Strings in Proteomics and Transcriptomics

Algorithmic and Combinatorial Questions in Mass Spectrometry and EST Clustering

Zsuzsanna Lipták

PhD thesis submitted to the
Technical Faculty of Bielefeld University, Germany
for the degree of Dr. rer. nat.

Supervised by
Dr. Sebastian Böcker

Referees
Dr. Sebastian Böcker, Prof. Dr. Jens Stoye

Defense on
July 11, 2005

for Nando

Abstract

This thesis treats two problem areas in bioinformatics which can both be beneficially formalized as string problems.

The first (and larger) part deals with *weighted string problems* as they arise from biotechnological mass spectrometry applications. In a mass spectrometry experiment—put in a simplified manner—the molecular mass of the sample molecules is determined. The aim is to identify the sample, either with or without additional information from a database, relying on the fact that the different building blocks of proteins (namely, amino acids) and of DNA molecules (nucleotides) have different molecular masses. Viewing proteins and DNA molecules as strings, this leads naturally to the definition of weighted strings as strings over a finite alphabet Σ with an additional weight (or mass) function $\mu : \Sigma \mapsto \mathbb{R}^+$.

We develop some results in weighted string combinatorics, and then present efficient algorithms for three weighted string problems which are motivated by mass spectrometry.

First, the *mass decomposition problem* is the problem of finding all compomers whose mass equals a query mass. Here, a compomer is an integer vector specifying the number of occurrences of each character of Σ. A compomer abstracts from the order of characters in a string and identifies instead all strings with the same number of occurrences of each character—since these strings all have the same mass. We also present simulation results and tables detailing the number of compomers for different biomolecules, in the ranges appropriate for mass spectrometry applications.

Next, we give several efficient algorithms for the *submass problem*: to test, for a given weighted string s and a query mass M, whether s has a substring with mass M; to find where such a substring occurs; and other variants. We present an algorithm for binary alphabets which runs in time logarithmic in the length of s. Furthermore, we present several algorithms for the problem where multiple masses are sought; these algorithms encode the submasses of s in a polynomial and rely for their efficiency on Fast Fourier Transform for polynomial multiplication.

The third weighted string problem we discuss is *de novo peptide sequencing*, i.e., recovering the amino acid sequence of a sample peptide from tandem MS data, a specialized mass spectrometry experiment. We describe an algorithm which is an enhancement of a dynamic programming algorithm presented by Chen *et al.* in 2000. We describe our implementation and present some simulation results.

The second part of the thesis deals with *EST clustering*. ESTs (expressed sequence tags) are short DNA sequences which are partial copies of mRNAs; they are produced in a high-throughput manner and submitted to public databases. In

EST clustering, the aim is to produce a partition of a large set of ESTs, where each cluster corresponds to a gene; thus enabling the researcher to identify which genes were being expressed. Clearly, the quality of the clustering is highly dependent on the dissimilarity measure (distance/similarity measure) for strings and on the clustering algorithm employed. We have developed a method for evaluating different string dissimilarity measures and clustering algorithms. Finally, we present simulation results for five dissimilarity measures and one clustering algorithm.

Acknowledgements

First and foremost I would like to thank Sebastian Böcker and Jens Stoye. Sebastian has been a wonderful supervisor, with lots of time for his students; he is always willing to explain or listen—assuming you are fast enough to follow. It's been great working with him, and I found his support and advice invaluable. Jens has been both a great boss and a supportive mentor over these past two years. Bielefeld is a prime location for bioinformatics research, and I am very happy to have been able to work here.

Thanks to Marcel Martin and Henner Sudek for implementing and re-implementing (and re-implementing ...) my algorithms. Particular thanks for being such smart guys and thinking so much before—or at least during—coding. It's been great working with you. Thanks, too, to Matthias Steinrücken; although he never programmed for me, he saved me in many a desperate situation when I had some C++ or general software question.

Thanks to the Deutsche Forschungsgemeinschaft (DFG), which has financed me within the Computer Science Action Program (BO 1910/1-2) for the whole period of my time at Bielefeld University.

Special thanks to Peter Widmayer at ETH Zürich, who supported me over a period of several years, and always gave me the freedom to choose what I wanted to do. I am grateful for the opportunities I was given at ETH and for all the things I learned there. I would also like to thank the team of the Forum for Women in Computer Science (Frauenförderung am Departement Informatik). Working with them has been a very enriching and formative experience for me while at ETH.

I would like to thank the South African National Bioinformatics Institute (SANBI) in Cape Town, in particular its head, Win Hide; as well as Scott Hazelhurst at the University of the Witwatersrand (Wits) in Johannesburg. They together financed a three-month research visit of mine at SANBI and Wits in 2002, and another one for one month in early 2003. These visits gave rise to our work on EST clustering, part of which is included in this thesis. I am looking forward to the continuation of this work. Special thanks to Scott Hazelhurst for being a friend as well as a wonderful collaborator. Thanks also to Cathal Seioghe, then at SANBI, who coordinated the masters course at the time, within which I taught three courses.

I want to express my thanks to my co-authors who have taught me how to do research in a team, and how to write papers together. These are: Sacha Baginsky, Nikhil Bansal, Sebastian Böcker, Mark Cieliebak, Thomas Erlebach, Wilhelm Gruissem, Scott Hazelhurst, Torsten Kleffmann, Matthias Müller, Arfst Nickelsen, Paolo Penna, Jens Stoye, Emo Welzl, and Judith Zimmermann. It has been a particularly enriching experience to work across scientific boundaries: for computer scientists and biologists to try to understand and learn from each other. In particular, many thanks to Win Hide at SANBI and to Sacha Baginsky at ETH for

spending countless hours trying to explain molecular biology to me. Special thanks go to Mark Cieliebak, with whom I shared an office and most of my work over a period of maybe two years at ETH: During this time, we taught each other how to be researchers.

Special thanks to the group Genome Informatics in Bielefeld, which is an amazingly nice collection of people. I hope that they have also benefitted from having had, for the first time, a woman academic among them; I trust the change consisted not only of no longer being able to tell the same jokes over lunch. On top of the many seminars, group meetings and academic discussions, from which I gained a lot, the relaxed atmosphere has been very enjoyable. In particular, thanks to Michael Kaltenbach (Mitch) who has been a great office mate; Constantin Bannert (Conni) for his unique and wonderful sense of humour; Klaus-Bernd Schürmann for initiating the afternoon coffee break; Sergio Carvalho for insights into the Brasilian way of life; Thomas Schmidt for organizing the pool evenings; Rileen Sinha for our many discussions on subtleties of the English language; Michael Sammeth and Gregor Obernosterer for lightening things up; Kim Rasmussen for advice on life with babies in Bielefeld; and the recently arrived Veli Mäkinen, Sven Rahmann, and Anton Pervukhin. And special thanks to Heike Samuel for her kindness in dealing with the day to day craze of a group of useless academics.

Many many thanks to Alexander Sczyrba, Ingo Schurr, Klaus-Bernd Schürmann, Christian Rückert, Hans-Michael Kaltenbach, Lisa Lampert, Mark Cieliebak, Ferdinando Cicalese, and Sacha Baginsky for proofreading parts of this thesis.

Finally, I want to thank my many great friends in Germany, Hungary, Switzerland, and the U.S., who have given me much support and love over these past years. I thank Ingo Schurr in particular, from all my heart, for always being there for me, as a friend and a fellow mathematician. I am grateful to my mother, who gave me so many opportunities, and to my father, who awakened my interest in mathematics at an early age. My daughter Réka Cicalese, born 8 January 2005, showed me wonderful new aspects of life and continues to do so every day. I thank Ferdinando Cicalese, friend, colleague, and husband, for having turned up in my life, and for being who he is: simply the most wonderful person in the world.

Contents

1 Introduction

This thesis is about two applications of the concept of strings in bioinformatics: weighted strings and string dissimilarity measures.

In recent years, focus in bioinformatics research has shifted away from structural genomics towards functional genomics: Since the completion of the draft version of the human genome [Int01, VAM+01], interest has increased in products of the later steps of protein expression and the general workings of the cell. The respective areas are referred to as transcriptomics, proteomics, and metabolomics, while understanding the whole system in all its intertwined complexities, referred to as systems biology, is now often viewed as the ultimate goal.

Nevertheless, the highly successful abstraction of viewing many biomolecules as *strings*, in the computer science meaning of the term, continues to play a vital role. One fairly new area of research is the study of strings over a finite alphabet where each character is assigned a positive real number, a *weight* or *mass*. In this thesis, we refer to these strings as *weighted strings*. Weighted strings appear naturally in the context of *mass spectrometry*, which has become the predominant analysis technique in proteomics. We treat some properties of weighted strings and their underlying combinatorics, and introduce efficient algorithms for several weighted string problems.

Another area that continues to hold interest is that of *dissimilarity* (or *distance*) measures between strings, be it for defining an evolutionary distance on genomes of different species, or, as in the case of *EST clustering*, for alleviating the effect of errors that occur in laboratory processes. EST clustering is a major challenge in interpreting the large amounts of EST data available in public databases. We present a method for evaluating clustering algorithms and string dissimilarity measures for EST clustering.

1.1 General biological background

The Central Dogma of Molecular Biology states that the flow of information in living organisms is from DNA to RNA to protein. Its implementation in the eukaryotic cell is depicted in Figure 1.1. This process is usually referred to as *protein expression*.

In the first step, *transcription* takes place: The double-stranded DNA is unwound, and one strand (the non-coding strand) is copied onto messenger RNA (mRNA). This is done by DNA-dependent RNA polymerase, which attaches to each base its Watson-Crick complement[1]; thus, the newly synthesized mRNA becomes the reverse complement of the non-coding strand, or, equivalently, an exact copy of the

[1] The Watson-Crick complements are: adenine (A)–thymine (T) and cytosine (C)–guanine (G) for DNA; in RNA, T is replaced by uracil (U).

other (the coding) strand, except that it contains a U in place of each T. The position where the transcription starts is called *transcription start site* (TSS), and where it ends, *polyadenylation site* (poly-A site). When the transcription process reaches the poly-A site, it is terminated and a so-called *poly-A tail*, a string of a number of A-bases, is added to the mRNA. This mRNA is also referred to as *pre-mRNA*.

The second step consists of *splicing* the pre-mRNA: Certain parts are cut out (spliced out), and the remaining stretches are bound together, respecting the original order. The stretches that are spliced out are called *introns* and those that remain *exons*. The spliced mRNA is sometimes referred to as *mature* mRNA, but usually just as mRNA.

Finally, the *translation* step follows. The mature mRNA is transported to the ribosome where a protein is assembled according to the sequence of bases in the mRNA: Each consecutive block of three bases (*codons*) codes for an amino acid, according to the *genetic code*. Not all of the mRNA is translated: at both ends, there are *untranslated regions* (UTRs). The 5'-UTR stretches up to the START-codon, where translation starts, and the 3'-UTR begins at the first STOP-codon, where translation ends.

The stretch of DNA that is transcribed is referred to as a *gene*. Genes either code for proteins or for RNA products; the process of RNA production is similar, except that the last step does not take place.

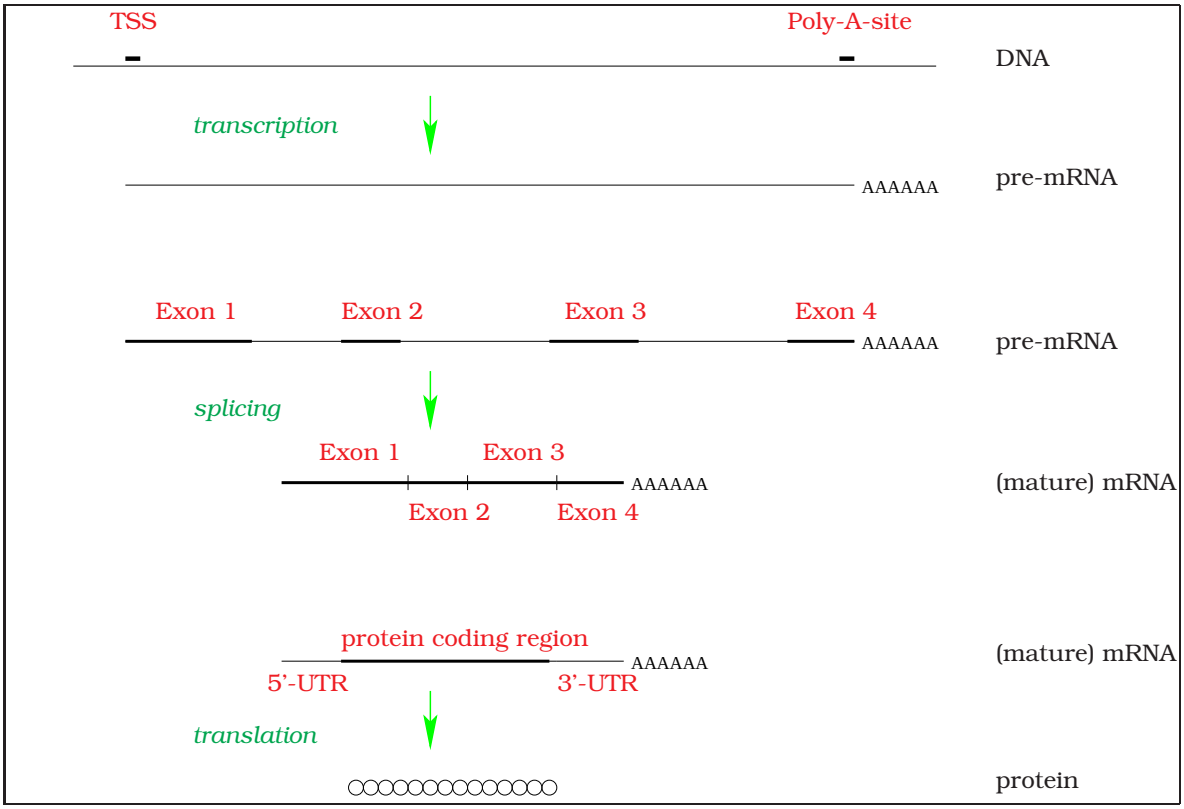


Figure 1.1: The Central Dogma of Molecular Biology (eukaryotes)

The totality of a species' DNA is called its *genome*. The study of the genome is referred to as *genomics*, consisting, among other things, of finding the sequence of bases, location of genes, regulatory elements, repeats (structural genomics), and their role in the complex process of the cell's functioning, including the correspondence of genes and protein products (functional genomics).

All proteins that are produced by a species' cells are referred to collectively as its *proteome*. Proteins are large molecules that play a fundamental role in all living organisms. They are made up of smaller molecules (amino acids) that are linked together in a certain order by peptide bonds. The sequence of amino acids constitutes the so–called *primary structure* of a protein. Protein size ranges from below 100 to several thousand amino acids, where a typical protein has length 300 to 600. Most proteins are made up of the 20 most common amino acids (listed on page 33). Short sequences of amino acids, typically around 20 amino acids, are often referred to as *peptides*. *Proteomics* is the study of a species' proteins: Under what circumstances are they expressed, by what types of cells, what is their function, how do they interrelate, how are they regulated. Expression studies, for instance, attempt to find which proteins are expressed at a given time, differentiating according to cell type, developmental stage, healthy versus pathological tissue, and many other things. For such studies, it is necessary to be able to identify the proteins isolated in the wetlab experiment. This can be done either using a protein database, or by de novo identification, i.e., without employing prior knowledge on existing proteins. For this analysis, very often mass spectrometry (MS) is used, a technique that makes use of the different molecular masses of the amino acids, the basic elements of proteins. A mass spectrometer determines, roughly speaking, the molecular masses of each of the sample molecules from the molecular mixture given as its input. The output thus consists of a list of masses along with their intensities, where ideally, each (mass, intensity)-pair corresponds to a molecule from the sample: The first entry is this molecule's mass, while the second is roughly proportional to the number of molecules of this type in the mixture. We will review in detail the technique of mass spectrometry in Chapter 2. The bioinformatics challenge lies in interpreting this output: First, to recover the primary structure (i.e., the order of amino acids) of the sample molecules from the output, and second, to do this in the presence of the great amount of noise and other errors that occur in mass spectrometry.

The *metabolome* consists of all substances present in the cell, such as sugars or lipids. Mass spectrometry can be and is increasingly employed in metabolomic studies; the string concept can no longer be applied here, but nonetheless, some of the same algorithmic problems occur.

Another way to gain information on which proteins are being expressed in a cell at a given time is to analyze its *transcriptome*, i.e., all mRNAs present in the cell. *Transcriptomics* can thus be viewed as lying inbetween genomics and proteomics: mRNA constites the connection between the genome and the proteome. One common way of analysing mRNA is by using *expressed sequence tags* (ESTs). ESTs are short DNA fragments which are partial copies of mRNAs captured in the cell; they are manufactured in a laboratory process, which we will review in Chapter 7. In order to identify which products were being expressed at the time of mRNA extraction in the cell, *EST clustering* is a method frequently employed: The ESTs produced from the extracted mRNAs are clustered in such a way that each cluster corresponds to

a gene, i.e., to a protein or RNA product. We will discuss EST clustering and the issues involved in detail in Chapter 8.

Two important divergences from the process described above make the analysis of transcriptome and proteome of an organism particularly challengig: First, alternative splicing (described in Chapter 7), and second, posttranslational modifications: these are modifications of the protein, such as phosphorylation, that occur *after* the translation step has taken place. Thus, the mass of substrings of the protein can no longer be derived from knowing the DNA sequence of the corresponding gene alone.

1.2 Weighted strings in computational biology

The two types of biological macromolecules traditionally under investigation in bioinformatics, DNA/RNA and proteins, can be conveniently viewed as strings over a finite alphabet: the alphabet of four nucleotides for DNA and RNA, and that of 20 amino acids for protein. The order of the characters is often referred to as the biomolecule's *primary structure*. By reducing biomolecules to their primary structure, we ignore the far more complicated 3-dimensional structure of these molecules, which play a decisive role in their function. However, in many applications, it suffices to know the primary structure, e.g. for database search. Viewing DNA/RNA and proteins as strings has proved a highly fruitful abstraction in bioinformatics research.

In mass spectrometry experiments, however, only the *mass* of the sample molecule is measured, which is—roughly speaking—the sum of the masses of its characters. Clearly, two molecules consisting of the same characters have the same mass, irrespective of the order of the characters. This leads to two closely related abstractions, weighted strings and compomers: A *weighted string* is a string over a finite weighted alphabet, i.e., an alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ with an additional mass (or weight) function $\mu : \Sigma \to \mathbb{R}^+$. The mass of a string is simply the sum of the masses of its characters. Given a string s over a (weighted or unweighted) ordered alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, its *compomer* $\mathrm{comp}(s)$ is a vector $(c_1, \ldots, c_k)$ with non-negative integer entries, where c_i counts the number of occurrences in s of the character σ_i.

Apart from mass spectrometry, the study of weighted strings and compomers has further applications in those problems on strings over an unweighted alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ where the focus of interest are not the strings themselves, but rather equivalence classes of strings defined by their compomers (i.e., by the multiplicities of characters). The compomer $(c_1, \ldots, c_k)$ thus represents all strings $s = s_1 \ldots s_n$ such that the cardinality of character σ_i in s is exactly c_i, for all $1 \leq i \leq k$. These objects have been referred to in recent publications variously as *compositions* [Ben03], *compomers* [Böc03a, Böc03b], *Parikh-vectors* [Sal03], *multiplicity vectors* [CEL+04]; related mathematical objects are *π-patterns* [ELP03]. A similar approach are so–called *Parikh-fingerprints* [AALS03, Did03]. Here, Boolean vectors of $(b_1, \ldots, b_k)$ are considered, where $b_i = 1$ if and only if character σ_i occurs in the string. Applications range from identifying gene clusters [Did03, SS04] to pattern recognition [AALS03], alignment [Ben03], or SNP discovery [Böc03b].

Even though weighted strings and compomers have ample applications and offer

a rich area of investigation in their own right, suprisingly little research on them exists to date. We will present some theoretical results on weighted strings and compomers, as well as efficient algorithms for several problems in the area.

1.3 String dissimilarity measures in computational biology

The notion of *distance* or *similarity* (which we collectively refer to as *dissimilarity*) between strings has been extensively used in bioinformatics research. Applications include computing the evolutionary distance between species, eliminating the effects of sequencing or other errors that occur in laboratory processes, and database search.

In phylogenetic studies (computing evolutionary distance), the underlying assumption is that the closer related two species, the more similar their genomes: viewed either on the string level—often restricted to highly preserved areas such as genes—or, increasingly, as regards the order of their genes. When eliminating sequencing or other errors, the aim is to correctly relate strings to each other which have been derived from the same original string, as in sequence assembly. Finally, database search for strings or substrings, as with BLAST [AGM+90] or FASTA [LP85], can be viewed as a mixture between these two: The aim is to find strings or substrings in the database which are similar to the query string. These can then be interpreted in one of two ways: either as related to the query string, e.g., the same protein from a different species; or as essentially equal to the query string, e.g., an alternatively spliced version of the same protein, a slightly differing genomic version due to interpersonal variation, or simply the effect of comparing two error–ridden laboratory copies of the original string.

The most commonly applied string similarity measures are alignment–based ones, such as the Levenshtein or edit distance [Lev66], often combined with some heuristics for speedup. However, subword–based measures are increasingly being employed, which compare subword–counts of the two strings. Most prominently, the q-gram distance [Ukk92] has been very successfully applied to approximate matching problems. For fixed $q \in \mathbb{N}$ (where $\mathbb{N}$ denotes the set of positive integers), the q-gram distance between two strings s and t is the sum, over all substrings w of length q, of the absolute difference in the number of occurrences of w in s and t. If we choose $q = 1$, this is just the L_1-distance of the two compomers of s and t, the so-called *compomer distance* of s and t.

In this thesis, we investigate the use of different string dissimilarity measures for EST clustering, and introduce a tool for evaluating them. The tool separates the three components used in EST clustering algorithms, namely the string dissimilarity measure, the clustering algorithm, and the clustering evaluation, and allows to test these individually, on simulated or real data.

1.4 Overview of the thesis

The thesis consists of two major parts: Chapters 2 through 6 deal with weighted string problems motivated by mass spectrometry applications, while Chapters 7, 8, and 9 treat EST clustering and string dissimilarity measures.

We begin with a brief introduction to mass spectrometry and to its application in biotechnology, and discuss what kinds of algorithmic problems it gives rise to (Chapter 2). This chapter also includes a short overview of some of the current bioinformatics literature on mass spectrometry.

Chapter 3 deals with weighted strings and some of their basic properties. This chapter contains all definitions and background for the following three chapters, which deal with particular algorithmic problems arising in mass spectrometry.

In Chapter 4, we discuss the mass decomposition problem, i.e., how to represent a query mass as the sum of given character masses. We introduce an efficient algorithm for producing all such decompositions of a query mass, and show how to use the data structure employed by the algorithm to solve several related problems. The chapter also includes simulation results as well as results on the number of decompositions of biomolecules.

Chapter 5 contains algorithms for the submass finding problem, i.e., whether a weighted string s has a substring t with a given query mass M. In particular, we give a very efficient algorithm for binary alphabets, and several algorithms based on Fast Fourier Transform of polynomial multiplication.

Chapter 6 deals with the de novo peptide sequencing problem and introduces an algorithm which enhances the dynamic programming algorithm introduced in [CKT+01]. The chapter also contains implementation details of our first prototype and some experimental results.

In the second major part of the thesis, we analyze the problem of EST clustering. We first give the biological and technical background in Chapter 7, followed by a chapter on EST clustering and string dissimilarity measures from a computer science perspective (Chapter 8). In Chapter 9, we give details of the implementation of a dedicated tool for evaluating EST clusterings, along with experimental results for a particular clustering algorithm.

We conclude with an outlook to future research and open problems in Chapter 10.

All implementation featured in the thesis was done either with the cooperation or under the supervision of the author. Parts of Chapters 3, 4, 5, 8, and 9 have been published in refereed conference proceedings or journals, and the contents of Chapter 6 as a technical report. Special thanks go to the co-authors of these publications.

Part I

Weighted Strings and Mass Spectrometry

2 Background I: Mass Spectrometry

This chapter starts with a brief introduction to mass spectrometry as applied to proteomics and genomics (Sections 2.1 and 2.2). Detailed introductions can be found in [Siu96, KS00]; see also the overview article [AM03]. Section 2.3 gives an overview of the bioinformatics problems that arise in mass spectrometry applications, including a brief overview of the research literature. Finally, Section 2.4 includes tables of the molecular masses of amino-acids, nucleotides, and atoms found in common biomolecules.

2.1 What is mass spectrometry?

A mass spectrometer is a device which, given a sample molecular mixture as input, determines the molecular mass of the sample molecules, or, more precisely, their mass-to-charge-ratio, commonly referred to as m/z. The mass spectrometer's output is referred to as a *spectrum*. Ideally, a peak in the spectrum indicates the presence of molecules of the corresponding m/z value in the sample, while the height of the peak, referred to as *intensity* or sometimes *relative abundance*, is proportional to the number of molecules with this m/z value. However, both the m/z values and the intensities are influenced by many other factors, as well; in particular, many of those influencing the intensity value remain unclear. Thus, an observed peak in a spectrum must be interpreted as merely a *hint* that there may be a significant number of molecules with the corresponding m/z value in the sample. See Figure 2.1 for an example of a mass spectrometry (MS) spectrum.

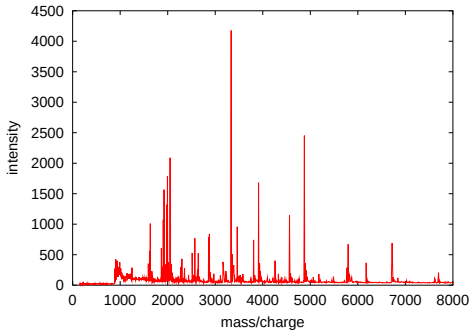


Figure 2.1: A mass spectrum (DNA base specific cleavage)

The process of measurement is indirect: First, the sample is ionized, and then some property is measured which can be correlated to its m/z value (e.g., time-of-flight), or specific ions are separated with a property correlated to a specific m/z value (quadrupole, ion trap). The exact procedures used depend on the particular

type of technology. We will sketch the two most prevalent ones, MALDI-TOF and ESI-Quadrupole.

Every mass spectrometer consists of three main components: the ionizer, where the sample molecules are charged; the analyzer, where, in a vacuum, the sample molecules are separated, according to their m/z value; and the detector, where the individual ions are detected.

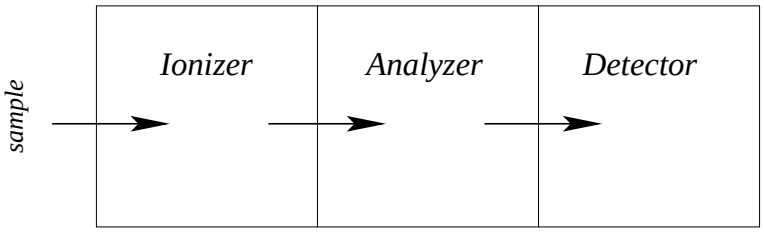


Figure 2.2: General principle of mass spectrometry

The ionizer

A large number of techniques for ionizing molecular mixtures have been introduced over time. Due to the fragile nature of biomolecules, however, the following two are predominant in biotechnology:

- **Matrix-Assisted Laser Desorption/Ionization (MALDI):** Due to Franz Hillenkamp and Michael Karas [KH88, HKBC91]. The sample is dissolved in a UV-absorbing compound, referred to as the "matrix", and left to dry on a probe. It is then shot at with a pulsed UV laser beam, which causes the matrix to evaporate, releasing and ionizing the sample molecules in the process. Ions with (mostly) single positive charge result from the reaction. These are then directed into the mass spectrometer. See Figure 2.3 for a schematic illustration.

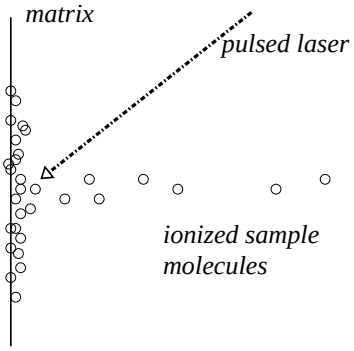


Figure 2.3: MALDI

- **Electrospray Ionization (ESI):** Due to John B. Fenn [WDYF85,FMM+89]. The sample is dissolved in a liquid solvent, and is sprayed from a small diameter needle in the presence of a strong electric field, creating highly charged droplets. The (usually) positive charge of the droplets causes them to move toward the negatively charged entrance lens of the instrument, and during this movement, the droplets are split into smaller charged droplets. (This phenomenon is known as a "Taylor cone": When the mutual repulsion on the surface of the droplet exceeds the forces of surface tension, ions are split off the droplet.) This process can be aided by a gas flow. The splitting process continues until the drops contain only single molecules, in proteins with as much as charge 50+. These molecules are then directed into the mass spectrometer. See Figure 2.4.

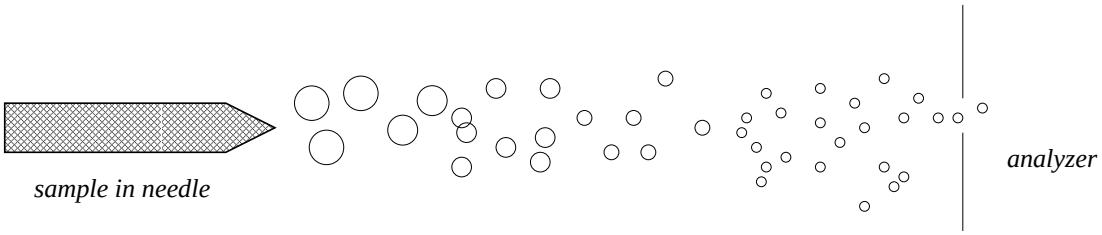


Figure 2.4: ESI

Other techniques include fast atom bombardment (FAB), electron ionisation (EI), and soft laser desorption (SLD). In fact, the Nobel Prize in Chemistry in 2002 was awarded jointly to John Fenn for ESI and Koichi Tanaka for SLD [MG02].

One important difference to keep in mind when interpreting mass spectra is that MALDI results in *singly charged ions*, while ESI leads to *multiply charged ions*. Molecular mass is measured in Dalton (Da), which is approximately the mass of one proton, and m/z in Thomson (Th), even though m/z values are often used as a unit-less ratio. Since the output of the mass spectrometer consists of mass-to-charge-ratios m/z rather than mass values, a sample molecule with molecular mass 1000 Da which is doubly charged by the addition of two protons would give rise to a measurement of 501 Th. Moreover, *a priori* it is impossible to distinguish, say, between a singly charged ion of mass 1000 Da, and a doubly charged ion of mass 500 Da. However, a distinction can be made using the isotopic patterns of the molecules: Isotopes differ in the number of neutrons they have in the nucleus, and they occur in nature with different frequencies, e.g., 98.892% of carbon atoms have 6 neutrons, while 1.108% have 7 neutrons. Thus, one ion will typically give rise to one main peak, and several isotopic peaks, which differ by 1 Da for each neutron if $z = 1$, and by 0.5 Da if $z = 2$.

The analyzer and the detector

Many different mass analyzer techniques are in use; again, we concentrate on two of the most common analyzer techniques for the sake of brevity: *time-of-flight* analyzers are predominantly used in connection with MALDI, and *quadrupole* analyz-

ers with ESI. We briefly sketch the technologies employed. Recall that the analysis takes place in a high vacuum.

- **Time-of-flight (TOF):** All ions are accelerated in an electric field with the same energy, and directed to drift accross the analyzer towards the detector. Thus, "smaller" (or "lighter") ions will arrive sooner at the detector, while "larger" (or "heavier") ions will take longer. The m/z value is then computed from the time measured between the acceleration and the arrival at the detector. In Figure 2.5, we sketch the principle of MALDI combined with a time-of-flight analyzer.

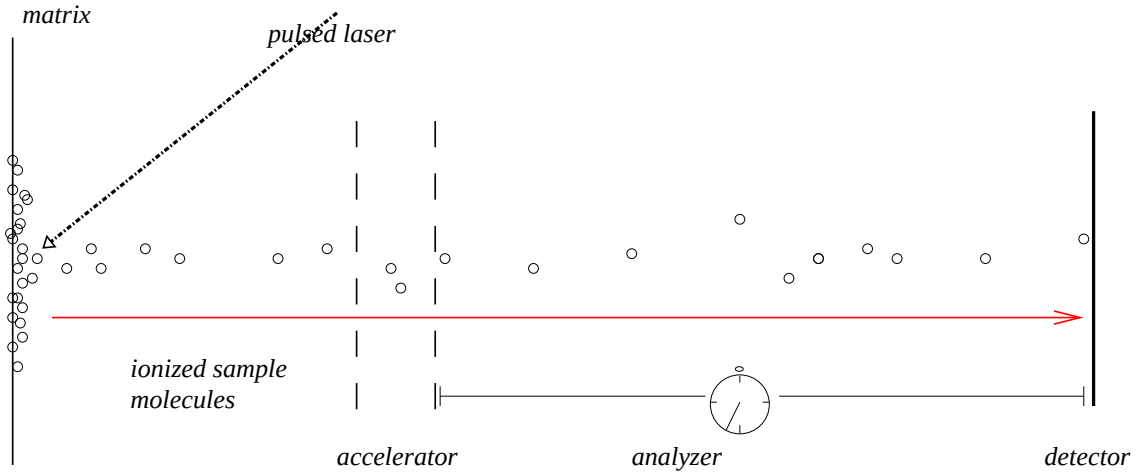


Figure 2.5: MALDI-TOF

- **Quadrupole mass filter:** The ions are directed to drift between four parallel rods with a specific electromagnetic field, which functions as a mass filter: Only ions with the appropriate m/z are allowed to pass straight through the field, while the others are diverted and filtered out. A spectrum is obtained by varying the properties of the field during the experiment. See Figure 2.6.

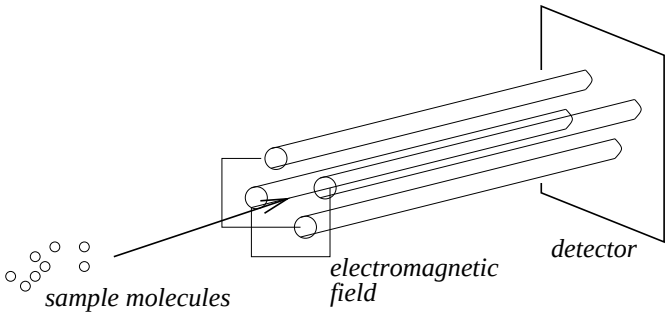


Figure 2.6: Quadrupole analyzer

The detector can measure the arrival of the ions by converting kinetic energy into an electrical current. Again, different technologies are used, but the details are not relevant to interpreting the spectrum.

2.2 Mass spectrometry in proteomics and genomics

Mass spectrometry is widely used in proteomics and genomics for identifying sample molecular mixtures. These may be protein mixtures, DNA molecules, or metabolites. In the following, we sketch how samples are prepared before they are introduced into the mass spectrometer, and then discuss three common applications of mass spectrometry in proteomics. Less commonly, mass spectrometry methods are also used for DNA molecules, for problems such as pathogen identification or SNP discovery. See Section 2.3 for more details.

Preparation of samples

Before introducing a sample into the mass spectrometer for analysis, it is separated in a first step. This is necessary since biomolecules are normally presented in mixtures, e.g. when extracted from the cell. A widely used separation method for proteins is two-dimensional gel electrophoresis (2DE), where the protein mixture is separated according to its size/mass, and its isoelectric point/pH-value. This is done in two steps, where in both, the mixture is allowed to migrate in one direction on a surface covered with a gel. First, the molecules are separated according to their size/mass, then introduced on a square probe where they are separated once more according to their pH-value. This results in spots where molecules are grouped according to these two values; selected spots are extracted and processed further. Other separation methods include liquid or gas chromatography (LC or GC), where the latter is used more commonly for metabolites.

The next step is, in many cases, a biochemical dissociation of the sample molecules. In the case of proteins, the molecular mixture is *digested*, i.e., a site-specific cleavage enzyme, most commonly trypsin, is used: Trypsin cuts after each arginine (one-letter-code R) and lysine (K), unless followed by a proline (P). For DNA molecules, *RNAse digestion* can be employed, consisting of four base-specific cleavage experiments: The molecules are cut after each T for thymine cleavage, after each C for cytosine cleavage etc.

The resulting molecular mixture is introduced into the mass spectrometer. Sample identification can then be achieved either by comparing the resulting mass spectra to a database, or *de novo*, i.e., without any external information.

Peptide mass fingerprinting

Peptide mass fingerprinting (PMF) [HWS03], also referred to as peptide mass mapping, is a method to identify proteins which relies on a protein database. The sample is digested, most commonly using trypsin as cleavage agent, resulting in a list of masses of tryptic peptides, the protein's *tryptic mass fingerprint*. This fingerprint is compared to a tryptic indexed database, i.e., a protein database which has

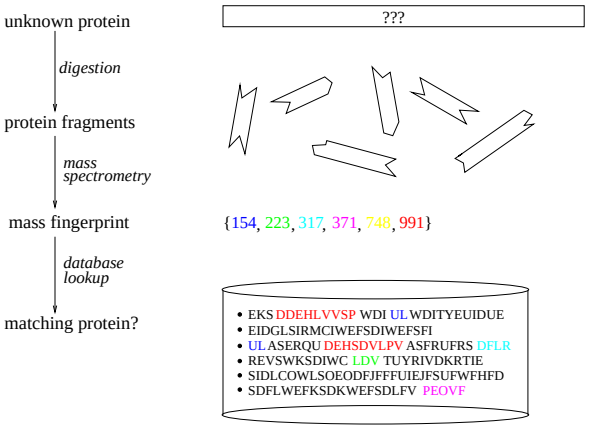


Figure 2.7: Protein identification with PMF

been preprocessed to include the tryptic fingerprints of all its entries as computed from the primary structure of the database entries. The term ”fingerprint” here is misleading: The list of masses cannot be unique, at least in theory, since two peptides with the same multiplicity of amino acids will have the same mass.

