijn allemaal rangloos.

We zien dus dat er door het omzetten van de boom in een tabulaire voorstelling (waarin enkel de ranghebbende taxa worden voorgesteld) veel informatie verloren gaat. Een stap naar speciekere classificaties is dus om een nieuwe methode te implementeren die rechtstreeks op de boom werkt en zo ook rekening houdt met tussenliggende rangloze taxa.

## 2.4 Boomgebaseerde LCA(\*)

De hierboven genoemde reden is niet de enige waarom het wenselijk is om een boomgebaseerde implementatie te maken. Zoals in Hoofdstuk 1 op pagina 1 beschreven is de laatste stap in de analyse om individuele identificaties van alle peptiden uit een eiwit(fragment) of DNA read tot één consensus te aggregeren.

In deze sectie diepen we het LCA\* algoritme verder uit. We starten met een implementatiebespreking van het gewone LCA algoritme en breiden die uit naar een algoritme voor LCA\*. Daarna bespreken we enkele problemen die voorkomen bij de omvorming en hun oplossingen.

De berekening moet natuurlijk zo snel mogelijk verlopen. Daarom zouden we dezelfde aanpak kunnen toepassen als UniPept om de LCA's van alle peptiden voor te berekenen. Op die manier zouden we alle resultaten in een databank kunnen opslaan, zodat we ze er in constante tijd uit kunnen halen. Die oplossing is helaas onmogelijk door het gigantische aantal mogelijke combinaties van taxa die als invoer kunnen worden gegeven. Er zijn op het moment van schrijven 1.267.511 taxa aanwezig in de NCBI taxonomy. Alle mogelijke subsets uit deze verzameling voorberekenen zou onmogelijk zijn. Bij het voorberekenen van de LCAs van alle peptiden (1.713.298.862) is dat wel mogelijk met voldoende opslagcapaciteit.

Iets wat we eventueel wel kunnen uitbuiten is de datastructuur waarop alle analyses zullen uitgevoerd worden. Die is namelijk altijd dezelfde.

### 2.4.1 Implementatie

In deze sectie bespreken we een algoritme om de LCA van een opgegeven lijst van nodes in een boomstructuur te berekenen. We tonen de tijd- en ruimtecomplexiteiten van deze implementatie aan en bespreken vervolgens hoe we het algoritme kunnen uitbreiden om als algoritme voor LCA\* te gebruiken.

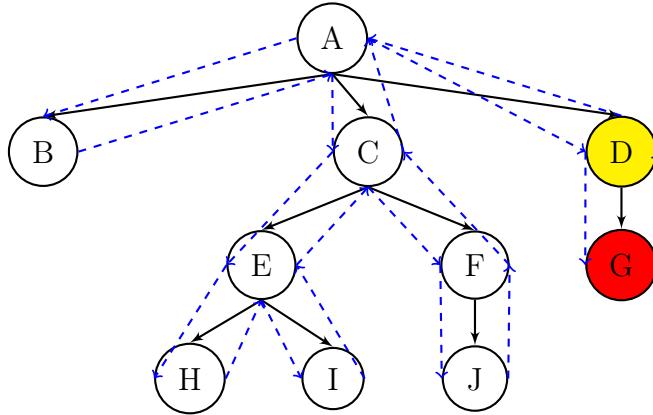
#### State of the art

Om het LCA probleem op te lossen, kijken we eerst naar de state of the art van dit probleem. We baseren ons op het artikel “Lowest Common Ancestors in Trees and Directed Acyclic Graphs” van Michael A. Bender en Martin Farach-Colton [1]. In het artikel wordt een algoritme beschreven om het LCA probleem in lineaire tijd te preprocessen. Dat kan door het LCA probleem om te vormen naar een range minimum query (RMQ) probleem. Na preprocessing kan in constante tijd de LCA van een paar nodes berekend worden.

#### Van LCA naar RMQ

Het LCA probleem is zoals we zien nauw verwant met het range minimum query (RMQ) probleem. Het RMQ probleem wordt geformuleerd als:

**Definitie 2.4.1.** RMQ Gegeven twee indices,  $i$  en  $j$ , in lijst  $A$ , geef de index in  $A$  van de kleinste waarde in de subarray  $A[i..j]$ .



Figuur 2.6: Illustratie van een Euler traversal door een boom. Hier duiden blauwe pijlen het pad aan dat doorheen de boom genomen wordt. De tabulaire voorstelling van dit pad is te vinden in Tabel 2.2.

Tabel 2.2: Tabulaire voorstelling van een Euler traversal door de boom uit Figuur 2.6.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Node	A	B	A	C	E	H	E	I	E	C	F	J	F	C	A	D	G	D	A
Diepte	0	1	0	1	2	3	2	3	2	1	2	3	2	1	0	1	2	1	0

Tabel 2.3: Tabel van eerste voorkomens van een node in Tabel 2.2.

A	B	C	D	E	F	G	H	I	J
0	1	3	15	4	10	16	5	7	11

Een belangrijke observatie hier is de volgende: de LCA van twee nodes,  $u$  en  $v$ , is de node die tijdens een Euler traversal van de boom tegenkomen wordt tussen  $u$  en  $v$  met de kleinste diepte. Wanneer we dus van twee nodes,  $u$  en  $v$ , de LCA willen berekenen, kunnen we eerst een Euler traversal op de boom uitvoeren waarbij we een lijst opstellen van dieptes van elke node die we tegenkomen op dit pad. Een voorbeeld van een Euler traversal is uitgewerkt in Figuur 2.6. De nodes worden met hun overeenkomstige diepte opgeslagen in een tabel. De tabulaire voorstelling van die Euler tour is te vinden in Tabel 2.2.

Eenmaal die tabel is opgesteld, kunnen we het RMQ algoritme toepassen op het deel tussen node  $u$  en node  $v$  van de tabel. Het deel tussen die nodes kunnen we vinden door gebruik te maken van de tabel van eerste voorkomens, weergegeven in Tabel 2.3. Door de bijhorende indices te nemen van node  $u$  en  $v$ , kunnen we in Tabel 2.2 RMQ toepassen op de lijst van dieptes van nodes tussen deze indices. Ter illustratie kunnen we bijvoorbeeld de LCA berekenen van nodes I en J. Nodes I en J komen volgens de tabel van eerste

Broncode 2.1: Voorbeeldcode om het LCA probleem te reduceren naar een probleem dat met RMQ kan worden opgelost.

```

1 def dfs_run(self, taxon, iteration, level):
2
3     euler_tour[iteration] = taxon.taxon_id
4     levels[iteration] = level
5     if not first_occurrences[taxon.taxon_id]:
6         first_occurrences[taxon.taxon_id] = iteration
7
8     for child in taxon.children:
9         iteration = dfs_run(child, iteration + 1, level + 1)
10
11    euler_tour[iteration] = taxon.taxon_id
12    levels[iteration] = level
13
14    return iteration + 1
15
16 euler_tour = [None]*(2*tree.length)-1)
17 levels = [None]*(2*tree.length)-1)
18 first_occurrences = [None]*tree.length
19
20 dfs_run(root_taxon, 0, 0)

```

voorkomens respectievelijk voor in de Euler traversal tabel op positie 7 en 11. De nodes tussen I en J zijn dus [I, E, C, F, J] met respectievelijke dieptes [3, 2, 1, 2, 3]. De node met de laagste diepte is dus C.

Formeel samengevat gaat de reductie van een LCA probleem naar een RMQ probleem als volgt (waarbij  $n$  het aantal nodes in de boom is):

1. Stel de `euler_tour` array van lengte  $2 * n - 1$  op die de identifier van de nodes bevat als we de boom langs het Euleriaanse pad aflopen;
2. Stel de `levels` array van lengte  $2 * n - 1$  op waarbij  $depth_i$  overeen komt met de diepte van overeenkomstige node `euler_tour[i]`;
3. Stel de `first_occurrences` array op van lengte  $n$  op, waarbij `first_occurrences[i]` overeen komt met het eerste voorkomen van node  $i$  in `euler_tour`.

Het opstellen van die drie arrays kan in één depth-first traversal door de boom gebeuren. De voorbeeldcode hiervoor is terug te vinden in Broncode 2.1 op pagina 15.

Wanneer deze drie arrays opgesteld zijn, komt het vinden van de LCA van twee nodes  $u$  en  $v$  neer op het volgende:

1. Zoek de index van  $u$  en  $v$  in de `first_occurrences` array. Dit geeft `first_occurrences[u]` en `first_occurrences[v]` die de positie van het eerste voorkomen van respectievelijk  $u$  en  $v$  in de `euler_tour` en de `levels` array aanduidt;

2. Bereken de RMQ in de subarray van `levels` van `first_occurrences[u]` tot `first_occurrences[v]`:  $\text{RMQ}_{\text{levels}}(\text{first\_occurrences}[u]:\text{first\_occurrences}[v])$ . Het resultaat hiervan is de index van de node met de kleinste diepte in de Euler traversal tussen  $u$  en  $v$ ;
3. De LCA van  $u$  en  $v$  bevindt zich nu in de `euler_tour` array op de index van het resultaat van voorgaande RMQ-bewerking:  

$$\text{LCA}(u, v) = \text{euler\_tour}[\text{RMQ}_{\text{levels}}(\text{first\_occurrences}[u]:\text{first\_occurrences}[v])].$$

## RMQ

De omzetting van het LCA probleem naar een RMQ probleem is dus zeker mogelijk in lineaire tijd ten opzichte van de nodes in de boom. RMQ zelf is echter geen eenvoudig probleem. In deze paragraaf wordt van een triviale implementatie vertrokken en wordt gaandeweg een meer optimale oplossing bekomen, gebruik makend van een aantal lemma's en observaties.

Er wordt gestart van een array  $A$  van lengte  $n$  met de eigenschap dat elk element met  $\pm 1$  verschilt van zijn buren. De  $\pm 1$  beperking ontstaat aangezien we het RMQ algoritme gebruiken op de `levels` array uit het voorgaande deel. Het RMQ probleem op arrays waar niet aan die eigenschap is voldaan, zullen we vanaf nu het RMQ probleem noemen. Wanneer wel aan deze eigenschap is voldaan, noemen we dat het RMQ $\pm 1$  probleem. In beide gevallen blijft het doel om, gegeven twee indexen in lijst  $A$ ,  $i$  en  $j$ , de index van de kleinste waarde in  $A$  te bekomen tussen  $i$  en  $j$ , met  $i$  en  $j$  inclusief.

**Naïeve oplossing** De naïeve oplossing is een oplossing die alle mogelijke combinaties van  $i$  en  $j$  voorberekent. Deze oplossing heeft een complexiteit van  $\Theta(n^3)$  om alles voor te berekenen wat kan worden gereduceerd tot  $\Theta(n^2)$  door gebruik te maken van dynamisch programmeren. Het uitvoeren van de effectieve RMQ vereist één lookup en kan dus in constante tijd gebeuren. Een preprocessing stap van  $\Theta(n^2)$  is echter veel te traag, zeker aangezien er snellere algoritmes bestaan.

**Snellere oplossing voor het algemene RMQ probleem** Er bestaat een snellere oplossing voor het algemene RMQ probleem. Het algemene idee is om elke mogelijke query voor te berekenen waarvan de lengte een macht van 2 is. Bij het opzoeken kunnen we dan de range opdelen in 2 (eventueel overlappende) blokken waarvan de lengte een macht van 2 is. Om in constante tijd tot een antwoord te komen, moeten we dus eerst voor elke  $i$  tussen 1 en  $n$  en elke  $j$  tussen 1 en  $\log n$  het kleinste element van de deelarray startend bij  $i$  met lengte  $2^j$  berekenen. Het resultaat hiervan voeren we in in de tabel  $M$  op positie  $[i, j]$ . Formeel wordt dit:  $M[i, j] = \min \{A[k], k = i..i + 2^j - 1\}$ . Zo hebben we voor elke mogelijke startindex alle mogelijke blokken binnen de array van een lengte van een macht van twee, startend op die index. Voorgaande berekening is het eenvoudigst te begrijpen aan de hand van een voorbeeld: voor de rij  $[1, 6, 2, 4]$  van lengte 4, geeft dit het volgende effect:

1	6	2	4
$\underline{M[1, 0] = \min A[1..1]}$			
$\underline{M[1, 1] = \min A[1..2]}$			
$\underline{M[1, 2] = \min A[1..4]}$			
	$\underline{M[2, 0] = \min A[2..2]}$		
	$\underline{M[2, 1] = \min A[2..3]}$		
	$\underline{M[2, 2] = \min A[2..5]}$		
		$\underline{M[3, 0] = \min A[3..3]}$	
		$\underline{M[3, 1] = \min A[3..4]}$	
		$\underline{M[3, 2] = \min A[3..6]}$	
			...

Met behulp van bovenstaande illustratie wordt duidelijk ingezien dat de volledige matrix  $M$  kan worden opgebouwd via dynamisch programmeren.

Formeel kan deze tabel opgebouwd worden volgens de formules, waarbij  $M[i, j]$  met  $j = 0$  overeen komt met het element op positie  $i$  in  $A$ .

$$M[i, j] = \begin{cases} M[i, j - 1], & \text{als } A[M[i, j - 1]] \leq A[M[i + 2^{j-1} - 1, j - 1]] \\ M[i + 2^{j-1} - 1, j - 1], & \text{anders} \end{cases}$$

In woorden omgezet betekent dit dat we in een blok van lengte  $2^j$  het minimum kunnen nemen van de blokken van lengte  $2^{j-1}$  waaruit dit blok van lengte  $2^j$  is opgebouwd. In bovenstaand voorbeeld kunnen we zo  $M[1, 1]$  vinden door het minimum te nemen van  $M[1, 0]$  en  $M[2, 0]$ . Op dezelfde wijze kan  $M[3, 1]$  gevonden worden door het minimum te nemen van  $M[3, 0]$  en  $M[4, 0]$ . Om dan  $M[1, 2]$  te vinden, nemen we het minimum van  $M[1, 1]$  en  $M[3, 1]$  die we zonet berekend hebben. Deze tabel kan dus worden opgevuld in  $\Theta(n \log n)$  tijd en ruimte.

Voor bovenstaand voorbeeld ziet die tabel er dan als volgt uit:

i	$j = 0$	$j = 1$	$j = 2$
1	0	0	0
2	1	2	2
3	2	2	2
4	3	3	3

De RMQ van een range van  $i$  tot  $j$  in  $A$  kan nu gevonden worden door twee (eventueel) overlappende blokken die deze volledige subarray bedekken. Om dat te bekomen zoeken we eerst het grootste blok met een lengte van een macht 2 dat binnen de subarray past. De lengte van dat blok kan eenvoudig berekend worden met de formule  $2^{\lceil \log(j-i) \rceil}$ . De RMQ van die subarray vinden we dan door het minimum te nemen van de RMQ op het blok dat start op index  $i$  en de RMQ van het blok dat eindigt op index  $j$ . Aangezien deze blokken een lengte van een macht van 2 hebben, heeft de preprocessing de minima van die blokken al voorberekend en kunnen we dus de RMQ in constante tijd berekenen van deze arbitraire subarray.

In bovenstaand voorbeeld kunnen we bijvoorbeeld de RMQ berekenen van index 0 tot 2. Hiervoor selecteren we twee blokken met lengte  $2^{\lceil \log(2-0) \rceil} = 2$ . Het eerste blok start op index 0, het laatste blok eindigt op index 2. De RMQ van deze twee blokken is al

voorberekend:  $A[M[1, 1]] = A[0] = 1$  en  $A[M[2, 1]] = A[2] = 2$ . De oplossing is dus 1, te vinden in A op index 0.

Dit algoritme kan dus de array A voorberekenen in  $\Theta(n \log n)$  tijd en ruimte. Opzoeken kunnen als gevolg van deze voorberekening in constante snelheid gebeuren.

**Oplossing voor het RMQ $\pm 1$  probleem** Het voorberekenen kan echter nog sneller wanneer de array A voldoet aan de  $\pm 1$  beperking. In het geval van de `levels` array is dat altijd zo aangezien we altijd van kind naar ouder of omgekeerd gaan.

Eerst wordt de array A verdeeld in blokken ter grootte van  $\frac{\log n}{n}$ . Daarna wordt een array,  $A'$  gedefinieerd van  $\frac{2*n}{\log n}$  elementen, waarbij het element op index  $i$  in  $A'$  het minimum element is van het  $i^{\text{de}}$  blok in A. Aangezien het RMQ algoritme de index van het kleinste element in een array teruggeeft en niet de waarde zelf, wordt een bijhorende array gedefinieerd,  $B$  waar  $B_i$  de positie aangeeft van het kleinste element in het  $i^{\text{de}}$  blok van  $A'$ .

Nu kan voorgaand algoritme voor het algemene RMQ probleem toegepast worden op de nieuwe array  $A'$ . Aangezien  $A'$  van lengte  $\frac{2*n}{\log n}$  is en het RMQ algoritme een tijdscomplexiteit heeft van  $n * \log n$ , gebeurt dit in  $\Theta\left[\frac{2*n}{\log n} * \log \frac{2*n}{\log n}\right] \approx \Theta(n)$  tijd.

Op basis van de huidige preprocessing van  $A'$  kunnen alle combinaties van  $i$  en  $j$  in constante tijd gevonden worden wanneer  $i$  en  $j$  op de grenzen van de blokken in A vallen. Dit is echter niet altijd het geval. Aangezien  $i$  en  $j$  binnen eenzelfde blok kunnen vallen, moeten we ook elk blok apart voorberekenen. Het is echter ook mogelijk dat  $i$  en  $j$  in verschillende blokken vallen. In dat geval wordt de RMQ als volgt berekend:

1. Bereken het minimum van  $i$  tot het einde van het blok waar  $i$  in valt;
2. Bereken het minimum van alle blokken in A tussen de blokken waar  $i$  en  $j$  in vallen;
3. Bereken het minimum van  $j$  tot het begin van het blok waar  $j$  in valt.

Het minimum van deze drie waarden is dan het kleinste element binnen de array A tussen  $i$  en  $j$ . Om stap 2 te berekenen kan gebruik worden gemaakt van de reeds voorberekende array  $A'$ . Stap 1 en 3 vergen echter wat meer werk. Er moet nu een snelle manier worden gevonden om een RMQ probleem te beantwoorden binnenin één blok.

Als op elk blok in A apart de algemene RMQ preprocessing zou worden gedaan, zouden we te veel tijd spenderen aan preprocessing. Daarom wordt gekeken of we het aantal mogelijke blokken kunnen beperken. Dat kan ook volgens volgende observatie: Wanneer twee arrays,  $X$  en  $Y$ , beide van lengte  $k$ , op elke positie met een constante waarde verschillen zodat  $X_i = Y_i + c$  voor elke  $i$  en een constante  $c$ , dan zijn alle mogelijke RMQ antwoorden hetzelfde voor beide arrays. Bijvoorbeeld gegeven arrays  $[1, 2, 1, 0]$  en  $[3, 4, 3, 2]$  dan geldt deze observatie voor  $c = 2$ . In beide blokken is het resultaat van de RMQ voor het volledige blok gelijk, namelijk index 3.

Deze observatie heeft een belangrijk gevolg. Nu kunnen we namelijk elk blok normaliseren door de eerste waarde uit dat blok af te trekken van alle waarden in datzelfde blok. Voor het voorbeeld hierboven geeft dit  $[1 - 1, 2 - 1, 1 - 1, 0 - 1] = [3 - 3, 4 - 3, 3 - 3, 2 - 3] = [0, 1, 0, -1]$ .

Broncode 2.2: Voorbeeldcode in Python voor de LCA berekening van 2 nodes, na uitvoering van Broncode 2.1 op pagina 15

```

1 def calc_lca_pair(first, second):
2     first_index = first_occurrences[first]
3     second_index = first_occurrences[second]
4
5     rmq_index = get_rmq(first_index, second_index)
6
7     return euler_tour[rmq_index]
```

Er kan nu worden aangetoond dat er  $\Theta(\sqrt{n})$  verschillende genormaliseerde blokken kunnen bestaan. Alle genormaliseerde blokken starten namelijk met 0 aangezien die waarde van zichzelf wordt afgetrokken. Wegens de  $\pm 1$  beperking die ook op deze genormaliseerde blokken geldt, bekomen we strings van lengte  $\frac{\log n}{2} - 1$ , waarbij  $n$  nog steeds de lengte is van de originele array  $A$ . Deze genormaliseerde blokken kunnen we ook omzetten in bitstrings waarbij een 1 aangeeft dat het getal op die index 1 hoger is dan zijn voorgaande getal, en 0 op dezelfde manier een verschil van -1 aangeeft. Dit resulteert in het bestaan van  $2^{\frac{\log n}{2}-1} = \frac{\sqrt{n}}{2} = \Theta(\sqrt{n})$  bitstrings, en dus  $\Theta(\sqrt{n})$  mogelijke genormaliseerde blokken.

Wat nu nog rest is het maken van tabellen voor elk genormaliseerd blok. Voor deze  $\Theta(\sqrt{n})$  blokken wordt een tabel aangemaakt waarin we alle antwoorden voor RMQ opvragingen binnen dat blok opslaan. Dit zijn er  $\frac{\log n}{2}^2 = \Theta(\log n^2)$  per blok, dus  $\Theta(\sqrt{n} \log n^2) \cong \Theta(n)$  in totaal. Opzoeken kunnen aan de hand van deze tabellen in constante tijd gebeuren.

Het laatste wat ons nog rest is het berekenen van welk blok in de originele array  $A$  bij welk genormaliseerd blok hoort. Dit kan eenvoudig door een mapping bij te houden van de bitstring na normalisatie.

Samengevat hebben we nu binnen een tijdscomplexiteit van  $\Theta(n)$  alle mogelijke RMQ $\pm 1$  vragen binnen blokken voorberekend, alsook alle RMQ vragen over meerdere blokken heen in dezelfde tijd. Door die voorberekening kunnen we in constante tijd de drie benodigde minima opvragen zoals hierboven aangegeven, en kunnen we nu voor elke opgegeven indices  $i$  en  $j$  het RMQ $\pm 1$  probleem oplossen in  $\Theta(1)$ .

De omzetting van LCA naar RMQ kan door middel van de depth-first traversal in  $\Theta(n)$  gebeuren en dient éénmalig te gebeuren om de benodigde arrays voor de RMQ preprocessing voor te berekenen. De opslag voor zowel de benodigde arrays van de boomstructuur als voor de interne arrays in de RMQ preprocessing bedraagt  $\Theta(n)$ .

**Implementatie** Aangezien er enkele C bibliotheken bestaan die bovenstaand algoritme voor het RMQ probleem implementeren, zullen we gebruik maken van zo'n bestaande bibliotheek. De bibliotheek die we hiervoor gebruiken is een implementatie geschreven door Hideo Bannai [8, 14] die voldoet aan de besproken snelheids- en ruimtecomplexiteiten. Het berekenen van de LCA van twee nodes gebeurt dan zoals geïllustreerd in Broncode 2.2.

### LCA van meerdere nodes

Met voorgaand algoritme kan gemakkelijk de LCA van twee nodes berekend worden. Het probleem wordt echter moeilijker wanneer we de LCA van meer dan twee nodes willen berekenen.