The comparison of the two fingerprints can also be done more precisely: Instead of only comparing the two lists of masses, one can compare the experimental spectrum itself to theoretical spectra of the database proteins, thus also taking the intensity values of the masses into account. See Figure 2.7 for a schematic view of PMF.

Tandem mass spectrometry

Tandem mass spectrometry (MS/MS) is another method that can be used for protein identification using a database. Tandem mass spectrometry consists of two phases. In the first phase, a spectrum of the sample peptide mixture is produced as for PMF. Then certain intense peaks in this spectrum are selected one after another, and for every such peak, ions giving rise to this peak are extracted. These ions are now subjected to another dissociation process, usually collision induced dissociation (CID), where they are transmitted to a high-pressure region of the tandem mass spectrometer containing gas molecules (so-called *MS/MS in space*). The collision with these gas molecules results in fragmentation of the sample ions. The majority of the ions fragment only once, thus the output of the experiment consists mainly of fragments that are either prefixes or suffixes of the original peptide string. The fragmentation can occur in several places between two residues along the backbone of the peptide, each fragmentation site giving rise to different types of ions, with slightly differing masses. The most common ones are so-called b-ions (prefixes) and y-ions (suffixes). Typical lengths of peptides are between 10 and 20 amino acids. See Figure 2.8 for a schematic view of MS/MS.

Now, for the identification, again a protein database is used. Details vary in different applications. One possibility, however, is to select in a first step candidate proteins from the database whose mass fingerprint is close to that of the experimen-

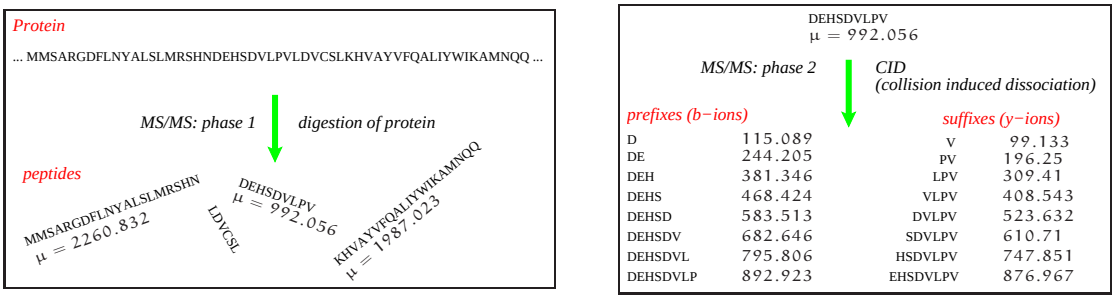


Figure 2.8: Tandem mass spectrometry (MS/MS)

tal protein(s); and then to compute theoretical tandem mass spectra of the peptides selected and compare them to the experimental spectra of the second phase. See Figure 2.9.

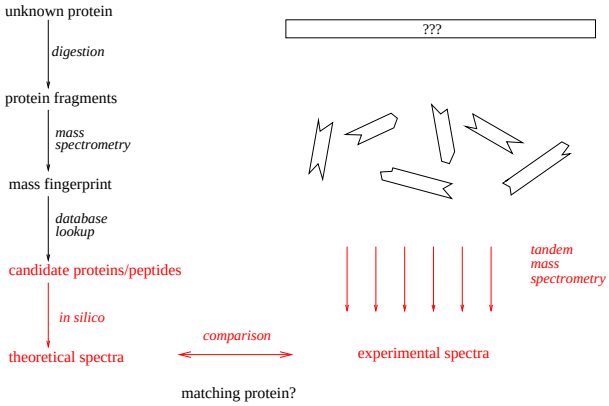


Figure 2.9: Protein identification with MS/MS

De novo peptide sequencing

A tandem mass spectrum of a peptide as described above can also be used for *de novo* peptide sequencing. The goal here is to deduce the primary structure of the peptide (i.e., the string) from its tandem mass spectrum. The underlying idea is that if the spectrum is of good quality, then it should contain peaks corresponding to each prefix and each suffix of the peptide, see Figure 2.10.

2.3 Algorithmic challenges in mass spectrometry

Interpreting mass spectrometry data for identifying biomolecules presents us with a number of interesting algorithmic and computational problems.

First, the measurements done by the mass spectrometer (the raw data) need to be converted into a spectrum as visualized in Figure 2.1, i.e., a list of $(m/z$, intensity)-pairs. This step involves *calibration*, the elimination of systematic shifts in the data, and *peak detection*, the discretization of the signal. Both of these are usually done

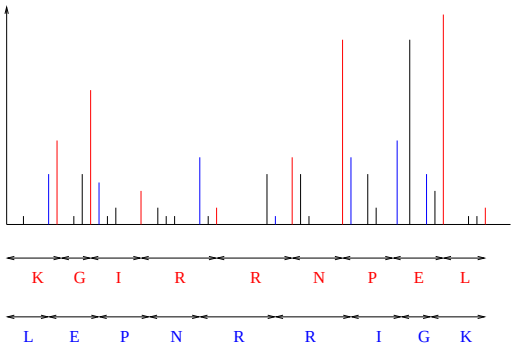


Figure 2.10: De novo sequencing with tandem mass spectrum: Prefixes (above) and suffixes (below) of the solution peptide KGIRRNPEL matching the experimental spectrum.

by the software of the mass spectrometer, but in fact deserve more attention, and are increasingly coming under the scrutiny of computer scientists.

All algorithms for interpreting mass spectra need to deal with the *large amount of noise* contained in the data, such as chemical substances present in the machine, residues from earlier experiments, and random noise. The fact that the intensity value does not in general correspond to the number of molecules measured eliminates any straightforward approach such as a general cutoff at a certain peak height. For example, a tandem mass spectrum ideally contains peaks corresponding to all b- and y-ions (prefixes and suffixes) of the sample peptide; at a typical length of around 10 amino acids, this would yield about 20 peaks. However, a real-world tandem mass spectrum consists of several hundred peaks. Thus, noise leads to *additional peaks* in the spectrum (w.r.t. to the sample). Another problem are *missing peaks*, i.e., peaks that should in theory be present but are not. Yet another challenge is interpreting the additional information relayed by the distribution of the different isotopes (esp. of carbon); thus a real peak is expected to have one or several small accompanying isotopic peaks. This is called *isotopic deconvolution*.

Protein identification with protein database

We give a schematic overview in Figure 2.11 of some of the algorithmic issues involved in protein identification using a protein database, either with PMF or with MS/MS. On the left, we see the experimental pipeline: From the unknown protein or protein mixture, protein fragments (peptides) are won by digestion; these are introduced into the mass spectrometer, which produces a mass fingerprint of the sample protein. In the case of PMF, the experimental part ends here; in the case of MS/MS, more mass spectrometry measurements follow producing tandem mass spectra of selected peptides. On the right hand side, we see the *in silico* processing of the database: First, candidate proteins and peptides have to be identified which could match the sample. For the comparison with the PMF of the sample protein, criteria have to be set as to what constitutes a good match. In addition, for MS/MS, theoretical tandem mass spectra of the peptides selected are produced to be com-

pared with the experimental peptides. Again, a good measure is sought as to when two spectra (in this case, an experimental and a theoretical one) match well.

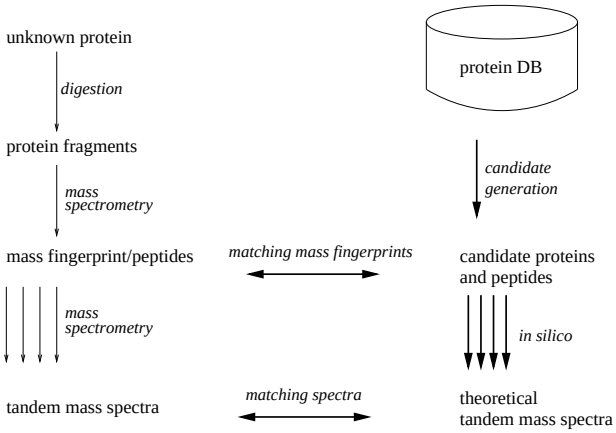


Figure 2.11: Some algorithmic challenges in protein identification with MS/MS data using a database (bold arrows)

Several papers have addressed the question of how to match theoretical (i.e., predicted) and experimental spectra, among them [PDT00, PMDT01], where a so-called spectral convolution was presented. A probabilistic model (SCOPE) was introduced in [BE01]. The software Sequest (see below) includes a method of how to match a predicted and an experimental tandem mass spectrum, described in [EMI94].

A suffix tree was used in [EL02] to efficiently store the large number of candidates from the database. For matching mass fingerprints, the simplest approach is to assume that the larger the number of matches, the more likely that the correct protein has been found. This approach was employed in [MHR93] and others. Another approach was introduced in [PHB93], where the database was preprocessed, computing the frequency of peptides within a certain mass range matching proteins of mass within a given range. This gives information as to the significance of a hit. The score is then normalized for proteins of a fixed average mass to eliminate the effect that heavier proteins have more matches. This approach was implemented in the software Mowse.

Another approach for protein identification with a database is presented in [LC03b], where the dynamic programming algorithm for de novo sequencing of [CKT+01] (see next section) is combined with a suffix tree based indexing of the protein database.

There are two widely used software packages for protein identification with a database. The first, Sequest [EMI94, YIEM95, YIEMS95, seq], matches tandem mass spectra to a protein database. It identifies proteins which have peptides (substrings) whose mass equals the total mass of the tandem mass spectrum (the parent mass); it assigns a score to each of these according to a scoring function; and then generates a theoretical tandem mass spectrum for the top 500 peptides, and compares these with cross-correlation analysis to the experimental spectrum.

The second, Mascot [PPCC99, CC02, mas], allows both interpretation of PMF data and MS/MS data. For the PMF, the scoring scheme is probability based: the prob-

ability that peptides of the given masses match by chance is estimated. Mascot is a further development of the software Mowse [PHB93].

The review [YI98] contains an in-depth discussion of protein database searching with mass spectrometry data, as well as more literature up to 1998.

De novo peptide sequencing with MS/MS data

De novo peptide sequencing with MS/MS data is the task to find the amino acid sequence that gave rise to a tandem mass spectrum, without querying a database.

Some early papers, such as [SMMK84, HWH86] solved the problem exhaustively, by generating all amino acid strings with the given total mass, and then comparing them to the tandem mass spectrum. Others [FGT+97, TJ01] transform the spectrum into a graph in which every connected path represents a possible sequence. They use different algorithms to select good matching sequences among the very large number of possible paths. The software Lutefisk [TJ97, TJ01, lut] is one implementation.

The spectrum graph approach was enhanced and the sequencing problem formulated in a graph theoretic setting in [DAC+99] but the algorithm was not described. In 2000, Chen *et al.* [CKT+01], introduced a dynamic programming algorithm for de novo peptide sequencing, which was the first efficient algorithm for this problem. It has since been improved in several directions, among them in [BE03] and [LC03a]. A different dynamic programming approach was introduced in [MZL03, MZL05] and implemented in a software named Peaks [MZH+03].

We present an enhancement of the approach of Chen *et al.* in Chapter 6.

DNA mass spectrometry

Mass spectrometry has been applied successfully to identify pathogens [HSB+03] and SNPs [Böc03b, RDPS+02] in DNA sequences. Moreover, algorithms for de novo sequencing of DNA sequences of short length (up to 200 bases) are also being developed [Böc03a, Böc04].

Combinatorial problems

Two basic combinatorial problems frequently arise in the context of interpreting MS or MS/MS data: The submass finding problem and the mass decomposition problem.

The *submass finding problem* can be stated as follows: Given a string s over an alphabet where each character has a mass (a *weighted alphabet*, see Chapter 3), and given a mass M, does s have a substring with mass M? If so, return such a substring, return the position in s of such a substring, or return all such substrings with their positions in s. This is motivated by database lookup of peptide mass fingerprints where random fragmentation is used, rather than digestion with a known enzyme. In this case, preprocessing the database and storing all submasses of all proteins is not feasible. We will devote Chapter 5 to this problem and its variants.

The *mass decomposition problem* is the problem of determining, for a given input mass M, all possible compomers with this mass. Hereby, a compomer (often called

composition in the mass spectrometry literature) is an equivalence class of strings, where two strings are equivalent if the number of occurrences of each character is equal, or, in other words, if one is a permutation of the other. We will devote Chapter 4 to this problem.

2.4 Mass tables

Tables 2.1 and 2.2 contain the molecular masses of the 20 most common amino acid residues and of the four deoxynucleotides. Table 2.3 contains the molecular weights of the most common bioatoms (the monoisotopic and average mass for phosphor are identical, because all but the isotope ^{31}P are radioactive). In Table 2.4, we list the names and 3- and 1-letter-codes of the amino acids for reference. In each of these tables, we give both the monoisotopic and the average mass. Here, the monoisotopic mass is the mass of a molecule whose elemental composition consists of the most abundant isotopes of those elements; the average mass, instead, is the weighted average of the isotopic masses, weighted by their abundance. We give the masses up to maximal precision; however, in applications, lower precisions are used, depending on the measurement accuracy of the machines. For many algorithms, the masses need to be scaled up to integers; then, the scaling factor depends on the desired precision.

amino acid	mol. composition	monoisotopic mass	average mass
A	$C_3H_5N_1O_1$	71.037113790 Da	71.079323045 Da
R	$C_6H_{12}N_4O_1$	156.101111044 Da	156.188746822 Da
N	$C_4H_6N_2O_2$	114.042927452 Da	114.104467719 Da
D	$C_4H_5N_1O_3$	115.026943030 Da	115.089069711 Da
C	$C_3H_5N_1O_1S_1$	103.009184490 Da	103.143711176 Da
E	$C_5H_7N_1O_3$	129.042593094 Da	129.116158896 Da
Q	$C_5H_8N_2O_2$	128.058577516 Da	128.131556905 Da
G	$C_2H_3N_1O_1$	57.021463726 Da	57.052233860 Da
H	$C_6H_7N_3O_1$	137.058911874 Da	137.142140206 Da
I	$C_6H_{11}N_1O_1$	113.084063982 Da	113.160590603 Da
L	$C_6H_{11}N_1O_1$	113.084063982 Da	113.160590603 Da
K	$C_6H_{12}N_2O_1$	128.094963024 Da	128.175293325 Da
M	$C_5H_9N_1O_1S_1$	131.040484618 Da	131.197889547 Da
F	$C_9H_9N_1O_1$	147.068413918 Da	147.178050372 Da
P	$C_5H_7N_1O_1$	97.052763854 Da	97.117549470 Da
S	$C_3H_5N_1O_2$	87.032028410 Da	87.078627759 Da
T	$C_4H_7N_1O_2$	101.047678474 Da	101.105716944 Da
W	$C_{11}H_{10}N_2O_1$	186.079312960 Da	186.215027571 Da
Y	$C_9H_9N_1O_2$	163.063328538 Da	163.177355085 Da
V	$C_5H_9N_1O_1$	99.068413918 Da	99.133501417 Da

Table 2.1: Molecular masses of amino acid residues (monoisotopic and average)

nucleotide	mol. composition	monoisotopic mass	average mass
adenine (A)	$C_{10}H_{12}N_5O_5P_1$	313.057605034 Da	313.211002878 Da
cytosine (C)	$C_9H_{12}N_3O_6P_1$	289.046371634 Da	289.185716855 Da
guanine (G)	$C_{10}H_{12}N_5O_6P_1$	329.052519654 Da	329.210307591 Da
thymine (T)	$C_{10}H_{13}N_2O_7P_1$	304.046037276 Da	304.197408032 Da

Table 2.2: Deoxynucleotide masses (monoisotopic and average)

element	symbol	monoisotopic mass	average mass
hydrogen	H	1.007825032	1.007975974
carbon	C	12.0	12.011137239
nitrogen	N	14.003074010	14.006726749
oxygen	O	15.994914620	15.999304713
phosphor	P	30.973761500	30.973761500
sulphur	S	31.972070700	32.064388131

Table 2.3: Common bioatoms

amino acid	3-letter-code	1-letter-code
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartate	Asp	D
Cysteine	Cys	C
Glutamate	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

Table 2.4: Amino acid code table

3 Combinatorics of Weighted Strings: Definitions, Problems, and Properties

This chapter is devoted to the theory of weighted strings.

Weighted string problems differ from traditional string problems in one important aspect: While there, substructures of strings (substrings, non–contiguous subsequences, particular types of substrings such as repeats, palindromes etc.) are under investigation, here we are only interested in *weights* of substrings. This means that, on the one hand, we lose a lot of the structure of strings: e.g. the weight of a string is invariant under permutation of letters; on the other, we gain the additional structure of the weight function, such as its additivity. For instance, the submass finding problem (see Section 3.1.3) seems to be related to the problem of searching in $X + Y$, where X and Y are two sets of numbers (see [Fre75] and [HPSS75]). However, we have been able to extend negative results which have been reached for that problem [CDF90]: We can show that this approach (using the naïve solution without preprocessing) cannot lead to an efficient algorithm for our problem. Likewise, using suffix trees, which can be applied to efficiently solve a large number of complex string problems, does not seem to help. The longest common substring problem [Gus97] also has very different characteristics. Another problem that may also appear to be close to the submass finding problem is maximum segment sum [Ben86]; however, it appears that it does not lead to good solutions, either.

In this chapter, we introduce the problems but do not deal extensively with algorithmic questions, since we will present in later chapters algorithms for several of these problems. We introduce weighted alphabets, weighted strings, and compomers, the central objects of much of this thesis. Section 3.1 contains definitions and some simple properties, as well as the problem statements. This is followed by a section on the theory of mass decomposition when all masses are non-negative integers, where we introduce the Frobenius number and detail the generating function approach to computing the number of decompositions of a given mass (Section 3.2). Finally, in Section 3.3, we discuss three functions defined for weighted strings: the number of substrings, of subcompomers, and of submasses. We present upper and lower bounds and extreme examples.

Note that in order to be more general, we give the definitions in this chapter for arbitrary mass functions, i.e., we allow the masses to be real numbers. The problems have quite different characteristics, however, if all masses are integers; for instance, our algorithms in Chapters 4 and 5 assume integer masses. For the applications, this is no real constraint, since there, the masses can best be viewed as rational numbers—which can be scaled up to integers, with the scaling factor dependent on the precision of the measurements and/or the desired precision of the output. In our simulations, we did exactly this; and it can be seen that our

algorithms perform well even with very high precisions (see Section 4.6).

Partial contents of this chapter have been published in [BL05a] and in [CEL+04].

3.1 Definitions and simple properties

A *weighted alphabet* (Σ, μ) is a finite alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ with a *weight* or *mass function* $\mu : \Sigma \to \mathbb{R}^+$. We refer to a string $s = s_1 \ldots s_n$ over Σ as a *weighted string*, where we extend μ canonically to Σ^* and assign

$$s \mapsto \mu(s) := \sum_{i=1}^{|s|} \mu(s_i), \tag{3.1}$$

the *weight* or *mass* of string s. Hereby, $|s|$ denotes the length n of string $s = s_1 \ldots s_n$. Since all masses are strictly positive, $\mu(s) = 0$ if and only if $s = \varepsilon$, where ε is the empty string. We denote by $t \sqsubseteq s$ that t is a non-empty substring of s, i.e., that there are positions $1 \leq i \leq j \leq |s|$ such that $t = s_i \ldots s_j$. For sake of simplicity, from now on we always assume that the alphabet is ordered.

3.1.1 Compomers

Compomers have been referred to by many different names, among them *Parikh-vectors, multiplicity vectors, compositions, composions* (see Section 1.2 for details on the literature). Compomers allow to abstract from the order of characters in a string, and simply count the number of occurrences of each character. In the following, we denote by $\mathbb{N}$ the set of positive integers and by $\mathbb{N}_0$ the set of non-negative integers. We denote the cardinality of a set X by $|X|$.

Definition 3.1.1. A *compomer* c over the (ordered) alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ is a vector with non-negative integer entries, i.e., $c \in \mathbb{N}_0^k$. Σ can be weighted or unweighted. We denote the empty compomer by $0 = (0, \ldots, 0)$. For a compomer $c = (c_1, \ldots, c_k)$, we set $|c| := \sum_{i=1}^k c_i$, the *length* of c. Moreover, if the alphabet is weighted with mass function μ, then we define $\mu(c) := \sum_{i=1}^k c_i \cdot \mu(\sigma_i)$, the *mass* of c.

We can now assign to each string s over Σ a compomer over Σ in a natural way:

Definition 3.1.2. Let s be a string over the finite (weighted or unweighted) ordered alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$. Then, $\mathrm{comp}(s) = (c_1, \ldots, c_k)$, where for $i = 1, \ldots, k$,

$$c_i := \left|\{j \mid 1 \leq j \leq |s|, s_j = \sigma_i\}\right| \tag{3.2}$$

is the compomer associated with s.

Thus, the i'th component of $\mathrm{comp}(s)$ counts the number of occurrences of character σ_i in s. Obviously, if $c = \mathrm{comp}(s)$, then $|c| = |s|$ and $\mu(c) = \mu(s)$. For the empty string, we have $\mathrm{comp}(\varepsilon) = 0$, and ε is the only string with this property.

Note that $\mu(s)$ depends on the existence of a mass function μ, while $\mathrm{comp}(s)$ is defined for all finite alphabets, weighted or unweighted. We now state some simple connections between strings and compomers.

Lemma 3.1.3. *Given a compomer* $c = (c_1, \ldots, c_k)$ *with* $|c| = n$*, the number of strings* s *with* $\mathrm{comp}(s) = c$ *is* $\binom{n}{c_1,\ldots,c_k} = \frac{n!}{c_1!\cdots c_k!}$.

Proof. Clearly, any string s with $\mathrm{comp}(s) = c$ has length n. There are $\binom{n}{c_1,\ldots,c_k}$ many ways of partitioning an n-set into subsets of sizes $c_1, \ldots, c_k$, which is exactly the number of ways of positioning the c_i many σ_i's, for $1 \leq i \leq k$. □

Lemma 3.1.4. *Given an integer* $n \geq 0$*, the number of compomers* c *with* $|c| = n$ *is*

$$\binom{n+k-1}{k-1}. \tag{3.3}$$

Proof. Consider the following graphical representation of a compomer $c = (c_1, \ldots, c_k)$ of length n. On a line of $n+k-1$ dots, place $k-1$ many crosses in the following way: Place a cross on dot number $c_1 + 1$, one on dot number $c_1 + c_2 + 2$, etc. Conversely, each placement of $k-1$ dots corresponds to a compomer by setting c_1 equal to the number of dots before the first cross, and c_i to that of the number of dots between the $(i-1)$'th and the i'th cross. There are obviously $\binom{n+k-1}{k-1}$ many ways to do this, thus follows the claim. For an example, see Figure 3.1.

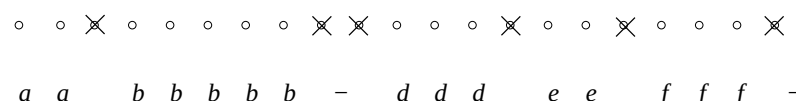


Figure 3.1: Graphical representation of compomer $(2, 5, 0, 3, 2, 3, 0)$ over the alphabet $\Sigma = \{a, b, c, d, e, f, g\}$.

□

Lemma 3.1.5. *Given an integer* $n \geq 0$*, the number of compomers* c *with* $|c| \leq n$ *is* $\binom{n+k}{k}$.

Proof. Follows from Lemma 3.1.4 and the identity $\sum_{i=0}^{m} \binom{i+r}{r} = \binom{m+r+1}{r+1}$ for binomial coefficients, or by introducing an additional "dummy" character $\$ \notin \Sigma$ and applying the lemma for alphabet size $k+1$. □

3.1.2 Compomer decompositions of masses

Given a weighted alphabet (Σ, μ), and a mass $M \in \mathbb{R}^+$, we are interested in decomposing M, i.e., in writing M as a sum of masses of characters from Σ.

Definition 3.1.6. Fix a weighted alphabet (Σ, μ). A mass $M \in \mathbb{R}^+$ is called *decomposable* over Σ if there is a compomer c with $\mu(c) = M$. Such a compomer c is called a *(compomer) decomposition* of M, or a *witness* of M, i.e., of the fact that M is decomposable. We denote by $\gamma(M)$ the number of decompositions of M over Σ.

Recall that the alphabet is ordered; thus if we have two characters with the same mass, then they give rise to different decompositions. For example, assume that

the alphabet is $\{a_1, a_2, a_3\}$ and $\mu(a_1) = 2 = \mu(a_2), \mu(a_3) = 3$, then $(2,0,1), (1,1,1)$ and $(0,2,1)$ are three different decompositions of 7.

As we will see in Section 3.2, if all character masses are relatively prime (or co-prime) positive integers, then there exists an integer $g \in \mathbb{N}$ such that all integers $m > g$ are decomposable but g is not. (If the masses are not relatively prime, then $g = +\infty$.) This number is referred to as the *Frobenius number* of the character masses.

We are interested in the following decomposition problems. Here, the weighted alphabet (Σ, μ), where $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, is fixed, with $a_i = \mu(\sigma_i)$, for $i = 1, \ldots, k$.

DECOMPOSITION DECISION PROBLEM:
Given a mass $M \in \mathbb{R}^+$, is M decomposable over Σ?

DECOMPOSITION ONE WITNESS PROBLEM:
Given a mass $M \in \mathbb{R}^+$, return one compomer decomposition of M, if one exists.

DECOMPOSITION ALL WITNESSES PROBLEM:
Given a mass $M \in \mathbb{R}^+$, return all compomer decompositions of M.

DECOMPOSITION COUNTING PROBLEM:
Given a mass $M \in \mathbb{R}^+$, determine $\gamma(M)$, the number of decompositions of M.

FROBENIUS PROBLEM:
For integer masses $\{a_1, \ldots, a_k\}$, determine the Frobenius number.

In Chapter 4, we will discuss known algorithms solving these problems, and introduce new ones for all but the counting problem.

3.1.3 Submasses and subcompomers

We now extend the notion of substring to compomers and masses.

Definition 3.1.7. Let s be a string over the weighted alphabet (Σ, μ). A mass $M \in \mathbb{R}^+$ is a *submass* of s if s has a substring t with $\mu(t) = M$, or, equivalently, if there exist positions $1 \leq i \leq j \leq |s|$ such that $\mu(s_i \ldots s_j) = M$. We refer to such a pair (i,j) as a *witness* of M in s (i.e., of the fact that M is a submass of s). Similarly, a compomer c is a *subcompomer* of s if there exist positions $1 \leq i \leq j \leq |s|$ with $\text{comp}(s_i \ldots s_j) = c$. Again, we call such a pair (i,j) a *witness* of c in s.

Note that we exclude 0 both as submass and as subcompomer. Thus, the notions of submass and subcompomer parallel the notion of non-empty substrings.

Definition 3.1.8. For a string s and a mass M, let $\kappa(M,s)$ denote the number of witnesses of M in s. For compomers c and strings t, $\kappa(c,s)$ and $\kappa(t,s)$ are defined similarly: Thus, $\kappa(t,s)$ is the number of occurrences of substring t in s, and $\kappa(c,s)$ the number of occurrences of substrings with mass M.

So, $\kappa(M, s) > 0$ if and only if M is a submass of s, and similarly for compomers and strings. Note that for any M, c, t:

$$\kappa(M, s), \kappa(c, s), \kappa(t, s) \leq |s|. \tag{3.4}$$

We can easily deduce the following relationship: For all $s, t \in \Sigma^*, c \in \mathbb{N}_0^k, M \in \mathbb{R}$:

$$\text{comp}(t) = c \text{ and } \mu(c) = M \quad \Rightarrow \quad \kappa(t, s) \leq \kappa(c, s) \leq \kappa(M, s). \tag{3.5}$$

In Chapter 5, we will introduce several algorithms for finding submasses and their witnesses in a given string.

Definition 3.1.9. For $s \in \Sigma^*$, let $\mathbf{s}(s), \mathbf{c}(s)$, and $\mathbf{m}(s)$ denote the number of different substrings, subcompomers, and submasses of s, respectively.

Section 3.3 contains a detailed discussion of these cardinalities. In Chapter 5, which is devoted to submasses of a string, we will develop algorithms for the following problems.

SUBMASS DECISION PROBLEM:
Given a string s and a mass M, is M a submass of s?

SUBMASS ONE WITNESS PROBLEM:
Given a string s and a mass M, return a witness of M in s, if there is one.

SUBMASS ALL WITNESSES PROBLEM:
Given a string s and a mass M, return all witnesses of M in s.

SUBMASS COUNTING PROBLEM:
Given a string s, how many submasses does s have, i.e., determine $\mathbf{m}(s)$.

3.2 Number of decompositions of a mass (integer masses)

Let a_i denote the mass of character σ_i, i.e., $a_i = \mu(\sigma_i)$, for $i = 1, \ldots, k$. For simplicity of exposition, we will assume throughout this section that the masses are pairwise different positive integers, and that they are ordered, i.e., that $0 < a_1 < \ldots < a_k$ holds.[1]

We are now interested in $\gamma(M)$, the number of decompositions of M over the masses $\{a_1, \ldots, a_k\}$. There is no "good" closed form known for $\gamma(M)$. The generating function method, detailed in Section 3.2.2, yields the best closed form, but it is only of theoretical interest due to its high computation cost. In Chapter 4, we will see how to compute $\gamma(M)$ using a dynamic programming algorithm.

[1] The only place where it is really necessary that the masses be pairwise distinct is Theorem 3.2.2.

3.2.1 The Frobenius number

We will see in this section that if $\gcd(a_1, \ldots, a_k) = 1$ and $k \geq 2$, every sufficiently large M has at least one compomer decomposition, i.e., there is an $M_0 \in \mathbb{N}$ such that $\gamma(M_0) = 0$ and for all $M > M_0$, $\gamma(M) > 0$. This M_0 is referred to as the *Frobenius number* $g(a_1, \ldots, a_k)$ of the a_i's. Note that conversely, if $d := \gcd(a_1, \ldots, a_k) > 1$, then there are arbitrarily large numbers that are not decomposable, since only multiples of d can be represented. It is not necessary, on the other hand, to assume that the masses be *pairwise* co-prime.

For $k = 2$, the Frobenius number has been long known:

Theorem 3.2.1 (J.J. Sylvester). *Let* $\gcd(a_1, a_2) = 1$. *Then,* $g(a_1, a_2) = (a_1 - 1)(a_2 - 1) - 1$.

Proof. We mainly follow [Bra42]. First, define $f(a_1, a_2)$ as the largest number N_0 that has no decomposition $N_0 = x_1a_1 + x_2a_2$ using *positive* integers x_1, x_2. Then, clearly, $g(a_1, a_2) = f(a_1, a_2) - (a_1 + a_2)$. We now show that $f(a_1, a_2) = a_1a_2$, which implies the claim. First, to see that a_1a_2 is not decomposable (using positive integers), assume otherwise. Then, there exist $x_1, x_2 > 0$ s.t. $a_1a_2 = x_1a_1 + x_2a_2$, implying $a_1(a_2 - x_1) = x_2a_2$. Since $\gcd(a_1, a_2) = 1$, it follows that $a_2 \mid a_2 - x_1$, i.e., that a_2 is a divisor of $a_2 - x$. In particular, $a_2 \leq a_2 - x_1$, a contradiction to $x_1 > 0$.

Now, let $M > a_1a_2$. Choose $a \leq x_1 \leq a_2$ such that $x_1a_1 \equiv M \mod a_2$ (such an x_1 exists because a_1 and a_2 are relatively prime). Then, $x_2 := \frac{1}{a_2}(M - x_1a_1) > 0$ is an integer, and $M = x_1a_1 + x_2a_2$. □

No explicit formulas for the Frobenius number are known for $k \geq 3$, but there has been considerable work on upper bounds. I. Schur in 1935 generalized Theorem 3.2.1 to an upper bound for arbitrary k:

$$g(a_1, \ldots, a_k) < (a_1 - 1)(a_k - 1). \tag{3.6}$$

This was improved upon by A. Brauer (1942) to what remains to date, to our knowledge, the best bound for the general case:

$$g(a_1, \ldots, a_k) \leq \sum_{i=2}^{k} a_i\left(\frac{d_{i-1}}{d_i} - 1\right), \quad \text{where } d_i = \gcd(a_1, a_2, \ldots, a_i),\ i = 1, \ldots, k. \tag{3.7}$$

Proofs for both (3.6) and (3.7) can be found in [Bra42].[2]

There has been considerable work on bounds for Frobenius numbers for special cases, see [Ram] for a recent survey; see also [Guy94] and papers quoted therein. The following asymptotic result can be shown using generating functions [Wil90]:

Theorem 3.2.2 (I. Schur). *Given* $0 < a_1 < \ldots < a_k \in \mathbb{N}$ *with* $\gcd(a_1, \ldots, a_k) = 1$. *Then,*

$$\gamma(M) \sim \frac{M^{k-1}}{(k-1)!a_1a_2 \cdots a_k} \qquad (M \to \infty).$$

[2] The paper [Bra42] includes results by A. Brauer's professor I. Schur as well as joint results of Brauer and Schur; however, since Schur was Jewish, he was prevented from publishing or lecturing in 1940's Germany and finally encouraged Brauer to publish these results alone.

Now, for $r = 0, \ldots, a_1 - 1$, let n_r be the smallest integer such that n_r is decomposable over $\{a_1, \ldots, a_k\}$ and $n_r \bmod a_1 = r$. Note that, if $n_r = x_1 a_1 + \ldots + x_k a_k$ is a decomposition of n_r, then $x_1 = 0$. The Frobenius number $g(a_1, \ldots, a_k)$ can now be computed as [BS62]:

Theorem 3.2.3 (Brauer and Shockley, 1962). $g(a_1, \ldots, a_k) = \max\{n_r \mid r = 0, \ldots, a_1 - 1\} - a_1$.

Proof. Let $N := \max\{n_r \mid r = 0, \ldots, a_1 - 1\}$ and $r_N = N \bmod a_1$. Then, by definition of $N = n_{r_N}$, $N - a_1$ cannot be decomposed, since $N - a_1 \equiv N \mod a_1$. Conversely, note that for any $M \in \mathbb{N}$, M is decomposable if and only if $M \geq n_r$ where $r = M \bmod a_1$. In other words, M is decomposable if and only if $M > n_r - a_1$, since $M \equiv n_r \mod a_1$. Now choose $M > N - a_1 \geq n_r - a_1$, with $r = M \bmod a_1$, thus M is decomposable, as claimed. □

In Chapter 4, we will introduce an algorithm for computing the n_r in time $\mathcal{O}(ka_1)$. Computing the Frobenius number is NP-hard [Ram96], thus no algorithm can be expected to have runtime polynomial in $\sum_i \log a_i$, the size of the input.

3.2.2 The generating function approach

In this section, we define a generating function whose coefficients equal $\gamma(M)$. We then find a function for the coefficients, which can be computed in a preprocessing step, and then evaluated for each query M. Of course, in general it is not possible to find a closed form for the coefficients of generating functions. However, in this case, we can exploit the fact that the roots of the denominator polynomial are complex roots of unity.

This generating function is one of the classical examples introduced when discussing how to use generating functions to count objects. Here, we are simply going into a bit more detail than is usually done. For an introduction to generating functions, see e.g. [GKP94], Chapter 7, or [Wil90]. We follow mostly [GKP94].

General case

Let $\gcd(a_1, \ldots, a_k) = 1$. Denote by $[z^n]F(z)$ the nth coefficient of the generating function $F(z) = \sum_{n \geq 0} f_n z^n$, thus $[z^n]F(z) = f_n$. Now define the generating function

$$G(z) = \sum_{n \geq 0} g_n z^n := \left(\sum_{n \geq 0} z^{na_1}\right)\left(\sum_{n \geq 0} z^{na_2}\right) \cdots \left(\sum_{n \geq 0} z^{na_k}\right). \tag{3.8}$$

Then g_n, the nth coefficient of $G(z)$, equals the number of compomers with mass n. To see this, note that $G(z)$ is a convolution, and each of the factor generating functions $\sum_{n \geq 0} z^{na_r}$, $r = 1, \ldots, k$, has coefficients which equal 1 if the index is a multiple of a_r, and 0 otherwise. Thus, each compomer $c = (c_1, \ldots, c_k)$ with $\mu(c) = \sum_{r=1}^{k} c_r \cdot a_r = n$ contributes exactly one to g_n.

We will apply the following theorem (cited from [GKP94]):

Theorem 3.2.4 (General Expansion Theorem for Rational Generating Functions). *If a generating function* $R(z)$ *can be written as* $R(z) = P(z)/Q(z)$, *where*

$Q(z) = q_0(1-\rho_1 z)^{d_1}\cdots(1-\rho_\ell z)^{d_\ell}$, *all* $\rho_r, r = 1,\ldots,\ell$, *distinct, and* $P(z)$ *is a polynomial of degree less than* $\deg Q$, *then*

$$[z^n]R(z) = f_1(n)\rho_1^n + \ldots + f_\ell(n)\rho_\ell^n, \tag{3.9}$$

where each f_r *is a polynomial of degree* $d_r - 1$ *and leading coefficient*

$$b_r = \frac{P(\frac{1}{\rho_r})}{(d_r-1)!q_0\prod_{j\neq r}(1-\frac{\rho_j}{\rho_r})^{d_r}}. \tag{3.10}$$

In order to be able to apply Theorem 3.2.4, we write $G(z)$ from (3.8) as follows, employing basic identities for generating functions:

$$G(z) = \frac{1}{1-z^{a_1}}\frac{1}{1-z^{a_2}}\cdots\frac{1}{1-z^{a_k}} = \frac{P(z)}{Q(z)}, \qquad \text{where } P(z)\equiv 1. \tag{3.11}$$

Now, each factor $(1-z^s)$, $s = a_1,\ldots,a_k$, can be explicitly decomposed into linear factors, since the zeros lie exactly at the complex sth roots of unity $\omega_s^j = e^{2\pi i\frac{j}{s}}$, $j = 0,\ldots,s-1$. Therefore,

$$(1-z^s) = -\prod_{j=0}^{s-1}(z-\omega_s^j) = -\prod_{j=0}^{s-1}(-\omega_s^j)(1-\frac{1}{\omega_s^j}z). \tag{3.12}$$

Thus, we get the desired form for $Q(z)$:

$$Q(z) = \underbrace{(-1)^k\prod_{r=1}^{k}\prod_{j=0}^{a_r-1}(-\omega_{a_r}^j)}_{q_0}(1-\rho_1 z)^{d_1}\cdots(1-\rho_K z)^{d_K}, \tag{3.13}$$

where K is the number of distinct roots of unity taken over all a_rs, and the ρs are inverses of the ω_r^js, and thus roots of unity themselves (of the same order). By comparing the constant coefficient with (3.11), we see that $q_0 = 1$, thus

$$Q(z) = (1-\rho_1 z)^{d_1}\cdots(1-\rho_K z)^{d_K}. \tag{3.14}$$

Now, by Theorem 3.2.4, we have that

$$[z^n]G(n) = f_1(n)\rho_1^n + \ldots + f_K(n)\rho_K^n =: g(n). \tag{3.15}$$

The polynomials f_j, $j = 1,\ldots,K$, have degree $d_j - 1$, where d_j is the multiplicity of zero ρ_j. Note that 1 has multiplicity exactly k, since it is a root of each of the factors $(1-z^{a_r})$, $r = 1,\ldots,k$, and all other roots have multiplicity $< k$, since we have assumed that the a_r are co-prime.

Thus, when we have a query M, we simply have to evaluate $g(M)$, which involves evaluating polynomials in M of degree at most k, and exponentiation to the power M, both of which require time $\mathcal{O}(\log M)$.

Pairwise co-prime masses

We now assume that the masses $a_1, \ldots, a_k$ are *pairwise* co-prime, i.e., that $\gcd(a_j, a_{j'}) = 1$ for $j \neq j'$.

Lemma 3.2.5. *The polynomial* Q *from* (3.11) *has a root* $\rho \neq 1$ *with multiplicity* d *if and only if there are* d *distinct numbers from* $\{a_1, \ldots, a_k\}$ *with a common factor, i.e., if there is* $A \subseteq \{a_1, \ldots, a_k\}$ *with* $|A| = d$ *and* $m > 1$ *such that for all* $a \in A$, $m \mid a$.

Proof. Obvious. □

Since $\gcd(a_j, a_{j'}) = 1$ for $j \neq j'$, by Lemma 3.2.5, the only root with multiplicity greater than 1 is 1, with multiplicity k, as before. Thus, all d_js equal 1, for $j = 2, \ldots, K$. Let $\rho_1 = 1$. From (3.15) we get

$$[z^n]G(n) = f_1(n) \underbrace{\rho_1^n}_{=1} + \underbrace{f_2(n)}_{\deg=0} \rho_2^n + \ldots + \underbrace{f_K(n)}_{\deg=0} \rho_K^n \tag{3.16}$$

$$= f_1(n) + b_2\rho_2^n + \ldots + b_K\rho_K^n, \tag{3.17}$$

where

$$b_r = \prod_{j \neq r} (1 - \frac{\rho_j}{\rho_r})^{-1}, \quad \text{for } r = 2, \ldots, K, \qquad \text{by Equation (3.10).} \tag{3.18}$$

Computation

The computation time (both for the general and the pairwise co-prime case) is more or less independent of the query M - to be more precise, the computation for the preprocessing is high but independent of M, while the query step runs in time $\mathcal{O}(\log M)$.

3.3 Number of substrings, subcompomers, submasses

In this section, we will investigate the three functions **s**, **c**, and **m**. Hereby, we denote by $|s|_\sigma = |\{i \mid 1 \leq i \leq |s|, s_i = \sigma\}|$, the number of occurrences of character σ in the string s.

Recall the definitions $\mathbf{s}(s) = |\{t \mid t \sqsubseteq s\}|$, $\mathbf{c}(s) = |\{\mathrm{comp}(t) \mid t \sqsubseteq s\}|$, $\mathbf{m}(s) = |\{\mu(t) \mid t \sqsubseteq s\}|$. Note again that these definitions exclude the empty string. For a string s with $|s| = n$, we have, by Equation (3.5), and the fact that the character masses $\mu(\sigma), \sigma \in \Sigma$, are strictly positive,

$$n \leq \mathbf{m}(s) \leq \mathbf{c}(s) \leq \mathbf{s}(s) \leq \frac{n(n+1)}{2}. \tag{3.19}$$

But are these bounds tight? The first inequality is, which can be seen by the fact that $\mathbf{m}(\sigma^n) = n$ for any $\sigma \in \Sigma$. For the last inequality, equality can be attained by the string $\sigma_1\sigma_2 \ldots \sigma_n$, where $\sigma_i \neq \sigma_j$ for $i \neq j$. However, this latter example requires that the alphabet have cardinality greater or equal to n; since we assume the alphabet

to be constant and finite, this example does not extend to arbitrary n. Furthermore, the first example string uses only one character, which is a special case again.

Thus, the question is, are there non-trivial strings S_n, T_n of length n with $\mathbf{m}(S_n)$, $\mathbf{c}(S_n)$, $\mathbf{s}(S_n) = \Theta(n^2)$, and $\mathbf{m}(T_n)$, $\mathbf{c}(T_n)$, $\mathbf{s}(T_n) = \Theta(n)$? In this section, we will show that this is indeed the case (where for $\mathbf{m}$, we need an additional strong condition on the mass function μ). Note that in order to exclude trivial examples, we are only interested in strings T_n which use all alphabet characters.

Let $|\Sigma| = k$. For $n \in \mathbb{N}, n \geq k$, define strings S_n and T_n:

$$S_n := \sigma_1^{m_1} \sigma_2^{m_2} \dots \sigma_k^{m_k}, \tag{3.20}$$

$$T_n := (\sigma_1 \dots \sigma_k)^m \sigma_1 \dots \sigma_r, \tag{3.21}$$

where $\sum_{i=1}^{k} m_i = n$ s.t. for all $i = 1, \dots, k$, $m_i = \lfloor \frac{n}{k} \rfloor$ or $m_i = \lfloor \frac{n}{k} \rfloor + 1$, i.e., all m_i are approximately equal, and $m = \lfloor \frac{n}{k} \rfloor, r = n \bmod k$. In particular, if n is a multiple of k, then $m_i = m$ for all $i = 1, \dots, k$, and $n = m \cdot k$. Then

$$S_n = \sigma_1^m \sigma_2^m \dots \sigma_k^m, \tag{3.22}$$

$$T_n = (\sigma_1 \dots \sigma_k)^m. \tag{3.23}$$

The following theorems state that the strings S_n and T_n maximize resp. minimize the three functions $\mathbf{s}, \mathbf{c}, \mathbf{m}$ up to a factor of 2.

Theorem 3.3.1 (Asymptotically maximal strings). *Let $n \in \mathbb{N}$ and $n_i \in \mathbb{N}$ for $i = 1, \dots, k$ such that $\sum_{i=1}^{k} n_i = n$.*

1. *S_n has quadratic values $\mathbf{s}, \mathbf{c}, \mathbf{m}$: $\mathbf{s}(S_n), \mathbf{c}(S_n) = \Theta(n^2)$, and for certain classes of weight functions, $\mathbf{m}(S_n) = \Theta(n^2)$.*

2. *S_n maximizes $\mathbf{s}$ up to a factor of 2, for n multiple of k and equal multiplicities: For $m = \frac{n}{k}$,*

$$\mathbf{s}(S_n) \geq \frac{1}{2} \max\{\mathbf{s}(s) \mid s \in \Sigma^n, |s|_{\sigma_i} = m \text{ for } i = 1, \dots, k\}.$$

3. *S_n maximizes $\mathbf{c}$: $\mathbf{c}(S_n) = \max\{\mathbf{c}(s) \mid |s| = n\}$. Moreover, for any fixed multiplicities n_i, $i = 1, \dots, k$, the string $\sigma_1^{n_1} \sigma_2^{n_2} \dots \sigma_k^{n_k}$ maximizes $\mathbf{c}$:*

$$\mathbf{c}(\sigma_1^{n_1} \sigma_2^{n_2} \dots \sigma_k^{n_k}) = \max\{\mathbf{c}(s) \mid s \in \Sigma^n, |s|_{\sigma_i} = n_i \text{ for } i = 1, \dots, k\}.$$

4. *There are classes of mass functions such that S_n maximizes $\mathbf{m}$: $\mathbf{m}(S_n) = \max\{\mathbf{m}(s) \mid |s| = n\}$. Moreover, for these mass functions, the string $\sigma_1^{n_1} \sigma_2^{n_2} \dots \sigma_k^{n_k}$ maximizes $\mathbf{m}$ for any fixed multiplicities $n_i \in \mathbb{N}_0$, $i = 1, \dots, k$:*

$$\mathbf{m}(\sigma_1^{n_1} \sigma_2^{n_2} \dots \sigma_k^{n_k}) = \max\{\mathbf{m}(s) \mid s \in \Sigma^n, |s|_{\sigma_i} = n_i \text{ for } i = 1, \dots, k\}.$$

Theorem 3.3.2 (Asymptotically minimal strings). *Let $n \in \mathbb{N}$.*

1. *T_n has linear values $\mathbf{s}, \mathbf{c}, \mathbf{m}$: $\mathbf{s}(T_n), \mathbf{c}(T_n), \mathbf{m}(T_n) = \Theta(n)$.*

2. T_n *minimizes* $\mathbf{s}$ *up to a factor of* 2, *for* n *multiple of* k *and equal multiplicities: For* $m = \frac{n}{k}$,
$$\mathbf{s}(T_n) \leq 2 \cdot \min\{\mathbf{s}(s) \mid s \in \Sigma^n, |s|_{\sigma_i} = m \text{ for } i = 1, \ldots, k\}.$$

3. T_n *minimizes* $\mathbf{c}$ *up to a factor of* 2, *for* n *multiple of* k *and equal multiplicities: For* $m = \frac{n}{k}$,
$$\mathbf{c}(T_n) \leq 2 \cdot \min\{\mathbf{c}(s) \mid s \in \Sigma^n, |s|_{\sigma_i} = m \text{ for } i = 1, \ldots, k\}.$$

4. *There are classes of mass functions such that* T_n *minimizes* $\mathbf{m}$ *up to a factor of* 2, *for* n *multiple of* k *and equal multiplicities: For* $m = \frac{n}{k}$,
$$\mathbf{m}(T_n) \leq 2 \cdot \min\{\mathbf{m}(s) \mid s \in \Sigma^n, |s|_{\sigma_i} = m \text{ for } i = 1, \ldots, k\}.$$

We devote the rest of this section to the proof of these two theorems.

3.3.1 Number of substrings of a given string

We first prove a lemma which we will need for the above theorems.

Lemma 3.3.3 (Linear and quadratic examples for s). *For all* $n \in \mathbb{N}$, *there exist strings* $S, T \in \Sigma^n$ *such that* $\mathbf{s}(S) = \Theta(n^2)$ *and* $\mathbf{s}(T) = \Theta(n)$ *and* $|S|_\sigma, |T|_\sigma \geq 1$ *for all* $\sigma \in \Sigma$. *In particular, for* $m, r, n_1, \ldots, n_k \in \mathbb{N}$ *such that* $r < k$ *and* $\sum_{i=1}^{k} n_i = n$,

1. *a)* $\mathbf{s}((\sigma_1 \ldots \sigma_k)^m) = (m-1)k^2 + \frac{1}{2}(k^2 + k)$,
   b) $\mathbf{s}((\sigma_1 \ldots \sigma_k)^m \sigma_1 \ldots \sigma_r) = (m-1)k^2 + \frac{1}{2}(k^2 + k) + r \cdot k$,
2. $\mathbf{s}(\sigma_1^{n_1} \sigma_2^{n_2} \ldots \sigma_k^{n_k}) = n + \sum_{1 \leq i < j \leq k} n_i \cdot n_j$.

Proof. 1.(a) For fixed length $n' \leq (m-1)k$, there are k different substrings of length n', namely $s_i \ldots s_{i+m}$ for $i = 1, \ldots, k$. There are k substrings of length $(m-1)k+1$, $k-1$ substrings of length $(m-1)k+2$, and so on, and finally, exactly one substring of length $m \cdot k = n$. Thus, $\mathbf{s}((\sigma_1 \ldots \sigma_k)^m) = (m-1)k \cdot k + \sum_{i=1}^{k} i$.

1.(b) In addition to substrings of $(\sigma_1 \ldots \sigma_k)^m$, each of the final r positions of s contributes k different new substrings, namely those beginning within the first block $\sigma_1 \ldots \sigma_k$ and ending in this position.

2. First consider substrings that start and end with the same character σ_i. For fixed $1 \leq i \leq k$, there are n_i different substrings of this type, yielding $\sum_{i=1}^{k} n_i = n$ different substrings. All other substrings start with some character σ_i and end with a different character σ_j, where $i < j$. For each pair (i, j), there are $n_i \cdot n_j$ different choices of the first and final positions, which all generate different substrings. Thus, $\mathbf{s}(\sigma_1^{n_1} \sigma_2^{n_2} \ldots \sigma_k^{n_k}) = n + \sum_{1 \leq i < j \leq k} n_i \cdot n_j$.

Since the alphabet size k is constant and $m = \lfloor \frac{n}{k} \rfloor$, we have $\mathbf{s}((\sigma_1 \ldots \sigma_k)^m \sigma_1 \ldots \sigma_r) = \Theta(n)$. On the other hand, let for all $i = 1, \ldots, k$, $n_i = \lfloor \frac{n}{k} \rfloor$ or $n_i = \lfloor \frac{n}{k} \rfloor + 1$ such that $\sum_{i=1}^{k} n_i = n$, i.e., let all n_i be roughly equal. Then this yields $\mathbf{s}(\sigma_1^{n_1} \sigma_2^{n_2} \ldots \sigma_k^{n_k}) = \sum_{i=1}^{k} n_i + \sum_{1 \leq i < j \leq k} n_i n_j \approx n + \binom{k}{2} \cdot (\frac{n}{k})^2 \approx n + \frac{1}{2}n^2 = \Theta(n^2)$.

□

For $n = m \cdot k$, the string $S_n = \sigma_1^m \sigma_2^m \dots \sigma_k^m$ is thus maximal up to a factor of 2 w.r.t. $\mathbf{s}$, since $\mathbf{s}(S_n) = n + \frac{k-1}{2k} n^2 \geq \frac{1}{2} \frac{n(n+1)}{2}$. In the next section, we will prove a lower bound on $\mathbf{c}(s)$ (Lemma 3.3.4), which depends on the multiplicities $|s|_{\sigma_i}$ of the different characters. For equal multiplicities $|s|_{\sigma_i} = m = \frac{n}{k}$ for all i, this lower bound is $\frac{m}{2}(k^2+k)$, implying $\mathbf{s}(s) \geq \mathbf{c}(s) \geq \frac{m}{2}(k^2+k)$. Since $\mathbf{s}((\sigma_1 \dots \sigma_k)^m) = (m-1)k^2 + \frac{1}{2}(k^2+k) \leq 2 \cdot \frac{m}{2}(k^2+k)$, the string $T_n = (\sigma_1 \dots \sigma_k)^m$ is minimal up to a factor of 2 w.r.t. $\mathbf{s}$.

Computation of $\mathbf{s}(s)$

The number $\mathbf{s}(s)$ can be computed using a suffix tree: In a suffix tree, each substring is represented by a unique path from the root. Thus, adding up the label lengths of the edges of the suffix tree of s will yield just $\mathbf{s}(s)$. The suffix tree of s can be computed in time $O(n)$, where $n = |s|$, see e.g. [Gus97]. The number of edges is linear in n, thus $\mathbf{s}(s)$ can also be computed in linear time. Moreover, we can enumerate all substrings of s in time $O(\mathbf{s}(s))$, if we only output tuple (i, j) for substring $t = s_i \dots s_j$. If we output each substring t itself, we obtain a runtime of $O(\sum_{t \sqsubseteq s} |t|)$.

3.3.2 Number of subcompomers of a given string

For a lower bound on $\mathbf{c}$, we define the index of the first occurrence of a character $\sigma \in \Sigma$ in a string $s = s_1 \dots s_{|s|}$ as $\text{Firstpos}_\sigma(s) := \min(\{i \mid s_i = \sigma\} \cup \{|s| + 1\})$.

Lemma 3.3.4 (Lower bound on c). *Let $n \in \mathbb{N}$ and $s = s_1 \dots s_n \in \Sigma^n$. Then*

$$\mathbf{c}(s) \geq \sum_{\sigma \in \Sigma} |s|_\sigma \cdot \text{Firstpos}_\sigma(s). \tag{3.24}$$

In particular, if $|s|_\sigma = m = \frac{n}{k}$ for all $\sigma \in \Sigma$, then $\mathbf{c}(s) \geq \frac{m}{2}(k^2 + k)$.

Proof. Let $x = s_n$. If x does not occur in $s_1 \dots s_{n-1}$, then appending x to $s_1 \dots s_{n-1}$ generates n new subcompomers, i.e.,

$$\mathbf{c}(s) = \mathbf{c}(s_1 \dots s_{n-1}) + n = \mathbf{c}(s_1 \dots s_{n-1}) + \text{Firstpos}_x(s_1 \dots s_{n-1}). \tag{3.25}$$

On the other hand, if x does occur in $s_1 \dots s_{n-1}$, then it generates at least $\text{Firstpos}_x(s_1 \dots s_{n-1})$ many new subcompomers, since for those substrings starting in positions $i = 1, \dots, \text{Firstpos}_x(s_1 \dots s_{n-1})$ and ending in $s_n = x$ it will hold that $|s_i \dots s_n|_x = |s_1 \dots s_{n-1}|_x + 1$. Thus, in both cases we get

$$\mathbf{c}(s) \geq \mathbf{c}(s_1 \dots s_{n-1}) + \text{Firstpos}_x(s_1 \dots s_{n-1}). \tag{3.26}$$

Applying this $n - 1$ times, we obtain

$$\mathbf{c}(s) \geq 1 + \sum_{i=2}^{n} \text{Firstpos}_{s_i}(s_1 \dots s_{i-1}). \tag{3.27}$$

Let $i \in \{2, \dots, n\}$. If s_i occurs in $s_1 \dots s_{i-1}$, then $\text{Firstpos}_{s_i}(s_1 \dots s_{i-1}) = \text{Firstpos}_{s_i}(s)$. In the sum above, this happens $|s|_{s_i} - 1$ times. For the first occurrence of character s_i in s, let $b = \text{Firstpos}_{s_i}(s)$; then $\text{Firstpos}_{s_i}(s_1 \dots s_{b-1}) = \text{Firstpos}_{s_i}(s)$ by

definition. Since $\text{Firstpos}_{s_1}(s) = 1$, we can write $1 + \sum_{i=2}^{n} \text{Firstpos}_{s_i}(s_1 \dots s_{i-1}) = \sum_{\sigma \in \Sigma} \text{Firstpos}_\sigma(s) \cdot |s|_\sigma$.

For fixed multiplicities $n_1, \dots, n_k$, the sum $\sum_{\sigma \in \Sigma} \text{Firstpos}_\sigma(s) \cdot |s|_\sigma$ is minimized over all strings with these multiplicities if all different characters occurring in s are positioned in the first positions of s, ordered ascending according to their multiplicities. In particular, if each letter occurs exactly m times, we obtain $\mathbf{c}(s) \geq m \sum_{i=1}^{k} i = \frac{m}{2}(k^2 + k)$.

□

Lemma 3.3.5 (Linear and quadratic examples for c). *For all $n \in \mathbb{N}$, there exist strings $S, T \in \Sigma^n$ such that $\mathbf{c}(S) = \Theta(n^2)$ and $\mathbf{c}(T) = \Theta(n)$ and $|S|_\sigma, |T|_\sigma \geq 1$ for all $\sigma \in \Sigma$. In particular, for $m, r, n_1, \dots, n_k \in \mathbb{N}$ such that $r < k$ and $\sum_{i=1}^{k} n_i = n$,*

1. *a)* $\mathbf{c}((\sigma_1 \dots \sigma_k)^m) = (m-1) \cdot (k^2 + 1 - k) + \frac{1}{2}(k^2 + k)$,
   b) $\mathbf{c}((\sigma_1 \dots \sigma_k)^m \sigma_1 \dots \sigma_r) = (m-1) \cdot (k^2 + 1 - k) + \frac{1}{2}(k^2 + k) + r(k-1)$,
2. $\mathbf{c}(\sigma_1^{n_1} \sigma_2^{n_2} \dots \sigma_k^{n_k}) = n + \sum_{1 \leq i < j \leq k} n_i \cdot n_j$.

Proof.

1.(a) Let $S := (\sigma_1 \dots \sigma_k)^m$. First consider only substrings with length $n' \leq (m-1)k$ and observe that for $n' = \ell \cdot k$, there is exactly one subcompomer $(\ell, \dots, \ell)$ for all substrings of length n'. Otherwise, if $n' = \ell \cdot k + p$ where $0 < p < k$, then for each $1 \leq j \leq k$, there is a substring t with subcompomer

$$|t|_{\sigma_i} = \begin{cases} \ell + 1 & \text{if there is } q \in \{0, \dots, p-1\} \text{ s.t. } i = (j+q) \bmod k, \\ \ell & \text{otherwise.} \end{cases} \tag{3.28}$$

Putting this together yields $(m-1)(1 + (k-1)k)$. Finally, for lengths $n' > (m-1)k$, the numbers of different subcompomers decrease one by one: There are k different subcompomers of substrings with length $(m-1) \cdot k+1$, $k-1$ with length $(m-1) \cdot k+2$ and so on, yielding $\sum_{i=k}^{1} i = \frac{1}{2}(k^2 + k)$ different subcompomers. Thus, $\mathbf{c}((\sigma_1 \dots \sigma_k)^m) = (m-1) \cdot (k^2 + 1 - k) + \frac{1}{2}(k^2 + k)$.

1.(b) This is a simple extension of *1.(a)*, noting that each of the last r positions of s will contribute $k-1$ new subcompomers for substrings ending in this position: For σ_j, $1 \leq j \leq r$, the compomer of the substring $s_i \dots s_{m \cdot k + j}$ will be new for all $i \in \{1, \dots, j, j+2, \dots, k\}$.

2. Observe that for string $S = \sigma_1^{n_1} \dots \sigma_k^{n_k}$, $\mathbf{c}(S) = \mathbf{s}(S)$, and thus, $\mathbf{c}(S) = n + \sum_{1 \leq i < j \leq k} n_i n_j$ by Lemma 3.3.3.

Similar to the proof of Lemma 3.3.3, we have $\mathbf{c}((\sigma_1 \dots \sigma_k)^m) = \Theta(n)$ for constant k and $\mathbf{c}(\sigma_1^{m_1} \sigma_2^{m_2} \dots \sigma_k^{m_k}) = \Theta(n^2)$ for roughly equal multiplicities m_i. □

For equal multiplicities $|s|_{\sigma_i} = m = \frac{n}{k}$ for all i, the lower bound on $\mathbf{c}$ is $\frac{m}{2}(k^2 + k)$. Since $\mathbf{c}((\sigma_1 \dots \sigma_k)^m) = (m-1) \cdot (k^2 + 1 - k) + \frac{1}{2}(k^2 + k) \leq 2 \cdot \frac{m}{2}(k^2 + k)$, the string $T_n = (\sigma_1 \dots \sigma_k)^m$ is minimal up to a factor of 2 w.r.t. $\mathbf{c}$.

The next two lemmas are used to prove a tight upper bound on $\mathbf{c}$ (Lemma 3.3.8). Hereby, we denote by $[\Lambda]$ the characteristic value of a proposition Λ, i.e., $[\Lambda] = 1$ if Λ is true, and $[\Lambda] = 0$ otherwise.

Lemma 3.3.6 (Maximal growth of c). *Let* $n \in \mathbb{N}, x \in \Sigma$ *and* $s = s_1 \dots s_n \in \Sigma^n$.

1. *If* s *does not contain character* x, *then* $\mathbf{c}(sx) = \mathbf{c}(s) + (n+1)$.
2. *If* s *contains character* x, *then* $\mathbf{c}(sx) \leq \mathbf{c}(s) + n - |s|_x + [s_n = x]$.

Proof. 1. Obvious.

2. There are $\mathbf{c}(s)$ different compomers of substrings starting and ending within s. Furthermore, n substrings of sx start within s and end in x. For each index $1 \leq i \leq n-1$ s.t. $s_i = x$, we have $\mathrm{comp}(s_i \dots s_n) = \mathrm{comp}(s_{i+1} \dots s_n x)$. Thus, none of these substrings has a new compomer. There are $|s|_x$ such substrings if $s_n \neq x$, and $|s|_x - 1$ otherwise. □

The next lemma shows that concentrating each character in blocks maximizes the number of subcompomers.

Lemma 3.3.7 (c–maximal strings). *Let* $n \in \mathbb{N}$ *and fix* $0 \leq n_1, \dots, n_k \in \mathbb{N}$ *such that* $\sum_{i=1}^{k} n_i = n$. *Then,*

$$\mathbf{c}(\sigma_1^{n_1} \dots \sigma_k^{n_k}) = \max\{\mathbf{c}(s) \mid s \in \Sigma^n, |s|_{\sigma_i} = n_i \text{ for all } i = 1, \dots, k\}. \tag{3.29}$$

Proof. By induction on n: For $n = 1$, the claim is obvious. Choose $s \in \Sigma^{n+1}$ and denote by $n_i := |s|_{\sigma_i}$ for $i = 1, \dots, k$. (Thus, we now have $\sum_{i=1}^{k} n_i = n+1$.) Up to relabeling (which leaves $\mathbf{c}$ invariant), we may assume that the last character of s is σ_k, thus we can write $s = s'\sigma_k$. If $n_k = 1$, then

$$\begin{aligned}
\mathbf{c}(s) &= \mathbf{c}(s') + (n+1) && \text{by Lemma 3.3.6} \\
&\leq \mathbf{c}(\sigma_1^{n_1} \dots \sigma_{k-1}^{n_{k-1}}) + (n+1) && \text{by the induction hypothesis} \\
&= \mathbf{c}(\sigma_1^{n_1} \dots \sigma_k^{n_k}) && \text{by Lemma 3.3.5.}
\end{aligned}$$

Otherwise, $n_k > 1$, and

$$\begin{aligned}
\mathbf{c}(s) &\leq \mathbf{c}(s') + n - |s'|_{\sigma_k} + 1 && \text{by Lemma 3.3.6} \\
&\leq \mathbf{c}(\sigma_1^{n_1} \dots \sigma_k^{n_k - 1}) + n - (n_k - 1) + 1 && \text{by the induction hypothesis} \\
&= n + \sum_{1 \leq i < j < k} n_i n_j + \sum_{i=1}^{k-1} n_i (n_k - 1) + n - (n_k - 1) + 1 && \text{by Lemma 3.3.5} \\
&= n + \sum_{1 \leq i < j < k} n_i n_j + \sum_{i=1}^{k-1} n_i (n_k - 1) + \sum_{i=1}^{k-1} n_i + 1 && \text{since } n + 1 = \sum_{i=1}^{k} n_i \\
&= (n+1) + \sum_{1 \leq i < j \leq k} n_i n_j \;=\; \mathbf{c}(\sigma_1^{n_1} \dots \sigma_k^{n_k}).
\end{aligned}$$

□

Lemma 3.3.8 (Tight upper bound on c). *Let* $s \in \Sigma^n$. *Then* $\mathbf{c}(s) \leq n + \sum_{1 \leq i < j \leq k} m_i m_j$, *where* $m_i = \lfloor \frac{n}{k} \rfloor$ *or* $m_i = \lfloor \frac{n}{k} \rfloor + 1$ *for* $i = 1, \dots, k$ *and* $\sum_{i=1}^{k} m_i = n$. *In particular, if* n *is a multiple of* k, *then* $\mathbf{c}(s) \leq \frac{k-1}{2k} n^2 + n$. *This latter bound is tight.*

Proof. Let $s \in \Sigma^n$. Denote by $n_i := |s|_{\sigma_i}$ for $i = 1, \ldots, k$. Then, by Lemma 3.3.7, $\mathbf{c}(s) \leq \mathbf{c}(\sigma_1^{n_1} \ldots \sigma_k^{n_k}) = n + \sum_{1 \leq i < j \leq k} n_i n_j$. Let $f(x_1, \ldots, x_k) := \sum_{1 \leq i < j \leq k} x_i x_j$. Function f attains its maximum on the set $B_n := \{(x_1, \ldots, x_k) \mid \sum_{i=1}^{k} x_i = n\}$ if all values are approximately equal, i.e., $\max\{f(B_n)\} = f(m_1, \ldots, m_k)$ where for all i, $m_i = \lfloor \frac{n}{k} \rfloor$ or $m_i = \lfloor \frac{n}{k} \rfloor + 1$ and $\sum_{i=1}^{k} m_i = n$. Moreover, since $\mathbf{c}(\sigma_1^{m_1} \ldots \sigma_k^{m_k}) = n + \sum_{1 \leq i < j \leq k} m_i m_j$, this bound is tight. If n is a multiple of k, then $m_i = \frac{n}{k}$ for all i, and thus:

$$\max\{\mathbf{c}(s) \mid |s| = n\} = n + \binom{k}{2} (\frac{n}{k})^2 = \frac{k-1}{2k} n^2 + n. \tag{3.30}$$

□

Computation of $\mathbf{c}(s)$

The question of the value of $\mathbf{c}(s)$ for a given string s is equivalent to the question of how many different compomers $\mathrm{comp}(t)$ the set $L_s := \{t \mid t \sqsubseteq s\}$ has. If we denote by $\Sigma^{\oplus}$ the free commutative monoid over Σ, then any language $L \subseteq \Sigma^*$ induces a subset $L^{\oplus}$ of $\Sigma^{\oplus}$, namely $L^{\oplus} := \{\prod_{a \in \Sigma} a^{|t|_a} \mid t \in L\}$ (see [ABB97]). Now, we have $\mathbf{c}(s) = |L_s^{\oplus}|$. We are not aware that $|L_s^{\oplus}|$ has been characterized in the literature.

We can compute $\mathbf{c}(s)$ trivially by enumerating all substrings of s, computing their compomers, and ordering them. This can be done in time $O(\mathbf{s}(s) \cdot \log(\mathbf{s}(s)))$.

3.3.3 Number of submasses of a given string

If we have $\mathbf{m}(s) = \mathbf{c}(s)$, all results of the previous section carry over: We can give a linear lower and tight quadratic upper bound, and can specify strings with a linear resp. quadratic number of submasses, the latter attaining the upper bound. We therefore define the UNIQUE DECOMPOSITION PROPERTY:

> A mass function μ has the UNIQUE DECOMPOSITION PROPERTY (UDP) if, for all strings s and t:
>
> $$\mu(s) = \mu(t) \iff \text{for all } \sigma \in \Sigma : |s|_\sigma = |t|_\sigma.$$

This just means that the masses are linearly independent over the integers. With the UDP, a mass M has at most one decomposition $M = \sum_{\sigma \in \Sigma} \nu(\sigma) \cdot \mu(\sigma)$ where $\nu(\sigma) \in \mathbb{N}_0$.

With the UDP, we have $\mathbf{m}(s) = \mathbf{c}(s)$ for all s. Note that this condition never holds if the masses are integers or rational numbers. If, however, we allow real numbers as masses, i.e., if $\mu : \Sigma \to \mathbb{R}^+$, then the masses can be chosen to satisfy the UDP. For example, if $\Sigma = \{a, b\}$, then μ with $\mu(a) = 1$ and $\mu(b) = \pi$ has the UDP.

We can even achieve that $\mathbf{m}(s) = \mathbf{c}(s)$ with integers if the masses may depend on the input size. Then, we can set $\mu(\sigma_i) := (n+1)^{i-1}$ for all $i = 1, \ldots, k$. However, this results in exponentially large masses.

Computation of $\mathbf{m}(s)$

In Chapter 5, we will give several algorithms for computing the number of submasses of a given string s.

4 Mass Decomposition Algorithms

In this chapter, we present an algorithm for solving the following problem. Fix a weighted alphabet (Σ, μ) where $\mu : \Sigma \mapsto \mathbb{N}$:

> DECOMPOSITION ALL WITNESSES PROBLEM:
> Given a mass $M \in \mathbb{N}$, return all compomer decompositions of M.

Note that this is the integer version of the general problem presented in Chapter 3 (page 38). For the mass spectrometry applications, integer weights can be achieved by scaling up the character masses by a chosen precision factor. We give simulation results in Section 4.6 for precisions 0.01 and 0.001, realistic precisions of current mass spectrometers.

Let $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ and $a_1, \ldots, a_k$ denote the character masses, i.e., $\mu(\sigma_i) = a_i$ for $i = 1, \ldots, k$, and let (w.l.o.g.) a_1 be the smallest character mass. Without loss of generality, we assume that the masses are all distinct; the generalization to non-distinct alphabets is straightforward. In [BL05b], we presented a new efficient algorithm for computing the Frobenius number $g(a_1, \ldots, a_k)$. The best known algorithm to date had been due to Nijenhuis [Nij79], which runs in time $\mathcal{O}(k\, a_1 \log a_1)$ or $\mathcal{O}(a_1(k+\log a_1))$, depending on the implementation of the priority queue used by the algorithm. It constructs a data structure of size a_1, which we refer to as "residue table," requiring $\mathcal{O}(a_1)$ additional memory for the construction; using the residue table, the Frobenius number can be easily computed. Our algorithm in [BL05b] also constructs the residue table but only requires $\mathcal{O}(ka_1)$ time and no additional memory. The algorithm has been implemented and found to be also fast in practice; both by us and by Wagon and co-workers [BHNW].

The algorithm presented in this chapter extends the ideas in [BL05b]. It constructs a data structure of size ka_1 in a preprocessing step, which can then be used to produce all witnesses for a query mass M. In addition, it also allows solving the DECISION, ONE WITNESS and FROBENIUS problems. We first describe the classical dynamic programming algorithm which is often employed for related problems; via backtracking in the table, all witnesses for the query can be produced.

The contents of this chapter have been partially published in [BL05a]. Implementation and simulations by Marcel Martin and Henner Sudek.

4.1 Related problems

The problem of identifying compomers that add up to a given mass can be posed as a Money Changing Problem: Given k positive integers $a_1, \ldots, a_k$ (the coin denominations), and an integer M, are there non-negative integers c_i such that $\sum_i c_i a_i = M$? In other words, is it possible to return exactly M as change? This problem is NP-complete, when both the alphabet and the query vary [Lue75], and can be solved

in pseudo-polynomial time by a dynamic programming algorithm [MT90]. It follows from the NP-completeness result for this problem that we cannot expect to find an algorithm that will solve this problem in polynomial time in all parameters of the input: the alphabet size k, the input size of the character masses $\sum_i \log a_i$ and that of the query, $\log M$.

Note that the problems considered here differ from what is often referred to as Coin Change Problem or Change Making Problem, where a decomposition of a query M using a *minimal* number of coins is sought. There, the decision problem is trivial, since usually, a coin of denomination 1 is assumed [MT90].

4.2 The classical dynamic programming algorithm

We sketch a dynamic programming algorithm, which is a variation of the classical algorithm usually introduced for the Coin Change Problem (where the number of coins is to be minimized), originally due to Gilmore and Gomory [GG65, MT90].

Given query mass M and the $a_1, \ldots, a_k$, a two-dimensional Boolean table B of size kM is constructed such that

$$B[i, m] = 1 \quad \text{if and only if} \quad m \text{ is decomposable over } \{a_1, \ldots, a_i\}. \tag{4.1}$$

The table can be computed with the following recursion: $B[1, m] = 1$ if and only if $m \bmod a_1 = 0$, and for $i > 1$,

$$B[i, m] = \begin{cases} B[i-1, m] & \text{if } m < a_i, \\ B[i-1, m] \vee B[i, m - a_i] & \text{otherwise.} \end{cases} \tag{4.2}$$

The table is constructed up to mass M, and then a straightforward backtracking algorithm computes all witnesses of M. Running time for solving the All Witnesses Problem is $\mathcal{O}(kM)$ for the table construction, and $\mathcal{O}(\sum_{c \in \mathcal{C}(M)} |c|) = \mathcal{O}(\gamma(M) \cdot \frac{1}{a_1} M)$ for computation of the witnesses, while storage space is $\mathcal{O}(kM)$.

For the DECISION and ONE WITNESS PROBLEMS, it suffices to construct a one-dimensional Boolean table A of size M, using the recursion $A[0] = 1$, $A[m] = 0$ for $1 \leq m < a_1$; and for $m \geq a_1$, $A[m] = 1$ if exists $1 \leq i \leq k$ s.t. $A[m - a_i]$, and 0 otherwise. Construction time is $\mathcal{O}(kM)$ and one witness c can be produced by backtracking in time proportional to $|c|$, which can be in the worst case $\frac{1}{a_1} M$. Of course, both of these problems can also be solved using the full table B.