In een boomstructuur waar een interne orde opgelegd is, zouden we de boom in-order kunnen overlopen. Daarna kunnen we uit alle quernodes de twee nodes opzoeken die het eerst en het laatst voorkomen tijdens deze in-order wandeling door de boom. Dit garandeert dat de twee geselecteerde nodes de buitenste nodes van alle quernodes zijn, en dat dus enkel de LCA van deze twee nodes moet berekend worden om tot een eindresultaat te komen voor alle quernodes.

Met deze aanpak zijn er echter twee aspecten die we in rekening moeten brengen. Ten eerste is het niet mogelijk een vaste volgorde op te leggen aan de taxa in de NCBI taxonomy. Enkel de kind-ouder relatie bestaat hier. Hoe een kind zich verhoudt ten opzichte van ouder of broers is niet vastgelegd. Het proberen opstellen van een in-order volgorde van een ouder met twee kinderen, A en B, kan dus resulteren in zowel [A, ouder, B] als [B, ouder, A]. Wanneer de  $\text{LCA}(A, \text{ouder}, B)$  berekend wordt, zal afhankelijk van de volgorde  $\text{LCA}(A, B)$  of  $\text{LCA}(B, A)$  berekend worden, wat wegens het symmetrisch zijn van LCA hetzelfde resultaat oplevert.

Ook belangrijk is dat de taxonomische boom geen binaire boom is. Stel dat de voorgaande ouder er nog een kind bij krijgt, C, dan kan het in-order overlopen volgende volgordes aannemen: [A, ouder, B, C], [A, B, ouder, C], [C, B, ouder, A], et cetera. Als we dan opnieuw de LCA berekenen van  $\text{LCA}(A, \text{ouder}, B)$  dan weten we niet of we  $\text{LCA}(A, B)$ ,  $\text{LCA}(A, \text{ouder})$ ,  $\text{LCA}(B, A)$ , ... moeten berekenen. Ook hier leveren alle mogelijkheden hetzelfde resultaat op, namelijk de ouder, maar we zullen later zien dat dit bij het LCA\* algoritme niet het geval is.

#### 2.4.2 Van LCA naar LCA\*

De aanpassing van LCA naar LCA\* lijkt op het eerste zicht vrij eenvoudig. Als we de LCA berekenen van twee nodes, en het resultaat van deze berekening is één van die twee nodes, dan weten we dat beide nodes op hetzelfde pad naar de wortel liggen. Als resultaat nemen we dan de node met de grootste diepte in plaats van deze met de kleinste diepte in het geval van LCA\*. De code die de LCA berekent uit Broncode 2.2 op pagina 19 wordt dan uitgebreid naar de code in Broncode 2.3 op de volgende pagina.

Deze aanpassing levert echter problemen op voor het toepassen van het LCA\* algoritme wanneer de LCA wordt berekend van meer dan twee nodes. Zoals uitgelegd in de vorige sectie, kan de LCA van meer dan twee nodes gevonden worden door het LCA algoritme toe te passen op de eerste en laatste quernode bij een in-order overlopen van de boom.

Stel dat we de boom er terug bij nemen met een ouder en drie kinderen: A, B en C en de ordening [A, B, ouder, C].  $\text{LCA}^*(\text{ouder}, A, B)$  met de methode van het nemen van de “buitenste” quernodes reduceert het probleem naar  $\text{LCA}^*(A, \text{ouder})$  wat met voorgaand algoritme resulteert in A. Hier is echter de informatie uit de tak van B verdwenen, A kan

Broncode 2.3: Uitbreiding van de voorbeeldcode uit Broncode 2.2 op pagina 19 in Python voor de LCA\* berekening van 2 nodes

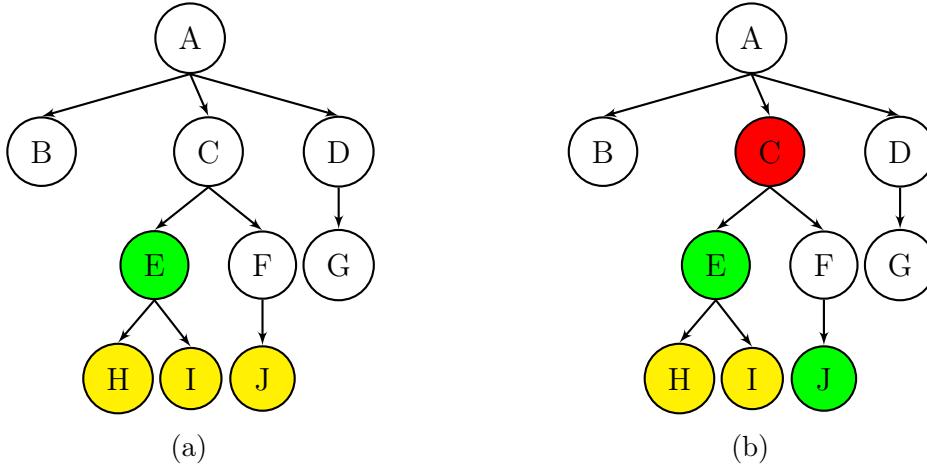
```

1 def calc_lca_pair(first, second):
2     first_index = first_occurrences[first]
3     second_index = second_occurrences[second]
4
5     rmq_index = get_rmq(first_index, second_index)
6
7     # Speciaal geval LCA*
8     if rmq_index == first_index:
9         lca_index = second_index
10    elif rmq_index == second_index:
11        lca_index = first_index
12    # Wanneer de RMQ index geen van beide originele indexen is, gebruik LCA
13    else:
14        lca_index = rmq_index
15
16    return euler_tour[lca_index]
```

namelijk nooit het resultaat zijn van  $\text{LCA}^*(A, B)$ . Ook een pre- en post-ordening hebben hetzelfde probleem, aangezien er geen specifieke ordening vast ligt voor de taxonomische boom. Bijvoorbeeld met de pre-ordening [ouder, A, B, C] is de  $\text{LCA}^*(\text{ouder}, B, C) = \text{LCA}^*(\text{ouder}, C) = C$ , waarbij de tak van B opnieuw ontrect wordt genegeerd. Als laatste middel kunnen we nog terug proberen te vallen op de Eulertraversal ordening, [ouder, A, ouder, B, ouder, C, ouder], maar ook hier geldt het tegenvoorbeeld:  $\text{LCA}^*(\text{ouder}, A, B) = \text{LCA}^*(\text{ouder}, B) = B$ , waarbij ditmaal de tak van A genegeerd wordt.

Een alternatief is het paarsgewijs toepassen van het LCA\*-algoritme. Gegeven een lijst van nodes kunnen we eerst voor de eerste twee nodes in die lijst de LCA\* berekenen. Het resultaat daarvan kan dan met LCA\* geaggregeerd worden met de derde node in de lijst. Op deze manier worden verder gegaan tot alle nodes in de lijst zijn geaggregeerd. Grote lijsten kunnen op die manier zelfs parallel verwerkt worden. Zo kan  $\text{LCA}^*(A, B, C, D)$  bijvoorbeeld berekend worden aan de hand van  $\text{LCA}^*(\text{LCA}^*(A, B), \text{LCA}^*((C, D)))$ . Voor het klassieke LCA-algoritme werkt dit probleemloos aangezien we nooit in de boom afdalen. Dit is echter niet het geval bij het LCA\*-algoritme. In Figuur 2.7 op de pagina hierna berekenen we bijvoorbeeld  $\text{LCA}^*(H, I, J)$ . Dit wordt paarsgewijs berekend door eerst  $\text{LCA}^*(H, I) = E$  te berekenen en daarna  $\text{LCA}^*(E, J) = C$ . Wanneer we echter een verkeerd tussenresultaat op hetzelfde pad als een volgende node bekomen, zoals in Figuur 2.8 op pagina 23, dan kan dit een verkeerd resultaat opleveren.  $\text{LCA}^*(B, C, D)$  wordt berekend aan de hand van  $\text{LCA}^*(B, C) = A$ , en daarna  $\text{LCA}^*(A, D)$ . Dit levert dat de lager liggende node D als LCA\* resultaat, waardoor de takken van B en C volledig genegeerd worden en het resultaat eveneens fout is.

De oorzaak van het probleem is te wijten aan het afdalen in de boom met het LCA\* algoritme wanneer een eerder bekomen LCA\* resultaat als tussenstap gebruikt wordt. Om



Figuur 2.7: Tussenstappen voor berekenen van  $\text{LCA}^*(H, I, J)$ . Gele nodes zijn querynodes, groene tussenresultaten en rode het eindresultaat. Via de iteratieve reductie geeft dit  $\text{LCA}^*(\text{LCA}^*(H, I), J)$ , wat resulteert in  $\text{LCA}^*(E, J)$ . Het uitrekenen hiervan geeft C als correct resultaat.

dit probleem op te lossen, kan gebruik gemaakt worden van een dieptelimiet in de boom, waar niet onder mag worden afgedaald. Deze dieptelimiet schuift telkens op naar het niveau van een node die het resultaat is van een  $\text{LCA}^*$  tussenstap waar niet wordt afgedaald. In het voorbeeld in Figuur 2.8b op de pagina hierna zou bij de berekening van  $\text{LCA}^*(B, C, D) = \text{LCA}^*(\text{LCA}^*(B, C), D)$  er na het verkrijgen van A als tussenresultaat van  $\text{LCA}^*(B, C)$  dus niet verder mogen worden afgedaald worden in een tak onder A, aangezien A reeds het tussenresultaat is van een gewone LCA stap op B en C. Na deze stap wordt de dieptelimiet op 0 gezet, aangezien resultaat A diepte 0 heeft. Wanneer een volgende  $\text{LCA}^*$  stap gedaan wordt op A en D, met dieptelimiet 0, dalen we niet af naar D aangezien de diepte van D (1) groter is dan deze limiet (0). Dit wordt geïllustreerd in Figuur 2.9 op pagina 24. De nieuwe notatie die we hiervoor invoeren is  $\text{LCA}_d^*$  waarbij  $d$  de huidige waarde is van de dieptelimiet.  $\text{LCA}^*(n_1, n_2, \dots, n_m)$  is dan een korte notatie voor  $\text{LCA}_\infty^*(n_1, n_2, \dots, n_m)$ .

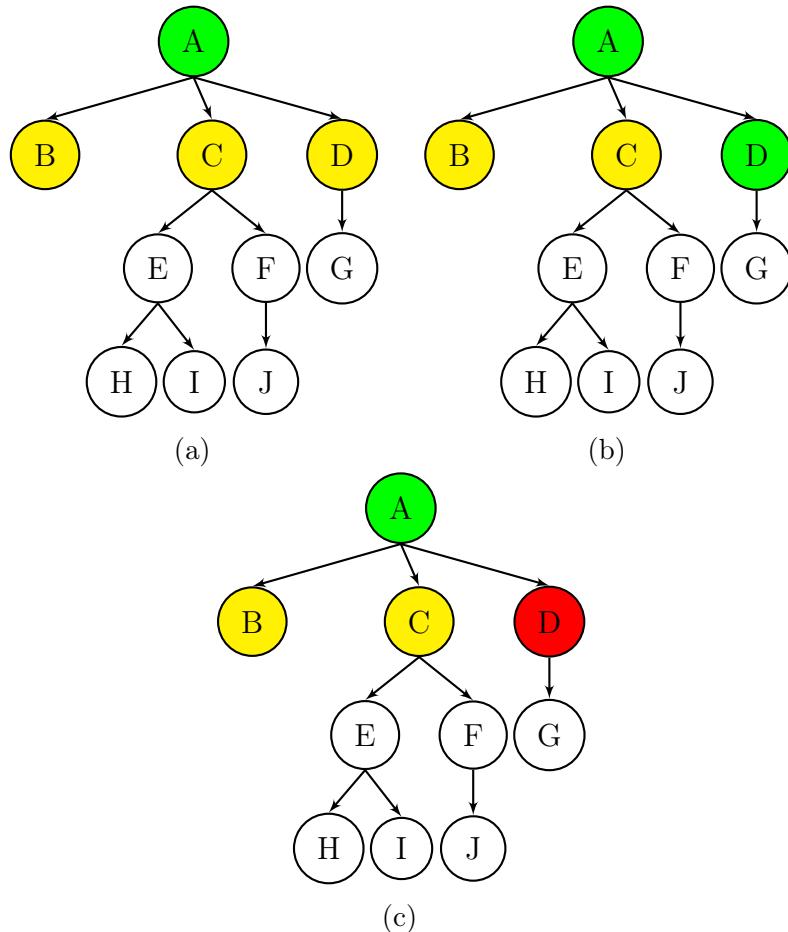
We breiden nu het codevoorbeeld uit Broncode 2.3 op pagina 21 uit naar Broncode 2.4, waarin rekening wordt gehouden met de dieptelimiet.

Broncode 2.4: Uitbreiding van de voorbeeldcode uit Broncode 2.3 op pagina 21 in Python voor de  $\text{LCA}^*$  berekening van 2 nodes. Deze code houdt rekening met de toegevoegde `depth_limit` parameter. Zoals besproken kan de `reduce` stap hier ook geparallelliseerd worden in plaats van met een accumulator te werken.

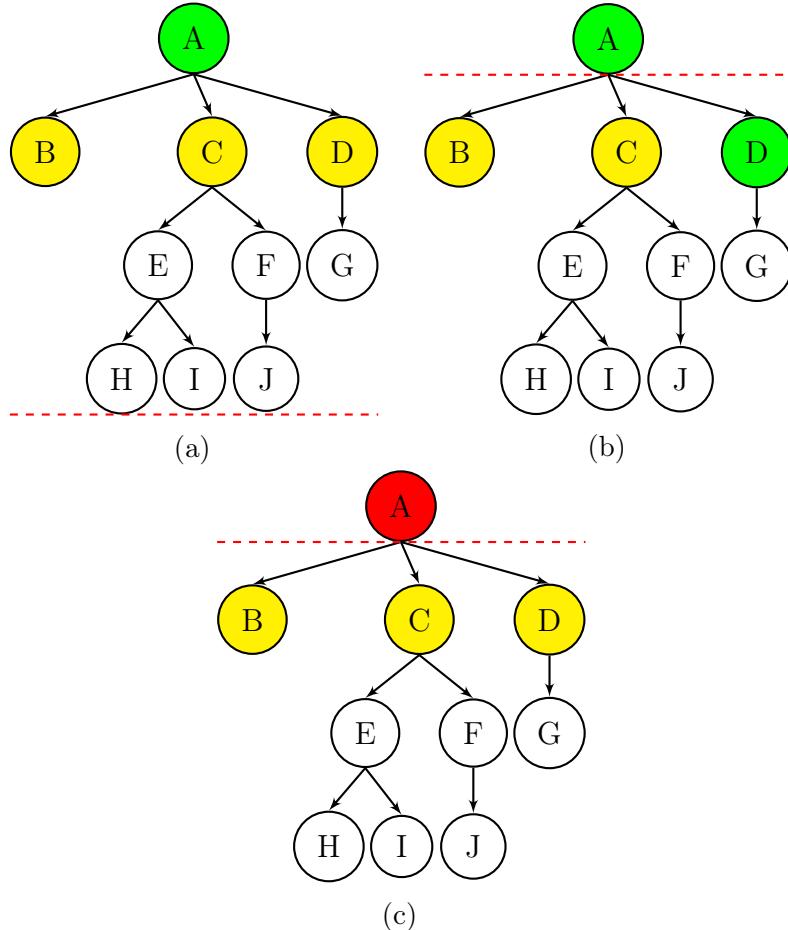
```

1 | def calc_lca_pair(acc, second, depth_limit):
2 |     depth_limit, first = acc
3 |
4 |     first_index = first_occurrences[first]
5 |     second_index = first_occurrences[second]
6 |

```



Figuur 2.8: Tussenstappen voor het berekenen van  $\text{LCA}^*(B, C, D)$ . Gele nodes zijn quaternodes, groene tussenresultaten en rode het eindresultaat. Via de iteratieve methode geeft dit  $\text{LCA}^*(\text{LCA}^*(B, C), D)$ , wat resulteert in  $\text{LCA}^*(A, D)$ . Het uitwerken van  $\text{LCA}^*(A, D)$  geeft echter foutief D als resultaat waarbij de takken van B en C volledig genegeerd worden.



Figuur 2.9: Tussenstappen voor het berekenen van  $\text{LCA}_{\infty}^*(B, C, D)$ . Gele nodes zijn querynodes, groene tussenresultaten en rode het eindresultaat. Met de paarsgewijze methode geeft dit  $\text{LCA}_0^*(\text{LCA}_{\infty}^*(B, C), D)$ . De binnenste LCA\* berekening geeft A als resultaat. Aangezien B en C beide deeltakken zijn van A wordt de dieptelimiet naar boven geschoven tot op de diepte van A, namelijk 0. Dit resulteert in  $\text{LCA}_0^*(A, D)$ . Het uitwerken hiervan zou met de oude methode D als resultaat geven, maar aangezien de diepte van D groter is dan de dieptelimiet geven we A als resultaat.

```

7      rmq_index = get_rmq(first_index, second_index)
8
9      # Speciaal geval LCA*
10     if rmq_index == first_index and levels[second_index] <= depth_limit:
11         lca_index = second_index
12     elif rmq_index == second_index and levels[first_index] <= depth_limit:
13         :
14         lca_index = first_index
15     # Wanneer de RMQ index geen van beide originele indexen is, gebruik
16     # LCA
17     else:
18         lca_index = rmq_index
19         depth_limit = levels[lca_index]
20
21
22 # Input: een lijst (of generator) van taxon identifiers
23 taxon_ids = iter([...])
24
25 first = next(taxon_ids)
26 depth_limit = sys.maxsize
27
28 _, lca_id = reduce(calc_lca_pair, taxon_ids, (depth_limit, first))

```

## 2.5 Toekomstig werk

In dit hoofdstuk bespraken we een nieuwe aggregatiemethode om verschillende taxa te aggregeren tot een zo specifiek mogelijk eindresultaat om te gebruiken bij de analyse van metagenomic samples. Deze nieuwe methode moet natuurlijk wel ergens kunnen worden gebruikt. De aangewezen plaats is hiervoor zijn de UniPept web services, zodat deze door zowel de UniPept command-line interface als door de webinterface kan gebruikt worden. In deze webservices is inmiddels al het commando `taxa2lca`[19] ingebouwd, die gebruik maakt van de huidige tabelgebaseerde berekening. Analoog aan dit commando zou dus een nieuwe commando, `taxa2treelca` kunnen worden ingebouwd. Aangezien alle calls naar deze services zo snel mogelijk moeten gebeuren, zou het dus aangeraden zijn dat de UniPept webservices de benodigde data voor het RMQ probleem in memory houden zodat een LCA\* vraag onmiddellijk kan worden beantwoord zonder dat er nog iets moet worden opgestart of opgebouwd.

Als alternatief voor onderzoekers die hun data niet graag over het internet sturen kunnen we ook een lokale versie van dit commando aanbieden. Dit houdt natuurlijk wel in dat de gebruiker de benodigde arrays en preprocessing lokaal moet doen. Voor de case study in Hoofdstuk 3 op pagina 27 werden al enkele scripts geschreven die een taxonomie kunnen inladen en deze omzetten in de benodigde arrays en datastructuren om het LCA\* probleem te kunnen oplossen. De gebruiker zou er dan voor kunnen kiezen om zelf een service te draaien die deze datastructuren in memory houdt, of om de voorberekende data

te serialiseren zodat deze niet telkens opnieuw hoeft voorberekend te worden.

Een laatste puntje dat verbeterd kan worden is het gebruik van de C bibliotheek. Momenteel wordt deze vanuit Python opgeroepen, waardoor bij elke oproep naar die bibliotheek de data moet worden omgezet van een Python datastructuur naar een C datastructuur. Aangezien de UniPept web en command-line interface in Ruby geschreven zijn zou die omzetting daar dus ook moeten gebeuren. Een herimplementatie van die bibliotheek in native Python/Ruby code zou deze omzetting, en dus ook het tijdsverlies dat ermee gepaard gaat, overbodig maken. Of de resulterende native code dan ook werkelijk sneller is dan de omzetting naar en de berekening in C zal gebenchmarkt moeten worden.

# Hoofdstuk 3

## Case Study: Benchmarking the UniPept Metagenomics Analysis Pipeline

In dit hoofdstuk evalueren we de prestaties van de UniPept Metagenomics Analysis Pipeline (UMAP). Dat werd onderzocht in een groepsproject van het vak Computationele Biologie, gedoceerd door professor Dawyndt, door een groep van vijf studenten (Robin Declerck, Simon Houbracken, Michael Niklaus, Ine Melckenbeeck, Bereket Tesfamariam Habtemariam) onder leiding van mijzelf.

In een tweede project binnen het vak werden, onder leiding van Felix Van der Jeugt, reeds geklassificeerde peptiden in de IGC databank opnieuw geklassificeerd met de UMAP. Zij concludeerden dat het grootste deel van de resultaten overeen kwamen tussen UniPept en IGC, maar dat UniPept meer (zowel true als false) positives rapporteert en daarenboven ook een specieker onderscheid tussen de verschillende rangen maakt. Het volledige onderzoek is te lezen in de scriptie: “Optimaliseren van de UniPept Shotgun Metaproteomics Analysis Pipeline”[9].

Ter herhaling doorloopt de UMAP enkele stappen, geïllustreerd in de inleiding in Figuur 1.5 op pagina 7. Voor elke DNA read in een metagenomics sample kunnen we een proteomics experiment doen. Hierbij zoeken we, bijvoorbeeld aan de hand van FragGeneScan, alle eiwitten op de DNA read in kwestie. De bekomen eiwitten worden opgedeeld in tryptische peptiden waarna elke peptide afzonderlijk op zijn taxonomische identificatie wordt afgebeeld door middel van het lowest common ancestor (LCA) algoritme. Aan de hand van het LCA\* algoritme, beschreven in Hoofdstuk 2 op pagina 8, kunnen we de bekomen taxa aggregeren naar een consensusclassificatie.

Voor dit benchmarkingsproces gaan we eerst op zoek naar een manier om te testen hoe goed de UMAP presteert op onbekende data en stellen we een aantal kernvragen die binnen dit onderzoek beantwoord moeten worden. Daarna worden de genomen stappen in het onderzoek besproken en vatten we de resultaten samen. “Goed presteren” definiëren we hier aan de hand van één hoofdzaak en enkele bijzaken. De toolchain moet hoofdzakelijk een zo specifiek mogelijke, maar wel correcte, identificatie van de invoerdata als resultaat geven. Daarnaast moet de identificatie ook snel kunnen gebeuren zonder al te veel hardwarebenodigdheden te vragen.

### 3.1 Bepalen van een benchmarkstrategie

Het bepalen van een benchmarkstrategie is cruciaal om tot betrouwbare resultaten te komen. Op het eind van deze case study willen we drie vragen kunnen beantwoorden: *i*) hoe accuraat is taxonomische identificatie met de UMAP, *ii*) hoe accuraat is die taxonomische identificatie wanneer ze wordt toegepast op onbekende data en *iii*) hoe goed presteert de UMAP op gesimuleerde DNA reads die fouten bevatten?

De eerste vraag kunnen we onderzoeken door bekende, volledig gesequeneerde genomen als invoer van de UMAP te gebruiken. Aangezien alle data uit deze genomen bekend is, en we er ook van uit gaan dat de taxonomische identificatie van de genomen correct is, verwachten we dat de resultaten zeer specifiek zullen zijn.