Finally, a variant computes the *number* $\gamma(M)$ of decompositions of M in the last row, where the entries are integers, using the recursion $C[i, m] = C[i-1, m] + C[i, m - a_i]$.

In the following sections, we will present more efficient algorithms which all use a data structure of size ka_1: the Extended Residue Table.

4.3 The extended residue table

In this section, we introduce the Extended Residue Table and its construction algorithm, EXTENDED ROUND ROBIN. Recall that a_1 is the smallest character mass.

Now, for any $M \geq 0$ and any $i, 1 \leq i \leq k$, it can be immediately seen that

$$M \text{ is decomposable} \quad \Rightarrow \quad M + a_i \text{ is decomposable.} \tag{4.3}$$

In particular, Equation (4.3) holds for $i = 1$. For $r = 0, \ldots, a_1 - 1$, let n_r be the smallest integer such that n_r is decomposable. Then, for any $M \geq 0$ with $M \equiv r \bmod a_1$,

$$M \text{ is decomposable} \quad \Leftrightarrow \quad M \geq n_r. \tag{4.4}$$

Thus, if we only want to know whether a mass is decomposable, we only need the values n_r for each $r = 0, \ldots, a_1 - 1$. However, for the DECOMPOSITION ALL WITNESSES PROBLEM, more information is needed: Namely, for each $r = 0, \ldots, a_1 - 1$ and each $i = 1, \ldots, k$, the smallest number $n_{r,i}$ congruent r modulo a_1 such that $n_{r,i}$ is decomposable over $\{a_1, \ldots, a_i\}$. More formally, the Extended Residue Table is a two-dimensional table of size ka_1, consisting of entries $ert(r, i)$ such that

$$ert(r, i) = \min\{n \mid n \equiv r \bmod a_1 \text{ and } n \text{ is decomposable over } \{a_1, \ldots, a_i\}\}, \tag{4.5}$$

where $ert(r, i) = \infty$ if no such integer exists.

Example 1. Let $k = 4$ and the masses be $6, 7, 11, 15$. We give the Extended Residue Table in Figure 4.1.

r	$a_1 = 6$	$a_2 = 7$	$a_3 = 11$	$a_4 = 15$
0	0	0	0	0
1	∞	7	7	7
2	∞	14	14	14
3	∞	21	21	15
4	∞	28	22	22
5	∞	35	11	11

Figure 4.1: Extended Residue Table for alphabet $6, 7, 11, 15$ (Example 1).

A variation of the Extended Residue Table consisting of only the last column was introduced in [BL05b]. Proofs of correctness can be found there. Here, we give a brief review of the basic construction algorithm. The pseudo-code can be found in Figure 4.2.

The table is constructed column by column. Let us look at column $i > 1$, and first assume that $\gcd(a_1, a_i) = 1$. We maintain a value n and an index r, where $r = n \bmod a_1$. In each step, n is updated $n \leftarrow n + a_i$, r is updated accordingly, and the next entry to be filled in is $ert(r, i)$, which will be assigned $\min(ert(r, i-1), n)$. Now we continue with $n \leftarrow ert(r, i)$.

For the example in Figure 4.1, let us look at column 3. We start at $ert(0, 3)$, which equals 0, add $a_3 = 11$, compute $r \leftarrow 11 \bmod 6 = 5$. Since $11 < ert(5, 2) = 35$, we set $ert(5, 3) \leftarrow 11$. Next, we have $n \leftarrow 11 + 11 = 22$, $r \leftarrow 22 \bmod 6 = 4$, and because $22 < 28 = ert(4, 2)$, we set $ert(4, 3) \leftarrow 22$. Then, $n \leftarrow 22 + 11 = 33$, and $r \leftarrow 33 \bmod 6 = 3$, but since $33 > 21 = ert(3, 2)$, we set $ert(3, 3) \leftarrow ert(3, 2) = 21$. The remaining two

entries also both get the corresponding values from the previous column, and we are done when we encounter $r = 0$ again.

In the case where $\gcd(a_1, a_i) = d > 1$, we need to repeat the simple algorithm above for $t = 0, 1, \ldots, d-1$ separately. Look at the last column in the example. We have $\gcd(6, 15) = 3$, so $t = 0, 1, 2$ (line 4 in Figure 4.2). Let's look at $t = 2$. We first have to find the minimum amongst the entries $ert(q, i-1)$, for $q = 2, 5$ (line 5), since this is the one we will start with (line 6). In this case, it is $ert(5, 3) = 11$, and thus $r = 5$. We set $ert(5, 4) \leftarrow 11$, add 15, end up with 26, which is larger than $14 = ert(2, 3) = ert(26 \bmod 6, 3)$, and thus we have $ert(2, 4) \leftarrow 14$. We are done after $1 = 6/3 - 1$ steps, since the next step would bring us back to residue class $r = 5$, which we have already processed.

Algorithm EXTENDED ROUND ROBIN

```
 1  initialize ert(0,i) = 0 and ert(r,i) = ∞ for r = 1,...,a_1 − 1, i = 1,...,k;
 2  for i = 2,...,k do
 3      d ← gcd(a_1, a_i);
 4      for t = 0,...,d − 1 do
 5          find n = min{ert(q,i − 1) | q mod d = t};
 6          ert(n mod a_1, i) ← n;
 7          if n < ∞ then repeat a_1/d − 1 times
 8              n ← n + a_i; r ← n mod a_1;
 9              n ← min{n, ert(r,i − 1)};
10              ert(r,i) ← n;
11          end;
12      done;
13  done.
```

Figure 4.2: Construction algorithm of the Extended Residue Table.

We summarize the complexity of the construction algorithm:

Theorem 4.3.1. *The* EXTENDED ROUND ROBIN *algorithm computes the Extended Residue Table of a weighted alphabet* $\{a_1, \ldots, a_k\}$ *with smallest mass* a_1 *in optimal runtime* $O(k\, a_1)$ *and extra memory* $O(1)$.

4.4 Finding all witnesses

In this section, we introduce an algorithm to solve the All Witnesses Problem using the Extended Residue Table. We analyze its complexity and compare it to the dynamic programming algorithm from Section 4.2. Finally, we discuss a possible heuristic improvement.

For producing all witnesses of query mass M, we use a recursive algorithm that, as in the simple backtracking using the DP tableau, maintains a current compomer c, an index i, and a current mass m. At step i, the entries $c_k, c_{k-1}, \ldots, c_{i+1}$ of compomer c have already been filled in, and the remaining mass $m = M - \sum_{j=i+1}^{k} c_j a_j$ will be decomposed over $\{a_1, \ldots, a_i\}$. The invariant at the call of FIND-ALL(m, i, c) is

that mass m is decomposable over $\{a_1, \ldots, a_i\}$ and $c_j = 0$ for $j = i, i-1, \ldots, 1$. We give the pseudo-code in Figure 4.3.

Algorithm FIND-ALL **(mass** M**, index** i**, compomer** c**)**

```
 1  if i = 1 then
 2      c_1 ← M/a_1; output c; return;
 3  end;
 4  lcm ← lcm(a_1, a_i); ℓ ← lcm /a_i;                  // least common multiple
 5  for j = 0, ..., ℓ − 1 do
 6      c_i ← j; m ← M − j a_i;                         // start with j pieces of a_i
 7      r ← m mod a_1; lbound ← ert(r, i − 1);
 8      while m ≥ lbound do                             // m is decomposable
 9          FIND-ALL(m, i − 1, c);                      //    over {a_1, ..., a_{i−1}}
10          m ← m − lcm; c_i ← c_i + ℓ;
11      done;
12  done.
```

Figure 4.3: Algorithm for finding all witnesses using the Extended Residue Table. For a mass M which is decomposable over $\{a_1, \ldots, a_k\}$, FIND-ALL$(M, k, 0)$ will recursively produce all witnesses with mass M.

4.4.1 Correctness of the algorithm

By construction, any compomer computed by FIND-ALL$(M, k, 0)$ will have mass M. Conversely, fix m decomposable over $\{a_1, \ldots, a_i\}$, $i > 1$, and let

$$\begin{aligned} N(m, i) \;:=\; & \{m' = m - n a_i \mid m' \geq 0 \text{ and} \\ & \quad m' \text{ is decomposable over } \{a_1, \ldots, a_{i-1}\}\}. \end{aligned} \tag{4.6}$$

We will show that for a call of FIND-ALL(m, i, c), the set of values m' for which a recursive call is made (line 9 in Figure 4.3) is exactly $N(m, i)$. Then it follows by induction over $i = k, k-1, \ldots, 2$ that, given $c = (c_1, \ldots, c_k)$ with mass M, the algorithm will arrive at call FIND-ALL$(c_1 a_1, 1, (0, c_2, \ldots, c_k))$ and thus output c (line 2): In the induction step, set $m = M - \sum_{j=i+1}^{k} c_j a_j$ and $m' = m - c_i a_i$.

In order to prove the claim, let $\ell := a_1 / \gcd(a_1, a_i) = \mathrm{lcm}(a_1, a_i)/a_i$, and $r_{q,m} := m - q a_i \bmod a_1$, for $q = 0, \ldots, \ell - 1$. Now consider the sets

$$\begin{aligned} N(m, i, q) \;:=\; & \{m' \geq \mathrm{ert}(r_{q,m}, i-1) \mid m' = m - n a_i, n \equiv q \bmod \ell\}, \\ & \text{for } q = 0, \ldots, \ell - 1. \end{aligned} \tag{4.7}$$

Observe that $N(m, i, q)$ is exactly the set of values for which a recursive call is made within the while-loop (line 9) for $j = q$ (line 5). Clearly, $\cup_{q=0}^{\ell-1} N(m, i, q) \subseteq N(m, i)$. On the other hand, let $m' = m - n a_i \in N(m, i)$. Further, let $r = m' \bmod a_1$ and $q = n \bmod \ell$. Since $r \equiv m' \bmod a_1$ and $m' = m - n a_i \equiv m - q a_i \bmod a_1$, we have $m - q a_i \bmod a_1 = r$. Since m' is decomposable over $\{a_1, \ldots, a_{i-1}\}$, it must hold that

$m' \geq ert(r, i-1)$ by property (4.5) of the Extended Residue Table. Thus, we have

$$N(m, i) = \bigcup_{q=0}^{\ell-1} N(m, i, q), \tag{4.8}$$

as claimed.

4.4.2 Complexity of the algorithm

As we have seen, step i of Algorithm FIND-ALL makes one recursive call for each $m' \in N(m, i, q)$, $q = 0, \ldots, \ell - 1$, where $\ell = \text{lcm}(a_1, a_i)/a_i$ (line 9). By (4.8), each of these calls will produce at least one witness. In order to check which m' are in $N(m, i, q)$, the algorithm enters the while-loop at line 8, and will thus make one unsuccessful comparison before exiting the loop. In the worst case, the current call FIND-ALL(m, i, c) will produce only one witness; in this case, we will have $\ell - 1$ additional comparisons. Since for all $i = 2, \ldots, k$, $\ell = \text{lcm}(a_1, a_i)/a_i \leq a_1$, we have

$$\text{number of comparisons for FIND-ALL}(M, k, 0) \quad \leq \quad k a_1 \gamma(M). \tag{4.9}$$

The previous discussion yields the following theorem:

Theorem 4.4.1. *Given the Extended Residue Table of a weighted alphabet* $\{a_1, \ldots, a_k\}$ *with smallest mass* a_1*, the* FIND-ALL *Algorithm solves the All Witnesses Problem for query* M *in time* $\mathcal{O}(k a_1 \gamma(M))$*.*

4.4.3 Runtime heuristic

The following simple heuristic may achieve a runtime improvement of algorithm FIND-ALL for certain weighted alphabets $\{a_1, \ldots, a_k\}$: First sort each column but the last of the Extended Residue Table in ascending order. Now instead of going through the values of $m - j a_i$ for $j = 0, \ldots, \ell - 1$ (line 5 in Figure 4.3) and comparing it to $ert(r, i-1)$ where $r = m - j a_i \bmod a_1$ (lines 7 and 8), we check each entry of the now sorted column $i-1$ in ascending order until we exceed the current value m. To this end, we need to find the value x such that, for given residue r modulo a_1, we have

$$m - x a_i = r \bmod a_1, \quad \text{with } x \text{ minimal.} \tag{4.10}$$

For finding x, we need to compute multiplicative inverses modulo a_1, if $\gcd(a_1, a_i) = 1$, resp. modulo $a_1/\gcd(a_1, a_i)$ in the general case. These can be computed with the Extended Euclidean Algorithm in a preprocessing step and stored. In addition, if $\gcd(a_1, a_i) > 1$, then column $i - 1$ needs to be split into separate blocks, and each block sorted separately.

Unfortunately, for our tests with amino acid masses, this heuristic significantly decreased the performance of the algorithm (data not shown). We suspect that here, the more complex operations (multiplication, modulo) outweigh any benefit of reducing the number of comparisons in the original FIND-ALL algorithm.

4.5 Solving related problems with the extended residue table

The Frobenius number $g(a_1, \ldots, a_k)$, the smallest number for which no decomposition over $\{a_1, \ldots, a_k\}$ exists, can be computed as [BS62]:

$$g(a_1, \ldots, a_k) = \max_{r=0,\ldots,a_1-1} \{ert(r, k)\} - a_1. \tag{4.11}$$

Given the Extended Residue Table, it can be decided in constant time whether a mass M is decomposable over $\{a_1, \ldots, a_k\}$:

$$M \text{ is decomposable over } \{a_1, \ldots, a_k\} \quad \Leftrightarrow \quad M \geq ert(M \bmod a_1, k). \tag{4.12}$$

Moreover, if one is interested in only *one* witness for each input, this can be solved using an additional vector, which we refer to as witness vector. This idea was first suggested but not detailed in [Nij79]. Along with the Extended Residue Table, we construct a vector w of length a_1 of pairs (index, multiplicity) as follows (see Figure 4.4):

Algorithm EXTENDED ROUND ROBIN WITH WITNESS VECTOR

```
 1  initialize ert(0,i) = 0 and ert(r,i) = ∞ for r = 1,...,a_1 − 1, i = 1,...,k;
 2  for i = 2,...,k do
 3      d ← gcd(a_1, a_i); w(0) ← (1,0);
 4      for t = 0,...,d − 1 do
 5          find n = min{ert(q, i − 1) | q mod d = t};
 6          ert(n mod a_1, i) ← n; counter ← 0;
 7          if n < ∞ then repeat a_1/d − 1 times
 8              n ← n + a_i; r ← n mod a_1; counter ← counter +1;
 9              if n > ert(r, i − 1)
10                  then n ← ert(r, i − 1); counter ← 0;
11                  else w(r) ← (i, counter);
12              end;
13              ert(r, i) ← n;
14          end;
15      done;
16  done.
```

Figure 4.4: The Round Robin Algorithm for the Extended Residue Table, computing in addition a witness vector w of size a_1.

The only change to Algorithm EXTENDED ROUND ROBIN in Figure 4.2 is the additional construction of the witness vector w (lines 3 and 11). We maintain a counter for the current number of a_i's that have been added up before we encounter a value in the previous column that is smaller than the current value of n. Note that n still assumes the minimum of the two values $\{n, ert(r, i-1)\}$ (line 9 in Figure 4.2 and lines 9-10 in Figure 4.4) as before. We update the entry $w(r)$ (line 11), unless $ert(r, i-1)$ is the smaller of the two.

Now the vector w can be used to construct a witness with the very simple algorithm shown in Figure 4.5. (Note that the entries of w are denoted 0 to $a_1 - 1$ to correspond with those of the Extended Residue Table, see Example 2.)

Algorithm FIND-ONE **(mass** M**)**
1 $r \leftarrow M \bmod a_1$; $m \leftarrow ert(r,k)$; $c_1 \leftarrow (M-m)/a_1$;
2 while $m \neq 0$ do
3 $(i,j) \leftarrow w(r)$;
4 $c_i \leftarrow j$;
5 $m \leftarrow m - ja_i$; $r \leftarrow m \bmod a_1$;
6 done;
7 return c.

Figure 4.5: Algorithm for computing one witness using the witness vector w.

Example 2. Let's look at the example $\Sigma = \{6,7,11,15\}$ again. Vector w will be $((1,0), (2,1), (2,2), (2,3), (2,4), (2,5))$ after the first iteration ($i = 2$). In the next iteration, $w(4)$ and $w(5)$ are updated with $w(5) = (3,1)$ and $w(4) = (3,2)$. In the final round, $w(3)$ and $w(4)$ are both set to $(4,1)$. We end up with vector $((1,0), (2,1), (2,2), (4,1), (4,1), (3,1))$. See Figure 4.6.

For query $M = 286$, we compute the witness $(44,1,0,1)$.

r	$a_1 = 6$	$a_2 = 7$	$a_3 = 11$	$a_4 = 15$	$w(r)$
0	0	0	0	0	(1,0)
1	∞	7	7	7	(2,1)
2	∞	14	14	14	(2,2)
3	∞	21	21	15	(4,1)
4	∞	28	22	22	(4,1)
5	∞	35	11	11	(3,1)

Figure 4.6: Extended Residue Table for alphabet $6,7,11,15$ with witness vector w (Example 2).

The construction algorithm EXTENDED ROUND ROBIN WITH WITNESS VECTOR uses $\mathcal{O}(a_1)$ additional space and time, compared to the EXTENDED ROUND ROBIN Algorithm, thus runs in time and space $\mathcal{O}(ka_1)$.

The witness vector w has the following property [BL05b]: For $r = 0, \ldots, a_1 - 1$, if $w(r) = (i,j)$, then

$$j = \max\{c_i \mid c = (c_1, \ldots, c_k), \mu(c) = ert(r,k)\}, \tag{4.13}$$

from which it directly follows that the number of iterations of the while-loop in line 2 (Figure 4.5) is at most $k-1$. Thus, the running time of Algorithm FIND-ONE is $\mathcal{O}(k)$.

We summarize:

Theorem 4.5.1. *Using the Extended Residue Table of size* $\mathcal{O}(ka_1)$*, whose construction time is* $\mathcal{O}(ka_1)$*, the* FROBENIUS *number can be computed in time* $\mathcal{O}(k)$*, and the*

DECISION PROBLEM *can be solved in time* $\mathcal{O}(1)$. *Using the Extended Residue Table with Witness Vector of size* $\mathcal{O}(ka_1)$, *whose construction time is* $\mathcal{O}(ka_1)$, *the* ONE WITNESS PROBLEM *can be solved in time* $\mathcal{O}(k)$.

4.6 Simulation results and $\gamma(M)$ for biomolecules

We implemented both the classical dynamic programming algorithm and our algorithm for the DECOMPOSITION ALL WITNESSES PROBLEM in C++. Running times on a SUN Fire 880, 900 MHz, for the amino acid alphabet, the DNA alphabet, and bioatoms are shown in Figures 4.7, 4.8, and 4.9. The DNA alphabet of size 4 uses average masses; the amino acid alphabet of size 19 monoisotopic masses, where the two amino acids of equal mass, I (Isoleucine) and L (Leucine), have been replaced by one. For the biocompomers, we consider the most abundant atoms in biomolecules (H, C, N, O, P, S). (See Section 2.4 for the masses.) We have run simulations for precisions 0.01 and 0.001. We plot the times for the preprocessing separately from the algorithms for producing all witnesses. Since our preprocessing algorithm EXTENDED ROUND ROBIN does not depend on the input mass, its running time is constant. Our preprocessing outperforms the dynamic programming by far, while the runtimes for producing all witnesses are very close. The plots for precision 0.001 start higher than at 0 because in the small mass area, there are no data points.

Obviously, the runtime of any algorithm that produces all witnesses for a query mass depends on the number of its decompositions. In Figure 4.10, we report, for different mass spectrometry applications, the number of compomers per mass, to call attention to the widely differing cases. Considering that the average amino acid mass is around 100 Da, a mass of 3000 Da corresponds to a string of average length 30. Nucleotide masses are all very close to 300 Da; thus, mass 10000 corresponds to strings of length up to around 35.

DNA compomers up to 10 000 Da

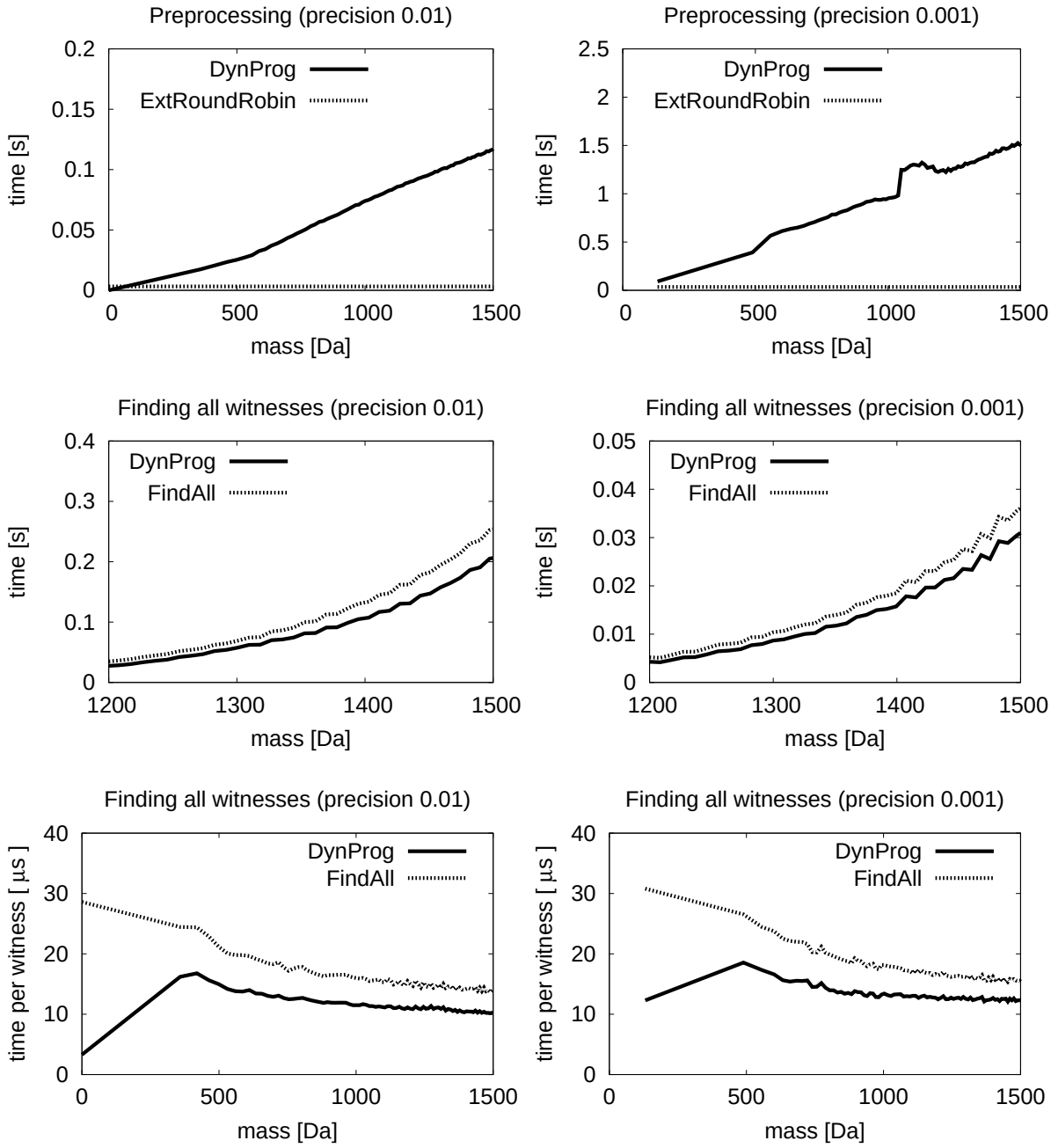


Figure 4.7: **Amino acid alphabet.** Runtime comparison for finding all witnesses with precision 0.01 Da (left) and 0.001 Da (right). Top: Preprocessing, center: query algorithm, bottom: query algorithm, runtime per witness.

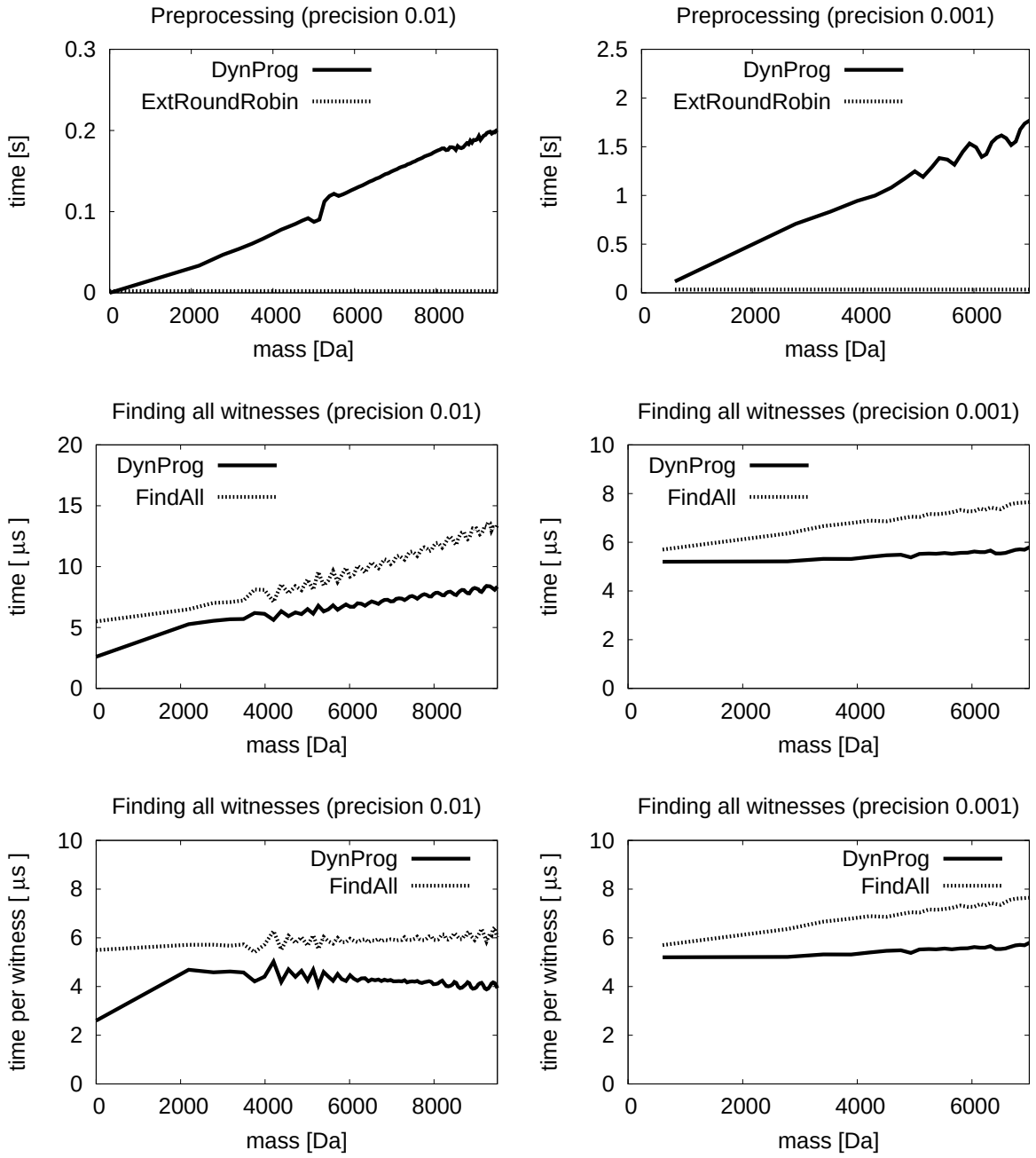


Figure 4.8: **DNA alphabet.** Runtime comparison for finding all witnesses with precision 0.01 Da (left) and 0.001 Da (right). Top: Preprocessing, center: query algorithm, bottom: query algorithm, runtime per witness.

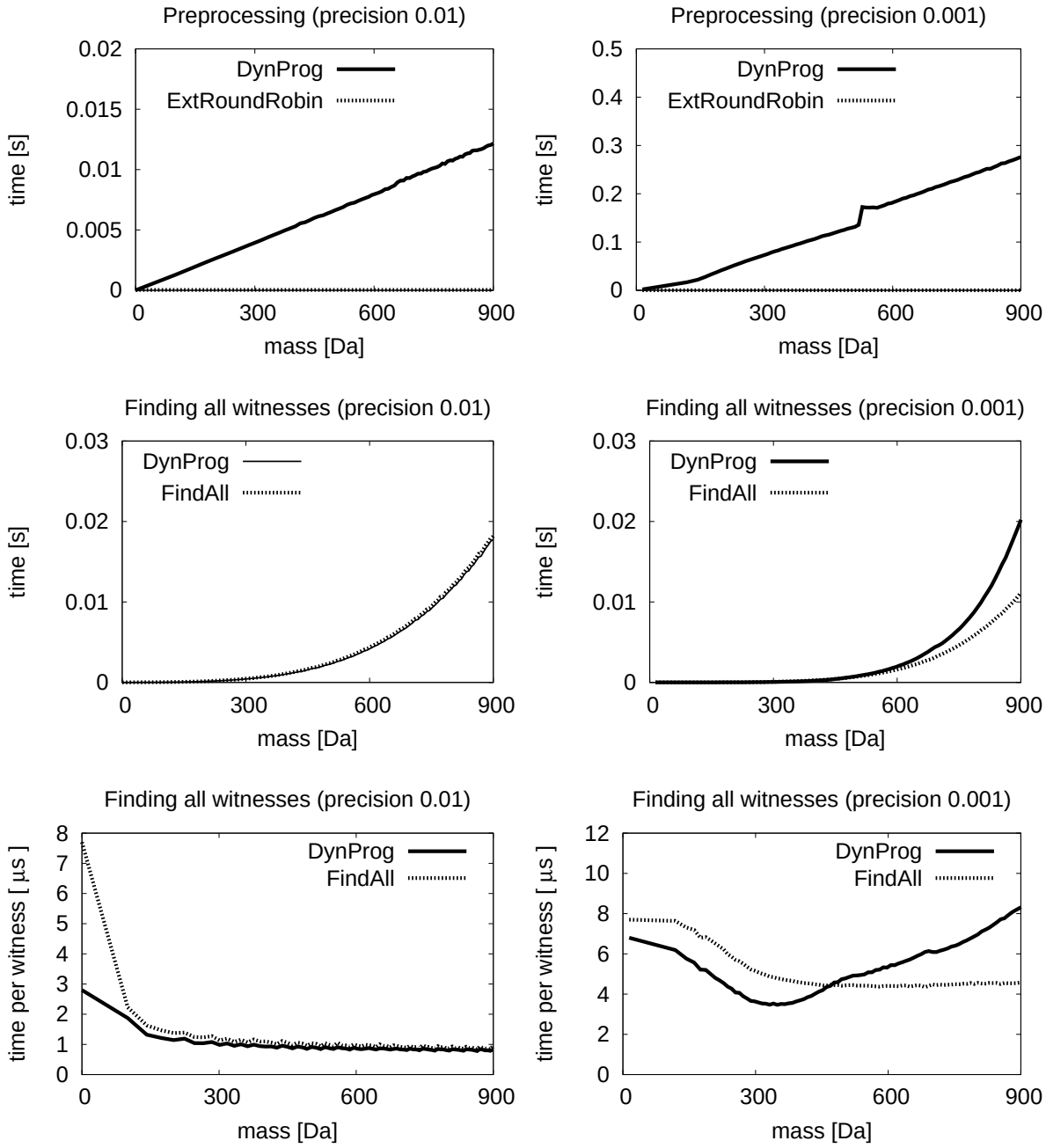


Figure 4.9: **Bioatoms alphabet.** Runtime comparison for finding all witnesses with precision 0.01 Da (left) and 0.001 Da (right). Top: Preprocessing, center: query algorithm, bottom: query algorithm, runtime per witness.

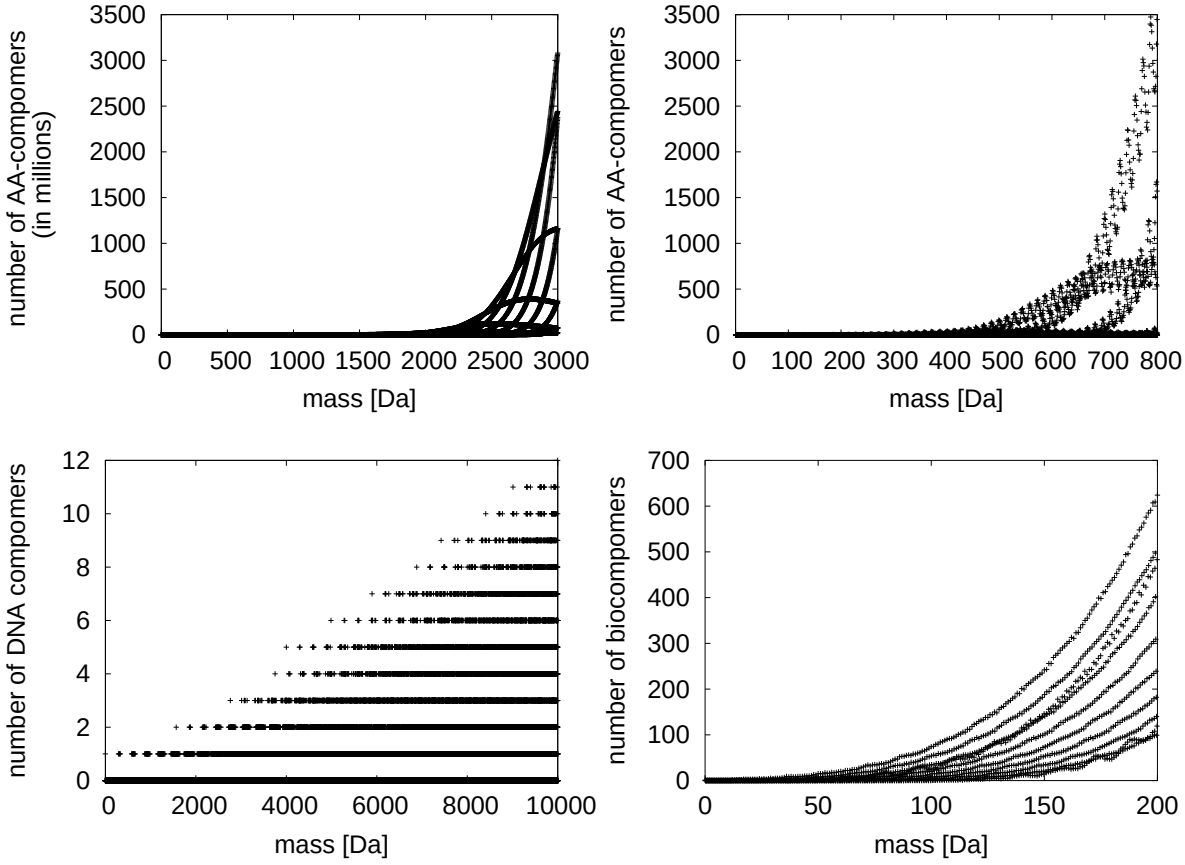


Figure 4.10: Top: Number of amino acid compomers up to 3000 Da (left) and up to 800 Da (right). Bottom: Number of DNA compomers up to 10 000 Da (left) and biomolecules up to 200 Da (right). Computation precision is 0.001 Da, output is merged within 0.1 Da.

5 Submass Finding Algorithms

In this chapter, we investigate variants of the SUBMASS DECISION PROBLEM. We recall the definitions from Chapter 3. Fix a weighted alphabet (Σ, μ).

SUBMASS DECISION PROBLEM:
Given a string s and a mass M, is M a submass of s?

SUBMASS ONE WITNESS PROBLEM:
Given a string s and a mass M, return a witness of M in s, if there is one.

SUBMASS ALL WITNESSES PROBLEM:
Given a string s and a mass M, return all witnesses of M in s.

SUBMASS COUNTING PROBLEM:
Given a string s, how many submasses does s have, i.e., determine $\mathbf{m}(s)$.

Let the length of s be n. We will first introduce simple algorithms for these problems. In Section 5.2, we present an algorithm which solves the SUBMASS DECISION PROBLEM in query time $\mathcal{O}(\log n)$, using storage space $\mathcal{O}(n)$. The idea is to preprocess and store the submasses using a simple geometric property in two dimensions. In Section 5.3, we present several algorithms for the multiple mass versions of the above problems, such as:

MULTIPLE SUBMASS DECISION PROBLEM:
Given a string s of length n and masses $M_1, \ldots, M_r$, return a subset $I \subseteq \{1, \ldots, r\}$ such that $i \in I$ if and only if M_i is a submass of s.

These latter algorithms rely for their efficiency on Fast Fourier Transform (FFT) for polynomial multiplication.

Parts of this chapter have appeared in [CEL+02], [CEL+04], and in [BCL04].

5.1 First solutions and overview of results

All these problems (submass decision, one witness, all witnesses, multiple masses, and counting) can be solved by one of several simple algorithms that we now describe. The first algorithm, which we refer to as LINSEARCH, moves two pointers along the string, one pointing to the potential beginning and the other to the potential end of a substring with mass M. The right pointer is moved if the mass of the current substring is smaller than M, the left pointer, if the current mass is larger than M. The algorithm solves each of the single-mass problems in $\mathcal{O}(n)$ time and uses $\mathcal{O}(1)$ space in addition to the storage space required for the input string

and the output. (Note that LINSEARCH also solves the SUBMASS ALL WITNESSES PROBLEM in $\mathcal{O}(n)$ time, independent of the number of witnesses, i.e., the output size, since $\kappa(M) \leq n$ for all submasses M of s. Recall that $\kappa(M)$ is the number of witnesses of M.)