De tweede vraag is echter moeilijker te beantwoorden. We kunnen namelijk geen analyse uitvoeren van nog niet gesequeneerde genomen, aangezien we dan geen objectieve manier hebben om in te schatten hoe accuraat het resultaat van de analyse zou zijn. Wat we wel kunnen, is dit soort genomen simuleren op basis van de complete genomen uit de vraag hierboven. Wanneer we de Figuur 2.2 op pagina 9 er nog eens bij halen, zien we dat in de UniPept Metaproteomics Analysis Pipeline de LCA van een peptide wordt berekend aan de hand van de eiwitten waarin dit peptide voorkomt. Deze eiwitten worden dan afgebeeld op hun overeenkomstige taxa, waarna via een aggregatie via LCA een consensustaxon bekomen wordt. Stel nu echter dat een genoom, en bij gevolg ook eiwitten uit dat genoom, niet gekend is, dan zal de resulterende mapping van een peptide naar eiwitten die eiwitten ook niet bevatten. Het proces van “niet gekend zijn” kunnen we dus simuleren door alle eiwitten die voorkomen in het oorspronkelijk genoom te schrappen uit de lijst van eiwitten bij een peptide horen. Door middel van deze filterstap (soort van leave-one-out strategie) kunnen we dus een ongekend genoom simuleren uit een wel gekend genoom en kunnen we toch nog de resultaten vergelijken.

De derde vraag kan op de volgende manier beantwoord worden. We kunnen aan de hand van een read simulator, zoals wgsim [11], reads met verschillende error percentages simuleren op basis van een volledig genoom. In deze reads kunnen we dan via een gene predictor, bijvoorbeeld FragGeneScan [17], eiwitfragmenten extraheren. Op deze eiwitten kunnen we kunnen we dan de UMAP uitvoeren en de resultaten vergelijken met de resultaten van de UMAP op het originele genoom.

We werken dus twee benchmarks uit. In de eerste nemen we de sequentie van een genoom als invoer, waaruit we alle eiwitten extraheren. Die eiwitten splitsen we vervolgens op in peptiden, waarop de UniPept Metaproteomics Analysis Pipeline wordt uitgevoerd. De resulterende lijst van taxa aggregeren we nu per proteïne met het LCA\* algoritme naar een consensustaxon. Als uitvoer krijgen we dan voor elke proteïne in het invoergenoom een taxonomische identificatie die we kunnen vergelijken met de taxonomische identificatie van het genoom zelf. Deze toolchain noemen we in het vervolg `pept2lca2lca*`. De naamgeving hiervan steunt op de genomen stappen in de analyse: van peptiden → lca en van lca → lca\*. In de volgende sectie zal deze naamgeving verduidelijkt worden.

Bij de tweede toolchain nemen we opnieuw een genoom als invoer, waaruit we opnieuw alle eiwitten extraheren. De eiwitten splitsen we op in peptiden. Deze peptiden verwerken

we op vergelijkbare wijze als hierboven, maar nu met een bijkomende tussenstap. Voor elke peptide wordt eerst een lijst van eiwitten opgevraagd waar het betreffende proteïne in voorkomt. Uit deze lijst worden dan de eiwitten gefilterd die reeds voorkomen in het invoergenoom. De resterende eiwitten worden verder verwerkt door ze af te beelden op hun taxa in de Unipept taxonomie, waarna ze worden geaggregeerd via het LCA algoritme. De lijst van resulterende taxa wordt vervolgens per proteïne geaggregeerd wordt de lijst van resulterende taxa per proteïne geaggregeerd met het LCA\* algoritme, wat eenzelfde lijst als bij de eerste toolchain oplevert. Naar deze toolchain verwijzen we vanaf nu met `pept2prot2filter2lca`, steunend op de overgangen van peptiden → eiwitten, eiwitten → gefilterde eiwitten, gefilterde eiwitten → lca\*.

Door één invoergenoom op beide manieren te verwerken, krijgen we nu één consensus-taxon per proteïne uit het genoom. Deze twee lijsten kunnen we nu onderling vergelijken om na te gaan hoeveel er precies aan specificatie verloren is gegaan door de filterstap. Voor een goede taxonomische identificatie door UMAP verwachten we dat de eerste lijst (onder filtering) bijna volledig zal bestaan uit taxa op *species* niveau. De tweede lijst (analyse op gesimuleerde onbekende genomen) kunnen we dan vergelijken met de eerste lijst om het verlies aan taxonomische identificatie in te schatten wanneer er wordt gewerkt met ongekende genomen.

Op deze twee resulterende lijsten van taxa wordt ook een onderscheid gemaakt tussen specificatieverlies en een verkeerde identificatie van een taxon. Wanneer het resulterende taxon op de lineage ligt van het taxon van het genoom in de invoer, maar niet van de rang *species* is, verliezen we aan specificatie. Dit zullen we aangeven met de term “match”. Wanneer de resulterende taxon niet op de lineage ligt van het taxon van het ingevoerde genoom, spreken we van een verkeerde identificatie, of “mismatch”.

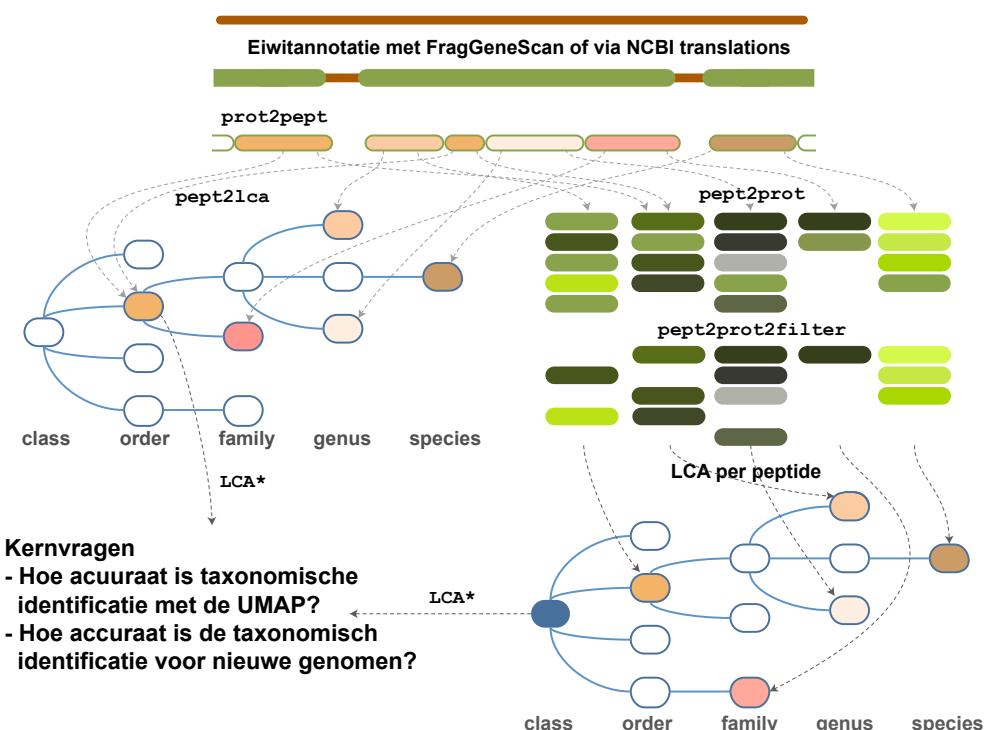
Als invoer voor dit benchmarkingsproces gebruiken we een lijst van ruim duizend volledig gesequeneerde genomen uit de NCBI databank. Beide toolchains worden geïllustreerd in Figuur 3.1 op de pagina hierna.

## 3.2 Implementatie

De implementatie van de twee hierboven beschreven toolchains steunt op een aantal tools en bibliotheken. In deze sectie bespreken we eerst deze lijst en geven we aan waar we de tools voor gebruiken. Daarna volgt de bespreking van het benchmarkingsproces aan de hand van een analysescript. Vervolgens staan we nog stil bij een ingreep in dat script om de snelheid ervan te verhogen.

### 3.2.1 Gebruikte tools en bibliotheken

- **Unipept-CLI:** zoals beschreven in de inleiding is de Unipept-CLI een interface in Ruby naar de webservices van Unipept. Hiermee kunnen alle methodes van de Unipept webservices aangesproken worden. We maken gebruik van de commando's `pept2lca`, die een lijst van peptides op hun LCA afbeeldt, en `pept2prot` die voor een



Figuur 3.1: Illustratie van beide processen. Alle eiwitten worden uit een genoom geëxtraheerd en opgesplitst in peptiden. Die peptiden worden dan verwerkt via de UniPept Metaproteomics Analysis Pipeline tot taxa (links) of worden (rechts) verwerkt door eerst alle eiwitten op te vragen waarin de peptide voorkomt. Daarna wordt die lijst van eiwitten gefilterd op eiwitten die voorkomen in het originele genoom. Vervolgens worden die eiwitten afgebeeld via LCA op hun consensustaxa in de UniPept taxonomie. Beide resulterende lijsten van taxa worden dan door een LCA\* stap per proteïne geaggregeerd en met elkaar vergeleken.

peptide een lijst van eiwitten oplevert waar die peptide in voorkomt. Met de Ruby `unipept-gem` komen ook de commando's `prot2pept`, die een een proteïne omzet in tryptische peptides, en `peptfilter` die te lange ( $>50$ ) of te korte ( $<5$ ) peptides filtert uit een lijst van opgegeven peptides.

- **Biopython:** Biopython [3] is een Python bibliotheek die een heel aantal tools voor computationele biologie bundelt. Wij zullen gebruik maken van de Entrez module [2]. Deze module zorgt ervoor dat we via de command-line toegang hebben tot de NCBI database om bijvoorbeeld alle eiwitten uit een genoom op te vragen.
- **UniProt mapping:** De UniProt mapping [16] service is een webservice die het vertalen van identifiers van eiwitten van en naar overeenkomstige identifiers uit verschillende databanken vereenvoudigt. Deze worden specifiek gebruikt om de identifiers van proteïne uit NCBI om te zetten naar UniProtKB identifiers.
- **RMQ-bibliotheek:** Een implementatie van een RMQ library [8, 14] in C die voldoet aan de besproken snelheids- en ruimtecomplexiteiten uit Hoofdstuk 2 op pagina 8.
- **Eigen scripts:** Deze scripts kunnen worden onderverdeeld in twee soorten:
  - Een reeks van **wrapper scripts** in Bash en Python die hierboven benoemde tools wrappen. Deze zijn gemaakt om bijvoorbeeld alle eiwitten op te vragen van een NCBI assembly of bijvoorbeeld om het resultaat van de `pept2lca` stap te aggregeren via LCA\*, gebruik makend van de RMQ-bibliotheek.
  - **Ondersteunende scripts** die de volledige taxonomische boom inladen en aan de hand van die data de benodigde arrays voor de preprocessing van RMQ opstellen.

### 3.2.2 Implementatie van de benchmark

In dit gedeelte overlopen we een analysescript in Bash, dat alle stappen van het volledige benchmarkingsproces uitvoert. De enige parameter waarmee dit script moet worden aangeroepen is een GenBank accession number van een assembly. Dit ziet er bijvoorbeeld uit als `GCA_000015425.1`.

```

1 #!/bin/bash
2 asm_id=$1 && shift
```

Aangezien alle scripts verspreid staan over meerdere folders en we ook een aantal (tussentijdse) resultaten wegschrijven, stellen we eerst enkele folders in en maken we die aan waar nodig. De gebruiker kan kiezen om de standaardwaarden van die folders te overschrijven aan de hand van opties van het Bash script.

```

3 # Save directory of the analysis script to know where to find the others
4 dir="$(dirname "$0")"
5 # Create a tmpdir and a datadir
```

```

7 tmpdir=$(mktemp -d -t "$asm_id.XXXXXXXXXX")
8 datadir=$tmpdir
9 rmqdatadir=""
10
11 while getopts "d:t:r:" opt
12 do
13   case $opt in
14     d)
15       datadir="$OPTARG/$asm_id"
16       ;;
17     t)
18       tmpdir="$OPTARG/$asm_id"
19       ;;
20     r)
21       rmqdatadir="-r $OPTARG"
22       ;;
23   ?)
24     usage
25     ;;
26   esac
27 done
28
29 mkdir -p $datadir
30 mkdir -p $tmpdir
31
32 echo "Writing data to $datadir"
33 echo "Writing tempdata to $tmpdir"

```

De eerste stap is het ophalen van de taxon identifier van de assembly. Dit gebeurt aan de hand van het `asm2taxid` script, dat een wrapper is rond de Entrez module in BioPython. Het resultaat hiervan is bijvoorbeeld 400667 voor genoemd GenBank accession number

```

34 # get the taxon ID of the assembly
35 tax_id=$(python3 $dir/../entrez/asm2taxid.py $asm_id)

```

We moeten natuurlijk ook alle peptiden hebben als brondata voor een analyse. Dit doen we aan de hand van het wrapperscript `asm2pept` dat, gegeven een assembly identifier, de bijhorende eiwitten uit de NCBI databank opvraagt en die eiwitten via de commando's `prot2pept` en `peptfilter` omzet naar een lijst van peptiden. Deze peptiden worden gegroepeerd onder het id van hun bijhorende proteïne opgeslagen in een FASTA bestand, genaamd `peptides.fasta`.

```

36 #  get the complete sequence and process it with:
37 #    - prot2pept
38 #    - peptfilter
39
40 echo "Getting peptides"
41 if [ ! -s "$tmpdir/peptides.fasta" ]
42 then
43   echo "No peptides found, downloading."
44   python3 $dir/../entrez/asm2pept.py $asm_id > "$tmpdir/peptides.fasta"
45 fi

```

Eens we de FASTA file van peptiden gegroepeerd hebben onder hun proteïne identifier, kunnen we de eerste toolchain (de taxonomische identificatie van volledige genomen) starten door `pept2lca` op dit bestand uit te voeren. `pept2lca` vraagt aan de UniPept webservices voor elke peptide zijn bijhorende taxonomische identificatie op. Het commando zorgt er ook voor dat alles gegroepeerd blijft onder zijn FASTA header.

De uitvoer hiervan wordt naar het `pept2lca2lca` (vandaar ook de naam van deze toolchain) commando gepiped. Het `pept2lca2lca` script is een wrapperscript rond de RMQ-bibliotheek die voor de groepen van taxa onder een proteïn identifier de LCA\* berekent. Als optioneel argument neemt dit script een taxon identifier. Wanneer deze taxon identifier wordt meegeleverd, voegt het script een extra kolom toe aan de uitvoer die een 1 bevat wanneer resulterende taxon een “match” is en 0 bevat wanneer de resulterende taxon een “mismatch” is.

De uitvoer wordt geschreven naar `pept2lca2lca.fst` wat het eindresultaat is van de eerste toolchain.

```

46 # analyse the complete sequence with and
47 # check whether resulting taxa come from the correct lineage
48 #     - pept2lca2lca
49
50 echo "Executing pept2lca2lca"
51 unipept pept2lca -i "$tmpdir/peptides.fst" \
52     | tee "$datadir/pept2lca.fst" \
53     | python3 $dir/../pept2lca2lca.py -c $tax_id $rmqdatadir \
54     > "$datadir/pept2lca2lca.fst"
```

De tweede toolchain vraagt een extra stap. Eerst worden aan de hand van het wrapper script `asm2seqacc` rond de Entrez module alle identifiers van sequences van een assembly opgevraagd. Deze worden vervolgens gepiped naar de UniProt translation service om de UniProt identifier van deze eiwitten te bekomen (hetgeen nodig is omdat we deze eiwitten later willen filteren).

Wegens een recente aanpassing van UnitProtKB waarbij alle redundante eiwitten zijn verwijderd en samengevoegd [25], is het mogelijk dat de vertaalservice een leeg antwoord teruggeeft. Zonder deze identifiers kunnen we geen filtering doen, en is deze toolchain dan ook overbodig. Dit blijkt het geval te zijn in ongeveer 27% van de assemblies. Voor de gevallen waar het wel slaagt gaan we verder met de toolchain.

```

55 echo "Getting uniprot ids"
56 # get the proteins uniprot ids which occur in the genome
57 if [ ! -s "$tmpdir/uniprot_protein_ids.txt" ]
58 then
59     echo "No uniprot ids found, downloading."
60     python3 $dir/../entrez/asm2seqacc.py $asm_id \
61     | python3 $dir/../entrez/seqacc2protid.py \
62     > "$tmpdir/uniprot_protein_ids.txt"
63
64     # Check if the file is still empty; no translation could be found.
65     if [ ! -s "$tmpdir/uniprot_protein_ids.txt" ]
66     then
```

```

67 |     echo "ERROR: It seems that the uniprot_protein_ids.txt file is still
68 |           empty.
69 | Pept2prot2filter has no use anymore now." >&2
70 |     exit
71 | fi

```

De laatste stap in de `pept2prot2filter2lca` toolchain is het uitvoeren van enkele wrapper scripts. We voeren eerst het `pept2prot` commando van de `uniipept` gem uit om alle eiwitten, behorend bij een peptide op te vragen. Daarna pipen we deze lijst door naar `pept2prot2filter` die op basis van de UniProt identifiers – die in de vorige stap werden opgehaald – de eiwitten uit de lijst filtert die voorkomen in het originele genoom. Als laatste voeren we nog de aggregatiestap uit. In deze stap worden eerst alle taxa behorend bij één enkele peptide geaggregeerd, daarna worden die taxa ook nog eens via het LCA\* algoritme per proteïne identifier geaggregeerd. Ook dit script neemt als parameter de taxon identifier van het origineel genoom om “matches” en “mismatches” te kunnen identificeren. Het resultaat komt terecht in het bestand `pept2prot2filter2lca.fst`.

```

72 | #      - pept2prot2filter2lca
73 | echo "Executing pept2prot2filter2lca"
74 | uniipept pept2prot -i "$tmpdir/peptides.fst" \
75 |   | $dir/../pept2prot2filter.sh "$tmpdir/uniprot_protein_ids.txt" \
76 |   | python3 $dir/../pept2prot2filter2lca.py -c $tax_id $rmqdatadir \
77 |   > "$datadir/pept2prot2filter2lca.fst"

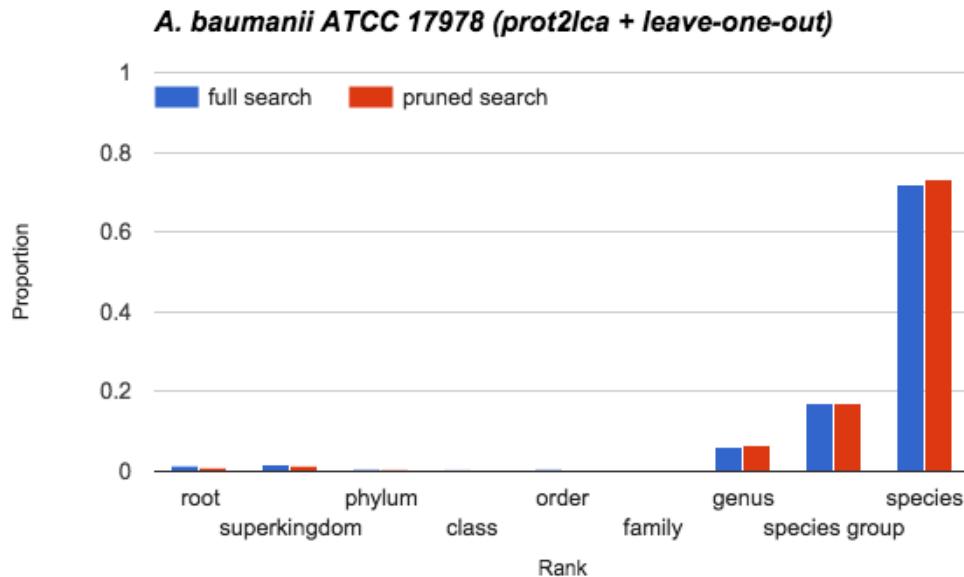
```

### 3.2.3 Snelheidsoptimalisatie

Nu we van beide toolchains een resultaat hebben, kunnen we deze beginnen te analyseren. De laatste stap in de tweede toolchain, en dan voornamelijk het commando `pept2prot` duurt echter heel lang. Voor 3300 eiwitten kan dit al snel enkele uren duren. De reden hiervoor is dat sommige peptiden zeer universeel zijn en dus in heel erg veel eiwitten voorkomen. Het peptide IEELR komt bijvoorbeeld in 162.827 eiwitten voor. Het `pept2prot` commando uitvoeren op 72.888 peptiden (1.1MB) resulteert bijvoorbeeld in een bestand van 64.949.837 lijnen ter grootte van 2.5GB. Dit is niet alleen 2.5GB die wordt gedownload van de UniPept server, maar ook 2.5GB aan data die de UniPept server zelf moet verwerken.

We kunnen dit optimaliseren volgens de volgende observatie. Stel dat we een genoom analyseren met de tweede toolchain waar de peptide IEELR een aantal keer in voorkomt. Dit zal alle 162.827 eiwitten ophalen. Wanneer we de bijhorende taxa bij de eiwitten aggregeren tot een consensustaxon zonder de filterstap zal de consensusclassificatie waarschijnlijk altijd in de minder specifieke rangen liggen, aangezien de peptide zo universeel voorkomt.

Wanneer we de filterstap wel uitvoeren en dus alle eiwitten van het originele genoom uit deze lijst van 162.827 eiwitten filteren, zullen er nog steeds heel veel van deze eiwitten overblijven en zal de aggregatie van hun bijhorende taxa nog steeds een weinig specifiek resultaat opleveren. We kunnen met andere woorden de eiwitten die universeel zijn tijdelijk uit de dataset halen zodat ze niet worden meegerekend in de `pept2prot` stap. Na deze stap



Figuur 3.2: Benchmark om te testen wat het effect is van de snelheidsoptimalisatie beschreven in Sectie 3.2.3 op pagina 34. Bij de “full search” doen we de tweede toolchain met alle eiwitten. “pruned search” laat voor de `pept2prot` stap alle universele peptiden weg en voegt ze na de `pept2prot` stap terug toe.

voegen we de uitgefilterde eiwitten terug bij het eindresultaat en gaan we verder met de toolchain.

Een benchmark van deze methode op de assembly met accession GCF\_000015425.1 voor *Acinetobacter baumannii* ATCC 17978, waarbij alle rangen boven family als universeel aanzien werden, resulteert in een zeer kleine shift van ongeveer 1.5% op van de hogere naar de lagere rangen. Dit wordt geïllustreerd in Figuur 3.2. Het resultaat van `pept2prot` werd hiermee gereduceerd van 64.949.837 naar 21.794.027 lijnen en leverde een tijdwinst van ongeveer 300% op.

Aangezien deze snelheidsoptimalisatie een gigantische tijdwinst oplevert met een minimaal effect op de identificatie, zullen we voor alle benchmarks gebruik maken van deze snelle methode. De nieuwe code voor de tweede toolchain passen we dan aan naar de volgende code. Hier splitsen we het `peptides.fst` op in universele en niet-universele peptiden. Dit onderscheid wordt voor elke peptide gemaakt aan de hand van de rang van zijn consensus-taxon in het `pept2lca.fst` bestand. Daarna voeren we op de gefilterde lijst het `pept2prot` commando uit, waarna we de lijsten terug samenvoegen en verder gaan met de toolchain.

```

72 | #      - pept2prot2filter2lca
73 | echo "Executing pept2prot2filter2lca"
74 |
75 | # Divide peptides in universal and non-universal peptides
76 | cat "$tmpdir/peptides.fst" \
77 |   | python3 $dir/..../commonpeptfilter.py "$datadir/pept2lca.fst"
78 |   | "$tmpdir/peptides.filteredout.fst" > "$tmpdir/peptides.filtered.fst"

```

```

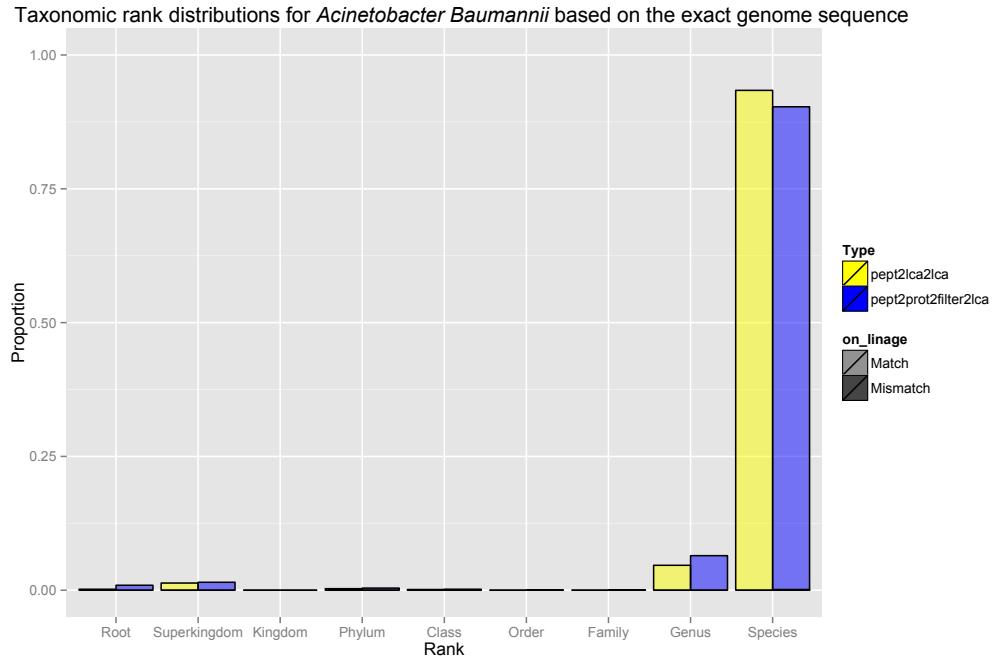
79 # Run pipeline
80 unipept pept2prot -i "$tmpdir/peptides.filtered.fst" \
81 | $dir/../pept2prot2filter.sh "$tmpdir/uniprot_protein_ids.txt" \
82 | {
83   read -r hdr;
84   echo $hdr;
85   sort -m - "$tmpdir/peptides.filteredout.fst" \
86   | python3 $dir/../pept2prot2filter2lca.py -c $tax_id $rmqdatadir \
87   > "$datadir/pept2prot2filter2lca.fst"
88 }
89 }
```

### 3.3 Resultaten

In deze sectie bespreken we de resultaten en beantwoorden we de drie vragen die in het begin van dit hoofdstuk gesteld werden. We bespreken eerst de resultaten van beide toolchains toegepast op het genoom, *Acinetobacter baumannii* ATCC 17978 en bekijken ook het effect van de analyse na het simuleren van reads op dit genoom. Daarna kijken we naar de resultaten van alle 1145 geanalyseerde genomen voor beide toolchains en bestuderen we het effect van de analyse op gesimuleerde onbekende genomen. Als laatste bekijken we de resultaten in een PCA plot en bespreken we enkele outliers.

In Figuur 3.3 op de volgende pagina staan de resultaten op het genoom van *Acinetobacter baumannii* ATCC 17978 van beide toolchains in een barchart weergegeven. Op Tabel 3.1 op de pagina hierna staat de overeenkomstige tabel met cijfers. Hierbij worden de resulterende taxa per rang proportioneel weergegeven. Met de `pept2lca2lca` toolchain worden 93% van de eiwitten op speciesniveau geklassificeerd. De overige 7% wordt voornamelijk op genusniveau gemapt. Wanneer we met `pept2prot2filter2lca` simuleren dat de eiwitten uit *Acinetobacter baumannii* ATCC 17978 niet gekend zijn, slagen we er nog steeds in om ongeveer 88% op speciesniveau te mappen. We zien hier een kleine shift naar links – wat logisch is aangezien we specifieke informatie schrappen. Wat ook opvalt is dat er geen zichtbaar percentage aan mismatches is bij zowel de `pept2lca2lca` toolchain (gele bars) als bij de `pept2prot2filter2lca` toolchain (blauwe bars).

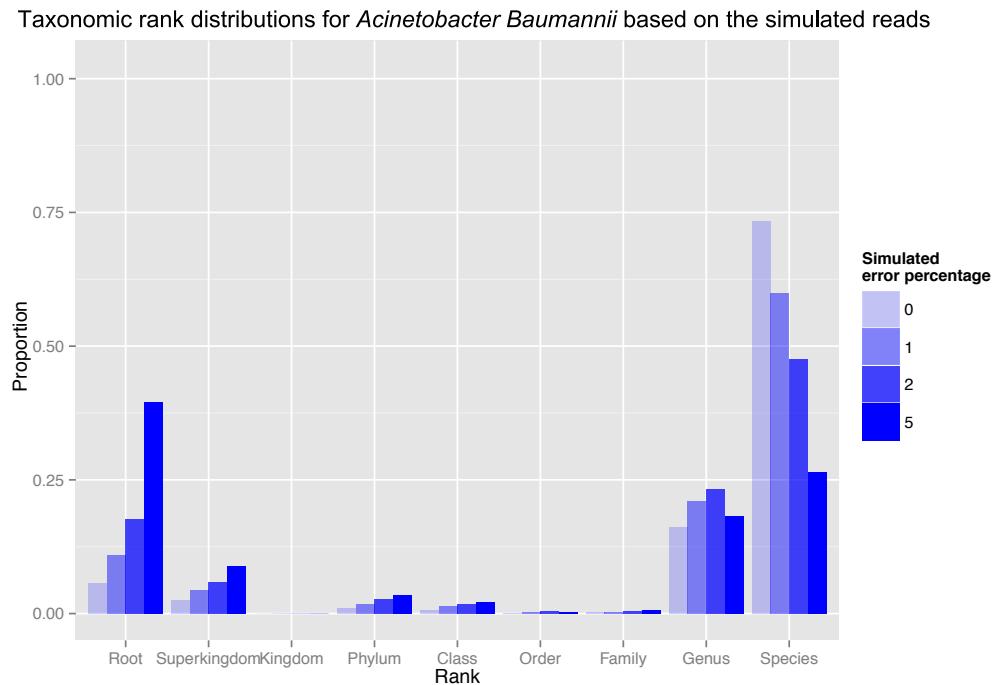
Wanneer we niet rechtstreeks van het genoom *Acinetobacter baumannii* ATCC 17978 vertrekken, maar eerst met wgsim reads van lengte 250 met verschillende foutpercentages (0, 1, 2 en 5 procent) op dat genoom simuleren en daarna met FragGeneScan de eiwitten extraheren, bekomen we de figuur in Figuur 3.4 op pagina 38. We zien een duidelijk verlies aan specifieke informatie wanneer het percentage van read errors omhoog gaat. Met een wgsim foutpercentage van 0% werden ongeveer 73% van de resultaten op speciesniveau gemapt. Dit is zo'n 20% minder dan wanneer alle eiwitten aan het genoom werden ontrokken. De rest van de resultaten is verspreid tussen root, superkingdom en genus waarbij genus het grootste aandeel heeft. Wanneer het foutpercentage toeneemt, zien we steeds een grotere shift naar de minder specifieke identificaties. Met een foutpercentage van 1% wordt nog meer dan de helft gemapt op speciesniveau. Vanaf dit foutpercentage



Figuur 3.3: Resultaat van het uitvoeren van beide toolchains op het genoom *Acinetobacter baumannii* ATCC 17978. Hierbij worden de resulterende taxa per rang proportioneel weergegeven.

Tabel 3.1: Resultaat van het uitvoeren van beide toolchains op het genoom *Acinetobacter baumannii* ATCC 17978. Hierbij worden de resulterende taxa per rang proportioneel weergegeven.

Rang	pept2lca2lca		pept2prot2lca	
	match	mismatch	match	mismatch
Root	7	0	35	0
Superkingdom	51	0	56	0
Kingdom	0	0	0	0
Phylum	11	0	15	0
Class	5	0	7	0
Order	0	0	1	0
Family	0	0	2	0
Genus	177	0	243	0
Species	3.548	0	3.399	5



Figuur 3.4: Overzicht van de taxonomische identificatie van resultaten van de `pept2lca2lca` toolchain wanneer deze wordt uitgevoerd op reads gesimuleerd door wgsim met verschillende foutpercentages.

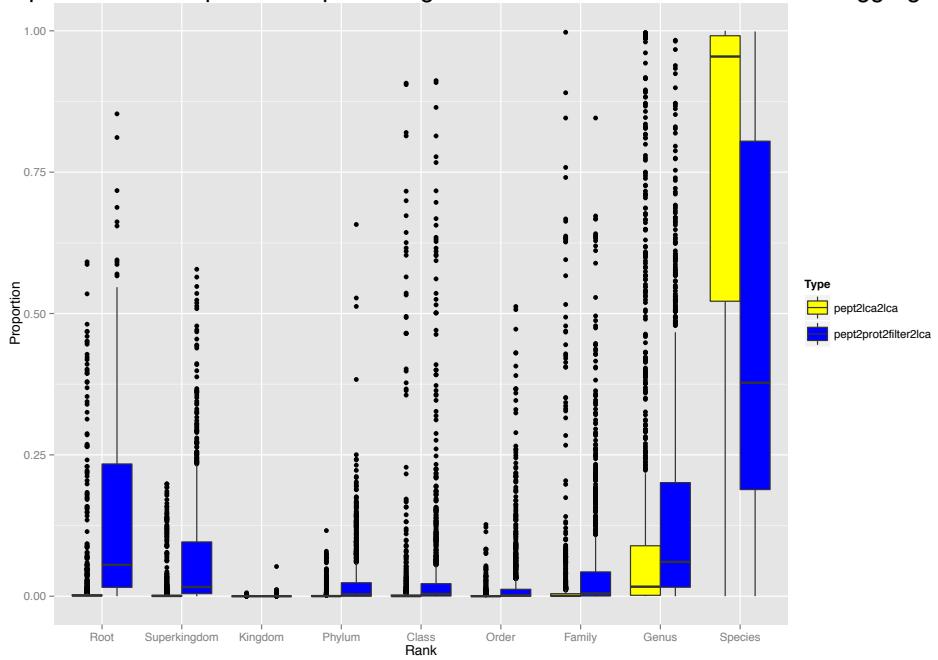
toeneemt naar 2% is dit al niet het geval meer. Bij foutpercentage 5% wordt het grootste aandeel zelfs op het rootniveau gemapt.

Om een globaal overzicht te krijgen van de accuraatheid van de taxonomische identificatie met de UMAP, krijgen we het beste resultaat met Figuur 3.5 op de pagina hierna. Deze figuur toont per rang voor beide toolchains een boxplot van de geaggregeerde resultaten voor 1145 genomen. We zien dat voor de `pept2lca2lca` toolchain bijna 97% van de resultaten op species niveau wordt gemapt, de overige 3% wordt volledig op genus niveau gemapt. Bij de `pept2prot2lca2lca` toolchain is dit ongeveer 38% op speciesniveau. De overige percentages worden voornamelijk verspreid tussen root, superkingdom en genus.

Een laatste, interessante plot is de PCA plot. PCA staat voor “Principal component analysis” en probeert de verschillen in de invoerdata te herleiden tot een aantal dimensies, 3 in dit geval, om een visuele voorstelling te kunnen maken van het verschil van de resultaten. Het resultaat hiervan, met als invoerdata de resultaten van `pept2lca2lca`, is te zien in Figuur 3.6 op pagina 40. We zien dat de meeste resultaten langs een as geclusterd worden die rechts in het midden start en lichtjes naar achter zakt. Vanuit dit punt in het rechts midden vertrekt ook een, weliswaar minder druk bevolkte, as naar de hoek rechts achteraan aan de bovenkant. In de volledige kubus lijken er ook enkele “verdwijnende” punten te zijn.

Links en rechtsboven van de PCA plot worden de barcharts van drie interessante outliers getoond. We zien duidelijk dat de twee outliers links van de figuur een ongewone verdeling vertonen van resultaten op de taxonomische ranks. De barplot linksboven lijkt

Per rank boxplot distribution plot of the percentage of taxonomic identifications for the aggregated results



Figuur 3.5: Deze figuur toont per rang voor beide toolchains een boxplot van de geaggregeerde resultaten voor 1145 genomen.

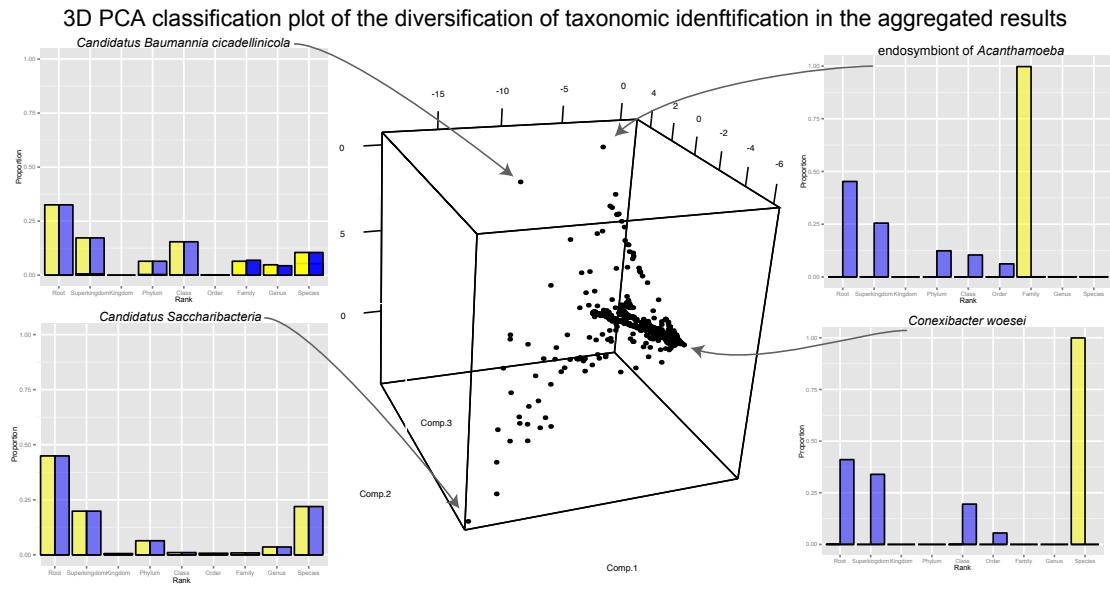
zelfs aan te geven dat alles onder orderniveau “mismatches” zijn en dus op een verkeerde tak zijn geïdentificeerd. Rechts bovenaan van de PCA plot wordt een outlier getoond van een endosymbiont van *Acanthamoeba*. Dit is blijkbaar een organisme dat volledig op familiyniveau gemapt wordt en, aangezien we helemaal geen hits krijgen op familiyniveau met de tweede toolchain, enkele zeer specifieke eiwitten bevat. Rechtsonder de PCA plot wordt een vergelijkbare verdeling getoond, maar in dit geval wordt het organisme met de *pept2lca2lca* toolchain wel volledig op species gemapt wordt.

## 3.4 Toekomstig werk

Zoals vermeld bij de besprekking van de PCA plot in de resultaten zijn er een heel aantal outliers te zien met hun bijhorend staafdiagram. Het zou zeer interessant zijn om te analyseren waarom deze bepaalde genomen dat specifieke resultaat hebben om zo eventueel verkeerd geklassificeerde organismes te ontdekken.

Momenteel zijn er ongeveer 2200 volledig gesequeneerde genomen onderzocht, waarbij bij ongeveer 600 enkel de eerste toolchain werd uitgevoerd wegens de beschreven problemen met de mapping tussen NCBI protein identifiers en UniProt identifiers. Er zijn echter ruim 4000 van dit soort genomen. Het zou dus zeker interessant zijn om alle genomen via de UniPept Metagenomics Analysis Pipeline te analyseren.

De gebruikte methode kan ook resistenter gemaakt worden tegen read errors door de



Figuur 3.6: PCA plot van de resultaten van de `pept2lca2lca` toolchain met omliggende enkele interessante outliers.

eiwitfragmenten op te delen in k-mers en daarvan telkens de LCA te berekenen, zoals beschreven in [27].

# Hoofdstuk 4

## Uitbreidingen van de UniPept command-line interface

De UniPept command-line interface[26] (CLI) is een verzameling van tools geschreven door Toon Willems gedurende zijn masterthesis in het academiejaar 2013-2014. Deze interface is ontwikkeld in Ruby en maakt gebruik van de UniPept HTTP API.[23].

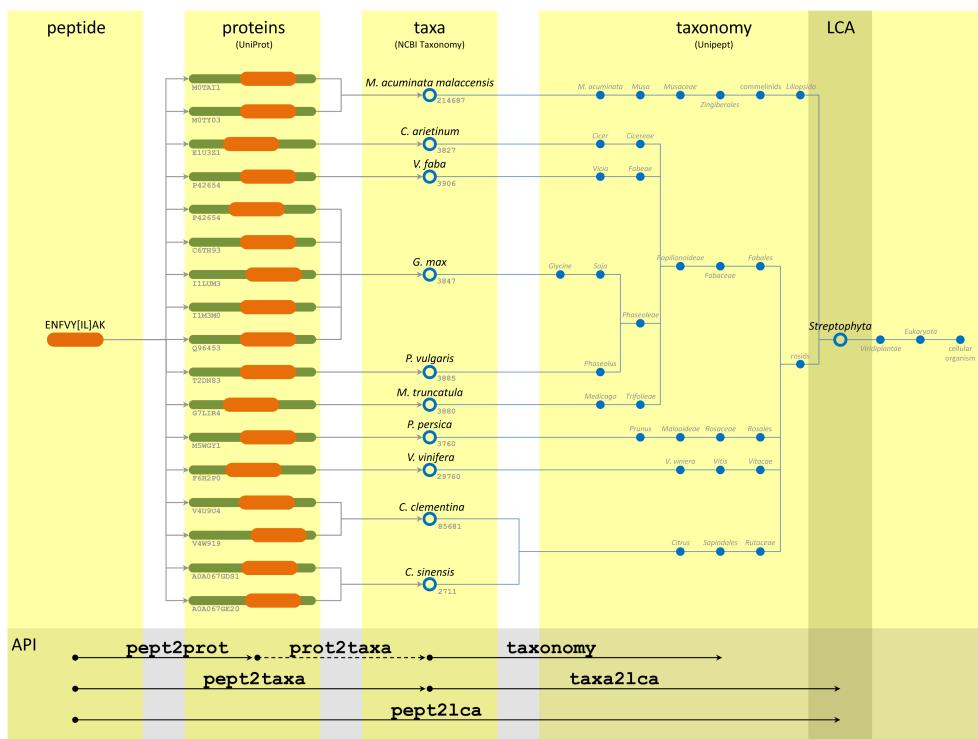
Gedurende de case study in Hoofdstuk 3 werd de CLI zeer intensief gebruikt. Door dit intensief gebruik zijn enkele fouten en problemen naar boven gekomen. In dit hoofdstuk bespreken we, na een korte inleiding tot de CLI, de oorzaak van die problemen en geven we ook aan hoe we ze hebben opgelost.

### 4.1 Inleiding tot de UniPept CLI

De UniPept command-line interface wrapt de vijf basismethodes van de UniPept webservices[23] in een command-line interface. De werking van deze commando's staat geïllustreerd in Figuur 4.1 op pagina 42.

- **pept2prot:** geeft een lijst van UniProt entries die een opgegeven peptide bevatten;
- **pept2taxa:** geeft een lijst van taxa geëxtraheerd uit UniProt entries die de opgegeven peptide bevatten;
- **pept2lca:** geeft de taxonomische lowest common ancestor voor een gegeven tryptische peptide;
- **taxa2lca:** geeft de taxonomische lowest common ancestor voor een gegeven lijst van taxon identifiers;
- **taxonomy:** geeft taxonomische informatie voor een gegeven taxon identifier.

Voor de CLI kan gebruikt worden, moet deze eerst lokaal geïnstalleerd worden. Aangezien alle commando's geïmplementeerd zijn in Ruby moet eerst een Ruby environment worden



Figuur 4.1: Illustratie van de vijf basiscommando's aangeboden door de UniPept webservices.

geïnstalleerd. De CLI kan daarna eenvoudig geïnstalleerd worden met het commando `gem`, de interne Ruby Package manager. De broncode van de CLI kan gevonden worden op de publieke UniPept-cli repository, <https://github.com/unipept/unipept-cli>.

```

1 $ gem install unipept
2 Successfully installed unipept-0.6.4
3 1 gem installed
4 Installing ri documentation for unipept-0.6.4...
5 Installing RDoc documentation for unipept-0.6.4...

```

Na de installatie van de UniPept gem is het `unipept`-commando beschikbaar. De namen van de methodes van de webservices komen overeen met de naam van de subcommandos van het UniPept CLI-commando. De helpfuncties en extra opties kunnen dan worden opgevraagd via de `--help` vlag:

```

1 $ unipept --help
2
3 COMMANDS
4 config
5 help      show help
6 pept2lca Give lowest common ancestor for given peptide
7 pept2prot Give protein information for given peptides
8 pept2taxa Single Peptide Search
9 taxa2lca Give lowest common ancestor for taxon ids
10 taxonomy Give NCBI taxonomy information on given input taxon ids

```

```

11
12 OPTIONS
13   -f --format      output format (available: json,csv,xml) (default: csv)
14   -h --help        show help for this command
15   --host          Override host setting
16   -i --input       input file
17   -o --output      output file
18   -q --quiet       don't show update messages
19   -v --version     print version

```

Zoals eerder vermeld steunt de CLI op de UniPept webservices. Vooraleer de commando's kunnen worden uitgevoerd moet dus een API server worden opgegeven via de `--host` vlag. Deze kan ook als standaardserver worden ingesteld via het `config` subcommando, zodat de `--host` vlag niet telkens opnieuw hoeft worden te meegegeven.

```

1 $ unipept --host 'api.unipept.ugent.be' pept2lca ENFVYIAK
2 peptide,taxon_id,taxon_name,taxon_rank
3 ENFVYIAK,35493,Streptophyta,phylum
4 $ unipept config host='api.unipept.ugent.be'
5 $ unipept pept2lca ENFVYIAK
6 peptide,taxon_id,taxon_name,taxon_rank
7 ENFVYIAK,35493,Streptophyta,phylum

```

Het voorgaande voorbeeld toont hoe het `unipept pept2lca`-commando wordt aangeroeopen voor de peptide ENFVYIAK. We kunnen ook alle eiwitten voor deze peptide opvragen via `unipept pept2prot`:

```

1 $ unipept pept2prot MVNENTR
2 peptide,uniprot_id,taxon_id
3 ENFVYIAK,Q96453,3847
4 ENFVYIAK,C6TH93,3847
5 ENFVYIAK,P42654,3906
6 ENFVYIAK,M0TY03,214687
7 ENFVYIAK,A0A067GDS1,2711
8 ENFVYIAK,C6TM63,3847
9 ...