Another simple algorithm, which we refer to as BINSEARCH, computes all submasses of s in a preprocessing step and stores them in a sorted array, which can then be queried in time $\mathcal{O}(\log n)$ for an input mass for the SUBMASS DECISION PROBLEM and the SUBMASS ONE WITNESS PROBLEM. The storage space required is proportional to $\mathbf{m}(s)$, the number of different submasses of string s, and is thus $\mathcal{O}(n^2)$, while the preprocessing time is $\Theta(n^2 \log \mathbf{m}(s))$. For the SUBMASS ALL WITNESSES PROBLEM, we need to store in addition all witnesses, requiring space $\Theta(n^2)$; in this case, the query time for mass M becomes $\mathcal{O}(\log n + \kappa(M))$.

Alternatively, we can use a Boolean array of size μ_s for storing all submasses of s, thus allowing constant time access for queries. We refer to this algorithm as BOOLEANARRAY. Then the query running time becomes $\mathcal{O}(1)$ and the storage space $\Theta(\mu_s)$, where $\mu_s = \mu(s)$. This yields query time $\mathcal{O}(1)$ for the SUBMASS DECISION PROBLEM and the SUBMASS ONE WITNESS PROBLEM, and $\mathcal{O}(\kappa(M))$ for the SUBMASS ALL WITNESSES PROBLEM.

All these algorithms can of course be extended for the multiple mass versions of the problems, by simply running the query algorithm r times for r query masses. This will yield, for LINSEARCH, a query time of $\Theta(rn)$ for the three multiple submass problems. For BINSEARCH, we get $\Theta(r \log n)$ for the MULTIPLE SUBMASS DECISION PROBLEM and the MULTIPLE SUBMASS ONE WITNESS PROBLEM, and for the MULTIPLE SUBMASS ALL WITNESSES PROBLEM, $\Theta(\max(r, K))$ query time, where $K = \sum_{i=1}^{k} \kappa(M_i)$ is the total number of witnesses for the query masses. Note that any algorithm solving the MULTIPLE SUBMASS ALL WITNESSES PROBLEM will have runtime $\Omega(K)$. For BOOLEANARRAY, the query time for the first two problems is then $\mathcal{O}(r)$, and for the MULTIPLE SUBMASS ALL WITNESSES PROBLEM, $\Theta(\max(r, K))$.

Finally, the SUBMASS COUNTING PROBLEM can be solved by simply computing all masses of substrings of s in time $\mathcal{O}(n^2 \log \mathbf{m}(s)) = \mathcal{O}(n^2 \log n)$ by computing $\mu(s_i \dots s_j)$ for each pair of (i, j) and sorting them. Alternatively, using a suffix tree structure and a linear suffix tree algorithm, in time $\mathcal{O}(\mathbf{s}(s) \log n)$.

In [CEL+04], we introduced an algorithm, referred to as LOOKUP, which solves the SUBMASS ONE WITNESS PROBLEM (and thus, also the decision problem) in $\mathcal{O}(\frac{n}{\log n})$ query time and $\mathcal{O}(n)$ storage space, using $\mathcal{O}(n)$ time and space for the preprocessing. The idea is to speed up LINSEARCH: During the query phase, the algorithm moves two pointers along the string pointing to the potential beginning and end of a substring with mass equalling query M. However, the pointers are not moved character by character, but rather block by block, where each block has length approximately $\log n$. For each pair of positions, a lookup step in a data structure is needed which tells us whether the current difference required (between query M and the mass of the current substring) can be achieved by moving one of the two pointers *within* the current block. If this is not possible, the lookup step also tells us which of the two pointers need to be moved for the next lookup. Carefully choosing the block size will yield the complexity claimed above. However, this is an asymptotic result only, since the constants in this running time are so large that

for a 20-letter alphabet and realistic string sizes, the algorithm is not applicable.

In this chapter, we introduce four new efficient algorithms for different variants of the submass finding problem. In Table 5.1, we present an overview of the runtimes for the multiple mass versions of the problems, in comparison to the two simple algorithms discussed before. We denote by $\mathcal{O}(A + B)$ that the preprocessing has runtime $\mathcal{O}(A)$ and the query time is $\mathcal{O}(B)$. All our algorithms assume that the masses are integers, i.e., that $\mu : \Sigma \to \mathbb{N}$. Recall that $\mu(s)$ denotes the total mass of the string s.

problem	LINS.	BINSEARCH	our algorithms	name/section
DECISION	$\mathcal{O}(rn)$	$\mathcal{O}(n^2 \log n + r \log n)$	$\mathcal{O}(n^2 + r \log n)$ *binary alph.* $\mathcal{O}(\mu(s) \log \mu(s) + r \log n)$	INTERVAL, 5.2 POLLY, 5.3.1
COUNTING	——	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(\mu(s) \log \mu(s))$	POLLY, 5.3.1
ONE WITNESS	$\mathcal{O}(rn)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(\mu(s) \log^3 \mu(s) + r \log n)$ *expected*	POLLYLASVEGAS, 5.3.2
ALL WITNESSES	$\mathcal{O}(rn)$	$\mathcal{O}(n^2 \log n + (r \log n + K))$	$\mathcal{O}((Kn\mu(s) \log \mu(s))^{1/2})$	POLLYDIVIDE, 5.3.3

Table 5.1: Runtime comparison of algorithms for multiple mass problems.

Of our two algorithms for the decision problem, the first one, INTERVAL, assumes that the string s is a string over a binary alphabet, while the second one, POLLY, works for arbitrary alphabets. Even though INTERVAL is in general slower—$\mathcal{O}(n^2)$ vs. $\mathcal{O}(\mu(s) \log \mu(s))$ for the preprocessing, with identical query time—, its storage requirements are much better, namely $\mathcal{O}(n)$ vs. $\mathcal{O}(\mathbf{m}(s))$. This is due to an efficient storage scheme which encodes masses according to their residue modulo the mass difference $|\mu(a) - \mu(b)|$, where a and b are the two characters of the alphabet.

5.2 An algorithm for binary alphabets

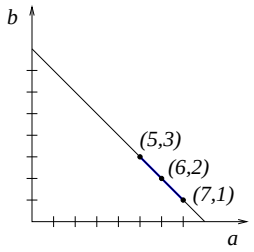
In this section, we present algorithm INTERVAL which solves the SUBMASS DECISION PROBLEM for an alphabet of size 2. It uses storage space $O(n)$ and has query time $O(\log n)$.

Let s be a string over $\Sigma = \{a, b\}$ of length n and fix $p \leq n$. Observe that, when sliding a window of size p over s, then, in one step, the multiplicities of a and b within the window change at most by one. We represent substrings of s by points in the $\mathbb{N}_0 \times \mathbb{N}_0$ lattice, where the two coordinates signify the multiplicities of a and b:

$$S_p := \{(i, j) \in \mathbb{N}_0 \times \mathbb{N}_0 \mid i + j = p, \text{ there is a substring } t \text{ of } s \text{ s.t. } |t|_a = i, |t|_b = j\}. \quad (5.1)$$

All points in S_p will lie on a line (a diagonal), and moreover, they will be neighbours. We will refer to such a set of neighbours on a line as an *interval*. Each such interval has two extremal points.

Example 3. $s = aaaaabaabb$. The figure shows the representation of all substrings of length $p = 8$. Extremal points of this interval are $(5,3)$ and $(7,1)$.



Assume for a moment that we know the multiplicities of a and b in M, e.g. $M = i \cdot \mu(a) + j \cdot \mu(b)$. Then we can easily find out whether M is a submass of s: We store the S_p's, for $1 \leq p \leq n$ by their extremal points during the preprocessing phase. Now we only have to check whether $(i,j) \in S_{i+j}$, which takes $O(1)$ time. This requires storage space linear in n. If, in addition, i and j were known to be the only feasible multiplicities of a and b (i.e., the unique solution of the equation $x \cdot \mu(a) + y \cdot \mu(b) = M$), then this algorithm would even decide whether M is a submass of s, and we would be done.

Unfortunately, we do not know the multiplicities of a and b in M. Without loss of generality, assume $\mu(a) < \mu(b)$. We define $d := \mu(b) - \mu(a)$ and use the residue of M mod d to look up a table. The table, generated during the preprocessing phase, contains representations of all submasses of s.

Let $M_p := \{\mu(t) \mid t \text{ is a } p\text{-length substring of } s\}$. Observe that consecutive elements of M_p (when sorted) differ by exactly d. Therefore, we can write $M_p = \{c_p + \ell \cdot d \mid \ell = 0, \ldots, n_p - 1\}$, where $c_p = \min M_p$ and $n_p = |M_p|$. Furthermore, $M_p = \{r_p + \ell \cdot d \mid \ell = a_p, \ldots, b_p\}$, where $r_p = (c_p \bmod d)$, $a_p = \lfloor \frac{c_p}{d} \rfloor$ and $b_p = a_p + n_p - 1$. This says that all submasses of the same length have the same residue modulo d.

Example 3 cont'd: Let $s = aaaaabaabb$ and $\mu(a) = 2$ and $\mu(b) = 7$. Then $d = 5$, and

$S_{10} = \{(7,3)\}$	$M_{10} = \{35\}$	$r_{10} = 0, a_{10} = 7, b_{10} = 7$
$S_9 = \{(7,2),(6,3)\}$	$M_9 = \{28,33\}$	$r_9 = 3, a_9 = 5, b_9 = 6$
$S_8 = \{(7,1),(6,2),(5,3)\}$	$M_8 = \{21,26,31\}$	$r_8 = 1, a_8 = 4, b_8 = 6$
$S_7 = \{(6,1),(5,2),(4,3)\}$	$M_7 = \{19,24,29\}$	$r_7 = 4, a_7 = 3, b_7 = 5$
$S_6 = \{(5,1),(4,2),(3,3)\}$	$M_6 = \{17,22,27\}$	$r_6 = 2, a_6 = 3, b_6 = 5$
$S_5 = \{(5,0),(4,1),(3,2),(2,3)\}$	$M_5 = \{10,15,20,25\}$	$r_5 = 0, a_5 = 2, b_5 = 5$
$S_4 = \{(4,0),(3,1),(2,2)\}$	$M_4 = \{8,13,18\}$	$r_4 = 3, a_4 = 1, b_4 = 3$
$S_3 = \{(3,0),(2,1),(1,2)\}$	$M_3 = \{6,11,16\}$	$r_3 = 1, a_3 = 1, b_3 = 3$
$S_2 = \{(2,0),(1,1),(0,2)\}$	$M_2 = \{4,9,14\}$	$r_2 = 4, a_2 = 0, b_2 = 2$
$S_1 = \{(1,0),(0,1)\}$	$M_1 = \{2,7\}$	$r_1 = 2, a_1 = 0, b_1 = 1$

Observe that $r_p = (p \cdot \mu(a) \bmod d)$. Thus, we may have the same residue modulo d for different values of p. Instead of storing a_p and b_p for each r_p individually (which could result in linear query time), we will store the union of all intervals which belong to the same residue r, sorted by their endpoints. This may result in merging intervals: If $[x,y]$ and $[y+1,z]$ occur as intervals for the same residue, they are replaced by $[x,z]$.

Example 3 cont'd: In the example, this yields the following preprocessed data. For residues 1 and 4, the intervals have been merged.

residue modulo d	union of intervals
0	$[2,5], [7,7]$
1	$[1,6]$
2	$[0,1], [3,5]$
3	$[1,3], [5,6]$
4	$[0,5]$

5.2.1 Algorithm INTERVAL

In the preprocessing phase, we compute the r_p's, a_p's, and b_p's as above. We then sort the r_p's, thus obtaining a sorted array $q_1, \ldots, q_m$, where $m \leq n$ (since different S_p's may have the same residue). For each q_ℓ, $1 \leq \ell \leq m$, we compute a list of interval endpoints which represents the union of all intervals $[a_p, b_p]$ with $r_p = q_\ell$. This list consists of one or more disjoint intervals, which we store in sorted order in an array A_ℓ.

Now, when querying whether a given mass M is a submass of s,

Algorithm INTERVAL

1. decompose $M = g \cdot d + r$, where $r = (M \bmod d)$ and $g \in \mathbb{N}$;

2. find index $\ell \in \{1, \ldots, m\}$ such that $r = q_\ell$, using binary search; if no such index can be found, then M is not a submass of s, and the algorithm outputs NO;

3. otherwise, find whether there is an interval $[a, b]$ in array A_ℓ such that $g \in [a, b]$, using binary search on (the left endpoints of) the intervals; M is a submass of s if and only if such an interval exists.

Since the total number of intervals to be stored is n, the storage space needed is $O(n)$. The first step of the query algorithm takes time $O(1)$. The second step takes time $O(\log n)$, since the number of different residues is at most n. The third step takes time $O(\log n)$, since the maximum number of intervals stored in one array A_ℓ is n. We obtain a total query time $O(\log n)$.

Theorem 5.2.1. *Algorithm* INTERVAL *solves the* SUBMASS DECISION PROBLEM *for binary alphabets with storage space* $O(n)$ *and query time* $O(\log n)$.

The problem in generalizing this approach to larger alphabets is that the algorithm relies on the crucial fact that points representing substrings of the same length lie on a line and form an interval. This does not generalize to higher dimensions, since there we only know that the points representing substrings of the same length are connected.

5.3 Submass finding with polynomials

In this section, we introduce algorithms for the multiple mass versions of the submass finding problems, i.e., algorithms solving the following problems: Fix a weighted alphabet (Σ, μ).

> MULTIPLE SUBMASS DECISION PROBLEM:
> Given a string s with $|s| = n$ and masses $M_1, \ldots, M_r$, return a subset $I \subseteq \{1, \ldots, r\}$ such that $i \in I$ if and only if M_i is a submass of s.

> MULTIPLE SUBMASS ONE WITNESS PROBLEM:
> Given a string s with $|s| = n$ and masses $M_1, \ldots, M_r$, return a subset $I \subseteq \{1, \ldots, r\}$ such that $i \in I$ if and only if M_i is a submass of s, and a set $\{(b_i, e_i) \mid i \in I, (b_i, e_i)$ is witness of M_i in $s\}$.

> MULTIPLE SUBMASS ALL WITNESSES PROBLEM:
> Given a string s with $|s| = n$ and masses $M_1, \ldots, M_r$, return a subset $I \subseteq \{1, \ldots, r\}$ such that $i \in I$ if and only if M_i is a submass of s, and for each $i \in I$, the set of all witnesses $W_i := \{(b, e) \mid (b, e)$ is witness of M_i in $s\}$.

In the rest of the chapter, we assume that all character masses are integers. Our algorithms are based on encoding the submasses of string s with appropriately chosen polynomials, and their efficiency derives from using Fast Fourier Transform (FFT) for polynomial multiplication. The first algorithm we introduce also solves the SUBMASS COUNTING PROBLEM.

Let μ_s denote the total mass of the string s, i.e., $\mu_s = \mu(s)$. Our algorithms all have running times which depend on μ_s, and space complexity $\mathcal{O}(\mathbf{m}(s))$. However, we can use the sparse polynomial multiplication technique of Cole and Hariharan [CH02] to give Las Vegas variants of our algorithms, where each term μ_s in the expected running time can be replaced by $\mathbf{m}(s)\,\mathrm{polylog}(\mathbf{m}(s))$. Thus, throughout this section, we present our runtimes as a function of μ_s with the understanding that μ_s is identical to $\mathbf{m}(s)$ up to polylogarithmic factors.

5.3.1 Searching for submasses using polynomials

In this section, we introduce the main idea of our algorithms, how to encode submasses via polynomials. We first prove some crucial properties, and then discuss algorithmic questions.

Define, for $0 \leq i \leq n$,

$$p_i := \sum_{j=1}^{i} \mu(s_j) = \mu(s_1 \ldots s_i), \tag{5.2}$$

the i'th prefix mass of s. In particular, $p_0 = \mu(\varepsilon) = 0$. The underlying idea is that any submass can be written as the difference between one prefix mass and another prefix mass; in particular, if $M = \mu(s_i \ldots s_j)$, then $M = p_j - p_{i-1}$, see Figure 5.1.

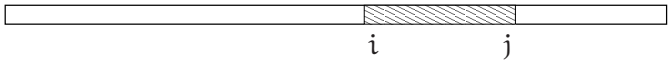


Figure 5.1: The submass $M = \mu(s_i \dots s_j) = p_j - p_{i-1}$.

We will capture this property by defining two polynomials whose product will have exactly the submasses of s (save for a shift) as the exponents of monomials with non-zero coefficients.

$$P_s(x) := \sum_{i=1}^{n} x^{p_i} = x^{\mu(s_1)} + x^{\mu(s_1 s_2)} + \dots + x^{\mu_s}, \tag{5.3}$$

$$Q_s(x) := \sum_{i=0}^{n-1} x^{\mu_s - p_i} = x^{\mu_s} + x^{\mu_s - \mu(s_1)} + \dots + x^{\mu_s - \mu(s_1 \dots s_{n-1})}, \tag{5.4}$$

and consider the product of $P_s(x)$ and $Q_s(x)$,

$$C_s(x) := P_s(x) \cdot Q_s(x) = \sum_{m=0}^{2\mu_s} c_m x^m. \tag{5.5}$$

Recall that we denote by $\kappa(M)$ the number of witnesses (i, j) of mass M in string s.

Lemma 5.3.1. *Let $P_s(x), Q_s(x)$ and $C_s(x)$ from Equations (5.3) through (5.5). Then for any $m \le \mu_s$, $\kappa(m) = c_{m+\mu_s}$, i.e., the coefficient $c_{m+\mu_s}$ of $C_s(x)$ equals the number of witnesses of m in s.*

Proof. By definition, we have

$$\begin{aligned} C_s(x) &= P_s(x) \cdot Q_s(x) = \sum_{j=1}^{n} x^{p_j} \sum_{i=0}^{n-1} x^{\mu_s - p_i} = x^{\mu_s} \cdot \sum_{1 \le i,j \le n} x^{p_j - p_{i-1}} \\ &= x^{\mu_s} \cdot \sum_{1 \le i \le j \le n} x^{\mu(s_i \dots s_j)} + x^{\mu_s} \cdot \sum_{1 \le j < i \le n} x^{-\mu(s_{j+1} \dots s_{i-1})}. \end{aligned}$$

Let $[x^i]A(x)$ denote the coefficient a_i of x^i of the polynomial $A(x) = \sum_j a_j x^j$. Then, for any $m \le \mu_s$,

$$\begin{aligned} \kappa(m) &= |\{(i,j) \mid \mu(s_i \dots s_j) = m\}| = [x^m]\left(\frac{1}{x^{\mu_s}} C_s(x)\right) \\ &= [x^{m+\mu_s}] C_s(x) = c_{m+\mu_s}. \end{aligned}$$

□

Lemma 5.3.1 immediately implies the following. Recall that for a proposition Λ, $[\Lambda] = 1$ if Λ is true, and 0 otherwise.[1]

[1] We stick to the use of $[,]$ both for the characteristic function of a proposition and for the coefficient of a polynomial, since this is standard notation, and the two meanings cannot be confused.

Corollary 5.3.2. *For* $C_s(x)$ *from* (5.5)*, the number of non-zero coefficients greater than* μ_s *equals the number of submasses of* s*:*

$$\sum_{m=\mu_s+1}^{2\mu_s} [c_m \neq 0] = \mathbf{m}(s),$$

Furthermore, $\sum_{m=\mu_s+1}^{2\mu_s} c_m = \frac{n(n+1)}{2}$.

Thus, polynomial C_s also allows us to compute the number of submasses of s.

Example 4. Let $s = baac$, $\mu(a) = 2, \mu(b) = 3, \mu(c) = 5$. Then,

$$\begin{aligned} P_s(x) &= x^3 + x^5 + x^7 + x^{12}, \\ Q_s(x) &= x^{12} + x^9 + x^7 + x^5, \quad \text{and} \\ C_s(x) &= x^8 + 2x^{10} + 3x^{12} + 2x^{14} + x^{15} + x^{16} + 2x^{17} + 2x^{19} + x^{21} + x^{24}. \end{aligned}$$

Dividing the terms $c_i x^i$ with $i > 12$ by $x^{12} = x^{\mu_s}$ yields $2x^2+x^3+x^4+2x^5+2x^7+x^9+x^{12}$. This yields the submasses $2, 3, 4, 5, 7, 9, 12$ with two witnesses for $2, 5$, and 7, and one witness for each of the other submasses.

Algorithm and analysis

We now present an algorithm to solve the Multiple Submass Decision Problem and the Submass Counting Problem. The algorithm primarily consists of computing polynomial $C_s(x)$.

Algorithm Polly

1. Preprocessing step:
   Compute μ_s, compute $C_s(x)$, and store in a sorted array all numbers $m - \mu_s$ for exponents $m > \mu_s$ where $c_m \neq 0$.
2. Query step:
   a) For the Multiple Submass Decision Problem: Search for each query mass M_i for $1 \leq i \leq r$, and return yes if found, no otherwise.
   b) For the Submass Counting Problem: Return size of array.

Correctness of Polly follows immediately from the previous lemmas.

Theorem 5.3.3. *Algorithm* Polly *solves the* Multiple Submass Decision Problem *in time* $\mathcal{O}(\mu_s \log \mu_s + r \log n)$ *and the* Submass Counting Problem *in time* $\mathcal{O}(\mu_s \log \mu_s)$.

Proof. The polynomial $C_s(x)$ can be computed efficiently using Fast Fourier Transform (FFT) [CT65], which runs in time $\mathcal{O}(\mu_s \log \mu_s)$, since $C_s(x)$ has degree $2\mu_s$. Hence, the preprocessing step takes time $\mathcal{O}(\mu_s \log \mu_s)$. The query time for the Multiple Submass Decision Problem is $\mathcal{O}(r \log \mathbf{m}(s)) = \mathcal{O}(r \log n)$. □

Instead of using a sorted array, we can store the submasses in an array of size μ_s (which can be hashed to $\mathcal{O}(\mathbf{m}(s))$ size) and allow for direct access in constant time, thus reducing the query time to $\mathcal{O}(r)$. As mentioned earlier, we can employ methods from [CH02] for sparse polynomials and reduce $\deg C_s$ to $\mathcal{O}(\mathbf{m}(s))$, the number of non-zero coefficients. However, for the rest of this section, we will refer to the running time as proportional to $\mu_s \log \mu_s$.

As an aside, note that $\mu_s \leq \mu_{\max} n$, where $\mu_{\max} = \max \mu(\Sigma)$. If the maximal mass can be viewed as a constant, this yields runtime $\mathcal{O}(n \log n)$ for the preprocessing step. It may not always be realistic to assume that $\mu_{\max}$ is constant, because in order to enforce that all masses be positive integers, a scaling of the masses may be necessary, which can blow them up significantly.[2] However, even in this case, the algorithm outperforms BINSEARCH for the SUBMASS DECISION PROBLEM as long as $\mu_{\max} = o(\frac{n}{\log n})$.

Along the same lines, for the SUBMASS COUNTING PROBLEM, our algorithm allows computation of $\mathbf{m}(s)$ in $\mathcal{O}(\mu_s \log \mu_s) = \mathcal{O}(n \mu_{\max} \log(n \mu_{\max}))$ time. The naïve solution of generating all submasses requires $\Theta(n^2 \log n)$ time and $\Theta(\mathbf{m}(s))$ space (with sorting), or $\Theta(n^2)$ time and $\Theta(\mu_s)$ space (with an array of size μ_s). Our algorithm thus outperforms this naïve approach as long as $\mu_{\max} = o(\frac{n}{\log n})$.

5.3.2 A Las Vegas algorithm for finding witnesses

We now describe how to efficiently find a witness for each submass of the string s. Our high level idea is the following: We first note that given a mass M, if we know the ending position j of a witness of M, then, using the prefix masses $p_1, \ldots, p_n$, we can easily find the beginning position of this witness. To do so, we simply do a binary search amongst the prefix masses $p_1, \ldots, p_{j-1}$ for mass $p_j - M$. Below, we will define two suitable polynomials of degree at most μ_s such that the coefficient of $x^{M+\mu_s}$ in their product equals the sum of the ending positions of substrings that have mass M.

Now, if we knew that there was a unique witness of mass M, then the coefficient would equal the ending position of this witness. However, this need not always be the case. In particular, if there are many witnesses with mass M, then we would need to check all partitions of the coefficient of $x^{M+\mu_s}$, which is computationally far too costly. To get around this problem, we look for the witnesses of M in the string s, where we do not consider all pairs of positions but instead random subsets of these.

By using the definition of $Q(x)$ from (5.4), set

$$R_s(x) := \sum_{i=1}^{n} i \cdot x^{p_i} \quad \text{and} \tag{5.6}$$

$$F_s(x) := R_s(x) \cdot Q_s(x) = \sum_{m=0}^{2\mu_s} f_m x^m. \tag{5.7}$$

In the following lemma, we use the definition of c_m from (5.5).

[2] This can be the case, e.g., for protein strings, where the amino acid masses are known up to a precision of more than 10^{-5}.

Lemma 5.3.4. *Let $m > \mu_s$. If $c_m = 1$, then f_m equals the ending position of the (sole) witness of $m - \mu_s$.*

Proof. By definition,

$$f_m = \sum_{(i,j) \text{ witness of } m} j$$

for any $m > \mu_s$. If $c_m = 1$, then, by Lemma 5.3.1, $m - \mu_s$ has exactly one witness (i_0, j_0). Thus, $f_m = j_0$. □

Example 5. We continue with Example 4 on string $s = baac$ and masses $2, 3, 5$ for characters a, b, c. We get $R_s(x) = x^3 + 2x^5 + 3x^7 + 4x^{12}$ and $F_s(x) = x^8 + 3x^{10} + 6x^{12} + 5x^{14} + x^{15} + 3x^{16} + 6x^{17} + 7x^{19} + 4x^{21} + 4x^{24}$. By checking the coefficients of $C_s(x)$, we see that among the exponents $m > \mu_s$, c_{15}, c_{16}, c_{21}, and c_{24} equal 1. Thus, with $F_s(x)$, we now know that the only witnesses of the submasses $3, 4, 9$, and 12 end at positions $1, 3, 4$, and 4, respectively.

The algorithm

We now present a Las Vegas algorithm for the MULTIPLE SUBMASS ONE WITNESS PROBLEM. In the algorithm, we first use polynomial $C_s(x)$ to generate a data structure containing all submasses of s. We then run a procedure which uses random subsets to try and find witnesses for each of these submasses. It outputs a set of pairs (m, j_m), where m is a submass of s, and j_m is the ending position of one witness of m. Then, for each query mass which is in this set, we find the beginning position of the witness in time $\mathcal{O}(\log n)$ with binary search within the prefix masses, as described above. For any remaining query masses which are submasses of s, we simply run LINSEARCH to find a witness.

Algorithm POLLYLASVEGAS

1. Compute $C_s(x)$ from Equation (5.5), and store all submasses of s.
2. Procedure TRY-FOR-WITNESS
   (i) For a from 1 to $2\log_2 n$, do:
   (ii) Let $b = 2^{-a/2}$. Repeat $24 \ln n$ times:
   (iii)
   - Generate a random subset I_1 of $\{1, 2, \ldots, n\}$, and a random subset I_2 of $\{0, 1, 2, \ldots, n-1\}$, where each element is chosen independently with probability b.
   - Compute $P_{I_1}(x) = \sum_{i \in I_1} x^{p_i}$, $Q_{I_2}(x) = \sum_{i \in I_2} x^{\mu_s - p_i}$ and $R_{I_1}(x) = \sum_{i \in I_1} i \cdot x^{p_i}$.
   - Compute $C_{I_1,I_2}(x) = P_{I_1}(x) \cdot Q_{I_2}(x)$ and $F_{I_1,I_2}(x) = R_{I_1}(x) \cdot Q_{I_2}(x)$.
   - Let $c_m = [x^m]C_{I_1,I_2}(x)$ and $f_m = [x^m]F_{I_1,I_2}(x)$.
   - For $m > \mu_s$, if $c_m = 1$ and if m has not yet been successful, then store the pair $(m - \mu_s, f_m)$. Mark m as successful.

3. Check which of the query masses is a submass of s by looking them up in the data structure generated in Step 1. Exclude all queries that are not submasses of s.

4. For all submasses amongst the queries M_ℓ, $1 \leq \ell \leq r$, which are marked as successful (i.e., an ending position was found by procedure TRY-FOR-WITNESS), find the beginning position with binary search amongst the prefix masses.

5. If there is a submass M_ℓ for which no witness was found, find one using LINSEARCH.

Analysis

We first give an upper bound on the failure probability of procedure TRY-FOR-WITNESS for a particular query mass M.

Lemma 5.3.5. *For a query mass M with $\kappa(M) = \kappa$, and $a = \lceil \log_2 \kappa \rceil$, consider the Step 2.iii of* POLLYLASVEGAS. *The probability that the coefficient $c_{M+\mu_s}$ of $C_{I_1,I_2}(x)$ for value a (as defined above) is not 1 is at most $\frac{7}{8}$.*

Proof. Let the witnesses of M be $\{(b_1, e_1), \ldots, (b_\kappa, e_\kappa)\}$. Clearly $0 \leq \kappa \leq n$. We first analyze the probability of the event that for this particular choice of a, the coefficient of x^{μ_s+M} in $C(x)$ is exactly 1. This is the case if and only if $|\{i : b_i \in I_1 \text{ and } e_i \in I_2, 1 \leq i \leq \kappa\}| = 1$. Now, for $1 \leq i \leq \kappa$, let E_i denote the event $E_i = \{b_i \in I_1\} \cap \{e_i \in I_2\}$. Since for any $i \neq j$, we have $b_i \neq b_j$ and $e_i \neq e_j$, it follows that E_i and E_j are independent events. Thus, the probability that exactly one of the E_i's holds is

$$\kappa 2^{-a} \cdot (1 - 2^{-a})^{\kappa-1} > \kappa 2^{-a} \cdot (1 - 2^{-a})^{2^a} \geq \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{8}.$$

The last inequality follows because $(1 - \epsilon)^{1/\epsilon} \geq \frac{1}{4}$ for any $\epsilon \leq \frac{1}{2}$. □

Lemma 5.3.6. *Procedure* TRY-FOR-WITNESS *does not find a witness for a given submass M with probability at most $1/n^3$. Moreover, the probability that the procedure fails for some submass is at most $1/n$.*

Proof. By Lemma 5.3.5 we know that for any fixed submass M, and for the particular choice of a, the probability that the random choice of I_1 and I_2 produces a unique witness for M is at least $1/8$. Since Step 2.iii is repeated $24 \ln n$ times, and all trials are independent of each other, the probability that there is no unique witness for any run is at most $(7/8)^{24 \ln n} \leq e^{-3 \ln n} = \frac{1}{n^3}$. This follows since $(1 - \epsilon)^{1/\epsilon} \leq e^{-1}$ for any $0 < \epsilon < 1$. Since there are at most $O(n^2)$ different submasses in a string of length n, using the union bound, the algorithm generates a witness for each distinct submass with probability at least $1 - n^2 \cdot \frac{1}{n^3} = 1 - 1/n$. □

Theorem 5.3.7. *Algorithm* POLLYLASVEGAS *solves the* MULTIPLE SUBMASS ONE WITNESS PROBLEM *in expected time $\mathcal{O}(\mu_s \log^3 \mu_s + r \log n)$.*

Proof. Denote the number of distinct submasses amongst the query masses by r', thus, $r' \leq r$. By Lemma 5.3.6, the probability that procedure TRY-FOR-WITNESS finds a witness for each of the $r' = O(n^2)$ submasses is at least $1 - 1/n$. In this case, the running time is the time for running the procedure, plus the time for finding witness beginning positions. On the other hand, the probability that the procedure fails to find a witness is at most $1/n$. In this case, we run LINSEARCH for the missing query masses, each in time $\mathcal{O}(n)$, thus at most in overall time $\mathcal{O}(r'n)$. Plugging it all together we get:

$$\begin{aligned} & \mathcal{O}(\underbrace{\mu_s \log \mu_s}_{\text{Step 1.}} + \underbrace{2 \log n}_{\text{Step 2.i}} \cdot \underbrace{24 \ln n}_{\text{Step 2.ii}} \cdot \underbrace{\mu_s \log \mu_s}_{\text{Steps 2.iii}}) \\ + \quad & \underbrace{\mathcal{O}(r \log n)}_{\text{Step 3.}} + (1 - \frac{1}{n})\mathcal{O}(r' \log n) + \frac{1}{n}\mathcal{O}(r'n) \\ = \quad & \mathcal{O}(\mu_s \log^3 \mu_s + r \cdot \log n). \end{aligned}$$

□

5.3.3 A deterministic algorithm for finding all witnesses

Recall that, given the string s of length n and r query masses $M_1, \ldots, M_r$, we are able to solve the MULTIPLE SUBMASS ALL WITNESSES PROBLEM in $\Theta(r \cdot n)$ time and $\mathcal{O}(1)$ space with LINSEARCH, or in $\Theta(n^2 \log n + r \log n)$ time and $\Theta(n^2)$ space with BINSEARCH. Thus, the two naïve algorithms yield a runtime of $\Theta(\min(rn, (n^2 + r) \log n))$.

Our goal here is to give an algorithm that outperforms the bound above, provided certain conditions hold. Clearly, in general it is impossible to beat the bound $\min(rn, n^2)$ because that might be the size of the output, K, the total number of witnesses to be returned. In the following, we concentrate on the case where $K \ll rn$.

First, consider two strings s and t and their concatenation st. We are interested in submasses of st with a witness that spans or touches the border between s and t. More precisely, we refer to a witness (i, j) as a *border-spanning* witness if and only if $i \leq |s| \leq j$. We can encode such witnesses again in a polynomial, using the definition of $P(x)$ from (5.3). The idea is that the mass of a border-spanning witness can be written as the sum of a prefix mass of s^R, the reverse string of s, and a prefix mass of t. Note that here, we also allow 0 as a submass.

Lemma 5.3.8. *For two strings s, t, and the polynomial*

$$D_{s,t}(x) := (x^0 + P_{s^R}(x)) \cdot (x^0 + P_t(x)) = \sum_{m=0}^{\mu(s)+\mu(t)} d_m x^m, \tag{5.8}$$

the coefficient d_m equals the number of border-spanning witnesses of m in $s \cdot t$.

Proof. Straightforward. □

Example 6. Let $s = ba, t = ac$, and the masses as before. We get $D_{s,t}(x) = x^0 + 2x^2 + x^4 + x^5 + 2x^7 + x^9 + x^{12}$. We compare these to the terms of $\frac{1}{x^{12}} C_{baac}(x)$ with

positive exponent in Example 4, since these yield all non-zero submasses of baac: $2x^2 + x^3 + x^4 + 2x^5 + 2x^7 + x^9 + x^{12}$. We see that the (sole) witness of 3 and one of the witnesses of 5 are not border-spanning witnesses.

The algorithm

The algorithm combines the polynomial method with LINSEARCH in the following way: We divide the string s into g substrings of approximately equal length. We then use polynomials to identify, for each query mass M and each witness (b, e) of M, which substrings the beginning and end index lie in. Then we use LINSEARCH on these substrings to actually find the witnesses. The crucial observation is given in Lemma 5.3.9. We now describe the details.

The string s is divided into g substrings of approximately equal length: $s = t_1 \cdot t_2 \cdots t_g$ (where we will choose g below), and denote by $M_{i,j} = \sum_{m=i+1}^{j-1} \mu(t_m)$. In particular, if $j \leq i + 1$, then $M_{i,j} = 0$. We illustrate in Figure 5.2.

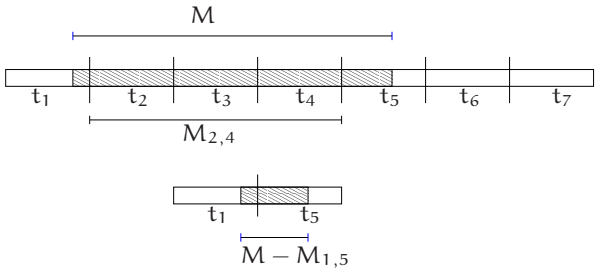


Figure 5.2: Illustration for algorithm POLLYDIVIDE.

In order to have a good choice for g, we need to know the total size of the output, $K = \sum_{\ell=1}^{r} \kappa(M_\ell)$. This we can obtain by computing $C_s(x)$ and then adding up the coefficients $c_{M_\ell + \mu_s}$ for $1 \leq \ell \leq r$. We now set $g = \lceil (\frac{Kn}{\mu_s \log \mu_s})^{\frac{1}{2}} \rceil$. Observe that if $Kn \leq \mu_s \log \mu_s$, then $g = 1$, in which case we are better off running LINSEARCH. So let $Kn > \mu_s \log \mu_s$.

In Step 2.(b) of the following algorithm, we modify LINSEARCH to only return border–spanning submasses. This can be easily done by setting the second pointer at the start of the algorithm to the last position of the first string, and by terminating when the first pointer moves past the first position of the second string.