```

## 4.2 Verwerken van FASTA files

Een recente toevoeging aan de commandolijn is de verwerking van FASTA files. Door deze toevoeging kunnen FASTA bestanden meegegeven worden als invoerbestand aan de commando's. Bij de resultaten worden de resultaten dan ook gegroepeerd onder hun overeenkomstige FASTA header. Tijdens de case study uit vorig hoofdstuk bleek dat FASTA files niet altijd correct verwerkt werden. Wanneer we dezelfde peptide toevoegen onder verschillende FASTA headers, kregen we in de output steeds enkel de laatst voorgekomen FASTA header te zien. Ook de eerdere FASTA headers werden bij die peptides vervangen door de laatste. Volgend voorbeeld illustreert het probleem:

```
1 | $ cat data/IITHPNFNGNTLDNDIMLIK
```

```

2 |>a|1
3 |IITHPNFNGNTLDNDIMLIK
4 |>b|2
5 |IITHPNFNGNTLDNDIMLIK
6 |
7 |$ unipept pept2lca -i data/IITHPNFNGNTLDNDIMLIK
8 |fasta_header,peptide,taxon_id,taxon_name,taxon_rank
9 |>b|2,IITHPNFNGNTLDNDIMLIK,9823,Sus scrofa,species
10 |>b|2,IITHPNFNGNTLDNDIMLIK,9823,Sus scrofa,species
11 |
12 |$ unipept pept2prot -i data/IITHPNFNGNTLDNDIMLIK
13 |fasta_header,peptide,uniprot_id,taxon_id
14 |>b|2,IITHPNFNGNTLDNDIMLIK,P00761,9823
15 |>b|2,IITHPNFNGNTLDNDIMLIK,C5IWV5,9823
16 |>b|2,IITHPNFNGNTLDNDIMLIK,F1SRS2,9823
17 |>b|2,IITHPNFNGNTLDNDIMLIK,P00761,9823
18 |>b|2,IITHPNFNGNTLDNDIMLIK,C5IWV5,9823
19 |>b|2,IITHPNFNGNTLDNDIMLIK,F1SRS2,9823

```

We zien dat de fasta header **b|2** wordt toegevoegd voor alle resultaten van de peptide **IITHPNFNGNTLDNDIMLIK**, ook wanneer de **a|1** de correcte header is, komt er header **b|2** te staan.

De oorzaak van het probleem lag bij de mapping van de FASTA headers. Het verwerken van het FASTA gedeelte gebeurt volledig aan de kant van de gebruiker. De server is dus nooit op de hoogte van het bestaan van de FASTA headers. Daarom moeten de command-line tools zelf een mapping bijhouden van peptides op FASTA headers. Daarna worden de peptides zonder headers naar de server gestuurd en worden de fasta headers bij het antwoord van de server terug op de peptides gemapt.

De onderstaande code illustreert hoe deze mapping gebeurt:

```

1 |# Input: een fastafайл, genaamd 'input' en een 'block'
2 |
3 |# Slaat de eerste header op in een tijdelijke variabele
4 |fasta_header = input.next
5 |
6 |# Breekt de invoer op in stukken (= batches) van 200
7 |input.slice(200) do |batch|
8 |  # Maak een hash aan die binnen een batch een peptide op een fasta
9 |  # header mapt
10 |  fasta_mapper = {}
11 |
12 |  # Itereert over de lijnen in 1 batch
13 |  j = 0
14 |  while j < batch.size
15 |    # Als de huidige lijn start met een > is deze een fasta header
16 |    if batch[j].start_with? '>'
17 |      # Overschrijf de fasta_header variabele met de huidige fasta header
18 |      fasta_header = batch[j]
19 |    else
20 |      # Voeg de fasta header toe aan de map
        fasta_mapper[batch[j]] = fasta_header

```

```

21     end
22     j += 1
23 end
24
25 # Verwijder de fasta headers uit de batch
26 batch -= fasta_mapper.values.uniq
27
28 # Stuur de batch zonder fasta headers naar de API ter verwerking
29 block.call(batch, fasta_mapper)
30 end

```

De oorzaak van het probleem lag op regel 9 van bovenstaande code. Door één enkele FASTA header op één peptide te mappen wordt de FASTA header overschreven wanneer er meerdere gelijke peptides binnen één batch aanwezig zijn. Als we er het voorbeeld van hierboven bijnemen, dan bevat de `fasta_mapper` na het overlopen van de eerste FASTA groep onder header `a|1` een mapping van '`IITHPNFNGNTLDNDIMLIK`' => '`a|1`', maar wanneer de tweede FASTA groep gelezen is, wordt de waarde overschreven naar `b|2`.

Bij het uitschrijven werd er per peptide die de API die als resultaat door de API wordt teruggegeven, een mapping gemaakt van peptide naar FASTA header. Aangezien de FASTA headers van peptides in bepaalde gevallen overschreden werden, werd de mapping soms fout gemaakt.

Om dit probleem op te lossen zouden we een mapping kunnen bijhouden van één peptide op meerdere fasta headers. Bij de output kunnen we dan berekenen welke header we moeten nemen afhankelijk van het aantal keer die peptide al is uitgeschreven. Na implementatie bleek de uitvoeringstijd echter veel trager (grootteorde 20) en veel meer belastend voor zowel de processor als het geheugen dan gewenst (ook al was de uitvoer correct).

Het mappen van peptides op hun FASTA header is dus geen goede oplossing voor genoemd probleem. In plaats van te werken met dit soort mapping kunnen we ook een lijst bijhouden van een inputpaar, bestaande uit FASTA header en peptide. De code om dit te bereiken ziet er als volgt uit:

```

1 # Input: een fastafie, genaamd 'input' en een 'block'
2
3 # Slaat de eerste header op in een tijdelijke variabele
4 fasta_header = input.next
5
6 # Breekt de invoer op in stukken (= batches) van 200
7 input.slice(200).do |batch|
8   # Maak een hash aan die binnen een batch een peptide op een fasta
     header mapt
9   fasta_input = []
10
11  # Gebruik een Set om de peptides op te slaan zodat we niet dezelfde
     peptide 2
12  # keer naar de server sturen
13  newsub = Set.new
14
15  # Itereert over de lijnen in 1 batch
16  sub.each do |s|

```

```

17   s.chomp!
18   if s.start_with? '>'
19     # Sla de huidige fasta_header op in een tijdelijke variabele
20     fasta_header = s
21   else
22     # Voeg het inputpaar toe aan de lijst
23     fasta_input << [fasta_header, s]
24     newsub << s
25   end
26 end
27
28 # Stuur de batch zonder fasta headers naar de API ter verwerking
29 block.call(newsub, fasta_input, ...)
30 end

```

Op deze manier hebben we (opnieuw aan de hand van het voorbeeld) de volgende variabelen: `newsub` die één element bevat: `IITHPNFNGNTLDNDIMLIK`, en `fasta_input` die de volgende lijst bevat: `[(a|1, IITHPNFNGNTLDNDIMLIK), (b|2, IITHPNFNGNTLDNDIMLIK)]`. `newsub` wordt dan naar de server gestuurd ter analyse.

Om de mapping terug te maken van peptide naar fasta header bekijken we eerst de structuur van resultaten voor de peptide. Commando's zoals `pept2lca` geven per invoerpeptide één resultaat terug. Commando's zoals `pept2prot` kunnen echter meerdere resultaten teruggeven per peptide:

```

1 $ curl 'api.unipept.ugent.be/api/v1/pept2lca.json?input[]=
2   IITHPNFNGNTLDNDIMLIK'
3 [
4   {
5     "peptide": "IITHPNFNGNTLDNDIMLIK",
6     "taxon_id": 9823,
7     "taxon_name": "Sus scrofa",
8     "taxon_rank": "species"
9   }
10 ]
11 $ curl 'api.unipept.ugent.be/api/v1/pept2prot.json?input[]=
12   IITHPNFNGNTLDNDIMLIK'
13 [
14   {
15     "peptide": "IITHPNFNGNTLDNDIMLIK",
16     "uniprot_id": "P00761",
17     "taxon_id": 9823
18   },
19   {
20     "peptide": "IITHPNFNGNTLDNDIMLIK",
21     "uniprot_id": "F1SRS2",
22     "taxon_id": 9823
23   },
24   {
25     "peptide": "IITHPNFNGNTLDNDIMLIK",
26     "uniprot_id": "C5IWV5",
27   }
28 ]

```

```

26     "taxon_id": 9823
27   }
28 ]

```

Ook bij het uitschrijven voeren we een verandering door. In de oude code werd de lijst van resultaten overlopen en werd op elk resultaat een FASTA header gemapt. Met de nieuwe code is dit niet langer mogelijk aangezien we de peptides die binnen één batch meerdere keren voorkomen maar één keer naar de server sturen. We moeten hier dus de `fasta_input` lijst overlopen en daar de resultaten op mappen. Dit gebeurt als volgt:

```

1 # Hervorm de resultaten van [{key1: value1, key2: value2, ...}]
2 # naar {value1 => {key1: value1, key2: value2, ...}}
3 data_dict = {}
4 data.each do |d|
5   data_dict[d.values.first.to_s] ||= []
6   data_dict[d.values.first.to_s] << d
7 end
8
9 # Itereer over de invoer
10 fasta_input.each do |input_pair|
11   fasta_header, id = input_pair
12
13 # Haal het resultaat voor het id op (indien die er is)
14 unless data_dict[id].nil?
15   data_dict[id].each do |r|
16     csv << ([fasta_header] + r.values).map { |v| v == "" ? nil : v }
17   end
18 end
19 end

```

Aangezien één peptide meerdere resultaten kan hebben, moeten we de lijst van de resultaten eerst hervormen, zoals gebeurt in regel 3 tot 7 van bovenstaande code. Dit is een preprocessing stap die ervoor zorgt dat we de bijhorende resultaten van een peptide eenvoudig op kunnen halen zonder telkens de hele resultatenlijst af te moeten lopen. Op regel 14 doen we ook nog een extra check of er wel degelijk een resultaat is voor de ingevoerde peptide. Het is namelijk mogelijk dat een peptide geen resultaat oplevert omdat hij niet gekend is door UniPept.

Deze aanpak heeft (naast correcte resultaten) een aantal voordelen ten opzichte van de oude. Zo worden peptiden die meerdere keren voorkomen binnen een batch maar één keer naar de server gestuurd ter analyse. Een tweede voordeel is dat hiermee een probleem met het `taxonomy`-commando is opgelost. De API geeft bij dit commando namelijk maar één resultaat terug wanneer hetzelfde taxon vaker worden verstuurd. Aangezien de resultaten nu worden overlopen aan de hand van de invoer en niet aan de hand van de resultaten zelf zal de CLI altijd evenveel resultaten opleveren als er invoerlijnen zijn.

De vraag is natuurlijk of de code nog steeds even performant is als (of performanter dan) de oude code, en niet dezelfde problemen vertoont als de vorige oplossing. Dit onderzoeken we in volgende sectie, waar we ook meteen een onderzoek voeren naar de optimale `batch_size` parameters voor de verschillende methoden.

## 4.3 Bepalen van de optimale batch size voor de UniPept CLI

De CLI stuurt niet alle data in één keer door naar de API. Het zou namelijk problemen opleveren als er meer data zou worden doorgestuurd dan de server in één keer kan verwerken, met fouten in de server tot gevolg. We moeten ook in het achterhoofd houden dat alle data over het netwerk verstuurd moet worden en dat er ook nog verwerking nodig is langs de kant van de client. De oplossing hiervoor is om de invoerdata op te delen in stukken, en die stuk voor de stuk naar de server te sturen om die sequentieel te verwerken. Zo wordt het netwerk niet overbelast door grote hoeveelheden data en moeten de server en client niet alle data in één keer bijhouden en verwerken. De grootte van de blokken (of batches) waarin de data wordt opgedeeld, wordt **batch size**, of de batchgrootte genoemd.

Verschillende commando's hebben ook verschillende verwerkingstijden en verschillende groottes van resultaten. Het **pept2lca** commando geeft bijvoorbeeld hoogstens één resultaat per peptide, maar voor diezelfde peptide kan **pept2prot** duizenden resultaten teruggeven.

Het komt er dus op neer om een optimale batchgrootte te vinden die voor een balans zorgt tussen de data die over het netwerk verstuurd kan worden, de data de server kan verwerken en de data die de client kan verwerken. De parameter moet daarenboven ook commando-afhankelijk zijn.

Om die parameter te bepalen werden enkele tijds- en CPU-metingen, waarbij de batchgrootte werd gevarieerd. Voor commando's die één resultaat per peptide teruggeven, werd het commando **pept2lca** gebruikt met batchgroottes 10, 100, 1000 en 10000. Voor commando's die meerdere resultaten teruggeven, werd **pept2prot** gebruikt met batchgroottes 5 en 10. Alle testen werden 5 keer herhaald per batchgrootte. Als invoer werd een dataset gebruikt die ongeveer 72000 peptides bevat.

Aangezien we, zoals beschreven in vorige sectie, wijzigingen hebben aangebracht in de command-line code is het interessant om de benchmarks ook op de oude code te laten lopen en zo in te schatten of de nieuwe code nog even performant is. De resultaten van deze benchmarks zijn te vinden in Tabel 4.1 op pagina 49. Uit deze tests kunnen we besluiten dat een grotere batchgrootte niet altijd beter is.

Bij het **pept2lca** commando zakt de uitvoeringstijd steeds naarmate de batchgrootte groter wordt, tot en met een batchgrootte van 1000. Vanaf die batchgrootte stijgt de uitvoeringstijd weer. De CPU-belasting neemt wel toe naarmate de batchgrootte groter wordt. De optimale batchgrootte zou hier dus 1000 zijn.

Bij de benchmarks voor **pept2prot** zien we een minimum uitvoeringstijd bij batchgrootte 5. Het CPU-gebruik ligt in alle gevallen tegen het maximum.

Als laatste zien we dat de nieuwe code voor FASTA files op alle vlakken zeer gelijkaardig presteert als de oude code. Bovendien is de nieuwe code ook volledig correct.

In de toekomst kan het interessant zijn om de batchgrootte te laten afhangen van meer parameters dan enkel het subcommando. Als de server druk bezet is, zou de batchgrootte bijvoorbeeld kleiner gemaakt kunnen worden. Het vragen van extra kolommen in het resultaat via de **--extra** vlag zou ook een invloed kunnen uitoefenen op de batchgrootte.

Tabel 4.1: Benchmarking van `pept2lca` met verschillende batchgroottes

(a) Oude code			(b) Nieuwe code		
Batchgrootte	Tijd (s)	CPU	Batchgrootte	Tijd (s)	CPU
10	8.9184	66%	10	8.6656	69%
100	3.4326	72%	100	3.4776	70%
1000	2.7184	81%	1000	2.9128	81%
10000	4.551	87%	10000	5.0994	82%

Tabel 4.2: Benchmarking van `pept2prot` met verschillende batchgroottes

(a) Oude code			(b) Nieuwe code		
Batchgrootte	Tijd (u)	CPU	Batchgrootte	Tijd (u)	CPU
2	6:24.24	98%	2	6:35.29	99%
5	5:28.35	99%	5	5:34.94	99%
10	7:35.05	99%	10	7:27.31	99%

## 4.4 Memory leak

Een tweede probleem dat naar boven is gekomen was dat de CLI commando's soms onnodig veel geheugen gebruikten, wat duidde op een memory leak. Na enkele experimenten leek het alsof de memory leak enkel optrad wanneer er zich een API-fout voordeed. Om dit te reproduceren in een stabiele omgeving werd een eigen API-server opgezet en werd de code van de API aangepast zodat elke request 1% kans had op falen. Daarna werd een grote FASTA file naar die API-server gestuurd en, inderdaad, het geheugengebruik bleef vrij stabiel tot wanneer er een API-fout optrad. Daarna schoot het geheugengebruik de hoogte in. Wat ook merkwaardig was, was dat niets nog werd uitgeschreven naar de output van zodra er één API-fout was opgetreden.

Door te debuggen werd het volgende gedrag geobserveerd. Hierbij werden telkens 5 requests met `batch_size` 2 uitgevoerd waarbij de API 20% kans had op falen.

```

1 $ unipept pept2prot -i peptides.10.fst
2 received a non-successful http response 500 API request failed! log can
3 be found in ~/.unipept/unipept-2015-04-20-10:52:19.log
4
5 $ unipept pept2prot -i peptides.10.fst
6 Writing header
7 Writing 0
8 Writing 1
9 Writing 2
10 Writing 3
11 Writing 4
12

```

```

13 $ unipept pept2prot -i peptides.10.fst
14 received a non-successful http response 500 API request failed! log can
15 be found in ~/.unipept/unipept-2015-04-20-11:06:56.log
16 Writing header
17 Writing 0
18 Writing 1

```

Bij de eerste test faalde de eerste request. Opeenvolgende requests werden niet meer uitgeschreven, al bleef het geheugengebruik toenemen wat deed vermoeden dat ze wel in het geheugen werden opgeslagen. De tweede test slaagde zonder enig probleem. Hierbij bleef het geheugengebruik consistent en laag. Bij de derde test faalde de tweede request, waarna het geheugen ook bleef toenemen.

De oorzaak van de memory leak bleek uiteindelijk te liggen aan de `batch_order` klasse. Deze klasse is verantwoordelijk voor het behouden van de volgorde van de batches. De volgorde binnen de batches wordt behouden aan de server kant, maar we hebben geen garantie dat de batches die in parallel naar de server worden gestuurd ook in de volgorde van versturen terug aankomen. Deze klasse bevat twee variabelen: een getal dat de index van de huidige batch bijhoudt en daarnaast ook `order-wachtrij`, een map waarbij indexen op batches gemapt worden. Wanneer deze klasse wordt geïnitialiseerd is de huidige index 0, en is de lijst van batches leeg. Wanneer de methode `wait` van deze klasse opgeroepen wordt met batch en zijn index, wordt gecontroleerd of die index gelijk is aan de huidige index in de klasse. We controleren met andere woorden of het de beurt is aan die batch om uitgeschreven te worden. Wanneer een batch wordt uitgeschreven, wordt de interne index verhoogd, en wordt nagegaan of er verder in de `order-wachtrij` nog blokken klaarstaan die uitgeschreven kunnen worden. Wanneer de `wait`-methode wordt opgeroepen met een index die niet gelijk is aan de interne index, wordt de batch in de wachtrij geplaatst.

Als we naar de code kijken die verantwoordelijk is voor het oproepen van de `batch-order` functie zien we het volgende:

```

1 request.on_complete do |resp|
2   if resp.timed_out?
3     $stderr.puts "request timed out, continuing anyway, but results
4     might be incomplete"
5   else
6     if resp.success?
7       # Parse het antwoord
8       ...
9
10      # Wacht tot het de beurt is aan die batch om uitgeschreven te
11      # worden
12      batch_order.wait(i) do
13        # Schrijf batch uit
14        ...
15      end
16    else
17      save_error(resp.response_body)
18    end
end

```

19 | **end**

De `wait`-methode wordt opgeroepen bij een succesvol antwoord van de API-server maar wanneer er een fout optreedt, wordt de methode niet opgeroepen. Dit heeft als gevolg dat er in de wachtrij van de `batch_order` klasse een gat ontstaat.

Stel bijvoorbeeld dat batches 1, 2, 4 en 5 meteen succesvol worden beantwoord, dan zullen batch 1 en 2 meteen worden uitgeschreven. Batches 4 en 5 worden in dat geval in de wachtrij geplaatst aangezien batch 3 nog niet is teruggekeerd van de server. Als batch 3 na een bepaalde tijd nog steeds geen antwoord heeft gekregen, dan zal de Typheous library een time-out geven en wordt enkel een foutbericht uitgeschreven.

Dit veroorzaakt een “gat” in de wachtrij aangezien de `wait`-methode niet wordt opgeroepen voor deze derde batch. Als gevolg zullen batches die volgen op een falende batch, worden opgeslagen in de wachtrij van de `batch_order`-klasse. Dit zorgt er niet alleen voor dat deze blokken in het geheugen blijven zitten, maar ook dat de succesvolle resultaten nooit worden uitgeschreven. Aangezien commando’s zoals `pept2prot` voor grote invoerbestanden een resultaat kunnen opleveren met bestandsgroottes in de gigabytes is dit problematisch.

De oplossing voor dit probleem is vrij eenvoudig: door ook de `wait`-methode op te roepen wanneer er een fout optreedt, zorgen we ervoor dat het gat in de wachtrij wordt opgevuld, zodat eventuele batches die staan te wachten om te worden uitgeschreven ook werkelijk uitgeschreven worden.

Het resultaat hiervan is in onderstaand voorbeeld te bekijken:

```
1 | $ unipept pept2prot -i peptides.10.fst
2 | Writing header
3 | Writing 0
4 | Writing 1
5 | Writing 2
6 | Writing 3
7 | received a non-successful http response 500 API request failed! log can
8 | be found in ~/.unipept/unipept-2015-04-20-11:18:48.log
9 | $ unipept pept2prot -i peptides.10.fst
10 | Writing header
11 | Writing 0
12 | Writing 1
13 | Writing 2
14 | Writing 3
15 | Writing 4
16 | $ unipept pept2prot -i peptides.10.fst
17 | Writing header
18 | Writing 0
19 | Writing 1
20 | received a non-successful http response 500 API request failed! log can
21 | be found in ~/.unipept/unipept-2015-04-20-11:29:09.log
22 | Writing 3
23 | Writing 4
```

We zien dat wanneer een API-fout optreedt, batches na de batch waarbij de fout is opgetreden wel worden uitgeschreven. Bij het laatste commando in bovenstaande voorbeel-

sessie zien we bijvoorbeeld dat de API een fout geeft bij de tweede request. Hierdoor wordt batch 2 niet uitgeschreven, maar daarna worden batch 3 en 4 wel succesvol uitgeschreven.

In een ideale wereld zou de API geen fouten mogen genereren, maar door de grote hoeveelheid data die door de server verwerkt wordt en over het netwerk wordt verstuurd, is het onmogelijk om fouten te vermijden. We kunnen wel een retry-strategie inbouwen bij fouten. Dit wordt besproken in de volgende sectie.

## 4.5 Retry-strategie bij fouten

API-fouten zijn helaas onvermijdbaar en hebben vaak zeer uiteenlopende oorzaken. We sturen ten eerste data over het internet. Hierbij kan het voorkomen dat de internetconnectie van de client (of zelfs van de server) even wegvalt, waardoor er datapaketten verloren gaan en nooit beantwoord worden. Ten tweede werken we met grote hoeveelheden data. Wanneer de server overspoeld wordt door aanvragen kan het zijn dat zijn resources ver zadigd raken en de server geen nieuwe aanvragen meer kan verwerken.

Aangezien fouten nooit volledig kunnen uitgesloten worden, moeten we dit aan de clientzijde proberen te remediëren. Momenteel wordt op de Typhoeus bibliotheek gesteund om de batches door te sturen naar de API-server ter verwerking. Typhoeus is een wrapper rond libcurl die het maken van HTTPS request, en vooral van het paralleliseren van requests, een stuk gemakkelijker maakt[22]. De Hydra klasse binnen Typhoeus is verantwoordelijk voor de parallelisatie van requests. Het doorsturen van batches werkt als volgt:

Broncode 4.1: Ruby code verantwoordelijk voor het sturen van requests naar de server en het verwerken van de resultaten

```

1 # Maak een nieuwe 'Hydra' die maximum 10 requests in parallel laat lopen
2 hydra = Typhoeus::Hydra.new(max_concurrency: 10)
3 num_req = 0
4
5 # Itereer over de invoer, opgedeelt in batches
6 peptide_iterator(peptides) do |batch, i, fasta_input|
7
8   # Maak een nieuwe request aan voor elke batch
9   request = Typhoeus::Request.new(...)
10
11  # Definieer de acties bij het afwerken van een request
12  request.on_complete do |resp|
13
14    if resp.success?
15      ...
16    else
17      if resp.timed_out?
18        ...
19      elsif ...
20      end
21    end

```

```

22 end

23
24 # Voeg de gemaakte request toe aan de Hydra queue
25 hydra.queue request

26
27 # Voor elke 200 requests die klaar staan, laat Hydra ze uitvoeren
28 # en wacht tot ze klaar zijn
29 num_req += 1
30 if num_req % 200 == 0
31     hydra.run
32 end

33
34 end

35
36 # Laat de Hydra nog eens lopen voor het geval dat de
37 # batches geen veelvoud zijn van 200
38 hydra.run

```

Samengevat worden altijd 200 batches klaargezet die bij een call naar `hydra.run` zullen verwerkt worden door Typheous. Hierbij zijn er altijd maximaal 10 die in parallel naar de API server gestuurd worden. Wanneer de 200 batches volledig verwerkt zijn, worden de volgende 200 batches verstuurd totdat ze allemaal zijn verwerkt.

Het zou zonde zijn om maar één keer te proberen om een request te versturen. Wie weet zou na een tweede keer proberen de request wel perfect verwerkt worden. Dit gedrag is vrij eenvoudig te implementeren. Typheous heeft een methode `queue_front`[15] om taken vooraan aan de Hydra queue toe te voegen. Door in het `else` blok van lijn 16 in bovenstaande code een call te doen naar die methode met de request als parameter, kunnen we de request opnieuw toevoegen voor verwerking.

We voegen de request toe aan het begin van de queue om de delay bij het uitschrijven, en zo ook het geheugengebruik, te beperken. Stel dat de eerste request uit een groep van 200 requests zou falen, dan zouden we moeten wachten tot de overige 199 requests zijn uitgevoerd vooraleer de request opnieuw kan verstuurd worden. Dat zou betekenen dat er in tussentijd 199 batches in de hierboven beschreven `batch_order` wachtrij staan te wachten om uitgeschreven te worden, en dus plaats innemen in het geheugen.

Als een request keer op keer faalt, bijvoorbeeld omdat de server volledig verzadigd is, willen we de request niet continu opnieuw naar de server sturen. Daarom moeten we een maximum opleggen op het aantal keer dat een request kan teruggestuurd worden. Dit kunnen we doen door de `Request`-klasse uit Typheous te patchen met twee extra variabelen: `retries`, het aantal keer dat een request mag uitgevoerd worden, en `current_retries`, het aantal keer dat een request reeds uitgevoerd werd.

Bovenstaande wijzigingen resulteren dan in Broncode 4.2 op pagina 53 voor het correct behandelen API-fouten.

Broncode 4.2: Aangepaste code voor het correct behandelen van API-fouten.

```

1 if resp.success?
2     ...

```

```
3 | else
4 |   if resp.timed_out?
5 |     ...
6 |   elsif ...
7 |   end
8 |
9 |   if request.current_retries != request.retries
10|     hydra.queue_front(request)
11|     request.current_retries += 1
12|   end
13| end
```

# Hoofdstuk 5

## Visualisatieframework voor UniPept

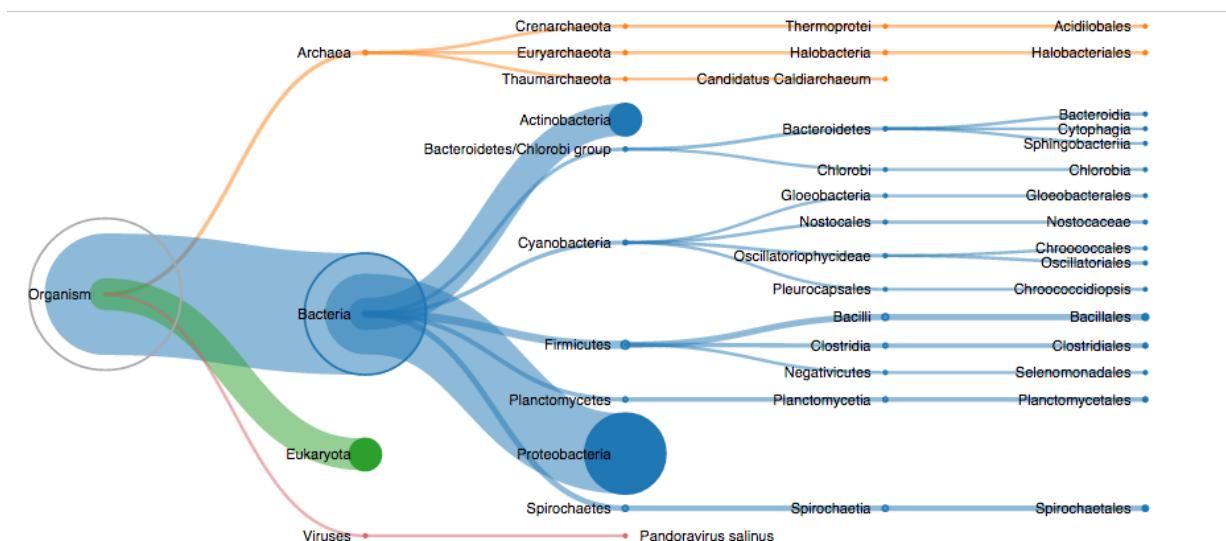
In dit hoofdstuk schetsen we eerst kort het overzicht van de huidige visualisaties in UniPept. We staan stil bij de nood aan en het praktisch nut van het abstracteren van die visualisaties in een apart framework en de voordelen hiervan. Daarna kijken we naar de implementatie van deze abstractie, waarbij we de abstractie van de `treeview`-visualisatie als voorbeeld uitwerken. Vervolgens gaan we verder in op enkele voorbeelden, use cases en optimalisaties van de `treeview`-visualisatie.

### 5.1 Overzicht van visualisaties in UniPept

Vooraleer we overgaan tot de bespreking van het visualisatieframework kijken we eerst welke visualisaties in UniPept verwerkt zijn. De eerste visualisatie die we bespreken is de `treeview`. Deze visualisatie geeft bij een tryptische peptidenanalyse een overzicht van de taxonomische portie van alle Uniprot records waarin een bepaalde peptide wordt gevonden en maapt deze voorkomens vervolgens op een boomstructuur. Een voorbeeld hiervan is te vinden in Figuur 5.1 op pagina 56. De `treeview`-visualisatie wordt ook gebruikt binnen de metaproteomics analyse. Gegeven een lijst van peptides uit een staal, wordt voor elke peptide de lowest common ancestor berekend. Vervolgens worden de lowest common ancestors gemapt op de boomstructuur.

Er zijn ook twee alternatieve visualisaties voor de tryptische peptidenanalyse. Het eerste alternatief is een `sunburst`-diagram, te zien in Figuur 5.2a op pagina 58. In deze visualisatie zien we in het centrum de `root`-taxon. Rond om rond deze centrumnode bevinden zich de kinderen van het `root`-taxon. Het is mogelijk om op een kind te klikken, zodat dat kind het nieuwe centrum van het diagram wordt. Door op de centrumnode te klikken wordt er een niveau uitgezoomd. Het tweede alternatief is de `treemap`-visualisatie. Dit is een 2D weergave van alle gevonden taxons in de analyse, waarbij de grootte van hun vlak overeenstemt met het aantal peptidesequenties die specifiek zijn tot op dat niveau. Door op een vlak te klikken kunnen we inzoomen. Deze visualisatie is te zien op Figuur 5.2b.

Ook bij de peptidome analyse worden twee visualisaties gebruikt. Bij een peptidome analysis worden verschillende volledig gesequeneerde genomen met elkaar vergeleken. Uit



Figuur 5.1: Voorbeeld van de treeview-visualisatie voor het weergeven van de taxonomische porties van alle UniprotKB eiwitten die de tryptische peptide “AALTER” bevatten.

deze analyses worden zaken afgeleid zoals de grootte van het aantal peptides binnen de core, pan of unique peptidome, of hoe goed verschillende genomen met elkaar clusteren. De eerste van deze twee die de groottes van de verschillende peptidomes aangeeft is de **pancore**-grafiek. Een voorbeeld hiervan is te zien in Figuur 5.2c op pagina 58. De clusteringsanalyse gebeurt aan de hand van een similariteitsmatrix waarbij donkere kleuren een grotere similariteit tussen twee genomen aanduiden, ondersteund door een klassieke phylogenetische boom. Een voorbeeld hiervan kan geraadpleegd worden in Figuur 5.2d op pagina 58.

Alle visualisaties zijn geschreven met behulp van D3. D3 is een JavaScript bibliotheek voor het maken van dynamische en interactieve visualisaties aan de hand van HTML, CSS en SVG[7].

## 5.2 Modularisatie en abstractie

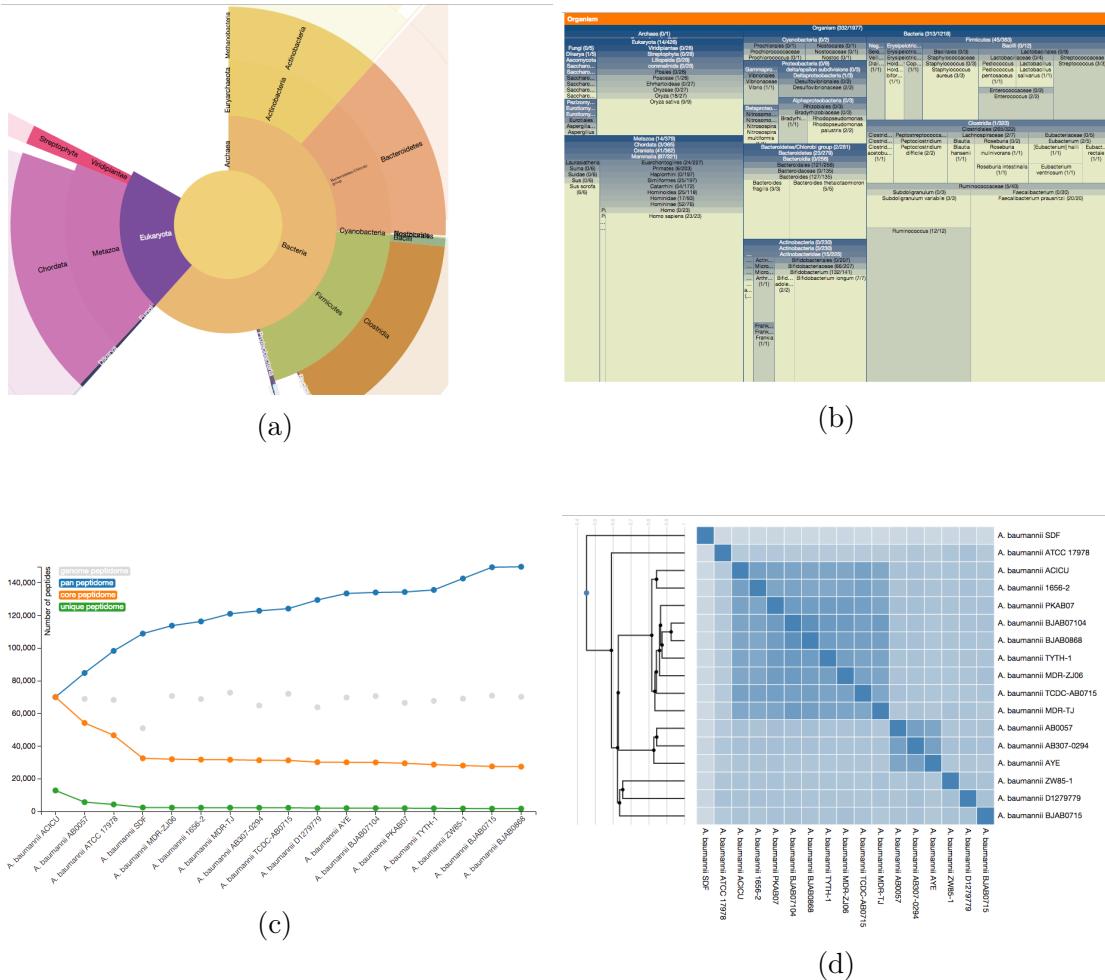
Momenteel zitten alle visualisaties volledig verankerd in UniPept en zijn ze zeer dicht verweven met de UniPept webinterface. Deze integratie brengt enkele voordelen met zich mee. Zo kan de code bijvoorbeeld eenduidig en specifiek geschreven worden voor één gewenst doel, zonder rekening te moeten houden met eventuele mogelijkheden die toch niet zullen voorkomen of zouden gewenst zijn. Volledige integratie heeft echter ook zijn nadelen. Zo is het niet mogelijk dezelfde visualisatie in een andere omgeving of voor een ander doel te gebruiken, zonder de achterliggende code aan te passen. Met de huidige implementatie kunnen er bijvoorbeeld geen twee treeviews naast elkaar weergegeven worden, kan er niet gemakkelijk andere data (zoals niet-biologische data) aan de boom gekoppeld worden, enzovoort.

Binnen de context van deze thesis wordt de treeview als een overzichtelijke visualisatie gebruikt om (delen van) de volledige NCBI taxonomie weer te geven. We gebruiken de treeview als visueel hulpmiddel om onder andere het effect van invalidaties van taxons op de treeview te visualiseren of om het effect van in ?? beschreven analyses weer te geven. Om dit te verwezenlijken moet het mogelijk zijn op een eenvoudige manier willekeurige data in te voeren en moeten de visualisatieopties snel kunnen ingesteld worden, zonder daarbij steeds de achterliggende code te moeten aanpassen. Daarnaast willen we ook meerdere treeviews op één pagina visualiseren, zodat niet telkens van scherm moet gewisseld worden bij het vergelijken van twee resultaten. Kort gesteld willen we vertrekken van een bepaald invoerformaat, om zo snel mogelijk een gevisualiseerd resultaat te zien krijgen, zonder de invoer in een webinterface in te geven.

In de informatica heet dit concept “modularisatie”: stukken code worden onderverdeeld in afgesloten modules waarbinnen ze apart worden ontwikkeld en onderhouden. De aparte modules kunnen vervolgens samen ingeplugged worden in een groter systeem. Door abstracties toe te voegen kunnen aparte modules niet enkel voor één doel gebruikt worden, maar kunnen ze ook dienst doen in andere projecten die niet per sé het visualiseren van data uit de metaproteomics als doel hebben.

Zoals eerder vermeld zijn de huidige visualisaties enkel beschikbaar binnen de UniPept webinterface. Dit werkt goed bij kleine hoeveelheden data, maar wanneer we grote datasets

## 5.2. MODULARISATIE EN ABSTRACTIEDSTUK 5. VISUALISATIEFRAMEWORK



Figuur 5.2: Overzicht van de visualisaties in UniPept. Linksboven zien we een Sunburst-voorbeeld van een analyse van de metaproteomics dataset van het “human gut”, met daar rechts van de treemap voor dezelfde dataset. Linksonder wordt een pancore visualisatie getoond van het peptidome van *Acinetobacter Baumannii* met rechts de similariteitsmatrix met bijhorende phylogenetische boom voor hetzelfde organisme.

willen inladen, duiken er problemen op en weigert de browser. We kunnen dit probleem oplossen door de visualisaties los te koppelen van de codebase van het huidige UniPept platform en door ze als zelfstandige modules aan te bieden. In combinatie met de bestaande command line tools kunnen we dan de visualisatiemodule gebruiken om de resultaten visueel weer te geven door de data via een webinterface in te laden en te verwerken.

Het modulariseren van die visualisaties kan dus een stap in de goede richting zijn om ze ook buiten de UniPept webinterface aan te bieden. Als we kijken naar de huidige staat van visualisaties buiten UniPept, moeten we vaststellen dat die niet optimaal is. Vaak zijn de visualisaties statisch en proberen ze te veel data in één opzicht weer te geven, wat het resultaat complex en onoverzichtelijk maakt. Het aanbieden van een open-source en een geabstraheerd visualisatieframework zou hier een oplossing voor kunnen bieden.

Dit is een argument voor modularisatie, maar ook abstractie speelt een rol. Zonder het toevoegen van abstractie kunnen de huidige visualisaties enkel gebruikt worden om één soort data voor te stellen, wat altijd in exact dezelfde visualisatie resulteert. Door het toevoegen van abstractie krijgt de gebruiker de mogelijkheid om de voorstelling van de boom eenvoudig te wijzigen, zonder daarbij aanpassingen in de onderliggende broncode te moeten maken.

## 5.3 Implementatie

De concepten van modularisatie en abstractie liggen dicht bij elkaar. Door iets modulair te maken, maak je het vanzelf abstracter. In deze sectie kijken we eerst hoe andere JavaScript frameworks dit realiseren. Daarna bespreken we de implementatie hiervan aan de hand van een hands-on voorbeeld van de treeview-visualisatie.

### 5.3.1 Modularisatie

Bootstrap[4] is waarschijnlijk het meest bekende Javascriptframework. Bootstrap biedt een hele reeks JavaScriptbibliotheeken aan om bepaalde ontwerppatronen op het web eenvoudiger te kunnen implementeren. Gebruikers kunnen alle functies samen in één pakket downloaden maar hebben ook de mogelijkheid om één functie te selecteren en enkel die te gebruiken. Aangezien de functionaliteiten van Bootstrap alleen, gedeeltelijk of volledig ingeplugged kunnen worden als modules, doet Bootstrap duidelijk aan modularisatie.

#### Case study: Bootstrap Carousel

Als we de code van Bootstrap in detail bekijken, zien we dat alle Bootstrap plugins eenzelfde coderichtlijn gebruiken. Broncode 5.1 op page 60 laat een (ingekort) voorbeeld zien van de Bootstrap `Carousel`-klasse.[5]. Eerst wordt een constructor<sup>1</sup> gedefinieerd om een `Carousel` aan te maken, waarna enkele standaardwaarden worden toegekend aan de klasse `Carousel`.

---

<sup>1</sup>Gebruik van OOP terminologie in JavaScript staat ter discussie, maar maakt het redeneren over de structuur eenvoudiger. Vanaf ECMAScript 6 zijn ook OOP keywords in JavaScript beschikbaar.

om de versie en andere standaardwaarden in te stellen. Daarna worden enkele methodes op de klasse gedefinieerd die toelaten om de staat van een object aan te passen.

Als we de functie `Plugin` tijdelijk overslaan zien we iets merkwaardigs. De functie `carousel` wordt gelijk gesteld aan een call naar de functie `Plugin` door middel van de lijn `$.fn.carousel = Plugin`. Dat wil zeggen dat we aan het `$` object – waarbij `$` een afkorting is van `jQuery` – een methode genaamd `carousel` toevoegen, waarbij oproepen in de vorm van `jQueryObject.carousel()` terechtkomen bij de methode `Plugin`.

Zo kunnen we bijvoorbeeld in een HTML-bestand een `div` maken, de `div` selecteren aan de hand van zijn `id` of `class`, en daarna de functie `carousel()` aanroepen. Op dit `div` object zal dan de functie `Plugin` worden aangeroepen.

Ook de lijn `$.fn.carousel.Constructor = Carousel` heeft wat meer uitleg nodig. Het toekennen van een klasse aan het `Constructor`-attribuut van een plugin zorgt ervoor dat de constructor van buitenaf toegankelijk is. Zo kan een programmeur bijvoorbeeld rechtstreeks een `Carousel` definiëren door middel van `var myCarousel = new $.fn.carousel.Constructor()` zonder gebruik te maken van de `Plugin` methode. De naamgeving van het `Constructor` attribuut is puur conventie.

Tot nu toe was alle code relatief rechtoe rechtaan. De echte “magie” om de `Carousel`-klasse modulair inplugbaar te maken, vindt plaats in de `Plugin` methode. In de eerste lijn van deze functie overlopen we via `this.each()` alle `jQuery` elementen waarop de methode – in dit geval – `carousel` is opgeroepen. Om ervoor te zorgen dat we verdere methodes kunnen schakelen aan de oproepende methode geven we de DOM-elementen ook terug[10].

Op elk DOM-element roepen we een anonieme functie op waarbij de `this` variabele (die een DOM element voorstelt) omvormen tot een `jQuery` object zodat `jQuery`methodes hier op kunnen worden opgeroepen. In de variabele `data` wordt opgeslagen of er reeds een `data`-attribuut is toegekend. Daarna stellen we een lijst op van opties via de `jQuery extend`-methode. Hierbij worden drie opties gemengd. i) de defaultwaarden van de `Carousel`-klasse, ii) eventuele attributen die meegegeven worden via `data`-attributen aan het DOM-element en iii) opties die worden meegegeven via de parameters van de `carousel`-functie.

Als de `data`-variabele nog niet ingesteld werd, stellen we die nu in. Om dat te doen zetten we het `data`-attribuut van het huidig DOM-element op `bs.carousel`, en maken we een nieuwe instantie van de `Carousel`-klasse met de gemende opties uit de vorige paragraaf. Het `data`-attribuut van het betreffende DOM-element geeft nu aan of er al een `carousel` is geïnitialiseerd of niet. Als we de `carousel`-functie oproepen op een DOM-element waar reeds een `carousel` is geïnitialiseerd, kunnen we afhankelijk van het type van de parameters verschillende acties uitvoeren op de het DOM-element, die dan de reeds geïnitialiseerde `carousel` bevat.

Broncode 5.1: Ingekorte Bootstrap code van de `Carousel`-plugin

```

1 // Creatie van de klasse Carousel
2 var Carousel = function (element, options) {
3     this.$element      = $(element)
4     this.options       = options
5     this.paused        = null

```

```

6  // Andere klassevariabelen kunnen hier worden gedeclareerd
7  ...
8 }
9
10 // Declaratie van globale variabelen
11 Carousel.VERSION = '3.3.4'
12 Carousel.TRANSITION_DURATION = 600
13 Carousel.DEFAULTS = {
14   interval: 5000,
15   ...
16 }
17
18 // Methoden kunnen worden toegevoegd via het prototype van het Carousel
19 Carousel.prototype.some_method = function (params) {
20   ...
21 }
22
23 // Toevoegen van de gemaakte klassen en methoden aan het jQuery object
24 $.fn.carousel = Plugin
25 $.fn.carousel.Constructor = Carousel
26
27 // Methode die bovenstaande Carousel klasse modulair inplugbaar maakt
28 function Plugin(option) {
29   return this.each(function () {
30     var $this = $(this)
31     var data = $this.data('bs.carousel')
32     var options = $.extend(
33       {},
34       Carousel.DEFAULTS,
35       $this.data(),
36       typeof option == 'object' && option
37     )
38
39     /* Initialiseert een nieuwe carousel wanneer $this geen data-
40      attribuut heeft met de waarde 'bs.carousel'. */
41     if (!data) $this.data('bs.carousel', (data = new Carousel(this,
42       options)))
43     // Voer methodes uit afhankelijk van de argumenten
44     if (typeof option === 'number') data.to(option)
45     else if ...
46   })
47 }
48

```

Kort samengevat: Bootstrap gebruikt een objectgeoriënteerde structuur waar, afhankelijk van het bestaan van een bepaald data-attribuut, een nieuwe instantie van een bepaalde klasse aan een jQuery element wordt gehangen of waar methodes worden aangeroepen op een reeds bestaande instantie. Het koppelen van instanties van een klasse aan een jQuery element is daarbij de motor om alles modulair te maken. Op die manier kan aan gelijk welk DOM-element een Bootstrapcomponent gehangen worden, zonder dat de interne code de

naam of id van dit object moet kennen.