Algorithm POLLYDIVIDE

1. Preprocesssing step:
   a) Compute μ_s and $C_s(x)$ as defined in (5.5), and compute $K = \sum_{\ell=1}^{r} c_{M_\ell + \mu_s}$. Set $g = \lceil (\frac{Kn}{\mu_s \log \mu_s})^{\frac{1}{2}} \rceil$.
   b) For each $1 \leq i \leq g$, compute $C_{t_i}(x)$.
   c) For each $1 \leq i < j \leq g$, compute $D_{t_i,t_j}(x)$ as defined in (5.8).
2. Query step:

a) Compute a witness-position-list for each query M_ℓ by iterating through all terms of the C_{t_i}'s and all terms of the D_{t_i,t_j}'s. The witness-position-list of M_ℓ contains exactly those i such that M_ℓ is a submass of t_i, and those pairs (i,j), such that $M_\ell - M_{i,j}$ is a border-spanning submass of $t_i \cdot t_j$.

b) For each $1 \leq \ell \leq r$,

   i. If M_ℓ's witness-position-list is empty, then return NO.

   ii. For each i in M_ℓ's witness-position-list, run LINSEARCH on t_i for M_ℓ and return all witnesses.

   iii. For each pair (i,j) in M_ℓ's witness-position-list, run LINSEARCH on $t_i \cdot t_j$ for submass $M_\ell - M_{i,j}$ and return all border-spanning witnesses.

Analysis

The following lemma shows the correctness of algorithm POLLYDIVIDE.

Lemma 5.3.9. *For* $1 \leq M \leq \mu_s$,

$$\kappa(M) = \sum_{i=1}^{g} [x^{M+\mu(t_i)}] C_{t_i} + \sum_{1 \leq i < j \leq g} [x^{M-M_{i,j}}] D_{t_i,t_j}(x).$$

Proof. First, observe that for any witness (b,e) of M, there is exactly one pair (i,j) such that b lies in string t_i and e in t_j. If $i = j$, then M is a submass of t_i and by Lemma 5.3.1 contributes exactly 1 to the coefficient $[x^{M+\mu(t_i)}]C_{t_i}(x)$. Otherwise, $i < j$, and $M - M_{i,j}$ is a submass of the concatenated string $t_i \cdot t_j$ with the witness (b',e'), where (b',e') is shifted appropriately (i.e., $b' = b - \sum_{i'<i} |t_{i'}|$ and $e' = e - \sum_{j'<j} |t_{j'}|$). Moreover, (b',e') is a border-spanning submass of $t_i \cdot t_j$. Thus, by Lemma 5.3.8, (b',e') contributes exactly 1 to $[x^{M-M_{i,j}}]D_{t_i,t_j}(x)$. □

For the runtime analysis of POLLYDIVIDE, we first show that the preprocessing step of POLLYDIVIDE has runtime $\mathcal{O}(g\mu_s \log \mu_s)$. To see this, observe that the time for computing the polynomials with FFT is

$$\begin{aligned}
\text{for } C_s(x)\text{:} \qquad & \mathcal{O}\left(\mu_s \log \mu_s\right), \\
\text{for the } C_{t_i}(x)\text{'s:} \qquad & \mathcal{O}\left(\sum_{i=1}^{g} \mu(t_i) \log(\mu(t_i))\right), \\
\text{for the } D_{i,j}(x)\text{'s:} \qquad & \mathcal{O}\left(\sum_{1 \leq i < j \leq g} (\mu(t_i) + \mu(t_j)) \log(\mu(t_i) + \mu(t_j))\right).
\end{aligned}$$

Together, the terms above yield

$$\begin{aligned}
&\mathcal{O}\Big(\mu_s \log \mu_s + \underbrace{\sum_{i=1}^{g} \mu(t_i) \log(\mu(t_i))}_{\leq\, \mu_s \log \mu_s} + \underbrace{\sum_{1 \leq i < j \leq g} \big(\mu(t_i) + \mu(t_j)\big) \log\big(\mu(t_i) + \mu(t_j)\big)}_{\leq\, g\mu_s \log \mu_s} \Big) \\
&= \mathcal{O}\left(g\mu_s \log \mu_s\right).
\end{aligned}$$

The upper bound on the third term follows from $\sum_{1\le i<j\le g}(\mu(t_i)+\mu(t_j)) = \sum_{i=1}^{g}(g-1)\cdot\mu(t_i) = (g-1)\mu_s$ and $\log(\mu(t_i)+\mu(t_j)) \le \log\mu_s$.

Now for the query time of PollyDivide: First, in Step 2a, we compute for each query M_ℓ the witness-position-list that contains all i such that $[x^{M_\ell+\mu(t_i)}]C_{t_i}(x) \neq 0$, and all (i,j) such that $[x^{M_\ell-M_{i,j}}]D_{t_i,t_j}(x) \neq 0$. These lists can be computed by iterating first through all non-zero coefficients c_m of each C_{t_i}, $1 \le i \le g$, and checking whether $m+\mu(t_i)$ is among the query masses. Recall that there are $\mathcal{O}(\mu_s \log\mu_s)$ many of these coefficients. Next, we iterate through all non-zero coefficients d_m of each D_{t_i,t_j}, $1 \le i < j \le g$, and check whether $m + M_{i,j}$ is among the query masses. Again, there are $\mathcal{O}(g\mu_s\log\mu_s)$ many coefficients to check. Together, we get a runtime of $\mathcal{O}(g\mu_s\log\mu_s)$ if we have constant access to the query masses, or $\mathcal{O}(g\mu_s\log\mu_s\cdot\log r)$ if they are stored in a binary array.

Now, in step 2b, for each query mass M_ℓ, we run LinSearch for each entry in the witness-position-list, thus at most $\kappa(M_\ell)$ many times. The LinSearch step for one entry takes at most $2n/g$ time. Thus, we get query time $\mathcal{O}(g\mu_s\log\mu_s + K\frac{n}{g})$. With $g = \lceil(\frac{Kn}{\mu_s\log\mu_s})^{\frac{1}{2}}\rceil$, the total runtime becomes $\mathcal{O}\big((Kn\mu_s\log\mu_s)^{\frac{1}{2}}\big)$, and we have thus proved the following theorem:

Theorem 5.3.10. *Algorithm* PollyDivide *solves the* Multiple Submass All Witnesses Problem *in time* $\mathcal{O}((Kn\mu_s\log\mu_s)^{\frac{1}{2}})$, *where* K *is the total number of witnesses, i.e., the output size.*

To better understand this result, let $\bar\kappa$ denote the average size of the output, i.e., $\bar\kappa = K/r$. Then the runtime is $(r\bar\kappa n\mu_s\log\mu_s)^{1/2}$. Recall that the running time of the combination of the naïve algorithms for the Multiple Submass All Witnesses Problem is $O(\min(rn, n^2\log n))$. Thus, our algorithm beats the running time of the naïve algorithms above if $\bar\kappa\mu_s\log\mu_s = o(rn)$ and $\bar\kappa r\mu_s\log\mu_s = o(n^3\log^2 n)$.

A variation for one witness per query

Algorithm PollyDivide can be straightforwardly adapted to only produce one witness per query mass, i.e., to solve the Multiple Submass One Witness Problem. Then its runtime becomes $\mathcal{O}((rn\mu_s\log\mu_s)^{\frac{1}{2}})$, i.e., somewhere between $\mathcal{O}(\mu_s\log\mu_s)$ and $\mathcal{O}(rn)$ (since rn needs to be the larger factor if we want to employ the algorithm). If, say, $r = \mathcal{O}(\mu_s)$, then we end up with a runtime of $\mathcal{O}(\mu_s\sqrt{n})$. Comparing this to the $\mathcal{O}(\mu_s \text{ polylog } \mu_s)$ runtime of PollyLasVegas leaves us with an extra $\sqrt{n}$ factor which we pay for the deterministic version.

6 De Novo Peptide Sequencing with Mass Spectrometry

In this chapter, we present the first prototype of the software AuDeNS for novo peptide sequencing. Recall the problem of de novo peptide sequencing: Given an MS/MS spectrum, find amino acid sequences which match the spectrum. (For more detailed description, see Section 2.2.) AuDeNS first preprocesses the input spectrum with a number of heuristics we refer to as "grass mowers." These result in a weighting of the input peaks. AuDeNS then employs a variant of a dynamic programming algorithm introduced in [CKT+01] to compute amino acid sequences matching the spectrum. Hereby, it uses the weighting obtained in the preprocessing step to score these solutions; only solutions within a user-specified cutoff of the optimal solution are computed.

We show the results of first simulations, which suggest that AuDeNS performs well (but not better) in comparison with Sequest [EMI94, seq], a frequently used software for peptide identification with database–lookup, and Lutefisk [TJ97, TJ01, lut], a de novo peptide identification tool.

The contents of this chapter have been published in [BCG+02], and parts are described in [Cie03]. The software has since been further developed, without the collaboration of this thesis' author.

6.1 Problem definition

Given a peptide string $p = s_1 \dots s_n$, its *dissociation pattern* is the set

$$D_p = \{m^p_{\text{parent}}, m^b_1, \dots, m^b_{n-1}, m^y_1, \dots, m^y_{n-1}\}, \qquad \text{where} \tag{6.1}$$

$$\begin{aligned}
m^p_{\text{parent}} &:= \sum_{i=1}^{n} \mu(s_i) + \text{offset}_{\text{parent}} && \text{parent ion (the entire peptide)} \\
m^b_r &:= \sum_{i=1}^{r} \mu(s_i) + \text{offset}_b, \quad 0 < r < n && \text{b–ions (its prefixes)} \\
m^y_r &:= \sum_{i=r}^{n} \mu(s_i) + \text{offset}_y, \quad 0 < r < n && \text{y–ions (its suffixes).}
\end{aligned}$$

Hereby, $\text{offset}_{\text{parent}}, \text{offset}_b, \text{offset}_y$ are positive real numbers. (Usually, $\text{offset}_{\text{parent}}$ and offset_y equal the mass of an OH_2 group plus that of a neutron, and offset_b equals the mass of a single neutron.)

A *tandem mass spectrum* S contains the *parent mass* m^S_{parent}, followed by a list of pairs $(m(i), a(i))$, $i = 1, \dots, N$, where the $m(i)$ are molecular masses, and $a(i)$ is the

abundance of $m(i)$. The entries are ordered w.r.t. their m–values. Typical values for N range from 35 to 900. A pair $(m(i), a(i))$ is often referred to as a *peak*, which derives from the customary visualization of mass spectra, see Figure 6.1. In the following, we call peaks that derive from ions of the original peptide *peptide peaks*, and the others *noise* or *grass*.[1] In addition, we are given a *mass tolerance* ϵ, the measurement error of the mass spectrometer.

A *solution* to a given spectrum S is a peptide p such that $|m^p_{\text{parent}} - m^S_{\text{parent}}| \leq \epsilon$. In addition, we would like to match the masses of D_p with the peptide peaks of the spectrum, i.e., find pairs $(m, m(i))$, $m \in D_p$ and peptide peak i of S, for which $|m - m(i)| \leq \epsilon$ holds. However, since it is not clear from the outset which peaks of S are peptide peaks, we allow that peaks not be matched, and in the extreme, that no peaks match with D_p. Thus, since the only necessary condition for a solution is that the parent masses match within the given mass tolerance, the solution cannot be unique. In particular, given one solution of length n, all of its permutations will match, typically a number exponential in n. Another reason for non–uniqueness of solutions is that the two amino acids isoleucine (I) and leucine (L) have exactly the same molecular mass. Increasing the mass tolerance causes further pairs of amino acids to become indistinguishable. Missing peptide peaks in the spectrum account further for non–uniqueness of the solution, e.g., an Asparagine (N) and two consecutive Glycines (G) cannot be distinguished if the peak corresponding to the first Glycine is missing, since $\mu(N) = 2 \cdot \mu(G)$.

The aim, therefore, is to output a ranked list of solutions such that the peptide that gave rise to the spectrum has high ranking. A *multi–sequence* is a finite set of sequences that we write as a regular expression, e.g., V(N|GG)GYSE(I|L)ER is short for the set {VNGYSEIER, VNGYSELER, VGGGYSEIER, VGGGYSELER}. Rather than listing feasible solutions individually, AuDeNS outputs a ranked list of multi–sequences.

6.2 AuDeNS: A tool for automated de novo peptide sequencing

In 2000, Chen *et al.* [CKT+01] introduced a de novo peptide sequencing algorithm that uses dynamic programming. The algorithm has two variants, but only the variant for noisy data is applicable to real-life applications. Chen *et al.* proved that the algorithm for noisy data has running time at most cubic in the number of peaks of the given spectrum but did not provide an implementation of their algorithms. Naïve use of the noisy variant is computationally too complex, since the number of potential solutions is too large. In addition, measurement errors need to be taken into account.

We have developed heuristics (which we refer to as "grass mowers") for assigning relevance values to the input peaks, and implemented a framework, AuDeNS, that

[1] Because of their appearance in the visualization, groups of small peaks are sometimes referred to as *grass*. Since much of this part of the input is not well interpretable, some of the data preprocessing is concerned with getting rid of this grass. This is the reason we call our data cleaning algorithms *grass mowers*.

first uses the heuristics to preprocess the spectrum, and then employs a modification of the noisy sequencing algorithm of [CKT+01] that can handle measurement errors. We assign relevances to the solutions and only enumerate those within a user–specified threshold relative to the maximal relevance value. The output of AuDeNS is a ranked list of "multi–sequences" (sequences that take inherent ambiguities of the input into account).

AuDeNS works in the following way: In a first step, it applies the mowers to the input data, assigning to each input peak i a relevance value $r(i)$, with the default being $r(i) = 1$. Hereby, each mower M uses a relevance factor Rel_M (which can be set as a parameter of AuDeNS), and the relevance value of peak i is then given by

$$r(i) := 1 + \sum_{M \text{ mower}} \mathrm{Rel}_M \cdot M(i), \tag{6.2}$$

where $M(i)$ is the value assigned to peak i by mower M. The relevance of a solution is then the sum of the relevances of the peaks matched by this solution. All mowers output values between 0 and 1, and thus, their output can be weighted against each other by the relevance factors specified by the user. In addition, the mowers each have parameters that can be specified (see Section 6.2.1 for details). It is an important aspect of AuDeNS that new mowers can be integrated with minimal effort.

In a second step, AuDeNS applies the sequencing algorithm. Hereby, the minimal quality of the solutions can be specified as a relative value as measured in comparison to the relevance r_{max} of a best solution, i.e., a $0 \leq \delta \leq 1$ such that all solutions with relevance greater or equal to the threshold $r_\delta = (1 - \delta) \cdot r_{max}$ are to be computed. First, a table is constructed and r_{max} is computed. Then, all solutions with relevance greater or equal r_δ are computed and output, using backtracking in the table.

Global parameters such as mass tolerance and relevance factor of the mowers allow for a fine–tuning of AuDeNS.

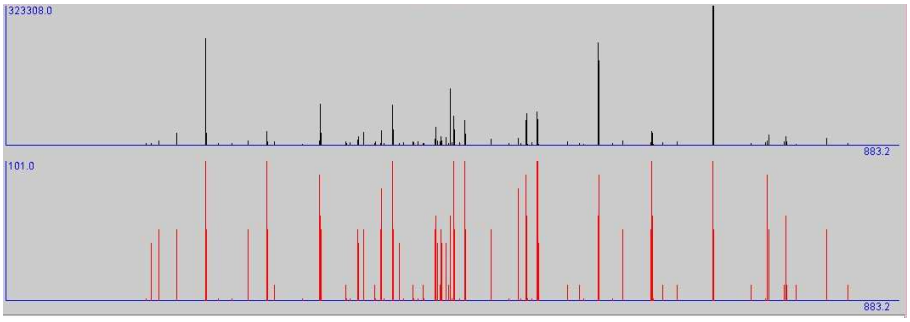


Figure 6.1: A tandem mass spectrum with corresponding relevance values. The x–axis represents the masses. The upper graph shows the abundance values of the masses on the y–axis, and the lower graph their relevance values.

6.2.1 The mowers

1. **Threshold Mower:**

   Peaks with very small abundance values, are unlikely to be peptide peaks. The threshold mower marks all peaks with an abundance value above a given (low) threshold.

2. **Window mower:**

   The window mower has two parameters: the size of a window W and a number k of peaks per window. It moves along the input, and, for each peak i, marks the k peaks with highest abundance within the window starting at $m(i)$, i.e., those k peaks with highest abundance in the set $\{j \mid m(i) \leq m(j) \leq m(i) + W\}$. The mower assigns each peak i a value proportional to the number of times it has been marked.

   Roughly speaking, high peaks are more likely to be caused by peptide ions than low peaks. The rationale for the window mower then is twofold: First, within any window of the approximate size of the smallest amino acid mass, there can be at most two peptide peaks, namely one b–ion and one y–ion.

   Second, when sequencing manually, contiguous regions of m/z values can be found such that the abundance of the peaks within *one* region do not differ very much, while they do differ between *different* regions. Regions are then scaled with different factors in order to level the height of the peaks, and then peaks which are high within their region are considered for the sequencing process [SJ01].

   The reasons are inherent technical characteristics of an ion trap that result in differential efficiencies of mass measurements over the entire mass range. During an MS/MS cycle, a selected peptide is excited by resonance excitation to accomplish collision induced dissociation. However, resonance excitation and resonance ejection are virtually identical, resulting in the possible loss or inefficient measurement of product ions during resonance excitation. As a consequence, low molecular mass ions are often underrepresented in an MS/MS spectrum (e.g., product ions with less than 30% of the parent mass are only rarely observed [Arn01]). Another reason for “regional differences” in mass measurement efficiencies are the inherent biochemical characteristics of amino acids, either resulting in efficient (e.g. Proline) or inefficient (e.g. Glycine) dissociation of a peptide bond [SJ01].

   Therefore, considering only the absolute abundance of the peaks is not sufficient to identify peptide peaks, but a method that takes the differences between regions into account is more appropriate, such as employed by the window mower.

3. **Isotope mower:**

   Typically, single ions give rise to more than one peak in an MS/MS spectrum due to isotopes. Isotopes differ in the number of neutrons they have in the nucleus, and they occur in nature with different probabilities (e.g., carbon has either 6 neutrons, with probability 98.892%, or 7 neutrons, with probability

1.108%). Thus, peaks without corresponding isotope peaks are rather unlikely to be caused by ions, and the number of isotope peaks of a single peak can be used to adjust the relevance of a peak.

The isotope mower has a parameter k, the number of isotopes required. It assigns a value to each peak i proportional to the number of isotopes present in the spectrum, i.e., for each $j, 1 \leq j \leq k$, it checks whether there is a peak with mass m such that $|m - (m(i) + j)| \leq \epsilon$.

4. **Intersection mower:**

   The intersection mower considers all spectra obtained from the same experiment as the current spectrum S that have the same parent mass $m \in [m^S_{\text{parent}} - \epsilon, m^S_{\text{parent}} + \epsilon]$. It then assigns each peak i in S a value proportional to the number of other spectra in which it is also contained.

   The rationale is the consideration that other spectra with the same parent mass that stem from the same experimental setup are likely to have been derived from the same peptide. Even though in theory, many different peptides will have the same parent mass (e.g., all permutations of the same amino acids), in reality it is not very likely that different peptides stemming from the same protein or from the same small number of proteins have the same mass. (Some preliminary database analysis has shown this probability to be only between 2 and 7%.)

5. **Complement mower:**

   If i is a peak in the spectrum which arose from a b–ion, then we expect the corresponding y–ion to be present in the spectrum, and vice versa. Therefore, for any peak i in the spectrum, we increase the relevance of i if the complement peak i' with $m(i') = m_{\text{parent}} - \text{offset}_{\text{parent}} + \text{offset}_b + \text{offset}_y - m(i)$ (within ϵ) is present. This mower is very closely related to the sequencing algorithm we use, since the algorithm heavily relies on pairs of complement peaks.

6.2.2 The sequencing algorithm

Our sequencing algorithm is based on the dynamic programming algorithm for noisy data introduced in [CKT+01]. The differences are very small: The algorithm in [CKT+01] maximizes the sum of weights of peak pairs (edges), while our algorithm maximizes the sum of the relevance values assigned to the peaks. We refer to this algorithm as WEIGHTED-CKTRC-ALGORITHM and present it briefly here.

The idea of the WEIGHTED-CKTRC-ALGORITHM is to generate a directed vertex–labelled graph $G = (V, E)$ with two special vertices x_0 and y_0, such that any directed path from x_0 to y_0 satisfying an additional constraint will correspond to a solution. For each peak i, there are two vertices $x_i, y_i \in V$, whose masses are the smaller and the larger value, respectively, of the mass of peak i and its complement w.r.t. the parent mass. The relevance $r(v)$ of a vertex v is the relevance of the correspondig peak assigned by the mowers. The reason for generating pairs (x_i, y_i) of vertices is that if a peak is a peptide peak, then it is either a prefix (a b–ion) or a suffix (a

y–ion)—and if the spectrum were perfect, then it would also contain its complement (see Section 6.2.1).

If two vertices have the same mass within the mass tolerance ϵ, then we merge them, and assign the new vertex the maximal relevance value among the merged vertices. The vertices are sorted such that $m(x_0) < m(x_1) < \ldots < m(x_N) < m(y_N) < \ldots < m(y_1) < m(y_0)$. (Because of the merging of vertices, the new value of n may have decreased, but we ignore this detail here.) Hereby, x_0 and y_0 are two new vertices with masses $m(x_0) = \text{offset}_b$ and $m(y_0) = m_{\text{parent}} - \text{offset}_{\text{parent}} + \text{offset}_b$, and both relevance 1. At this point, for each pair $(x_i, y_i), i = 1, \ldots, N$, we know that it either constitutes noise, or one is a prefix of the peptide and the other a suffix—but we do not know which is which.

G contains a directed edge (u, v) if $m(v) - m(u)$ can be written as the sum of some amino acid masses within the mass tolerance (see Section 6.2.3 for details). Call a directed path P in G k–*compatible* if P contains at most one vertex of each pair $(x_i, y_i), i = 1, \ldots, k$. Any N–compatible directed path P in G from x_0 to y_0 corresponds to a solution to the input, because it represents a partial list of prefixes.

We will now fill in a table Q of size $(N + 1) \times (N + 1)$ that will be used to compute paths from x_0 to y_0. Define $w(P)$, the *pathweight* of the directed path P in G, as $w(P) := \sum_{v \in P} r(v)$. Set

$$\begin{aligned} Q(i,j) := \max\{w(L) + w(R) \mid\ & L \text{ directed path from } x_0 \text{ to } x_i, \\ & R \text{ directed path from } y_j \text{ to } y_0, \\ & \text{and } L \cup R \text{ is } \max(i,j)\text{–compatible.}\} \end{aligned}$$

This definition implies $Q(i, j) = 0$ if no such paths exist, since $\max \emptyset = 0$. The table Q has the property that $Q(i, j) > 0$ if and only if there is a path L from x_0 to x_i and a path R from y_j to y_0 such that $L \cup R$ is $\max(i, j)$–compatible. It can be filled in using the crucial observation that the maximum path for a given pair x_i, y_j, $i < j$, can be computed using all maximal paths for pairs x_i, y_k, for $k < j$. Since $j > i$, y_j can be added to any such pair $L \cup R$ without violating the compatibility condition. The situation is analogous for the case where $i > j$. Thus, $Q(i, j)$ can be computed as follows:

$$Q(i,j) = \begin{cases} \max_{0 \le k < j}\{Q(i,k) \mid (y_j, y_k) \in E \text{ and } (i = k = 0 \text{ or } Q(i,k) > 0)\} + r(y_j) & \text{if } i < j \\ 0 & \text{if } i = j \\ \max_{0 \le k < i}\{Q(k,j) \mid (x_k, x_i) \in E \text{ and } (k = j = 0 \text{ or } Q(k,j) > 0)\} + r(x_i) & \text{if } i > j \end{cases}$$

The value of a maximal path is now $r_{\max} = \max\{Q(i, j) \mid (x_i, y_j) \in E\}$. Note that $r_{\max} = 0$ means that there is no feasible solution to the input, i.e., the parent mass cannot be written as a sum of amino acid masses within the given error tolerance ϵ. Now all paths within the given threshold are enumerated recursively via backtracking in the table.

6.2.3 Details of efficient implementation

1. **Enumeration of the multi–sequences:**

Entry $Q(i,j)$ contains the maximum weight of any path from x_0 to x_i and from y_j to y_0. Thus, the table Q can be used in a backtracking algorithm to recursively enumerate all paths from x_0 to y_0 whose weights are above a given threshold. The use of a threshold allows for pruning the tree of computation generated by the backtracking process in early stages. This makes the time spent in the recursion proportional, not to the total number of possible paths, but to the number of paths that are of interest (whose weights are above the threshold).

2. **Deciding whether a mass is a sum of amino acids**

   To decide whether a given mass can be represented by a sum of masses of certain amino acids and to list all such amino acid sequences, we work with an array of Boolean variables $b_0, \ldots, b_M$. Variable b_i represents masses $m \in [i\Delta m, (i+1)\Delta m)$. Let $m_i = i\Delta m + \Delta m/2$ be the center mass of the interval represented by b_i. The maximal index M depends on the maximal mass M_{max} considered and is computed as $M = \lceil M_{max}/\Delta m \rceil$. We use $\Delta m = 0.01$ Da and $M_{max} = 1000$ Da.

   The variables b_i are initialized as follows: If the mass interval represented by b_i contains the mass of any single amino acid, b_i is set true, otherwise b_i is set to false. This can be done in $20 + M = O(M)$ time. In a second phase, we run from b_0 to b_M and set b_i true, if there is an amino acid mass a such that the variable b_j containing $m_i - a$ is true. The second pass takes $20M = O(M)$ time steps since there are 20 amino acids.

   To answer the question whether a mass sum m measured with error ϵ can be represented by a sum of masses of certain amino acids, we check all variables b_i that represent part of the interval $(m - \epsilon, m + \epsilon)$. If one of them is true, the answer is yes, if all are false, then the answer is no.

3. **Enumeration of subsequences:**

   To enumerate all amino acid sequences for a mass sum m and an error ϵ, we proceed as follows: For all true b_i's that represent part of the interval $(m - \epsilon, m + \epsilon)$, and for all amino acid masses a, we test whether the variable b_j containing $m_i - a$ is true. If so, we store the letter of amino acid a and recursively enumerate all sequences for mass m_j. This algorithm, however, enumerates all permutations of all possible sequences. To avoid this, in recursion depth d we only consider amino acids whose letters are lexicographically larger or equal to the animo acid letter chosen in depth $d - 1$. This way, only distinct sequences with respect to permutation are output.

6.3 First experimental results

The running time of AuDeNS depends on the number of sequences computed. To create the best 30 sequences, AuDeNS takes less than one second to read, mow, and sequence a spectrum, on a PC with 700 MHz and 256 MB RAM.

mower	relevance	other parameters
threshold	40	threshold 8000
window	10	number of peaks 2, window 50
isotope	10	number of isotopes 1
complement	40	
intersection	0	

Table 6.1: Parameter settings for AuDeNS in our experiments

We compared the output of AuDeNS to the results of Sequest and Lutefisk. Lutefisk needs 2 seconds up to several minutes on the same computer and same spectrum to output the best 0 to 5 solutions. However, Lutefisk outputs individual peptide sequences, as opposed to multi–sequences of AuDeNS. Even without systematically tuned parameters of the mowers, the best sequence found by Sequest for a spectrum is very often among the first 30 sequences created by AuDeNS. Otherwise, there are many almost correct sequences among the output. Three selected example outputs are shown in Figures 6.2 to 6.4. The mower parameters had been set as shown in Table 6.1. The global parameter ϵ was set to 0.5 Da.

740.0 V(N|GG)GYSE(I|L)E(R|GV)
735.0 V(N|GG)GY(I|L)C(I|L)E(R|GV)
716.0 V(N|GG)GYAGS(I|L)E(R|GV)
715.0 V(N|GG)GYES(I|L)E(R|GV)
715.0 V(N|GG)GYDT(I|L)E(R|GV)
715.0 V(N|GG)GYTPME(R|GV)
711.0 V(N|GG)GYSGA(N|GG)E(R|GV)
705.0 V(N|GG)GYTD(I|L)E(R|GV)

Figure 6.2: Sequest sequence VNGYSEIER has the highest rating in the AuDeNS output.

655.0 (AG|Q|K)A(I|L|N|GG)AAA(I|L)(N|GG)(AG|Q|K)
655.0 (AG|Q|K)(N|GG)AAAA(I|L)(N|GG)(AG|Q|K)
644.0 (AE|IS|LS|TV|CP)(I|L)AAA(I|L)(N|GG)(AG|Q|K)
615.0 (AG|Q|K)A(I|L|N|GG)AAA(I|L|N|GG)(I|L)(AG|Q|K)
615.0 (AG|Q|K)(N|GG)AAAA(I|L|N|GG)(I|L)(AG|Q|K)
605.0 (AG|Q|K)PSAAA(I|L)(N|GG)(AG|Q|K)
605.0 (AG|Q|K)(N|GG|D)AAAA(I|L)(N|GG)(AG|Q|K)

Figure 6.3: Sequest sequence AEIAAALNK is at third position in the AuDeNS output.

Even though our tool does not perform as well as Lutefisk at the moment, we

496.0 (AG|Q|K)AE(N|GG)(AG|Q|K)SGFFE
495.0 (AG|Q|K)AE(N|GG)AAEFFE
495.0 (AG|Q|K)AE(N|GG)(AG|Q|K)GSFFE
486.0 (AG|Q|K)AE(I|L)GS(N|GG)YFE
486.0 (AG|Q|K)AE(AG|Q|K)(N|GG)SGFFE
485.0 (AG|Q|K)AE(I|L)GS(N|GG)YFE
485.0 (AG|Q|K)AE(N|GG)AA(N|GG)YFE
485.0 (AG|Q|K)AE(AG|Q|K)(AG|Q|K)EFFE
485.0 (AG|Q|K)AE(AG|Q|K)(N|GG)GSFFE
485.0 (AG|Q|K)AE(I|L)E(AG|Q|K)YFE
485.0 (AG|Q|K)AE(N|GG)(AG|Q|K)(AG|Q|K)YFE
485.0 (AAG|AQ|AK)E(N|GG)(AG|Q|K)SGFFE
484.0 (AAG|AQ|AK)E(N|GG)AAEFFE
484.0 (AAG|AQ|AK)E(N|GG)(AG|Q|K)GSFFE
481.0 (AG|Q|K)AE(I|L)ESGFFE
480.0 (AG|Q|K)AE(I|L)EGSFFE
475.0 (AAG|AQ|AK)E(I|L)GS(N|GG)YFE
475.0 (AAG|AQ|AK)E(AG|Q|K)(N|GG)SGFFE
474.0 (AAG|AQ|AK)E(I|L)GS(N|GG)YFE
474.0 (AAG|AQ|AK)E(N|GG)AA(N|GG)YFE
474.0 (AAG|AQ|AK)E(AG|Q|K)(AG|Q|K)EFFE
474.0 (AAG|AQ|AK)E(AG|Q|K)(N|GG)GSFFE
474.0 (AAG|AQ|AK)E(I|L)E(AG|Q|K)YFE
474.0 (AAG|AQ|AK)E(N|GG)(AG|Q|K)(AG|Q|K)YFE
471.0 (AG|Q|K)AE(AG|Q|K)CST(I|L)FE
470.0 (AG|Q|K)AE(I|L)E(AG|Q|K)YFE
470.0 (AAG|AQ|AK)E(I|L)ESGFFE
469.0 (AAG|AQ|AK)E(I|L)EGSFFE
466.0 (AG|Q|K)AE(N|GG)(AG|Q|K)SGFFE
465.0 (AG|Q|K)AE(N|GG)AAEFFE

Figure 6.4: Sequest sequence AKELQEYFK does not appear within the first 30 sequences of AuDeNS but many similar sequences do, e.g., (AAG|AQ|AK)E(I|L)E(AG|Q|K)YFE.

believe that it can be developed to match or even outperform Lutefisk for a number of reasons:

1. In our experiments, AuDeNS has much lower running times than either Lutefisk or Sequest, due to a fast algorithm and efficient implementation.

2. AuDeNS is a framework that is capable of having new mowers added to it with minimal effort. The mowers we employ at the moment are heuristics that are plausible but need further fine–tuning, in particular with regard to the parameters.

3. Even without having systematically tuned the parameters of AuDeNS, our out-

put compares relatively well with that of Lutefisk and Sequest.

We conclude from our first experiments that AuDeNS constitutes a promising approach to de novo peptide sequencing. Clearly, more work needs to be done in refining the mowers and adding new ones; in fine-tuning the parameters, possibly using machine learning algorithms with a training set of mass spectra where the correct sequence is known; and others.

Part II

String Dissimilarity Measures and EST Clustering

7 Background II: Expressed Sequence Tags

In this chapter, we introduce expressed sequence tags (ESTs), discuss their production process and typical properties, and introduce the problem of EST clustering.

7.1 Why ESTs and EST clustering?

The collection of all mRNAs (messenger RNAs) present in the cell is referred to as the *transcriptome*. The capture of (mature) mRNA that is on its way to the ribosome constitutes a method of measuring *gene expression*, i.e., of detecting which genes are expressed in the cell. This can be done, for instance, with regard to the type of cell (type of tissue), developmental stage, or healthy versus diseased tissue. In addition, mRNA capture has been used for *gene discovery*, where a gene had not previously been identified via its protein product or via other gene detection methods. More recently, mRNA capture has been increasingly used to detect events of *alternative splicing* (see Section 7.3).

Since capture of full-length mRNAs is technically very challenging, information about the transcriptome is most often collected using *expressed sequence tags* (ESTs). ESTs are short cDNA (complementary DNA) transcripts of mRNAs, which are produced in a high-throughput manner. They are available in large numbers in public databases, and thus constitute an easily available source of information. *EST Clustering* is the problem of partitioning (clustering) a set of ESTs into subsets where each subset (cluster) corresponds to a gene, i.e., two ESTs are members of the same cluster if and only if they have been derived from mRNAs that are transcripts of the same gene.

Since the completion of the sequencing of the human genome, it has become apparent that the number of genes is far lower than the estimated number of expression products (proteins and RNA products). EST clustering can be used for identifying products of alternative splicing, as well as for gene discovery and for measuring gene expression.

7.2 What are ESTs?

An EST (*expressed sequence tag*) is a short cDNA that is manufactured in order to gather information about mRNAs present in the cell (see Figure 7.1). First, all mature mRNAs are extracted from the cell. Then, a reverse complement DNA copy (single-stranded cDNA) is made of the mRNA by reverse transcription, and the mRNA is digested (destroyed and thus separated from the cDNA strand). Using

DNA polymerase, a complementary cDNA-strand is synthesized, and the double-stranded cDNA is then inserted into a vector, typically a plasmid with which the *E. coli* bacterium is infected. The bacterium is then allowed to reproduce, thus creating many copies (*clones*) of the original cDNA. Finally, the DNA is extracted from the bacterium and the clone insert sequenced. Hereby, knowledge of the vector DNA is employed; primers can be created adjacent to each end of the insert, and thus the insert is sequenced once from the 5'-end and once from the 3'-end. This yields a pair of ESTs that are reads of the same cDNA, and which may or may not overlap. Typically, ESTs are between 300 and 500 bp (base pairs) long.

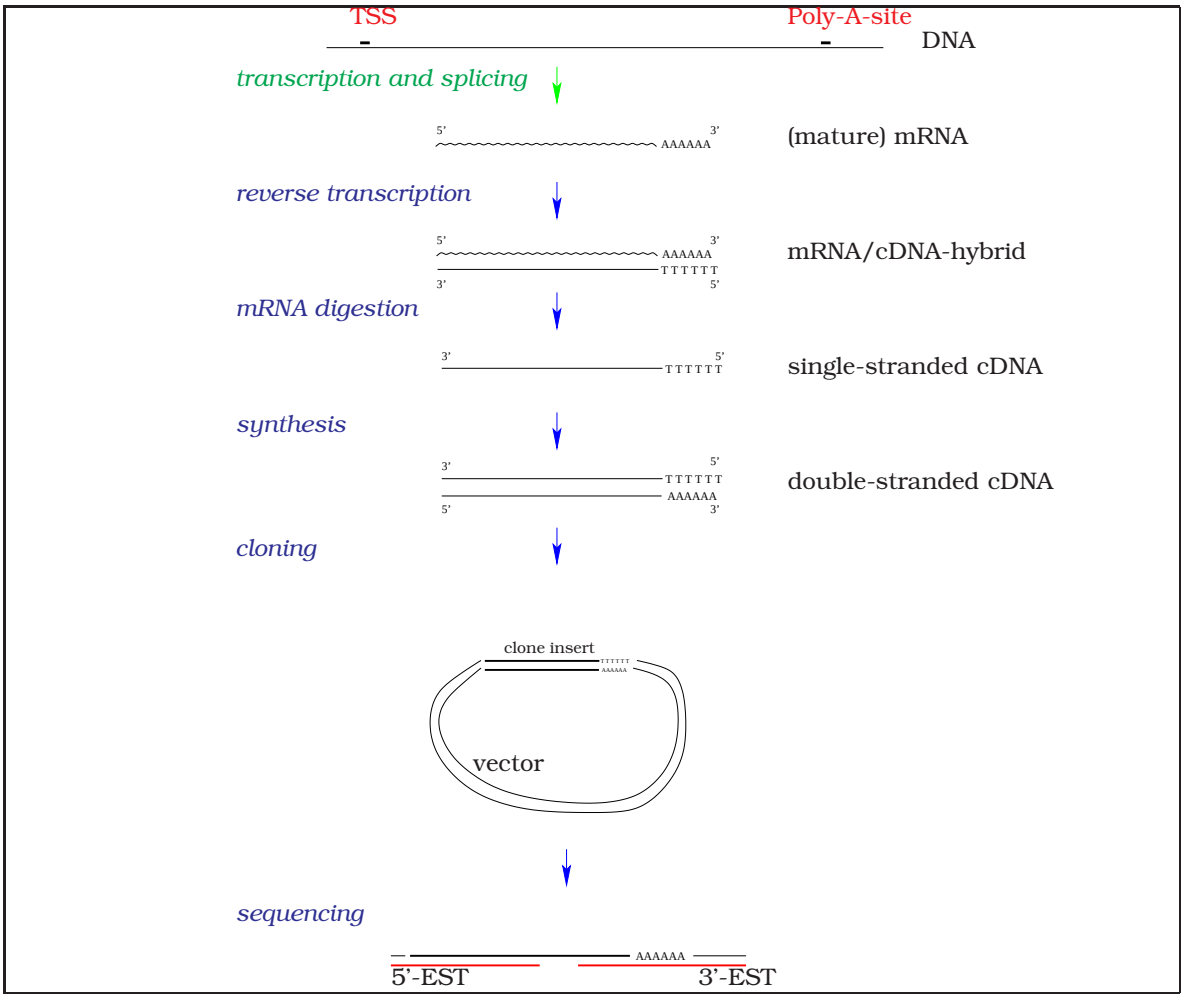


Figure 7.1: EST manufacture (partially adapted from [Chr01], p.6)

7.3 Properties of ESTs

ESTs are manufactured in a high-throughput manner, and they are more error-prone than DNA reads that are used for genome sequencing. Errors occur at all stages of EST production:

- Extraction of total mRNA of the cell: In this step, other sequences may also be captured, such as
  - rRNAs (ribosomal RNA)
  - pre-mRNAs (before splicing, thus including introns)
  - mitochondral DNA (approx. 1-2 %)
  - human genomic DNA (from handling the experimental setup)
- Reverse transcription:
  - reverse transcriptase works with a 1:3000 error rate
  - cDNA length: cDNA can fall off the template mRNA too soon due to exhaustion of reverse transcriptase; since up to around 1999, reverse transcription was always made starting at the 3'-end of the mRNA, this produced many more 3'-end reads than 5'-end reads.
- Cloning:
  - the linker used for insertion of the cDNA into the vector may be included in the read
  - contamination with vector DNA
  - chimeras caused by recombination in vector
- Sequencing and sequencing software:
  - lane-tracking errors (in particular before the introduction of the capillary gel method)
  - robotics errors (spillage)
  - base-calling errors: the first approx. 30 bases are bad quality
  - where no base can be identified, an N is inserted; however, due to mistakes in the base-calling software, an N may represent two bases or even no base at all.
  - incorrect bases or indels.
- Bad annotation: ESTs are often poorly and unreliably annotated in the public databases, e.g. the tissue specified is incorrect, or sometimes even the organism is the incorrect one. In particular, clone-linking information, i.e., the information of which two ESTs form a pair (reads of opposite ends of the cDNA of the same clone) is often not reliable.

EST errors

Due to the processes detailed above, typical problems when trying to cluster ESTs include:

- *Contamination*: Contaminants can include vector, rRNA (ribosomal RNA), mitRNA (mitochondrial RNA), genomic sequence; and possibly sequence from other species.

- *Single base errors:* First, single bases may be read incorrectly using Sanger Sequencing; or, single base errors may be due to the reverse transcription. Second, polymerase decay may cause an increase in the rate of errors as the EST is read (there is gentle decay for the bulk of the EST followed by a very rapid decay at the end). Finally, interference from the primer makes the beginning parts of reads particularly unstable.

- *Stuttering:* Stuttering is caused by a problem in the sequencing process: the sequencer slips, and a portion of the EST is re-read. Stuttering can occur anywhere, but is most likely to occur after repeated Gs or Ts.

- *Ligation:* Ligation occurs when two ESTs bond together, giving the appearance of a new EST. The two ESTs that join together need not come from adjacent parts of an mRNA; indeed, they ususally come from different mRNAs.

Alternative splicing and other problems

Not all mature mRNA transcripts from the same gene are identical: This phenomenon is known as *alternative splicing*. Forms of alternative splicing include:

- *exon skipping:* in addition to the introns, some of the exons are also spliced out;

- *alternative transcription start site:* transcription begins at a different position;

- *alternative polyadenylation site:* transcription ends at a different position;

- *alternative splice sites:* the position of the exon/intron boundary is moved (exon truncation/exon elongation).

Usually, one transcript is seen as the "canonical" transcript, while the other forms are referred to as alternative transcripts. The transcript viewed as canonical is the one that includes all exons, uses specific TSS (transcription start site) and poly-A (polyadenylation) sites, and is spliced at specific splice sites. This has some justification in that TSS, poly-A site, and splice sites can be detected with some certainty independently of the proof of transcripts, but especially regarding alternative splice sites, the singling out of one particular product seems problematic. Disregarding alternative start, poly-A, and splice sites, for a gene with k exons, there are, in theory, 2^k-1 different forms: namely all possible choices of exons except the empty set. However, research seems to concentrate on detecting exon skipping events where only one exon has been skipped, which yields only k different forms.

Another problem is posed by the occurrence of *paralogous genes*: These are pairs of genes in one organism that are the product of a duplication event in the species' history. Since they are very similar, an EST that is derived from one gene is likely to also fit to its paralog. Thus, if EST clustering is done on the basis of sequence similarity, ESTs of paralogous genes are likely to be clustered together.

Finally, *repeats* and so-called *low-complexity regions* pose a problem in clustering, as they do in sequence assembly. Repeats are substrings that occur more than once

within the genome; they can be around 10 bp to several thousand bp long. Low-complexity regions are strings that have little variation in their base content, e.g. long stretches of CG pairs.

7.4 EST clustering

EST clustering is done both in individual EST projects (e.g. gene expression projects), as well as when researchers try to gather information based on the large number of ESTs publicly available in EST databases. In either case, one is faced with a large set (up to million or more) of ESTs, which needs to be interpreted. This leads to the problem of *EST Clustering*, which has the goal of producing a partition of the original set such that each cluster corresponds to a gene, i.e., two ESTs are members of the same cluster if and only if they have been derived from mRNAs that are transcripts of the same gene. See also [HMP+99] for more background on ESTs and EST clustering.

All EST clustering tools include some preprocessing of the data. Usually, certain substrings are masked out, such as known repeats (from repeat databases), vector DNA (from databases of the vector used in the experiment). Using the example of two of the most commonly used EST clustering packages, UniGene [BS95] and StackPack [CvGG+01], we demonstrate some other heuristic techniques employed.

UniGene employs a strategy referred to as *clustering in stages*. The information that is available for two ESTs to belong to one cluster is weighted according to its source (so-called *links*): mRNA-EST-links, EST-EST-links, and clone links. Then clusters are formed, where only those strings are grouped together that have strong links, i.e., links with high weights.

StackPack employs a number of postprocessing steps to the clustered data: Using different software tools, a consensus is generated for each cluster, then the consensus is evaluated, several consensi generated (if products of alternative splicing can be detected), some clusters are split (if they represent paralogous genes), and other clusters are merged using clone-pair information.

In the next chapter, we will concentrate on the clustering method, i.e., the step after the preprocessing and before the postprocessing.

8 EST Clustering

In this chapter, we give an overview of some literature relevant to EST clustering and to the methodology for EST clustering components which we propose in Chapter 9. This is followed by an overview of different string dissimilarity measures, clustering algorithms, and clustering validity indices used for EST clustering. The contents of this chapter have been published in [ZLH04].

8.1 Literature on and software for EST clustering

Several software packages exist that perform EST clustering; among the most widely used are UniGene [BS95], TIGR [QCL+01], and StackPack [CvGG+01], for all of which a database of EST clusters exists. The clustering packages usually consist of several steps that include contamination and repeat masking, clustering, and consensus computation. Here, we focus on the *clustering step*. For the clustering, different algorithms are used, both with respect to the *clustering algorithms*, and to the criteria (*dissimilarity measures*[1]) according to which sequences are clustered together. In the following, we shall refer to the string dissimilarity measure plus the clustering algorithm together as the *clustering method*. We will give exact definitions in the following sections.

In spite of the variety of EST clustering methods, to the best of our knowledge, no rigorous technique has been put forward for evaluating which one of the existing or newly suggested ones is best suited. Instead, in most studies, it is not the underlying clustering algorithms and sequence dissimilarity measures that are being evaluated but the output: The output is evaluated either according to expert knowledge, or according to how well it corresponds to the output of some software that has proved to be "good" in the past. While we believe that the former is a valid and, in fact, invaluable method of evaluation, it is in most cases not feasible. The latter, on the other hand, hinders innovation for intrinsic reasons. Moreover, since ESTs are produced according to different processes, they can have different properties (such as the distribution of errors of different types), and thus, different algorithms may be more or less appropriate depending on the type of input data.

Most literature on *EST clustering* proposes a particular EST clustering algorithm, and validates the clustering computed by comparing it to similar tools, or by expert examination of the outcome. This is the case, for example, for literature on well known clustering tools such as UniGene [BS95], TIGR [QCL+01], and StackPack [CvGG+01]. In 1999, Hartuv *et al.* [HSL+99] proposed an EST clustering algorithm that employs a string dissimilarity measure and clustering algorithm which

[1] Recall that we refer both to similarity and dissimilarity/distance measures simply as *dissimilarity measures*.

both differ from the ones commonly used. The validation of the clusterings is made both with simulated and with real input data: For simulated data, the result is compared to the known ideal clustering, while for real data, it is compared to a very carefully generated clustering, which is judged correct by the authors. The authors do not claim that their algorithm produces the best clustering, but that it allows for a large speedup as a preprocessing step to a more careful but more costly EST clustering method. Recently, two new EST clustering algorithms using suffix tree-type data structures were suggested: In [MCJ03], Malde *et al.* use suffix arrays, and the clusterings computed are compared to the output of other clustering tools. Kalyanaraman *et al.* [KAKB03], validate their clusterings by comparing them to clusterings computed by aligning the ESTs to the genome. Burke *et al.* [BDH99] compare the results of d^2-cluster, the clustering method used by StackPack, to results produced by UniGene. It is shown that d^2-cluster is more sensitive, and this claim is supported by *(i)* examining the resulting clusters using biological expertise, and *(ii)* by deriving upper bounds on the probability of incorrectly clustering sequences. In addition, a comparison with the Smith-Waterman algorithm is performed, with the explicit assumption that it produces correct results. A large-scale comparison of four EST assembly tools was conducted by Liang *et al.* in [LHP+00], using both real and simulated ESTs; the study includes a comprehensive discussion of the exact behaviour of the four programs and their sensitivity to different parameters. The focus there, however, is on the assembled sequences (tentative consensi) rather than on the clusters produced.

Some approaches to *protein clustering*, e.g. [BSS+01, KSV00], implicitly assume one clustering algorithm to be better than others, and then use that as a benchmark. Several studies have been conducted on clustering for *gene expression*, to be more exact, for clustering data from DNA expression arrays. While Wicker *et al.* [WDRP02] concentrate on finding the optimal number of clusters, Yeung *et al.* [YHR01] develop an internal clustering validation index, which they use to evaluate the quality of clusterings produced by three different clustering algorithms, both on real and on simulated data. This validation index is a (simpler) variation of one of the four different indices of internal clustering validation used by Datta and Datta [DD03], who compare six clustering methods for microarray data. This study is closest to the present one in that it aims at comparing different clustering methods, without claiming that one particular validation or clustering method is optimal. Instead, after having demonstrated that the outcome varies significantly depending on the clustering method used, Datta and Datta offer "some guidelines in the choice of a clustering technique to be used in connection with a particular microarray data set." The review by Quackenbush *et al.* [Qua01] discusses relevant issues in expression analysis with microarray data, including an overview of different clustering algorithms and distance measures, with the focus on the former; several of these are compared on a simulated data set.

One example of a large-scale comparison of two *string (dis)similarity measures* is work by Nash *et al.* [NBG01]. Here, the Smith-Waterman algorithm and BLAST are compared for pairwise alignment of protein sequences, and it is shown that in some cases, BLAST finds matches that Smith-Waterman does not. Even though the implicit assumption is made that the Smith-Waterman algorithm produces the 'correct' answer, no method is supplied for evaluating the quality of the results.

Our methodology as presented in Chapter 9 is most similar to that used in *phylogenetic studies*, where a phylogenetic tree is synthetically generated according to some evolutionary model, and then phylogenetic algorithms are evaluated according to how well they can reconstruct the known tree from the leaf data, see e.g. [MWW02, SEM98]. We are not aware of any study that attempts a rigorous comparison of EST clustering methods, separating the effects of the string dissimilarity measure and the clustering algorithm.

8.2 Terminology

ESTs are produced in a laboratory process, which we detailed in Chapter 7: The mRNAs present in the cell (the *transcriptome*) are extracted, reverse transcribed, inserted into vectors, cloned, and then sequenced. As the reverse transcription is comparatively error prone, the resulting ESTs are only approximate substrings of the original mRNAs (save the RNA/DNA substitution of T for U). These mRNAs, having undergone the process of splicing, are related to the original genes in the known way: They can be seen as the concatenation of certain substrings of the gene (the exons) such that the exon order in the gene is preserved. The phenomenon of alternative splicing means that ESTs that have been derived from two different mRNAs of the same gene do not need to have overlaps; instead, they can have similar substrings which need not be at the ends, or even be contiguous. In addition, ESTs are more error-prone than, for instance, sequences for shotgun sequencing. Let $\Sigma = \{A, C, G, T\}$ denote the set of bases.[2] Let $G \subseteq \Sigma^+$ be the set of (not necessarily known) genes.

Definition 8.2.1 (EST Clustering). Given a finite set S of strings (ESTs) over Σ, find a partition $\mathcal{C} = C_1, \ldots, C_k$ of S such that there exist strings (genes) $g_1, \ldots, g_k \in G$ where, for all $1 \leq i \leq k, s \in S : (s \in C_i \iff s$ has been derived from $g_i)$. We refer to the sets C_i as *clusters*.

Here, a *partition* of a set S is a collection of disjoint subsets whose union equals S. Note that Definition 8.2.1 does not require that the genes g_i be specified. The right side of the equivalence is kept in informal terms because the question of how to capture formally the (physical) process of an EST sequence having been derived from a gene, is one of the topics we discuss here.

For a set X, let $\binom{X}{2}$ denote the set of its subsets with cardinality 2. For a partition $\mathcal{C}$ of S and $s \in S$, denote by $\mathcal{C}(s)$ the unique cluster $C \in \mathcal{C}$ such that $s \in C$. Let $\mathcal{C}$ and $\mathcal{D}$ be two partitions of the same set S. We call $\mathcal{C}$ a *refinement* of $\mathcal{D}$ if for all $s, t \in S$: if $\mathcal{C}(s) = \mathcal{C}(t)$, then $\mathcal{D}(s) = \mathcal{D}(t)$. We call it a *proper refinement* of $\mathcal{D}$ if in addition $|\mathcal{C}| > |\mathcal{D}|$.

For strings w, s with $|w| \leq |s|$, let $\mathrm{freq}_s(w) := |\{i \mid w = s_i \ldots s_{i+|w|-1}\}|$, the number of times w occurs in s. Note that this definition allows overlapping occurrences of w.

[2] In fact, ESTs are strings over the alphabet $\Sigma \cup \{N, X\}$, where characters N and X are interpreted either as any of the four characters from Σ or as strings over Σ^+; usually, the presence of an N is an artifact of the sequencing process, while X's are inserted during masking. For the sake of simplicity, we omit this from our definition.

Furthermore, let $\mathrm{occ}_s(w) = 1$ if w is a substring of s, and 0 otherwise. Substrings are often referred to as *subwords*.

In the absence of genes to be compared with, some notion of similarity or dissimilarity is used to cluster individual sequences together. This requires a function, or *dissimilarity measure* of pairs of strings $D : \Sigma^+ \times \Sigma^+ \to \mathbb{R}$. If D is symmetric, obeys the triangle inequality, is positive, and $D(s,t) = 0$ implies $s = t$, then it is called a *metric*; if $D(s,t) = 0$ can hold for some $s \neq t$, then it is a *pseudo-metric*. For string dissimilarity measures, sometimes neither is the case. Given a dissimilarity measure D and a positive integer m (the window size), define

$$\widehat{D}_m : \Sigma^+ \times \Sigma^+ \to \mathbb{R} : \quad \widehat{D}_m(s,t) := \min\{D(s',t') \mid s' \sqsubseteq s, t' \sqsubseteq t, |s'| = |t'| = m\}. \tag{8.1}$$

$\widehat{D}_m(s,t)$ is the minimum dissimilarity of any pair of substrings (windows) of length m of s and t. If m is clear, we simply write $\widehat{D}$ for $\widehat{D}_m$.

8.3 String similarity and distance

Two different, but closely related, concepts are in common use in the literature on strings: That of *similarity* and that of *dissimilarity/distance*. Both are usually represented by a function $D : \Sigma^+ \times \Sigma^+ \to \mathbb{R}$, but the former type takes on higher values the closer (more similar) two strings are, while the latter decreases in this case. For the sake of consistency, we view all measures as dissimilarity measures; for alignment (a measure of similarity), we transform the score function into a penalty function to achieve a measure of dissimilarity (see below).

String dissimilarity measures employed in approximate string matching include those that are based on *alignment* and those that are based on *subword* comparisons. Other approaches exist, such as information theoretic ones (as cited in the review [VA03]), but are not commonly used. In approximate string matching of biological sequences, the most widely employed string dissimilarity measure is BLAST [AGM+90]. For an overview of approximate string matching, see [Nav01, Gus97].

Alignment and edit distance

The *Levenshtein distance* [Lev66], also referred to as *unit cost edit distance*, of strings s, t is the minimum length of a sequence of edit operations transforming s into t, where admissible edit operations are substitutions, insertions, and deletions of characters. Enhanced cost functions include different cost attached to different types of operations, and the cost of an operation depending on the characters involved (e.g. substitution of a for c may have different cost from substitution of b for c). The Levenshtein distance is a metric. An optimal sequence of such transformations can be visualised as an *alignment* of the two strings, by placing them under each other character by character, possibly inserting free spaces (*gaps*), such that there are no two gaps in one column. By assigning penalties for mismatches or gaps in an alignment, we can compute an alignment score, a measure of *similarity*. More enhanced scoring functions include affine or more general types of gap penalty functions: here, the score given to a character aligned with a gap may

depend on the number of consecutive gaps before. The alignment score of two strings is the score of an optimal alignment. To find high-similarity substrings in two strings, a *local alignment* is sought. The definition differs in that the alignment need not continue to the ends of the strings. Another variant is *end-space free global alignment*, where gaps at the ends of either string have zero cost. Since we discuss measures of *dissimilarity*, we will use alignment scoring functions which assign positive values to mismatches and gaps, and negative values to matches, and will refer to such functions as *penalty functions*.

Computing an optimal local alignment score can be done with the well-known dynamic programming algorithm of Smith-Waterman. To improve performance, heuristic algorithms like BLAST [AGM+90] and FASTA [LP85] are used, which employ filtering techniques to isolate areas where matches are likely to happen. Most biological clustering methods use BLAST as the underlying dissimilarity measure, among them UniGene [BS95] and TIGR [QCL+01].

Word frequency counts

Fix a subword size $q \in \mathbb{N}$, thus $q > 0$. For 'small' q, substrings of length q have been referred to as q-*words*, q-*mers*, or q-*grams*. Any string $s \in \Sigma^+$ can be mapped to its q-gram vector, or word-frequency vector $\mathrm{freq}_s \in \mathbb{N}^{|\Sigma|^q}$, whose w'th entry is just $\mathrm{freq}_s(w)$. The q-*gram distance* [Ukk92] of two strings is

$$D_{q\text{–gram}}(s,t) := \sum_{w \in \Sigma^q} |\mathrm{freq}_s(w) - \mathrm{freq}_t(w)|. \tag{8.2}$$

This is the L_1- (or Manhattan) distance of the vectors freq_s and freq_t. However, $D_{q\text{–gram}}$ is only a pseudo-metric, since $D_{q\text{–gram}}(s,t) = 0$ for all s, t with identical word frequency vectors. Ideas derived from the q-gram distance have been implemented for database search in the QUASAR project [BCF+99], in the software SWIFT [RSMpt], which can also be employed for clustering ESTs. Note that for $q = 1$, we get exactly the L_1-distance of the two compomers of s and t, the so-called *compomer distance* (cf. Chapter 3).

Another dissimilarity measure using the word frequency vectors is referred to as d^2 [TBDS90]:

$$D_{d^2,q}(s,t) := \sum_{w \in \Sigma^q} (\mathrm{freq}_s(w) - \mathrm{freq}_t(w))^2. \tag{8.3}$$

This is the squared L_2 (or Euclidean) distance of the frequency vectors, hence the name. Note that $\sqrt{D_{d^2,q}}$ is a pseudo-metric, but $D_{d^2,q}$ is not, because it does not obey the triangle inequality. In the more general form, fix a lower and an upper bound l, u on the word size, and set $D_{d^2}(s,t) := \sum_{q=l}^{u} D_{d^2,q}(s,t)$. For EST-clustering, experimental evidence has shown that fixing q is satisfactory, and commonly $D_{d^2,6}$ is used. However, it should be noted that this measure is, in theory, not a good approximation of edit distance: There exist sequences s, t of length 100 such that $D_{d^2,6}(s,t) = 0$ but the unit edit distance of s and t is 30. The StackPack EST clustering package uses a variant of $\widehat{D}_{d^2,6}$ with window size $m = 100$.

Fingerprints and subword occurrences

Fingerprints have been employed in computational biology for physical mapping of DNA (see [Pev00, Chapter 3]) and, more recently, for EST clustering in [HSL+99]. Given a finite set $P \subseteq \Sigma^+$ of words, and a string $s \in \Sigma^+$, the *fingerprint* of s is the set $P \cap \{t \mid t \sqsubseteq s\}$, or its representation as a Boolean vector in $\{0,1\}^{|P|}$ whose i'th entry is $occ_s(p_i)$ for some enumeration $p_1, \ldots, p_n$ of P. Extending P to all Σ^q, we get the Boolean vector $occ_s \in \{0,1\}^{|\Sigma|^q}$ with w'th entry $occ_s(w)$. Again, any distance measure on Boolean vectors can be applied to define a dissimilarity measure on strings, such as the Hamming distance:

$$D_{q\text{–occurrence}}(s,t) := \sum_{w \in \Sigma^q} |occ_s(w) - occ_t(w)|. \tag{8.4}$$

Another simple dissimilarity measure using subword occurrences is a Boolean function we refer to as *common word*, which assigns to two sequences distance 0 if they share a subword of fixed size K, and 1 otherwise. Formally,

$$D_{cword}(s,t) := \begin{cases} 0 & \text{if there exists } w, |w| = K, w \sqsubseteq s, t, \\ 1 & \text{otherwise.} \end{cases} \tag{8.5}$$

Since K is typically 'large' (say, around 20), we use a different variable from q, which is usually thought of as being 'small', typically up to 10. Note that this dissimilarity measure is symmetric, but does not obey the triangle inequality, and can have value 0 for non-identical strings; thus, it is not even a pseudo-metric. The advantage of the common word function is that clustering can be implemented in linear time, at least for reasonable size K. The clustering is in general too coarse, but provided K is chosen well, a better clustering can be produced by refining it, and thus this method can provide a good initial clustering. It is used, e.g. in [KAKB03], for determining the order in which to compare the sequences (done with an alignment-based measure); in [MCJ03], a measure is employed that combines several non-contiguous common words into one dissimilarity score.

We define a further dissimilarity measure, *Boolean q-grams*, which is an extension of D_{cword}, and in some sense complementary to $D_{q\text{–occurrence}}$: It counts the number of common q-grams; however, as a dissimilarity measure, we define it to be negative for each of these common q-grams: $D_{B\ q\text{-gram}}(s,t) := -\sum_{w \in \Sigma^q} occ_s(w) \cdot occ_t(w)$. So, $D_{B\ q\text{-gram}}$ is the negative scalar product of the Boolean vectors defined above. Again, this dissimilarity measure is not a metric, and not even a pseudo-metric.

Sliding windows

In EST clustering, local regions of high similarity are sought. This is reflected by, e.g. computing local alignments. For subword-based measures, a 'sliding window' of a fixed size m is used instead. Pairs of subsequences of size m are compared; comparing all such pairs yields $\widehat{D}_m(s,t)$. Note that even if D is a metric, $\widehat{D}$ in general is not, since the triangle inequality may be violated.

To increase time efficiency, sometimes not all pairs of windows are compared. For instance, the StackPack EST clustering tool uses a variant of $\widehat{D}_{d^2,6}$, where

all windows in one sequence are compared with every k'th window in the other sequence:

$$D_{d^2\ \mathrm{asym}}(s,t) := \min\left\{D_{d^2,6}(s', t^{(i)}) \mid |s'| = m, s' \sqsubseteq s, i = 0, \ldots, \lfloor |t|/k \rfloor\right\}, \tag{8.6}$$

where k is the skip size, and $t^{(i)} := t_{i\cdot k+1} \ldots t_{\min(i\cdot k+m,|t|)}$ is the substring of t starting at position $i \cdot k + 1$. Note that this measure is not symmetric, which is why we refer to it as *asymmetric* d^2. In contrast, we will refer to $D_{d^2\ \mathrm{sym}} := \widehat{D}_{d^2}$, which compares all pairs of windows, as *symmetric* d^2.

8.4 Clustering algorithms

Data clustering is the task of grouping together a set of objects into subgroups according to some property or properties. There are a large variety of clustering algorithms and a fair amount of terminological inconsistency in the literature. See [JMF99], whose terminology we follow, for an introduction. We assume in the following that $D : X \times X \mapsto \mathbb{R}$ is a dissimilarity measure on the set of objects X.

For EST clustering, most often hierarchical clustering algorithms are used. A sequence of increasingly fine partitions of the data is produced, starting from the whole set as one cluster, where each partition is a refinement of the previous one. The process stops when a partition is found that is fine enough according to some previously defined criterion. Most EST clustering algorithms are *single linkage* (nearest neighbour, transitive closure): Two objects x and y are clustered together if there is a finite sequence $x = x_1, x_2, \ldots, x_{k-1}, x_k = y$ such that for all $1 \leq i < k$, $D(x_i, x_{i+1}) < \theta$, for some threshold θ. If the threshold is set beforehand, single linkage can be viewed as a partitional clustering algorithm; in its general form, it is a hierarchical clustering algorithm where, usually, different levels correspond to different values of θ. Other types of clustering algorithms, such as *complete linkage* or k-*means*, are not commonly used for EST clustering.

Clustering can be *seeded* or *unseeded*. In seeded clustering, the number of clusters is known beforehand, and a member of each cluster (the *seed*) is supplied as part of the input: in EST clustering, this is typically a full-length mRNA. In unseeded clustering, no additional information is used and, in particular, the number of clusters is unknown.

Of the EST clustering methods mentioned in Section 8.1, UniGene uses a hierarchical seeded clustering algorithm, where mRNAs are used as seeds, and different hierarchies represent different types of merging stages: e.g. edges that connect two ESTs are judged less reliable than those connecting an EST to an mRNA. TIGR uses single linkage clustering and produces a tentative consensus for each cluster. TGICL [PHL+03], a recent enhancement of TIGR, also uses single linkage, and so do both [KAKB03] and [MCJ03]. StackPack [CvGG+01] uses single linkage and different hierarchy levels, where the lowest (least confident) level is "clone linking": Two clusters are merged if they contain two end reads of the same clone. Hartuv *et al.* [HSL+99] use the HCS clustering algorithm: Highly connected (i.e., particularly dense) subgraphs in a threshold graph are output as clusters.

The prevalence of single linkage is due to the fact that EST clustering constitutes a type of local alignment to an unknown reference sequence (the gene). Thus, the

traditional drawback of single linkage, namely that it creates 'elongated clusters,' is a desired effect in EST clustering.

8.5 Clustering evaluation

The quality evaluation of a clustering algorithm is referred to in the literature as *cluster validity analysis*, *clustering evaluation*, or *goodness of fit*. A large number of different methods are in use. They can be grouped into methods of *internal* and *external* assessment [JMF99]: Internal assessment methods validate some criterion of internal consistency, while external methods compare the resulting clusters to an ideal solution. In the following, we only consider methods of *external* assessment. A score of the goodness of fit of the two clusterings is computed, which is referred to as a *validity index*. For comparing partitions, commonly used validity indices include the Rand Index, the Jaccard Index, and the Minkowski Index. For an overview of cluster validity indices, see [Dub93, JD88]. Less formal validation techniques include counting the number of exactly matching clusters or comparing the number of singleton clusters (as in [MCJ03] and [BDH99] resp.). Two measures commonly employed in the biological literature are *sensitivity* and *specificity*. Below, we give formal definitions for these validity indices.

Let S be the ground set of size $|S| = n$, the two partitions under consideration $\mathcal{C} = \{C_1, \ldots, C_k\}$ and $\mathcal{D} = \{D_1, \ldots, D_\ell\}$. All indices mentioned above are functions of the number of unordered pairs of elements that were "treated alike" and "treated differently" by the two clusterings. Set

$$\begin{aligned}
a_1 &= \left|\left\{\{s,t\} \in \tbinom{S}{2} \mid \mathcal{C}(s) = \mathcal{C}(t) \text{ and } \mathcal{D}(s) = \mathcal{D}(t)\right\}\right|, \\
a_2 &= \left|\left\{\{s,t\} \in \tbinom{S}{2} \mid \mathcal{C}(s) \neq \mathcal{C}(t) \text{ and } \mathcal{D}(s) \neq \mathcal{D}(t)\right\}\right|, \\
d_1 &= \left|\left\{\{s,t\} \in \tbinom{S}{2} \mid \mathcal{C}(s) = \mathcal{C}(t) \text{ and } \mathcal{D}(s) \neq \mathcal{D}(t)\right\}\right|, \\
d_2 &= \left|\left\{\{s,t\} \in \tbinom{S}{2} \mid \mathcal{C}(s) \neq \mathcal{C}(t) \text{ and } \mathcal{D}(s) = \mathcal{D}(t)\right\}\right|.
\end{aligned}$$

Further, we set $a = a_1 + a_2$ and $d = d_1 + d_2$. Observe that a is the number of agreements and d the number of disagreements between the two clusterings. In the biological literature, it is customary to speak of 'true/false positives/negatives.' If we view $\mathcal{C}$ as the correct clustering, then a_1 is the number of true positives, a_2 the number of true negatives, d_1 the number of false negatives, and d_2 the number of false positives. We give the definitions of the above validity indices in Table 8.1, transforming the definition of the Minkowski Index into one employing the values introduced above.

The Rand Index can also be corrected for chance by the expected difference to a randomly chosen clustering with the given number of clusters, see [HA85]. We have decided to use the uncorrected Rand Index because it is not realistic to assume that the number of clusters is known. Also, correction using the expected score of a random clustering is assumed to be small and often not worth the additional computational effort.

Rand Index	$\frac{a}{\binom{n}{2}}$
Jaccard Index	$\frac{a_1}{a_1+d}$
Minkowski Index	$\left(\frac{2d}{2(a_1+d_1)+n}\right)^{\frac{1}{2}}$
Sensitivity	$\frac{a_1}{a_1+d_1}$
Specificity	$\frac{a_1}{a_1+d_2}$

Table 8.1: Common validity indices

9 A Method for Evaluating String Dissimilarity Measures and Clustering Algorithms for EST Clustering

In this chapter, we introduce a methodology for evaluating the different components involved in EST clustering, depending on the type of input data. We then introduce the tool ECLEST which we developed for EST clustering evaluation, and present simulation results. For the simulations, we used artificial data generated by the tool ESTSim, which was developed and implemented by Scott Hazelhurst at the University of Witwatersrand for the purposes of this study [HB03].

The software ECLEST was implemented by Judith Zimmermann within her Diplom thesis [Zim03], which was carried out under the supervision of the author of this thesis and Prof. Peter Widmayer at ETH Zürich.

The contents of this chapter have been published in [ZLH04].

9.1 Evaluation method

We propose a rigorous method for evaluating the suitability of string dissimilarity measures and clustering algorithms for EST clustering, depending on the characteristics of the input data. We distinguish between four different components:

1. the input data,
2. the string dissimilarity measure,
3. the clustering algorithm, and
4. the clustering validity index, according to which the quality of the output is judged.

As *input data*, we generate *simulated* ESTs from real cDNAs or full-length mRNAs according to specifiable error parameters, using the tool ESTSim (*EST Simulator*) [HB03]. ESTSim generates simulated ESTs from input cDNAs or mRNAs according to user-specifiable parameters, and outputs EST-like sequences, identifying which input sequence they were derived from.

We then compute a partition of the set of simulated ESTs with our tool ECLEST (*Evaluator for CLusterings of ESTs*). ECLEST computes clusterings of a set of input sequences, using specified *string dissimilarity measures* and *clustering algorithms*, where both components can be chosen independently. It then computes a score of the resulting partition by comparing it to the ideal partition, using a specified *validity index*. Again, the validity index can be chosen independently of the other

two components. ECLEST has a modular architecture, which allows different string dissimilarity measures, clustering algorithms, and clustering validity indices to be used.

We believe that using simulated data rather than real ESTs is better suited for this type of study, for two reasons: First, for real ESTs, the ideal clustering is never known and can at best be approximated by, for example, aligning ESTs to the genome, or by relying on annotation information, or both. Therefore, any evaluation will be subject to possible errors made during the experimental phase. Second, simulated data allow for careful tuning of parameters, such as average sequence length or single base error distribution, and thus for estimating the impact of an individual parameter on the methods used: This is, of course, impossible with real data.

We demonstrate the fitness of our method by presenting the results of a test study we carried out on 699 cDNA sequences from a mammalian gene collection. From these, we derived approximately 16,000 simulated ESTs and clustered them using five different string dissimilarity measures and a single linkage clustering algorithm, computing the clustering quality scores using the Rand Index (for details, see Section 9.3). We ran two sets of tests, where we varied the error types, in the first simulating error types and levels as laid down in the guidelines of NCBI, and in the second, increasing these error levels. We were able to draw statistically significant conclusions as to the quality of clusterings produced using these different string dissimilarity measures.

9.2 ECLEST: A tool for evaluating EST clusterings

The ECLEST tool (Evaluator for CLusterings of ESTs) was implemented in Java 1.4. It takes as input a set of DNA-strings in FASTA-format and a text-file specifying the ideal clustering. It then computes and evaluates a clustering of the input strings, using

1. a specified string dissimilarity measure;
2. a specified clustering algorithm; and
3. a specified clustering validity index.

The input data may be real ESTs, or artificial data such as produced by the dedicated tool ESTSim (cf. Section 9.3.2).

The interfaces are designed in such a way that ECLEST can be easily extended by new algorithms in any of the three categories. In the current version, five dissimilarity measures, one clustering algorithm, and one clustering validity index have been implemented.

The clustering algorithm implemented is single linkage clustering. A partition is computed with the following property: Given threshold θ,

$$\mathcal{C}(s) = \mathcal{C}(t) \iff \text{exist } s = s_1, s_2, \ldots, s_{k-1}, s_k = t \text{ s.t. for all } 1 \leq i < k, d(s_i, s_{i+1}) < \theta. \tag{9.1}$$

dissim. measure	parameters	definition
end-space free alignment	penalty function f (end-space free)	$D_{esfa}(s,t) =$ $\min\{f(A) \mid A \text{ global alignment of } s,t\}$
common word	word size K	$D_{cword}(s,t) = \begin{cases} 0 & \text{if } \exists w, \lvert w\rvert = K, w \sqsubseteq s,t \\ 1 & \text{otherwise} \end{cases}$
symmetric d^2	bounds u,l window size m	$D_{d^2\ sym} = \widehat{D}_{d^2}$, where $D_{d^2}(s,t) = \sum_{\lvert w\rvert=l}^{u}(freq_s(w) - freq_t(w))^2$
asymmetric d^2	bounds u,l window size m skip size k	$D_{d^2\ asym}(s,t) = \min\{D_{d^2}(s',t^{(i)}) \mid \lvert s'\rvert = m,$ $s' \sqsubseteq s, i = 0,\ldots,\lfloor\lvert t\rvert/k\rfloor\}$, using D_{d^2} as above and $t^{(i)} = t_{i\cdot k+1}\ldots t_{\min(i\cdot k+m,\lvert t\rvert)}$
Boolean q-grams	word size q window size m	$\widehat{D}_{B\ q\text{-gram}}$, where $D_{B\ q\text{-gram}}(s,t) = -\sum_{\lvert w\rvert=q} occ_s(w)\cdot occ_t(w)$

Table 9.1: Dissimilarity measures currently implemented in ECLEST. All parameters are specified in a configuration file.

The algorithm can be implemented efficiently with a union-find data structure, starting with a singleton cluster for each element.

For the clustering validation, we implemented the Rand Index (see Table 8.1 on page 107). The dissimilarity measures are listed in Table 9.1. The implementation we used is the Smith-Waterman dynamic programming algorithm for the end-space free alignment; a truncated suffix tree, which is a slight modification of Ukkonen's algorithm [JU91], for the common word measure; an incremental computation with dynamic lists for the symmetric d^2 [Haz04], and modifications of these for the assymetric d^2 and the Boolean q-grams.

For further details, including full implementation details, see [Zim03]. The ECLEST manual, as well as the Java code, can be found at

`http://bibiserv.techfak.uni-bielefeld.de/esteval/`,

including instructions on how to extend the application.

9.3 Suitability evaluation for single linkage clustering

We ran two sets of experiments, where we compared the five dissimilarity measures above, having fixed the clustering algorithm (single linkage) and one validation index (Rand Index). We used simulated data which we generated using the tool ESTSim [HB03], which had been developed for the purpose of producing EST–like sequences from input DNA sequences. We first briefly introduce the software ESTSim, then detail the data we used, the parameters of the dissimilarity measures, and close with a report of our results.