### Huidige Treeview implementatie in Unipept

Bootstrap gebruikt hier dus een strategie die een oplossing biedt voor de tekortkomingen van de `treeview`-visualisatie. Een gemodereerd deel van de oorspronkelijke code is te zien in Broncode 5.2 op page 62. Hierbij wordt alle code omvat in één functie, `constructTreeview`. Bij een oproep van die functie wordt de binnenste `init`-functie opgeroepen die enkele zaken initialiseert. De functie bindt eerst de `reset`-functie aan een knop met id `treeview-reset`, waarna de functie de benodigde functies oproeft om de treeview ook werkelijk te tekenen, zijnde `redraw`- en `draw`. Onderaan wordt ook de methode `reset` publiek beschikbaar gemaakt door deze (en eventueel andere publieke methodes) te binden aan de `that` variabele. Het `that` object wordt op het einde van de omvattende functie teruggegeven, zodat de publieke functies via het object kunnen opgeroepen worden.

De methode `constructTreeview` kan zo door een gebruiker opgeroepen worden, waarbij de gebruiker het resultaat (het `treeview`-object) kan opslaan om methodes op dit object op te roepen.

Broncode 5.2: Gemodereerde TreeView code

```

1 /**
2  * Maakt een Treeview object die de visualisatie representeert
3 */
4 var constructTreeview = function constructTreeview(args) {
5     //***** Private variabelen *****/
6
7     // parameters
8     var that = {},
9         multi = args.multi,
10        data = args.data;
11
12    // private variabelen
13    var privateVariable = name;
14
15    //***** Private methoden *****/
16
17    // Initialiseert een Treeview
18    function init() {
19        // init controls
20        initControls();
21
22        // draw everything
23        redraw();
24        draw(data);
25
26        ...
27    }
28

```

```

29     // Initieert controllers
30     function initControls() {
31         // bindt de reset-methode aan het #treeview-reset element
32         $("#treeview-reset").click(that.reset);
33     }
34
35     // Hertekent de treeview
36     function redraw() {
37         // Maak de treeview leeg
38         $("#d3TreeView").empty();
39
40         ...
41
42         // Tekent de d3 visualisatie op het #d3TreeView element
43         svg = d3.select("#d3TreeView").append("svg")
44
45         ...
46     }
47
48     // Tekent de boom
49     function draw(data) { ... }
50
51     /****** Publieke methoden *****/
52
53     // Zet de visualisatie in zijn oorspronkelijke staat terug
54     that.reset = function reset() {
55         zoomListener.scale(1);
56         rightClick(root);
57     };
58
59     // initialiseert het object
60     init();
61
62     // geef het object terug
63     return that;
64 };

```

De code die net werd beschreven functioneert goed wanneer die enkel binnen Unipect gebruikt wordt, maar is zeker niet modulair genoeg om als framework aan te bieden. Het grootste verschil tussen de twee benaderingen is het binden van de visualisatie aan een DOM-element. Bij een Bootstrap plugin gebeurt dit door een methode (in voorgaand voorbeeld `carousel`) op te roepen op een jQuery object. Bij de huidige implementatie in Unipect wordt dit element ingesteld in de code zelf. Methodes worden in Bootstrap plugins rechtstreeks op het element opgeroepen, zonder dat het Javascript object ergens dient te worden opgeslagen. De koppeling tussen het element en de plugin/visualisatie die vasthangt aan dat element gebeurt intern via een `data`-attribuut. In de implementatie in Unipect moet hiervoor het JavaScript object zelf bijgehouden worden en worden methodes rechtstreeks op dit object opgeroepen.

Praktisch houdt dit in dat er op meerdere plaatsen in de Unipect-visualisatie verwezen

wordt naar een HTML klasse of id. Dit is nadelig als we deze visualisaties zouden willen aanbieden aan gebruikers buiten Unipept. Wanneer er bijvoorbeeld een gebruiker twee verschillende bomen onder elkaar zou willen weergeven, is dit onmogelijk zonder de hele omvattende methode te dupliceren en de id waaraan de visualisaties worden gekoppeld aan te passen, of door het visualisatieobject mee te geven als extra parameter aan de `constructTreeview` functie. Ook het koppelen van acties zoals binden van de `reset`-methode aan meerdere objecten is met deze strategie niet mogelijk.

Als we de Unipept visualisaties in een extern visualisatieframework willen aanbieden, is het wenselijk dat de gebruiker de code nooit zelf hoeft aan te passen, maar die gewoon vanuit zijn eigen JavaScriptcode kan aanroepen. Door de Bootstrap `Plugin`-methode toe te passen op de bestaande code en intern enkele veranderingen door te voeren, kunnen we de interne code volledig loskoppelen van onderliggende DOM-elementen, zodat de gebruiker deze plugin – met natuurlijk de nodige documentatie – als een blackbox kan behandelen zonder intern aanpassingen te hoeven aan brengen.

## Implementatie

De herwerkte code na het integreren van de Bootstrap `Plugin`-functie is terug te vinden in Broncode 5.3 op page 64. Erg verschillend van een combinatie van de vorige twee codefragmenten is dit niet. We zien onderaan dat de `treeview`-methode wordt gekoppeld aan de `Plugin`-methode, analoog met de Bootstrap `carousel`-functie. In de `Plugin`-methode koppelen we een data-attribuut `vis.treeview` aan het opgegeven jQuery-object wanneer dit attribuut nog niet is ingevuld. Bij het aanmaken van een `TreeView` object geven we naast het DOM-element waarop de `treeview`-methode is aangeroepen, ook de standaard opties en eventuele opgegeven opties mee. In de `init`-methode van deze `TreeView` klasse koppelen we nu de D3 visualisatie aan het opgegeven DOM-element. Methodes die van buitenaf beschikbaar moeten te zijn, zoals de `reset`-methode, worden aan de variabele `that` toegekend die uiteindelijk wordt teruggegeven.

Broncode 5.3: (Ingekorte) code van de Unipept Treeview plugin

```

1 // Constructor van het TreeView-object
2 var TreeView = function TreeView(element, options) {
3     var that = {};
4
5     // Private methodes
6     function init() {
7         ...
8
9         svg = d3.select(element).append("svg")
10        ...
11    }
12
13    ...
14
15    // Publieke methodes

```

```

16     that.reset = function reset() { ... }
17
18     init();
19
20     return that;
21 };
22
23 TreeView.DEFAULTS = { ... }
24
25 $.fn.treeview = Plugin;
26 $.fn.treeview.Constructor = TreeView;
27
28 // Pluginmethode
29 function Plugin(option) {
30     return this.each(function () {
31         var $this = $(this);
32         var data = $this.data('vis.treeview');
33         var options = $.extend(
34             {},
35             TreeView.DEFAULTS,
36             $this.data(),
37             typeof option === 'object' && option
38         );
39
40         if(!data) {
41             $this.data('vis.treeview',
42                     (data = new TreeView(this, options)));
43         }
44         if(typeof option === 'string') { data[option](); }
45     });
46 }

```

Dit zorgt ervoor – net zoals in de Bootstrap Carousel case study – dat er een treeview-visualisatie kan worden gehangen aan eender welk DOM-element door het oproepen van de `treeview`-functie. Eén argument is verplicht, namelijk het `data`-argument dat de te visualiseren data bevat. Publieke methoden, zoals de `reset`-methode kunnen worden opgeroepen door de methodenaam als `string` mee te geven aan de `treeview`-function: `$('#treeviewDiv').treeview('reset')`. Meer voorbeelden zullen worden uitgewerkt in Sectie 5.4 op pagina 67.

### 5.3.2 Abstractie

De huidige visualisaties in UniPept zijn, zoals eerder beschreven, specifiek gemaakt om data uit de UniPept metaproteomics pipeline grafisch weer te geven. Als we willen dat gebruikers ook de visualisaties kunnen gebruiken voor andere soorten data, moeten we de specifieke functies abstraheren. Daarnaast willen we ook een stuk controle in de handen van de gebruiker leggen. Een gebruiker wil bijvoorbeeld de kleur van de visualisatie kunnen bepalen, de hoogte en breedte van de visualisatie veranderen, de grootte van de nodes

beïnvloeden, etc. We moeten er dus voor zorgen dat de gebruiker zelf (extra) methodes kan implementeren, die de standaardmethodes vervangen.

We maken onderscheid tussen twee vormen van abstractie. Ten eerste moet de gebruiker via parameters instellingen kunnen overschrijven, zoals bijvoorbeeld de breedte van de visualisatie, de afstand tussen de nodes, etc. Ten tweede moet de gebruiker ook functies die inspelen op de data kunnen overschrijven. Het eerste deel hiervan wordt reeds opgelost door het uitbreiden van de standaard opties met opties die de gebruiker meegeeft aan de Plugin-methode beschreven in Sectie 5.3.1 op pagina 64. Stel dat de gebruiker de hoogte en breedte van zijn visualisatie wil opgeven, dan kan hij dit doen door de functie `treeview` op een DOM-element op te roepen, waarbij hij naast de te visualiseren data ook de argumenten `width` en `height` meegeeft. De oproep om een visualisatie van 1400 pixels breed op 800 pixels hoog te maken met als data een object genaamd “data” kan er dan bijvoorbeeld zo uitzien:

```
1  $("#d3TreeView").treeview({
2      data: data,
3      width: 1400,
4      height: 800
5});
```

Functies overschrijven kan echter niet zo eenvoudig. Het verschil met gewone algemene parameters is dat we bij functies willen dat ze afhankelijk zijn van een bepaalde invoer, bijvoorbeeld de naam van de visualisatiecomponent in kwestie. De eenvoudigste oplossing hiervoor is om de te overschrijven functies als standaardmethodes te implementeren op dezelfde hoogte als waar de standaard waarden gedefinieerd worden. Zo kunnen we bijvoorbeeld een standaardfunctie implementeren die de opvulkleur van de nodes in de treeview afhankelijk maakt van het al dan niet hebben van “kinderen”. Dit zou bijvoorbeeld als volgt als standaardmethode kunnen worden geïmplementeerd:

```
1 TreeView.NODE_FILL_COLOR = function(d) {
2     if (d.selected) {
3         return d._children ? d.color || "#aaa" : "#fff";
4     } else {
5         return "#aaa";
6     }
7};
```

Deze standaardfunctie kunnen we nu meegeven als parameter aan het `TreeView.DEFAULTS`-object dat onder meer ook de bovenstaande standaardwaarden bevat via `nodeFillColor`: `TreeView.NODE_FILL_COLOR`.

Indien een gebruiker een andere functionaliteit wenst dan de standaardfunctionaliteit, dan kan hij zelf een functie declareren die bijvoorbeeld altijd rood teruggeeft:

```
1 // Declaratie van eigen kleurfunctie ter vulling van nodes
2 var nodeFillColor = function(d) {
3     return 'red';
4 }
5 // Meegeven van eigen kleurfunctie aan de Treeview-bibliotheek
```

```

7 | $("#d3TreeView").treeview({
8 |   data: data,
9 |   nodeFillColor: nodeFillColor
10| });

```

Alle functies die de gebruiker moet kunnen overschrijven of aanpassen, kunnen op deze manier geïmplementeerd worden.

Niet alleen de uitvoer (het visuele resultaat) moet generiek zijn, maar ook de invoer. Momenteel is de invoer een JSON-object met vier attributen: `id`, `name`, `data` en `children`. Deze vaste attributen zijn ook nodig om dezelfde data in andere visualisaties voor te stellen. Aan het `data`-attribuut kunnen elementen worden toegevoegd die specifiek zijn voor de gewenste visualisatie. In de UniPept-visualisaties wordt bijvoorbeeld altijd gebruik gemaakt van een `count`-, `rank`- en `self_count` attribuut. Deze optionele attributen kunnen dan opgeroepen worden in de functies die de standaardmethodes vervangen. Aangezien we met hiërarchische visualisaties werken, moeten ook “kinderen” aan “ouders” toegevoegd kunnen worden. Dit laatste gebeurt door het toevoegen van een lijst van JSON-objecten van dezelfde vorm als hierboven geschreven aan het `children`-attribuut van zijn “ouder”.

Aangezien de invoer momenteel al generiek en door middel van het `data`-attribuut ook naar wens uitbreidbaar is, kunnen we besluiten dat de invoer voldoende geabstraheerd is.

## 5.4 Case studies

In deze sectie wordt een overzicht van de huidige mogelijkheden geschetst aan de hand van enkele use cases, telkens met de betreffende code en het resultaat.

### 5.4.1 Modularisatie: vergelijken van resultaten van toolchains

We beginnen met een eenvoudig voorbeeld van de modularisatie. Stel dat een gebruiker een metagenomics analyse heeft uitgevoerd, gebruik makend van de `pept21ca21ca` toolchain met de command-line tools, en dit resultaat visueel wil weergeven. Het resultaat van de analyse is een lijst van taxons, hun rang en id, gegroepeerd per fasta header. De gebruiker kan daarna via een script `tax2tree.py` het resultaat omzetten in een JSON-object dat hij rechtstreeks kan inplussen in een visualisatie door middel van het HTML bestand in Broncode 5.4 op pagina 68. Deze code kan gewoon in een lokaal HTML bestand worden ingegeven en geeft als resultaat Figuur 5.3.

Stel nu dat de gebruiker een nieuwe analyse wil uitvoeren op het genoom in kwestie, maar nu met de `pept2prot2filter21ca` toolchain en de twee resultaten wil vergelijken. Dan kan dit eenvoudig door op het nieuwe resultaat het `tax2tree.py` script uit te voeren, en de broncode uit Broncode 5.5 toe te voegen aan de broncode uit Broncode 5.4.

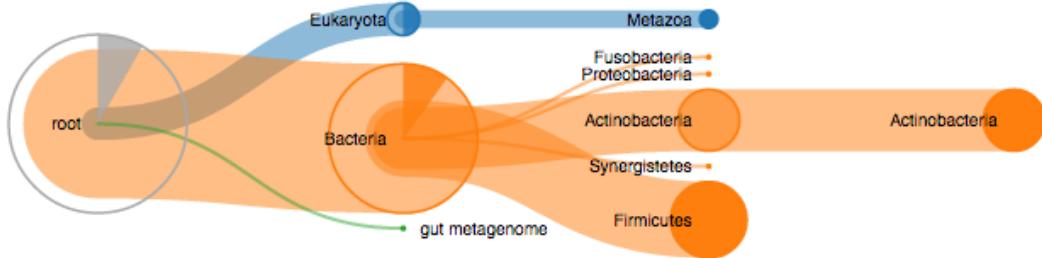
Dit geeft de visualisatie zoals weergegeven in Figuur 5.4. Een visuele vergelijking van de twee resultaten toont al snel dat er twee deeltakken verdwenen zijn, iets niet zo snel zou zijn opgevallen door de tekstresultaten te vergelijken.

Broncode 5.4: Voorbeeldcode ter illustratie van modularisatie van de treeview-visualisatie

```

1 <html>
2   <head>
3     <title>Unipept Visualisation Framework</title>
4
5     <!-- Inladen van jQuery, Bootstrap en d3 -->
6     ...
7
8     <!-- Inladen Visualisatiemodules -->
9     <link href="../assets/stylesheets/tooltip.css" rel="stylesheet">
10    <script src="../assets/javascripts/treeview.js"></script>
11
12   <script>
13     $(function() {
14       // Inladen van het te visualiseren JSON-bestand
15       d3.json("../sample-data/pept2lca2lca.json", function(error, data) {
16         // Initialisatieaanroep van de treeview-methode
17         $("#d3TreeView-pept2lca2lca").treeview({data: data});
18       });
19
20       // Binden van de reset-actie aan de #treeview-reset knop
21       $("#treeview-reset-pept2lca2lca").click(function() {
22         $("#d3TreeView-pept2lca2lca").treeview('reset');
23       });
24     });
25   </script>
26 </head>
27 <body>
28   <span class="dir">
29     <a class="btn btn-xs btn-default" id="treeview-reset-pept2lca2lca"
30       title="reset"
31       visualisation"><i class="glyphicon glyphicon-repeat"></i></a>
32   </span>
33   <span class="dir text">Scroll to zoom, drag to pan, click a node to
34   expand, right click a node to set as root</span>
35   <div id="d3TreeView-pept2lca2lca"></div>
36 </body>
37 </html>
```

 Scroll to zoom, drag to pan, click a node to expand, right click a node to set as root

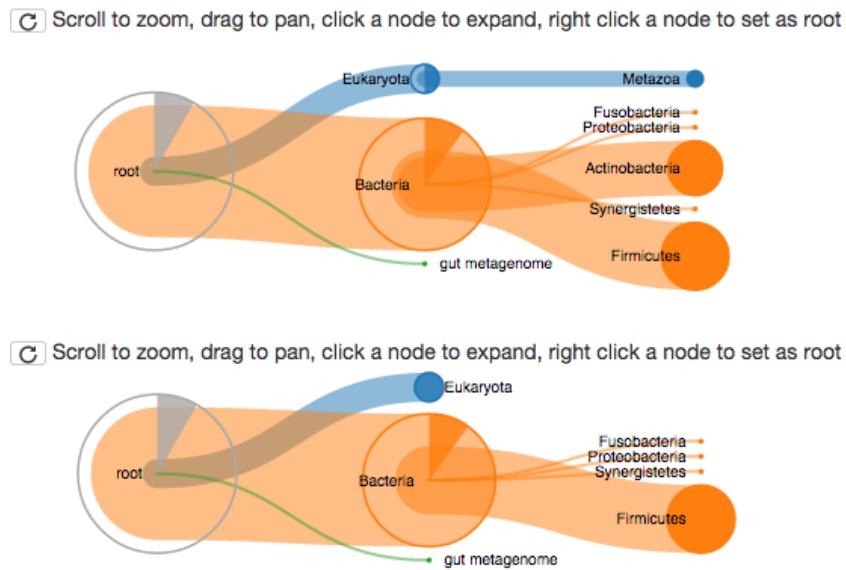


Figuur 5.3: Resultaat van de HTML-code in Broncode 5.4

Broncode 5.5: Uitbreiding van Broncode 5.4 waarbij een tweede treeview is toegevoegd

```

1 ...
2
3 // Inladen van het nieuwe te visualiseren JSON-bestand
4 d3.json("../sample-data/pept2prot2filter2lca.json",
5   function(error, data) {
6     // Initialisatieaanroep van de treeview-methode
7     $("#d3TreeView").treeview({data: data});
8   }
9 );
10
11 // Binden van de reset-actie aan de #treeview-reset knop
12 $("#treeview-reset-pept2prot2filter2lca").click(function() {
13   $("#d3TreeView-pept2prot2filter2lca").treeview('reset');
14 });
15 );
16
17 ...
18
19 <!-- Toevoegen van nieuwe reset-knop en div in de HTML body-->
20 <span class="dir">
21   <a class="btn btn-xs btn-default" id="treeview-reset-
22     pept2prot2filter2lca"
23     title="reset
24     visualisation"><i class="glyphicon glyphicon-repeat"></i></a>
25 </span>
26 <span class="dir text">Scroll to zoom, drag to pan, click a node to
27 expand, right click a node to set as root</span>
<div id="d3TreeView-pept2prot2filter2lca"></div>
```



Figuur 5.4: Visualisatie van de HTML-code in Broncode 5.5

### 5.4.2 Abstractie: Overschrijven van standaardmethodes

Met een tweede case study tonen we aan hoe de gebruiker bepaalde functies kan overschrijven met zijn eigen implementatie.

Indien de gebruiker een grafische zoektocht willen starten naar bepaalde speciale gevallen in de taxonomie, zoals taxons die rangloos zijn en daarnaast ook enkele taxons willen invalideren die aan bepaalde voorwaarden voldoen, dan kan hij de volgende procedure gebruiken. De gebruiker kan van een standaard JSON-representatie van de boom starten, waarvan hij het `data`-attribuut aanvult met twee eigen attributen `rank` en `valid_taxon`. Een taxon is rangloos wanneer zijn rang gelijk is aan “no rank” en een taxon is `invalid` wanneer het `valid_taxon` attribuut gelijk is aan 0. Het JavaScript-gedeelte van de code kan teruggevonden worden in Broncode 5.6 op pagina 71.

Eerst definiëren we de twee gewenste functies: een functie die de randkleur van de node aanpast en een functie die de opvulkleur van de node aanpast. In beide gevallen geven we “red” terug wanneer het `valid_taxon`-attribuut van het `data`-attribuut op 0 staat. Wanneer de `rank` gelijk is aan “no rank” geven we “grey” terug. In alle andere gevallen vallen we telkens terug op de standaardimplementatie die beschikbaar is via het conventionele `Constructor`-attribuut van de `treeview`-functie. Deze functies geven dan uiteindelijk mee aan de oproep naar de `treeview`-methode zodat die functies de standaardimplementaties overschrijven.

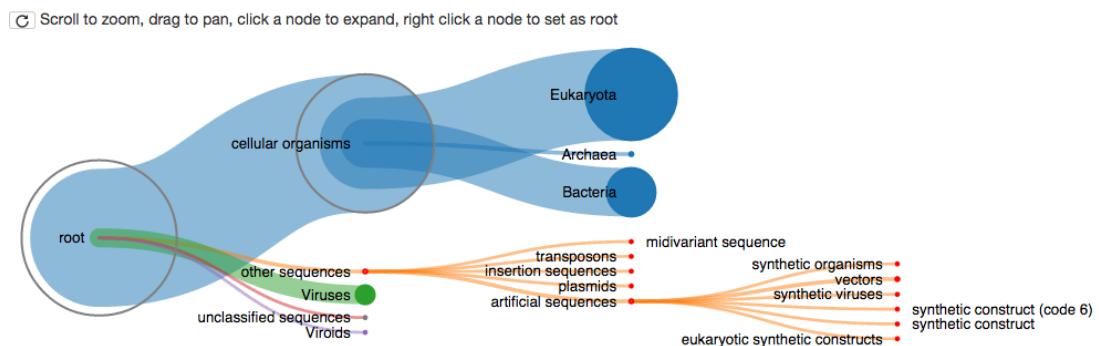
Het resultaat van Broncode 5.6 is te zien in Figuur 5.5. We zien inderdaad dat de nodes die geen rang hebben zoals onder meer `root` en `cellular organisms`, een grijze rand krijgen en dat geïnvalideerde nodes, zoals de volledige tak van de “other sequences”, rood opgevuld worden. Bij de overige nodes wordt de standaard opvul- en randkleur gebruikt.

Broncode 5.6: JavaScriptgedeelte van de voorbeeldcode ter illustratie van abstractie van de treeview-visualisatie

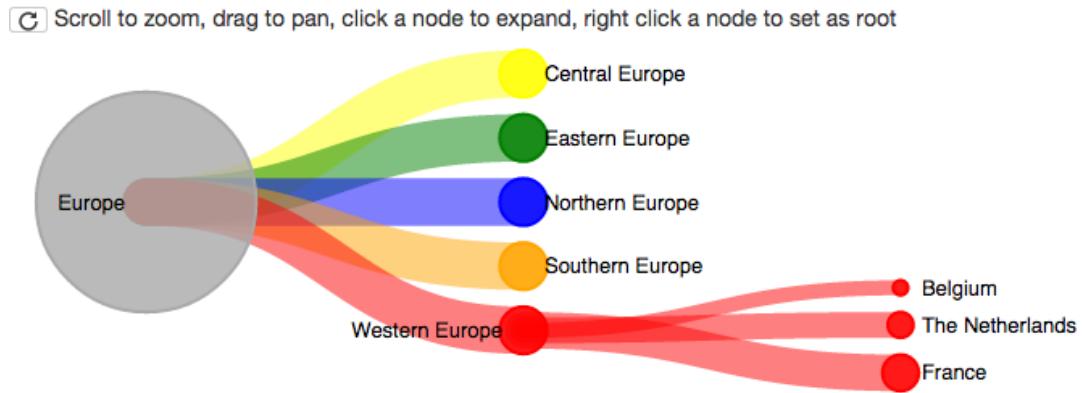
```

1 // Eigen implementatie van de nodeStrokeColor functie
2 var nodeStrokeColor = function(d) {
3   if(d.data.valid_taxon !== 1) {
4     return "red";
5   }
6   if (d.data.rank === "no rank") {
7     return "gray";
8   }
9   return $.fn.treeview.Constructor.NODE_STROKE_COLOR(d);
10 };
11
12 // Eigen implementatie van de nodeFill functie
13 var nodeFillColor = function(d) {
14   if(d.data.valid_taxon !== 1) {
15     return "red";
16   }
17   return $.fn.treeview.Constructor.NODE_FILL_COLOR(d);
18 }
19
20 d3.json("../sample-data/hierarchical-taxons-flat.json",
21 function(error, data) {
22   if (error) return console.warn(error);
23
24   /* Initieert de treeview voor het #d3TreeView element waarbij we de eigen
25    functies meegeven */
26   $("#d3TreeView").treeview({
27     data: data,
28     nodeStrokeColor: nodeStrokeColor,
29     nodeFillColor: nodeFillColor,
30   });
31 });

```



Figuur 5.5: Resultaat van de HTML-code in Broncode 5.6



Figuur 5.6: Resultaat van de visualisatie van niet-biologische invoerdata uit Broncode 5.7

### 5.4.3 Niet-biologische invoerdata

Zoals in de inleiding vermeld, willen we niet enkel de visualisaties specialiseren op biologische data. We kunnen zonder problemen eender welke hiërarchische data omvormen tot een JSON-object dat kan gevisualiseerd worden in de **treeview**. Zo kunnen we bijvoorbeeld een opdeling van Europa definiëren aan de hand van JSON in Broncode 5.7 op pagina 73. Dit resulteert in de visualisatie in Figuur 5.6 op pagina 72.

## 5.5 Optimalisaties aan de bestaande **treeview**-visualisatie

Aan de bestaande treeview-visualisatie konden ook nog enkele verbeteringen worden doorgevoerd.

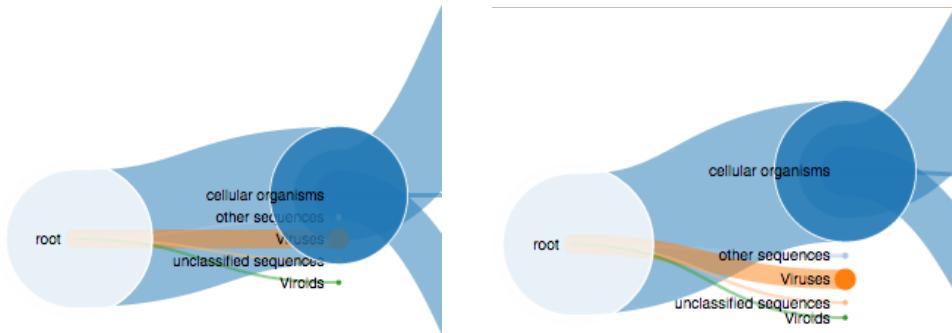
Een eerste probleem was dat de huidige code ervoor zorgde dat de opgegeven hoogte altijd behouden werd. Bij kleine visualisaties was dit geen probleem. Wanneer er echter een grote hoeveelheid data werd weergegeven, dan zorgde dit ervoor dat kleine nodes soms onder grote nodes terechtkwamen. Hierdoor waren sommige nodes niet meer aanklikbaar. Dit effect is te zien op Sectie 5.5 op pagina 74. De oorzaak hiervan was dat de D3 **tree** layout werd ingesteld met een vaste hoogte via `d3.layout.tree().size([height, width])`.

Om dit probleem tegen te gaan heeft D3 een ingebouwde functie, `tree.seperation()`[21]. Deze functie zorgt ervoor dat een meegegeven functie kan gebruikt worden om de afstand tussen de nodes te bepalen. De functie vereist wel dat ook de `nodeSize` van de `d3.tree` wordt ingesteld. De resulterende code kan worden gevonden in Broncode 5.8 op page 72. Samengevat wordt de afstand tussen de nodes berekend aan de hand van hun `nodesize`, met enkele pixels extra ruimte ertussen. Wanneer beide nodes niet dezelfde ouder hebben, wordt de extra ruimte nog wat uitgebreid zodat er een duidelijkere scheiding is tussen groepen kinderen van verschillende oudernodes. Het resultaat van deze aanpassing is te zien op Sectie 5.5 op pagina 74. Alle nodes hebben nu evenveel ruimte tussen elkaar en overlappen niet, waardoor alle nodes aanklikbaar blijven.

Broncode 5.7: JSON-representatie van een opdeling van Europa als invoer voor de treeview-visualisatie.

```

1  {
2      "id": 1,
3      "name": "Europe",
4      "data": {
5          "count": 100,
6          "self_count": 100
7      },
8      "children": [
9          {
10             "id": 2,
11             "name": "Central Europe",
12             "data": {
13                 "count": 20,
14                 "self_count": 20,
15                 "color": "yellow"
16             }
17         },
18         {
19             "id": 6,
20             "name": "Western Europe",
21             "data": {
22                 "count": 20,
23                 "self_count": 20,
24                 "color": "red"
25             },
26             "children": [
27                 {
28                     "id": 7,
29                     "name": "Belgium",
30                     "data": {
31                         "count": 5,
32                         "self_count": 5
33                     }
34                 },
35                 {
36                     "id": 8,
37                     ...
38                 }
39             ]
40         }
41     ]
42 }
```



Figuur 5.7: Voorbeeld van voor (links) en na (rechts) het oplossen van overlappende nodes van de `treeview`-visualisatie. Links zien we dat de nodes “other sequences” en “viruses” onder node zitten van “cellular organisms”. Rechts is dit probleem opgelost: geen enkele node overlapt, waardoor ze aanklikbaar blijven.

Broncode 5.8: Dynamische schaling van plaats tussen nodes in de `treeview`-visualisatie

```

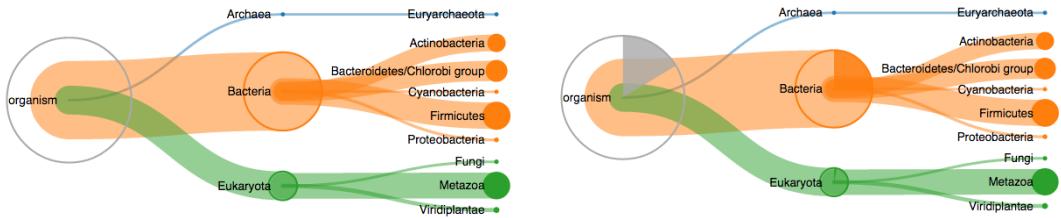
1 | tree = d3.layout.tree()
2 |   .nodeSize([2, 10]);
3 |   .separation(function(a, b) {
4 |     var width = (nodeSize(a) + nodeSize(b)),
5 |         distance = width / 2 + 4;
6 |     return (a.parent === b.parent) ? distance : distance + 4;
7 |   });

```

Een tweede probleem, ditmaal specifiek toegepast op de metagenomics data, is het volgende: stel dat we een hiërarchische visualisatie willen weergeven van het aantal gevonden taxons in een sample data, dan kan dat worden weergegeven op de `treeview`-visualisatie waarbij de grootte van nodes overeen komt met het aantal samples gevonden in die tak van de boom. In sectie 5.5 op pagina 75 wordt de bijhorende visualisatie gemaakt ter analyse van een stoelgangstaal van een mens. Het is duidelijk te zien dat de grootste aantallen gevonden worden in de tak van de bacteriën en van de eukaryoten. Wat echter niet meteen zichtbaar is, is het onderscheid tussen hoeveel peptides worden gemapt op het niveau van de bacteriën zelf, en hoeveel eigenlijk dieper in die tak worden gevonden. We kunnen aan de grootte van de nodes inschatten hoeveel dit ongeveer zou kunnen zijn, maar naarmate een node meer en meer kinderen heeft, wordt dit steeds.

Aangezien de nodes worden voorgesteld door middel van cirkels is het aangewezen om dit als een taartdiagram aan te duiden met twee delen: het doorzichtige deel is specifiek voor die tak, maar niet voor de node, terwijl het opgevulde deel het aantal elementen voorstelt specifiek voor de node. Door op de node zelf visueel aan te duiden hoeveel peptides specifiek tot op dat niveau en niet dieper worden gemapt, kunnen we dit proces gemakkelijker maken.

Momenteel worden de nodes in D3 weergegeven door een `circle` achteraan toe te voegen



Figuur 5.8: Voorbeeld van voor (links) en na (rechts) het toevoegen van slices in de nodes van de treeview-visualisatie. Links is moeilijk in te schatten hoeveel procent van de eiwitten op organismen niveau gemapt worden. Rechts kunnen we dit wel meteen inschatten aan de hand van de gekleurde slice op de node.

aan de node-objecten. Deze `circle` krijgt een aantal attributen zoals straal, lijndikte, opvullingskleur en lijnkleur. Op dezelfde manier dat we een cirkel toevoegen, kunnen we ook een D3 `arc` toevoegen[18]. Dit resulteert in de volgende code:

```

1 // Definieert een arc-scale zodat deze altijd tussen 0 en 2 Pi ligt
2 arcScale = d3.scale.linear().range([0,2*Math.PI]);
3
4 // Definieert een arc-object
5 var arc = d3.svg.arc()
6   .outerRadius(nodeSize)
7   .startAngle(0)
8   .endAngle(arcSize);
9
10 // Voegt de arc toe aan de node wanneer deze wordt upgedated
11 nodeUpdate.select("path")
12   .duration(duration)
13   .attr("d", arc);

```

Wanneer een node niet is opengeklapt wordt altijd de volledige node gevuld. Het resultaat van die aanpassing is te zien op Sectie 5.5 op pagina 75. De nodes “organism”, “Bacteria” en “Eukaryota” hebben nu allemaal een slice die aangeeft hoeveel peptides er in die dataset specifiek op die node gemapt zijn. Dit is in één oogopslag te zien, zonder dat het aantal moet worden afgeleid uit de grootte van de kinderen.

Naast deze twee wijzigingen, werden ook nog enkele geduplicateerde code in functies ondergebracht, maar dit laten we hier nu buiten beschouwing.

## 5.6 Benchmarking

We kunnen ons de vraag stellen hoe performant de treeview-visualisatie is en of deze performant blijft naarmate we meer en meer data toevoegen. Momenteel wordt de visualisatie zonder problemen gebruikt om de resultaten van UniPept Metaproteomics Analysis Pipeline visueel weer te geven. Wanneer we echter met metagenomics data werken is deze vaak veel groter dan metaproteomics data, wat misschien mogelijk problemen kan opleveren.

De ultieme manier om dit te testen is om de grootst mogelijke dataset in te laden voor metagenomics data: de volledige NCBI taxonomie [20]. Deze taxonomie bevat op het moment van schrijven 1.267.511 taxa. Een screenshot van deze visualisatie kan gevonden worden in Figuur 5.9 op pagina 77. Aangezien de visualisatie dynamisch en interactief is, kan deze ook in zijn geheel bekijken worden op <https://github.ugent.be/pages/unipept/unipept-visualizations/samples/treeview-ncbi-taxonomy.html>.

Vanaf een lokale machine duurt het gemiddeld vier seconden om de visualisatie te laden. Vanaf de Github Pages is dit ongeveer 51 seconden. Let wel dat deze data volledig in JavaScript geladen wordt. Dit gebeurt volledig client side wat dus ook wil zeggen dat alle data eerst gedownload dient te worden, alvorens de visualisatie kan worden opgezet. De data die in deze visualisatie geladen wordt, is als platte tekst ongeveer 147 MB groot. Met een connectie van 25 mbit per seconde is van die 51 seconden dus ongeveer 47 seconden download, en vier seconden effectieve rendering.

Bij dit laatste is het belangrijk dat we een onderscheid maken tussen data die lokaal geladen wordt en data die via een externe server in de browser geladen wordt. Aangeboden visualisaties op een website zullen hoogstwaarschijnlijk nooit zo groot zijn als de volledige NCBI taxonomie. Wanneer data van deze omvang wordt gevisualiseerd, dan zal dit meestal het gevolg zijn van een analyse op een lokale machine, waarbij de data van dezelfde machine afkomstig is en men niet gebonden is aan een beperkte downloadsgeschwindheid.

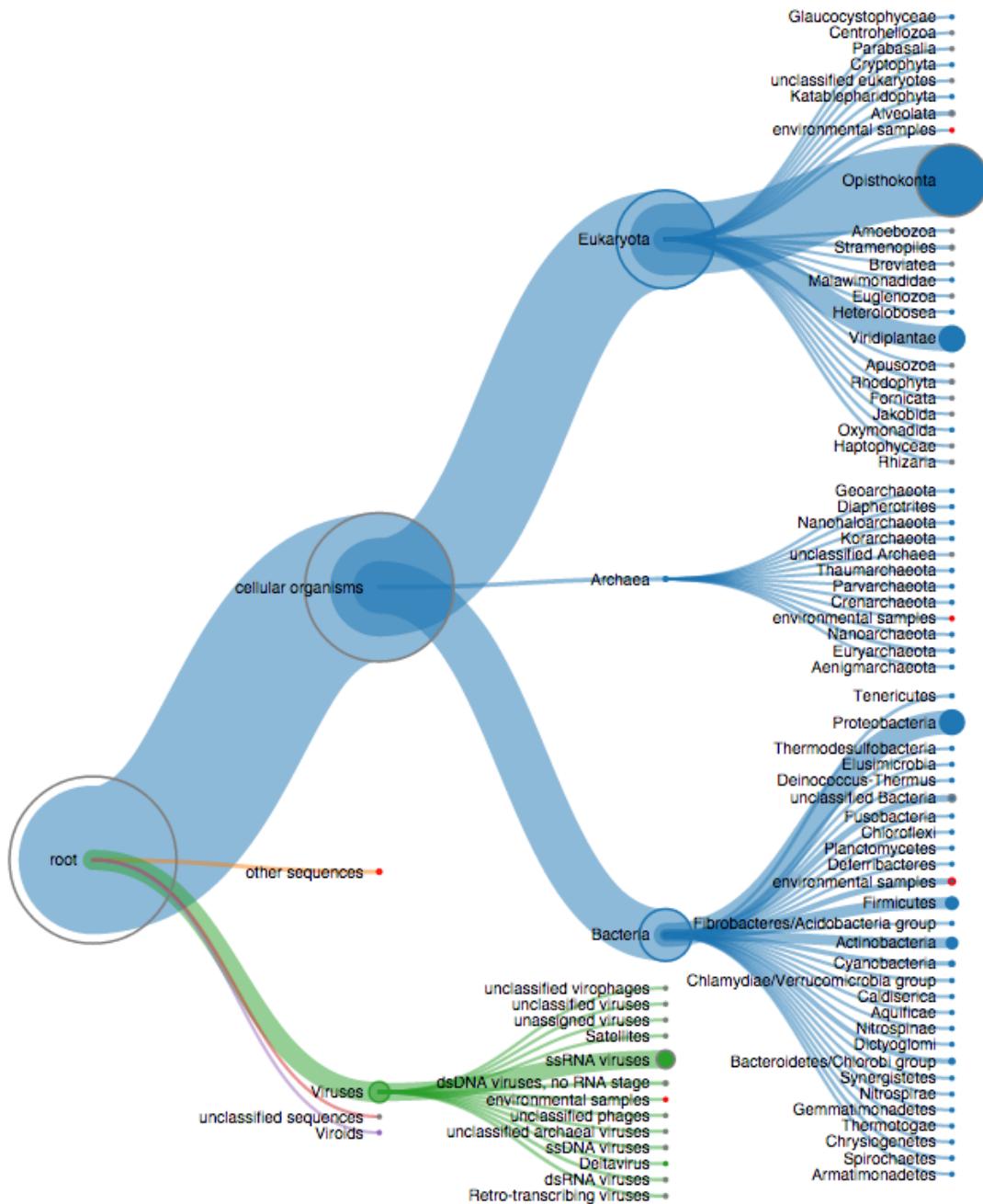
Een ander punt is dat de visualisatie ook in alle gevallen responsief dient te blijven, ook met grote hoeveelheden data. Aangezien D3 enkel de data toont en animeert die zichtbaar is, is dit geen probleem. We hebben ondervonden dat zelfs wanneer er heel veel nodes openstaan de visualisatie nog steeds responsief blijft.

## 5.7 Conclusie

De visualisaties zijn zoals die initieel in Unipept ingebed zaten, waren niet alleen geschikt om binnen Unipept gebruikt te worden voor de visualisatie en analyse van metaproteomische of metagenomische data, maar kunnen dus ook voor andere doeleinden gebruikt worden. Hiervoor kunnen de visualisaties – zoals overlopen in Sectie 5.1 – op dezelfde manier als de treeview-visualisatie omgevormd worden tot abstracte en modulaire modules, die dan terug in Unipept kunnen ingeplugged worden. De plugins kunnen daarnaast ook gebruikt worden om de resultaten van de command-line interface te visualiseren zonder te steunen op het volledige Unipept-platform en zouden zelfs kunnen gebruikt worden om niet-biologische data te analyseren.

## 5.8 Toekomstig werk

In dit hoofdstuk werd als voorbeeld de treeview-visualisatie geabstraheerd en omgevormd. Zoals echter in de inleiding tot de verschillende visualisaties in Unipept werd vernoemd zijn er nog meer visualisaties dan enkel de treeview. Om tot een volwaardig visualisatieframework



Figuur 5.9: Screenshot van (een deel van) de volledig NCBI taxonomie gevisualiseerd met de `treeview`-visualisatie.

te komen moeten niet enkel de andere visualisaties worden omgevormd, maar moeten we het inplussen van data ook zo gemakkelijk mogelijk maken voor de gebruiker. Hiervoor zouden we de Unipept command-line interface kunnen uitbreiden met een parameter die bijvoorbeeld automatisch het bekomen resultaat visualiseert. Een hele reeks features die momenteel in Unipept zijn ingebouwd zoals het opslaan van een visualisatie als afbeelding of het fullscreen bekijken van een visualisatie zouden moeten worden overgenomen. Het zou ook handig zijn mocht de visualisatiedata eenvoudig kunnen worden verdeeld onder onderzoekers die aan hetzelfde project werken.

# Bibliografie

- [1] Michael A. Bender e.a. „Lowest common ancestors in trees and directed acyclic graphs”. In: *J. Algorithms* 57 (2005), p. 18.
- [2] *Bio.Entrez*. <http://biopython.org/DIST/docs/api/Bio.Entrez-module.html>.
- [3] *Biopython - Biopython*. [http://biopython.org/wiki/Main\\_Page](http://biopython.org/wiki/Main_Page).
- [4] *Bootstrap · The world's most popular mobile-first and responsive front-end framework*. <http://getbootstrap.com/>.
- [5] *bootstrap/carousel.js at master · twbs/bootstrap*. <https://github.com/twbs/bootstrap/blob/master/js/carousel.js>.
- [6] UniProt Consortium e.a. „UniProt: a hub for protein information”. In: *Nucleic Acids Research* (2014), gku989.
- [7] *D3.js - Data-Driven Documents*. <http://d3js.org/>.
- [8] *Hideo Bannai*. <http://www.i.kyushu-u.ac.jp/~bannai/>.
- [9] Felix Van der Jeugt. *Optimaliseren van de Unipept Shotgun Metaproteomics Analysis Pipeline*.
- [10] *jQuery Chaining*. [http://www.w3schools.com/jquery/jquery\\_chaining.asp](http://www.w3schools.com/jquery/jquery_chaining.asp).
- [11] H Li. *lh3/wgsim*. <https://github.com/lh3/wgsim>.
- [12] Bart Mesuere e.a. „The Unipept Metaproteomics Analysis Pipeline”. In: *Proteomics* 15 (2015), p. 1437–1442.
- [13] Bart Mesuere e.a. „Unipept: tryptic peptide-based biodiversity analysis of metaproteome samples”. In: *Journal of proteome research* 11.12 (2012), p. 5773–5780.
- [14] *Misc Software*. <http://www.i.kyushu-u.ac.jp/~bannai/software/misc/>.
- [15] *Module: Typhoeus::Hydra::Queueable - Documentation for typhoeus/typhoeus (master)*. [http://www.rubydoc.info/github/typhoeus/typhoeus/Typhoeus/Hydra/Queueable:queue\\_front](http://www.rubydoc.info/github/typhoeus/typhoeus/Typhoeus/Hydra/Queueable:queue_front).
- [16] *Retrieve / ID mapping*. <http://www.uniprot.org/help/uploadlists>.
- [17] Mina Rho, Haixu Tang en Yuzhen Ye. „FragGeneScan: predicting genes in short and error-prone reads”. In: *Nucleic acids research* 38.20 (2010), e191–e191.

- [18] *SVG Shapes - mbostock/d3 Wiki*. <https://github.com/mbostock/d3/wiki/SVG-Shapes#arc>.
- [19] *taxa2lca / Unipept*. <http://unipept.ugent.be/apidocs/taxa2lca>.
- [20] *Taxonomy browser (root)*. <http://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi>.
- [21] *Tree Layout - mbostock/d3 Wiki*. <https://github.com/mbostock/d3/wiki/Tree-Layout#separation>.
- [22] *typhoeus/typhoeus*. <https://github.com/typhoeus/typhoeus>.
- [23] *Unipept API / Unipept*. <http://unipept.ugent.be/apidocs>.
- [24] Michaël Vyverman e.a. „A Long Fragment Aligner called ALFALFA”. In: *BMC Bioinformatics* 16.1 (2015), p. 159.
- [25] *What is a redundant proteome? Can reference proteomes become redundant? Can reviewed entries be deleted because of the proteome redundancy reduction?* [http://www.uniprot.org/help/proteome\\_redundancy\\_faq](http://www.uniprot.org/help/proteome_redundancy_faq).
- [26] Toon Willems. *Evolutionaire reconstructie door clustering van het bacteriele peptidoom*.
- [27] Derrick E Wood en Steven L Salzberg. „Kraken: ultrafast metagenomic sequence classification using exact alignments”. In: *Genome Biol* 15.3 (2014), R46.