9.3.1 Using ESTSim for Creating Benchmarks of Simulated EST Sets

ESTSim [HB03] has been designed to create large sets of "realistic" but artificial EST sequences. The rationale behind using synthetic data in benchmarking is the following. When using real data, many unknown characteristics can and do influence the outcome; in particular, with real ESTs it is not known with complete certainty which genes they have been derived from. The use of artificial test data enables us to produce data with a range of different properties. Thus, the effect of different error models on the effectiveness of the use of certain EST clustering methods can be tested precisely, which would be impossible using real data.

ESTSim creates artificial ESTs from a set of given cDNA sequences using criteria specified by the user. The artificial creation of ESTs in this way will lead to the creation of an EST set whose exact final clustering is known, because each artificial EST carries an identifier of the sequence it was derived from. So, when testing an EST clustering method, the output can be evaluated by comparing it to the known ideal clustering. The approach of ESTSim is similar to that of GenFrag [EB94], though ESTSim is specifically tailored to EST data and supports more sophisticated error models. Here, we give a brief overview; full details can be found in [HB03].

ESTSim simulates the production of ESTs from cDNAs or any genomic-like data with a variety of models. It has been designed in such a way as to generate different types of data, simulating the different types of errors and error rates, according to the biological processes used in different laboratories.

Given an input sequence (mRNA or DNA), it is split into fragments (according to user parameters), the fragments copied a user–specified number of times, and then each fragment is mutated according to user–specified error models. The errors modeled include single base errors, stuttering, and ligation, each with parameters that can be specified by the user. Contamination and repeats have not been modeled because typically, these are masked out in a preprocessing phase for EST clustering.

For an example of the probability distribution of different single base errors, see Figure 9.1. ESTSim has been built in a modular way so that it is relatively easy to build in new error models. However, we believe that the number of parameters already at the user's choice allow for a large enough design space to build realistic data sets.

9.3.2 Data used in the experiments

For our experiments, we used ESTSim to generate simulated ESTs from a collection of human cDNAs from a mammalian gene collection at `http://mgc.nci.nih.gov/`. Non-human cDNAs are removed by using BLAST and common contaminant information. Since we do not want to include the effects of alternative splicing, we removed all but one cDNA sequence per gene, by performing complete pair-wise comparison with BLAST: If the similarity (e-value) exceeded 10^{-85}, one of the two sequences was removed, because it is likely that they correspond to the same gene. This yields a set of 699 cDNAs, which we split up into sets of 4 to 10 sequences each, such that the overall length (number of nucleotides) in each set is roughly

the same. We refer to each of these sets as a *test set*; there are 134 such test sets. Ten of these we used for preliminary testing.[1]

We generated two sets of simulated ESTs for each of the test sets, called *experiments* in the following. The two experiments differed by the parameters used for ESTSim. Each experiment consisted of running ECLEST, for each test set, on the ESTs generated by ESTSim from this test set, with the five dissimilarity measures detailed below, and computing the validity index for each of the five clusterings produced. We ran 10 preliminary tests for estimating the distribution of the dissimilarity measures and for setting up our hypotheses, and used the remaining 124 test sets (for technical reasons, 123 for the second experiment) to test these hypotheses. Each test set included between 200 and 300 ESTs as input to ECLEST.

The first experiment simulates high quality ESTs that meet the standards laid down by NCBI. The probability graph of a single base error, taking into account random noise, polymerase decay, and primer interference is shown in Figure 9.1. A very modest amount of stuttering is permitted: The parameter was chosen such that stuttering would happen on average with probability 0.005 on a sequence of 10 Gs. In the second experiment, we increased most of the parameters of ESTSim in such a way as to produce ESTs that have twice as high error probabilities for most error types. In both experiments, ESTs of length between 300 and 500 bp were produced, where every base of the original cDNAs appears on average in 5 ESTs; no reverse reads are produced. We also ignore ligation, to exclude that two clusters be merged artificially.

We list the settings used for our experiments in Table 9.2. Briefly, `samplerandom` x y z w generates fragments of the input sequence of random length between x and y; each base will appear in z copies on average; and w is the proportion of reverse complement copies. Regarding the parameters, α is the probability that an arbitrary base changes randomly; β is the margin (number of bases) at the beginning of the sequence where errors are most likely to occur; γ is the probability that an error occurs in the margin; ζ and ξ are two paramaters involved in the formula defining the effect of the polymerase decay at the end of the sequence; η is the probability that stuttering happens after a repeat; θ is the ligation parameter; κ, λ, and μ define the proportion of single base errors that are substitutions, deletions, and insertions, respectively; and ν gives the proportion of changes resulting in an N. See [HB03] for a more detailed description.

	manner of EST generation	α	β	γ	ζ	ξ	η	θ	κ, λ, μ	ν
Experiment 1	`samplerandom` 300 500 5 0	0.005	30	0.04	1	2	20	0	10 each	0
Experiment 2	`samplerandom` 300 500 5 0	0.01	50	0.06	2	3	0	0	10 each	0

Table 9.2: Parameters for ESTSim in the two experiments.

[1] This includes determining the number of test sets we needed: These preliminary tests revealed that for statistically significant statements, we needed 120 test sets, hence the total number of test sets.

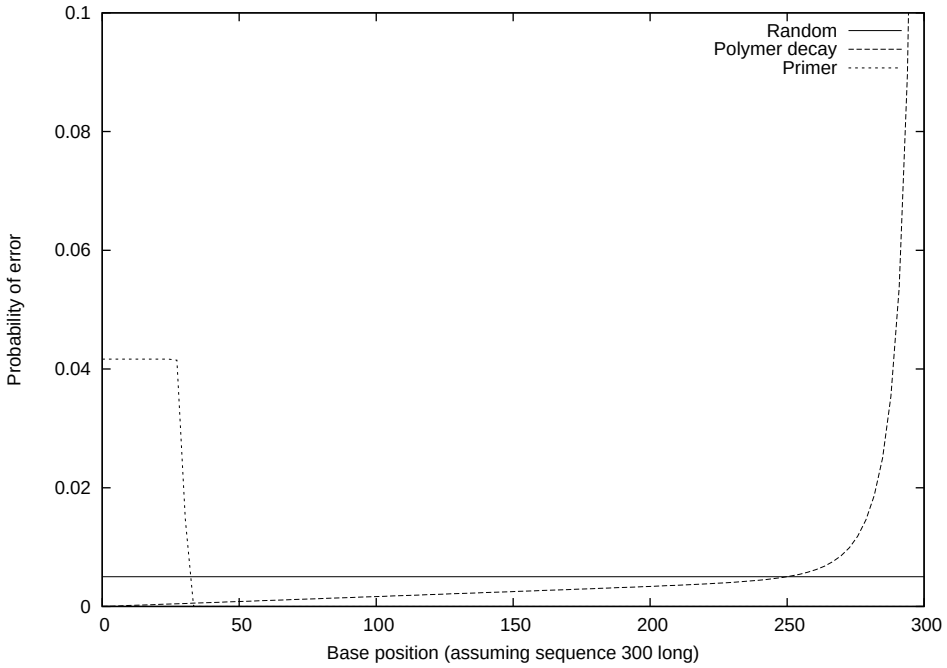


Figure 9.1: Error probability functions for different single base errors. For clarity, the y-axis is cropped at $y = 0.1$.

9.3.3 Dissimilarity measures compared

We compare the five dissimilarity measures detailed in Section 9.2; the parameters are shown in Table 9.3. We chose these dissimilarity measures for the following reasons. BLAST is frequently used as dissimilarity measure in EST clustering, thus in our alignment measure, we use a penalty function which closely emulates BLAST parameters. We refrained from using BLAST itself, because we are aiming for a clean comparison which does not include the many heuristics used in a full BLAST implementation. Asymmetric d^2 is used by StackPack [CvGG+01]: We set all parameters accordingly, except for the threshold, which we optimised ourselves. We chose symmetric d^2 as a comparison to asymmetric d^2, in order to see whether the efficiency gained by comparing only every 50'th window of one of the two sequences can be justified: If the quality of the clusterings decreases dramatically, it cannot. Common word seems a good dissimilarity measure as a preprocessing step because of very efficient implementations; in addition, it may already produce good quality clusterings. Finally, we chose Boolean q-grams because they, too, are used in EST clustering. The thresholds θ were set heuristically by running a few tests and finding the thresholds that maximize the validity score.

9.3.4 Results

We ran 10 preliminary test sets to estimate the distributions of the clustering scores computed with the different dissimilarity measures, and to set up our hypotheses. We then ran 124 test sets to test our hypotheses. We summarize the results in

measure	parameters	θ
alignment	penalty function $f(\text{match}) = -1$, $f(\text{mismatch}) = +3$, $f(\text{gap opening}) = +5$, $f(\text{gap extension}) = +2$, $f(\text{end space gap}) = 0$	-20
symmetric d^2	fixed word size 6 ($u = l = 6$), window size $m = 100$	50
asymmetric d^2	fixed word size 6 ($u = l = 6$), window size $m = 100$, skip size $k = 50$	50
common word	word size $K = 19$	1
Boolean q-grams	word size $q = 6$, window size $m = 100$	-13

Table 9.3: Dissimilarity measures used in our experiments with parameters and thresholds.

Table 9.4: Here, we report the mean and standard deviation of the clustering scores in the two experiments, taken over 134 tests for Experiment 1, and 133 tests for Experiment 2. (To be exact, since the distributions are unknown, the values are *(i)* the average, and *(ii)* the average squared difference from the average.) Tables 9.5 and 9.6 show how the dissimilarity measures compared to each other in the two experiments, in absolute numbers and in percentages (rounded to .1%). Entry $a/b/c$ in cell (i, j) denotes that measure i produced a better clustering than measure j in a number (percent) of cases; i and j tied in b cases; and j produced a better result in c cases. We visualize the results for Experiment 2 in Figures 9.2 and 9.3.

	NCBI-parameters (Experiment 1)		doubled NCBI-param.s (Experiment 2)	
	Mean	St. dev.	Mean	St. dev.
alignment	0.9861	0.0181	0.9834	0.0199
symm. d^2	0.9768	0.0221	0.9568	0.0306
asymm. d^2	0.9718	0.0233	0.9471	0.0334
cword	0.8674	0.2037	0.8405	0.2284
Bool. q-grams	0.8262	0.0586	0.8171	0.0548

Table 9.4: Rand Index of the clusterings compared with ideal clusterings, with two types of input parameters.

	alignment	symm. d^2	asym. d^2	cword
alignment				
symm. d^2	1.5/41.0/57.5			
asymm. d^2	0.7/25.4/73.9	0/70.1/29.9		
cword	14.2/36.6/49.3	36.6/15.7/47.8	43.3/9/47.8	
Bool. q-grams	0/0/100	0/0/100	0/0/100	23.1/0/76.9

Table 9.5: Comparison of measures, for 134 tests with NCBI-like parameters (Experiment 1), in percent.

	alignment	symm. d^2	asym. d^2	cword
alignment				
symm. d^2	0.8/5.3/94			
asymm. d^2	0.8/0.8/98.5	0/23.3/76.7		
cword	3/39.1/57.9	45.1/3.8/51.1	49.6/0.8/49.6	
Bool. q-grams	0/0/100	0/0/100	0/0/100	30.1/0/69.9

Table 9.6: Comparison of measures, for 133 tests with doubled NCBI-parameters (Experiment 2), in percent.

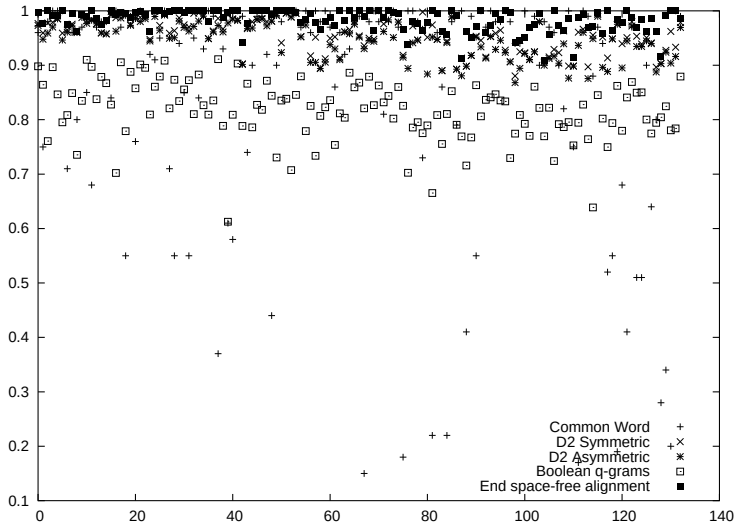


Figure 9.2: Rand Index for the five different dissimilarity measures for the 133 test sets in Experiment 2.

Statistical hypothesis testing

From the preliminary test sets it could be seen that the convenient assumption of normal distribution was untenable, thus we were restricted in the kind of statistical tests we could use. In all cases, we used either the Friedman ranking test or the binomial test. The Friedman test utilises only a ranking of the scores rather than their values. Another limitation is that the Friedman test has been designed for continuous distributions, an assumption that slightly skews the results when several identical scores (*ties*) appear in one test. We therefore ran the test also by only using those test sets where the results were different for all dissimilarity measures in question; however, this reduced dramatically the number of test sets that could be evaluated.

Because of the missing normal distribution assumption, we were unable to make quantitative statements about *how much* better one dissimilarity measure performs than another. All our statements (acceptance or rejection of hypotheses) have been made with a 95% confidence. We used the statistics program R, version 1.6.1 [The], to evaluate our data. The first results are:

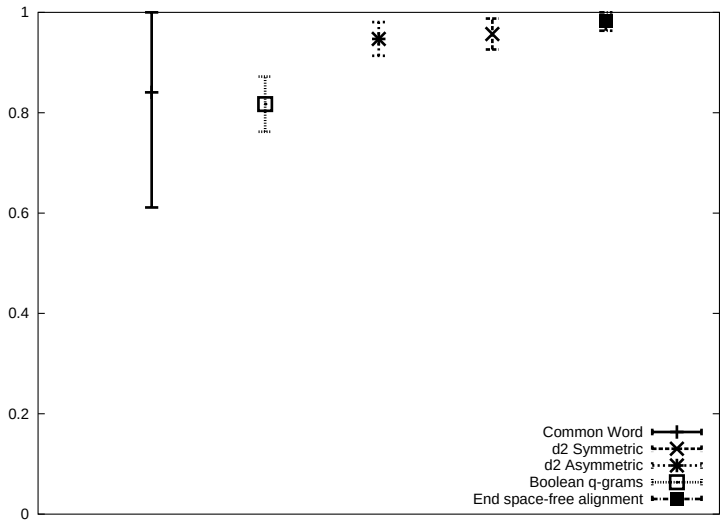


Figure 9.3: Mean and standard deviation of the Rand Index for the five different dissimilarity measures (Exp. 2).

1. In both experiments, the clustering scores of the individual dissimilarity measures do not follow a normal distribution.

2. In both experiments, the clustering scores of the five dissimilarity measures do not have the same distribution.

The second statement could be shown both with data that included ties, and with the much smaller data set without ties. Next we compared those dissimilarity measures where it seemed reasonable to assume that either they differed significantly or that they were similar. Since the distribution of the common word measure was very different from the others, we excluded it from these individual comparisons. Finally, we tested whether certain dissimilarity measures can be used as preprocessing for others in order to achieve a good result. For technical reasons, this type of hypothesis was only tested in the first experiment, where ESTs of NCBI-quality were produced. The results are summarised in Table 9.7.

9.4 Conclusion

We have made two contributions: First, we developed a rigorous methodology for comparing string similarity measures and clustering algorithms for the purposes of EST clustering, and introduced a tool, ECLEST, that implements this method. Second, we presented a preliminary study for the commonly used single linkage clustering algorithm, using common dissimilarity measures and two sets of typical values for EST error models.

Our evaluation methodology comprises independently choosing the string dissimilarity measure, the clustering algorithm, and a clustering validity index to be employed, and computing a clustering of the data as well as a quality score. The

Hypothesis	with probability NCBI-param.s (Experiment 1)	with probability doubled NCBI-param.s (Experiment 2)
symmetric d^2 performs as well as alignment	< 0.5	> 0.05
asymmetric d^2 performs as well as symmetric d^2	> 0.5	not: > 0.05
symmetric d^2 performs better than Boolean q-grams	> 0.95	> 0.95
asymmetric d^2 performs better than Boolean q-grams	> 0.95	> 0.95
common word followed by symmetric d^2 performs as well as symmetric d^2 alone	> 0.95	—
common word followed by alignment performs better than symmetric d^2 alone	not: > 0.5	—

Table 9.7: Statistical results (95% confidence). Hypotheses that could not be validated are marked by "not". A dash '—' denotes that the hypothesis was not tested.

system ECLEST implements all three components; at present, five similarity measures, one clustering algorithm, and one validity index have been implemented. As input, either real ESTs or simulated ESTs can be used such as generated by the tool ESTSim.

The particular experiments we ran illustrated the value of the approach. They show that certain dissimilarity measures yield better results than others, but also that some measures compute very similar results. This gives algorithm designers greater choices in tradeoff considerations, when time and space limitations are vital. Furthermore, it gives rigorous ways of justifying the choice of certain heuristics to speed up the clustering process.

10 Conclusion

In this thesis, we discussed problems from two areas of bioinformatics, namely mass spectrometry and EST clustering. Both areas give rise to string problems which are of interest in their own right, as well as relevant to applications.

We have shown that a wealth of challenging combinatorial problems arise from mass spectrometry (MS) applications. We presented a consistent formal framework for the theory of weighted strings and gave efficient algorithms for several weighted string problems. In particular, the mass decomposition and the submass finding problems are encountered frequently when dealing with biotechnological MS data; the former when asking whether and how a mass can be written as a mass of amino acids (for proteins) or of nucleotides (for DNA); the latter when dealing with MS data where non-site-specific digestion was used.

Our algorithm for the DECOMPOSITION ALL WITNESSES PROBLEM uses a data structure whose size and construction time only depend on the alphabet and not on the query, and can thus be constructed in a preprocessing step. The running time for a query M is then only dependent on the size of this data structure and on the number of compomers with mass M, and not on M itself. This is the first algorithm that solves the problem in time not dependent on the query but only on the output size. Moreover, the data structure can be used to solve several other mass decomposition problems, as well (DECISION, ONE WITNESS, FROBENIUS).

For the submass finding problems, we presented a very efficient algorithm for binary alphabets solving the SUBMASS DECISION PROBLEM, which runs in time $\mathcal{O}(\log n)$ and uses space $\mathcal{O}(n)$ where n is length of the string s queried. Even though no biological applications are apparent at present for an alphabet of size 2, it may be possible to extend the underlying ideas to larger alphabets. Furthermore, we described several algorithms for different variants of the submass finding problem (DECISION, COUNTING, ONE WITNESS, ALL WITNESSES) for multiple query masses. These employ an encoding of submasses in polynomials and derive their efficiency from Fast Fourier Transform for polynomial multiplication.

We presented an algorithm for the DE NOVO PEPTIDE SEQUENCING PROBLEM which enhances a known dynamic programming algorithm. We described its implementation and first simulation results.

The other area we treated in this thesis is EST clustering and the use of different string dissimilarity measures. We presented a framework and a dedicated tool to evaluate the impact of the choice of dissimilarity measure and clustering algorithm on clustering quality. EST clustering is the method of choice for extracting information from the vast amount of publicly available EST sequences. It is thus of vital importance to develop stringent methodologies for improving its effectiveness.

Open Problems

The thesis has opened up a number of directions for further research.

One particularly interesting challenge on the algorithmic side is to extend the ideas behind the mass decomposition algorithm presented in Chapter 4 in such a way that we can solve the DECOMPOSITION COUNTING PROBLEM; at the moment, we can only solve it using the classical dynamic programming algorithm. Another open question is whether the algorithm INTERVAL for the submass finding problem (Section 5.2) can be extended to larger alphabets, in order to benefit from its high efficiency. The algorithms for the submass problems using polynomials (Section 5.3) should be implemented and compared to other algorithms in applications. Finally, the search for a good de novo peptide sequencing algorithm continues.

On the theoretical side, one open problem is how to efficiently store compomers, in particular the set of compomers of a given string. Solving this problem would have a large number of applications not only when the objects of interest are the compomers themselves (such as in mass spectrometry), but also in those where for certain substrings, only the multiplicity but not the order of characters is of interest (such as gene clusters or certain alignment algorithms).

In the EST clustering area, we plan to extend the framework by implementing more string dissimilarity measures and clustering algorithms, and to run further experiments on simulated data as well as on carefully chosen sets of real data.

Bibliography

[AALS03] A. Amir, A. Apostolico, G. Landau, and G. Satta. Efficient text fingerprinting via Parikh mapping. *Journal of Discrete Algorithms*, 1(5-6):409–421, 2003.

[ABB97] J.-M. Autebert, J. Berstel, and L. Boasson. Context–free languages and pushdown automata. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, vol. 1*, chapter 3, pages 111–174. Springer, 1997.

[AGM+90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[AM03] R. Aebersold and M. Mann. Mass spectrometry-based proteomics. *Nature*, 422:198–207, 2003.

[Arn01] D. Arnott. Basics of triple-stage quadropole/ion-trap mass spectrometry: precursor and neutral loss scanning. Electrospray ionisation and nanospray ionisation. In P. James, editor, *Proteome Research: Mass Spectrometry*, pages 11–29. Springer, 2001.

[BCF+99] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array. In *Proc. of the 3rd Annual International Conference on Computational Molecular Biology (RECOMB'99)*, pages 77–83, Lyon, France, 1999. ACM Press.

[BCG+02] S. Baginsky, M. Cieliebak, W. Gruissem, T. Kleffmann, Zs. Lipták, M. Müller, and P. Penna. AuDeNS: A tool for automatic de novo peptide sequencing. Technical Report 383, ETH Zurich, Department of Computer Science, Oct. 2002.

[BCL04] N. Bansal, M. Cieliebak, and Zs. Lipták. Efficient algorithms for finding submasses in weighted strings. In *Proc. of the Fifteenth Annual Combinatorial Pattern Matching Symposium (CPM 2004)*, volume 3109 of *LNCS*, pages 194–204, 2004.

[BDH99] J. Burke, D. Davison, and W. Hide. D2_cluster: A validated method for clustering EST and full-length cDNA sequences. *Genome Research*, 9(11):1135–1142, 1999.

[BE01] V. Bafna and N. Edwards. SCOPE: A probabilistic model for scoring tandem mass spectra against a peptide database. *Bioinformatics*, 17(Supplement 1):S13–S21, 2001.

[BE03] V. Bafna and N. Edwards. On de novo interpretation of tandem mass spectra for peptide identification. In *Proc. of the Seventh Annual International Conference on Computational Molecular Biology (RECOMB'03)*, pages 9–18, 2003.

[Ben86] P. J. Bentley. *Programming Pearls*. Addison–Wesley, 1986.

[Ben03] G. Benson. Composition alignment. In *Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI'03)*, pages 447–461, 2003.

[BHNW] D. E. Beihoffer, J. Hendry, A. Nijenhuis, and S. Wagon. Faster algorithms for Frobenius numbers. Submitted.

[BL05a] S. Böcker and Zs. Lipták. Efficient mass decomposition. In *Proc. of the ACM Symposium on Applied Computing (ACM-SAC'05)*, pages 151–157, 2005.

[BL05b] S. Böcker and Zs. Lipták. The money changing problem revisited: Computing the Frobenius number in time $O(ka_1)$. In *Proc. of the Eleventh International Computing and Combinatorics Conference (COCOON'05)*, 2005.

[Böc03a] S. Böcker. Sequencing from compomers: Using mass spectrometry for DNA de-novo sequencing of 200+ nt. In *Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI'03)*, pages 476–497, 2003.

[Böc03b] S. Böcker. SNP and mutation discovery using base-specific cleavage and MALDI-TOF mass spectrometry. *Bioinformatics, Supplement 1 (ISMB'03)*, pages i44–i53, 2003.

[Böc04] S. Böcker. Weighted sequencing from compomers: DNA de novo sequencing from mass spectrometry data in the presence of false negative peaks. In *Proc. of the German Conference on Bioinformatics (GCB'04)*, pages 13–23, 2004.

[Bra42] A. Brauer. On a problem of partitions. *Amer. J. Math.*, 64:299–312, 1942.

[BS62] A. Brauer and J. E. Shockley. On a problem of Frobenius. *J. Reine Angew. Math.*, 211:215–220, 1962.

[BS95] M. Boguski and G. Schuler. ESTablishing a human transcript map. *Nature Genetics*, 10(11):369–371, 1995.

[BSS+01] E. Bolten, A. Schliep, S. Schneckener, D. Schomburg, and R. Schrader. Clustering protein sequences—structure prediction by transitive homology. *Bioinformatics*, 17(10):935–941, 2001.

[CC02] D. M. Creasy and J. S. Cottrell. Error tolerant searching of uninterpreted tandem mass spectrometry data. *Proteomics*, 2:1426–1434, 2002.

[CDF90] M. Cosnard, J. Duprat, and A. G. Ferreira. The complexity of searching in $X + Y$ and other multisets. *Information Processing Letters*, 34:103–109, 1990.

[CEL+02] M. Cieliebak, T. Erlebach, Zs. Lipták, J. Stoye, and E. Welzl. Algorithmic complexity of protein identification: Searching in weighted strings. In *Proc. of the 2nd IFIP International Conference of Theoretical Computer Science (TCS'02)*, pages 143–156, 2002.

[CEL+04] M. Cieliebak, T. Erlebach, Zs. Lipták, J. Stoye, and E. Welzl. Algorithmic complexity of protein identification: Combinatorics of weighted strings. *Discrete Applied Mathematics*, 137(1):27–46, 2004.

[CH02] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. of the 34th Symposium on the Theory of Computing (STOC'02)*, pages 592–601, 2002.

[Chr01] A. Christoffels. *Generation of a human gene index and its application to disease candidacy*. PhD thesis, University of the Western Cape, 2001.

[Cie03] M. Cieliebak. *Algorithms and Hardness Results for DNA Physical Mapping, Protein Identification, and Related Combinatorial Problems*. PhD thesis, Eidgenössische Technische Hochschule (ETH) Zürich, 2003. Diss ETH no. 15258.

[CKT+01] T. Chen, M.-Y. Kao, M. Tepel, J. Rush, and G. M. Church. A dynamic programming approach to de novo peptide sequencing via tandem mass spectrometry. *J. Comp. Biol.*, 8(3):325–337, 2001. A preliminary version appeared in the Proceedings of the Symposium on Discrete Algorithms (SODA 2000) in 2000.

[CT65] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[CvGG+01] A. Christoffels, A. van Gelder, G. Greyling, R. Miller, T. Hide, and W. Hide. STACKdb: Sequence tag alignment and consensus knowledgebase. *Nucleic Acids Research*, pages 234–238, 2001.

[DAC+99] V. Dančík, T. A. Addona, K. R. Clauser, J. E. Vath, and P. A. Pevzner. De novo peptide sequencing via tandem mass spectrometry: A graph-theoretical approach. *J. Comp. Biol.*, 6(3/4):327–342, 1999.

[DD03] S. Datta and S. Datta. Comparisons and validation of statistical clustering techniques for microarray data. *Bioinformatics*, 19(4):459–466, 2003.

[Did03] G. Didier. Common intervals of two sequences. In *Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI'03)*, pages 17–24, 2003.

[Dub93] R. Dubes. Cluster analysis and related issues. In C. Chen, L. Pau, and P. Wang, editors, *Handbook of Pattern Recognition & Computer Vision*, pages 3–32. World Scientific, River Edge, NJ, 1993.

[EB94] M. Engle and C. Burks. GenFrag 2.1: new features for more robust fragment assembly benchmarks. *Computer Applications in the Biosciences*, 10(5):567–568, 1994.

[EL02] N. Edwards and R. Lippert. Generating peptide candidates from amino-acid sequence databases for protein identification via mass spectrometry. In *Proc. of the 2nd International Workshop on Algorithms in Bioinformatics (WABI'02)*, pages 68–81, 2002.

[ELP03] R. Eres, G. M. Landau, and L. Parida. A combinatorial approach to automatic discovery of cluster-patterns. In *Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI'03)*, pages 139–150, 2003.

[EMI94] J. K. Eng, A. L. McCormack, and J. R. Y. III. An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database. *Journal of the American Society for Mass Spectrometry (JASMS)*, 5:976–989, 1994.

[FGT+97] J. Fernandez-de-Cossio, J. Gonzalez, T. Takao, Y. Shimonishi, G. Padron, and V. Besada. A software program for the rapid sequence analysis of unknown peptides involving modifications, based on MS/MS data. In *45th ASMS Conference on Mass Spectrometry and Allied Topics, Slot 074*, 1997.

[FMM+89] J. Fenn, M. Mann, C. Meng, S. Wong, and C. Whitehouse. Electrospray ionisation for mass spectrometry of large biomolecules. *Science*, 246:64–71, 1989.

[Fre75] M. L. Fredman. Two applications of a probabilistic search technique: Sorting $X+Y$ and building balanced search trees. In *Conference Record of Seventh Annual ACM Symposium on Theory of Computing (STOC'75)*, pages 240–244, 1975.

[GG65] P. Gilmore and R. Gomory. Multi-stage cutting stock problems of two and more dimensions. *Oper. Res.*, 13:94–120, 1965.

[GKP94] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[Guy94] R. Guy. *Unsolved Problems in Number Theory*. Springer, 1994.

[HA85] L. Hubert and P. Arabie. Comparing partitions. *Journal of Classification*, 2:193–218, 1985.

[Haz04] S. Hazelhurst. An efficient implementation of the d^2 distance function for EST clustering: preliminary investigations. In *Proc. of the Annual Conference of the South African Institute of Computer Scientists and Information Theoreticians (SAICSIT'04)*, pages 1–6, 2004.

[HB03] S. Hazelhurst and A. Bergheim. ESTSim: A tool for creating benchmarks for EST clustering algorithms. Technical Report TR-Wits-CS-2003-1, School of Computer Science, University of the Witwatersrand, 2003.

[HKBC91] F. Hillenkamp, M. Karas, R. Beavis, and B. Chait. Matrix-assisted laser desorption/ionization mass spectrometry of biopolymers. *Analytical Chemistry*, 63:1193A–1203A, 1991.

[HMP+99] W. Hide, R. Miller, A. Ptitsyn, J. Kelso, C. Gopalkrishnan, and A. Christoffels. EST Clustering Tutorial, 1999.

[HPSS75] L. Harper, T. Payne, J. Savage, and E. Straus. Sorting $X + Y$. *Communications of the ACM*, 18(6):347–349, 1975.

[HSB+03] R. Hartmer, N. Storm, S. Böcker, C. P. Rodi, F. Hillenkamp, C. Jurinke, and D. van den Boom. RNAse T1 mediated base-specific cleavage and MALDI-TOF MS for high-throughput comparative sequence analysis. 31(9):e47, 2003.

[HSL+99] E. Hartuv, A. Schmitt, J. Lange, S. Meier-Ewert, H. Lehrach, and R. Shamir. An algorithm for clustering cDNAs for gene expression analysis. In *Proceedings of the 3rd Annual International Conference on Computational Molecluar Biology (RECOMB 99)*, pages 188–196. ACM, 1999.

[HWH86] C. Hamm, W. Wilson, and D. Harvan. Peptide sequencing program. *CABIOS*, 2(2):115–118, 1986.

[HWS03] W. J. Henzel, C. Watanabe, and J. T. Stults. Protein idendification: The origins of peptide mass fingerprints. *J. Am. Soc. Mass Spectr.*, 14:931–942, 2003.

[Int01] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409:860–921, 2001.

[JD88] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice Hall, 1988.

[JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[JU91] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In A. Tarlecki, editor, *Proceedings of the 16th Symposium on Mathematical Foundations of Computer Science. (MFCS '91)*, volume 520 of *LNCS*, pages 240–248, Berlin, Germany, 1991. Springer.

[KAKB03] A. Kalyanaraman, S. Aluru, S. Kothari, and V. Brendel. Efficient clustering of large EST data sets on parallel computers. *Nucleic Acids Research*, 31(11):2963–2974, 2003.

[KH88] M. Karas and F. Hillenkamp. Laser desorption ionisation of proteins with molecular masses exceeding 10.000 Daltons. *Analytical Chemistry*, 60:2299–2301, 1988.

[KS00] M. Kinter and N. E. Sherman. *Protein Sequencing and Identification Using Tandem Mass Spectrometry*. Wiley, 2000.

[KSV00] A. Krause, J. Stoye, and M. Vingron. The SYSTERS protein sequence cluster set. *Nucleic Acids Research*, 28(1):270–272, 2000.

[LC03a] B. Lu and T. Chen. A suboptimal algorithm for de novo peptide sequencing via tandem mass spectrometry. *Journal of Computational Biology*, 10(1):1–12, 2003.

[LC03b] B. Lu and T. Chen. A suffix tree approach to the interpretation of tandem mass spectra: Applications to peptides of non-specific digestion and post-translational modifications. *Bioinformatics, Supplement 2 (ECCB)*, pages ii113–ii121, 2003.

[Lev66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybern. Control Theory*, 10:707–710, 1966.

[LHP+00] F. Liang, I. Holt, G. Pertea, S. Karamycheva, S. Salzberg, and J. Quackenbush. An optimized protocol for analysis of EST sequences. *Nucleic Acids Research*, 26(18):3657–3665, 2000.

[LP85] D. Lipman and W. Pearson. Rapid and sensitive protein similarity searches. *Science*, pages 1435–1441, 1985.

[Lue75] G. S. Lueker. Two NP-complete problems in nonnegative integer programming. Technical Report TR-178, Department of Electrical Engineering, Princeton University, March 1975.

[lut] `http://www.immunex.com/researcher/lutefisk/`.

[mas] `http://www.matrixscience.com/`.

[MCJ03] K. Malde, E. Coward, and I. Jonassen. Fast sequence clustering using a suffix array algorithm. *Bioinformatics*, 19(10):1221–1226, 2003.

[MG02] K. Markides and A. Gräslund. Mass spectrometry (MS) and nuclear magnetic resonance (NMR) applied to biological macromolecules. Advanced information on the Nobel Prize in Chemistry 2002, October 2002.

[MHR93] M. Mann, P. Hoffsetjrup, and P. Roepstorff. Use of mass spectrometric molecular weight information to identify proteins in sequence databases. *Biol. Mass Spectrom.*, 22(6):338–345, 1993.

[MT90] S. Martello and P. Toth. *Knapsack Problems*. John Wiley and Sons, Chichester, 1990.

[MWW02] B. Moret, L.-S. Wang, and T. Warnow. Towards new software for computational phylogenetics. *IEEE Computer*, 35(7):55–64, 2002.

[MZH+03] B. Ma, K. Zhang, C. Hendrie, C. Liang, M. Li, A. Doherty-Kirby, and G. Lajoie. PEAKS: Powerful software for peptide de novo sequencing by MS/MS. 17(20):2337–2342, 2003.

[MZL03] B. Ma, K. Zhang, and C. Liang. An Effective Algorithm for the Peptide De Novo Sequencing from MS/MS Spectrum. In *Proc. of 14th Symposium on Combinatorial Pattern Matching (CPM'03)*, pages 266–277, 2003.

[MZL05] B. Ma, K. Zhang, and C. Liang. An effective algorithm for the peptide de novo sequencing from ms/ms spectrum. *Journal of Computer and System Sciences*, 70:418–430, 2005.

[Nav01] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[NBG01] H. Nash, D. Blair, and J. Greffenstette. Comparing algorithms for large-scale sequence analysis. In *Proc. of the Second IEEE International Symposium on Bioinformatics and Bioengineering*, pages 89–96. IEEE Computer Society Press, 2001.

[Nij79] A. Nijenhuis. A minimal-path algorithm for the "money changing problem". *Amer. Math. Monthly*, 86:832–835, 1979. Correction in *Amer. Math. Monthly*, 87:377, 1980.

[PDT00] P. A. Pevzner, V. Dančík, and C. L. Tang. Mutation-tolerant protein identification by mass-spectrometry. In R. Shamir, S. Miyano, S. Istrail, P. Pevzner, and M. Waterman, editors, *Proc. of the Fourth Annual International Conference on Computational Molecular Biology (RECOMB'00)*, pages 231–236. ACM Press, 2000.

[Pev00] P. A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. MIT Press, 2000.

[PHB93] D. Pappin, P. Hoffsetjrup, and A. Bleasby. Rapid identification of proteins by peptide-mass fingerprinting. *Current Biology*, 3(6):327–332, 1993.

[PHL+03] G. Pertea, X. Huang, F. Liang, V. Antonescu, R. Sultana, S. Karamycheva, Y. Lee, J. White, F. Cheung, B. Parvizi, J. Tsai, and J. Quackenbush. TIGR gene indices clustering tools (TGICL): a software system for fast clustering of large EST datasets. *Bioinformatics*, 19(5):651–652, 2003.

[PMDT01] P. A. Pevzner, Z. Mulyukov, V. Dančík, and C. L. Tang. Efficiency of database search for identification of mutated and modified proteins via mass spectrometry. *Genome Res.*, 11(2):290–299, 2001.

[PPCC99] D. N. Perkins, D. J. Pappin, D. M. Creasy, and J. S. Cottrell. Probability-based protein identification by searching sequence datases using mass spectrometry data. *Electrophoresis*, 20:3551–3567, 1999.

[QCL+01] J. Quackenbush, J. Cho, D. Lee, F. Liang, I. Holt, S. Karmycheva, B. Parviyi, G. Pertea, R. Sultana, and J. White. The TIGR gene indices: analysis of gene transcript sequences in higly sampled eukaryotic species. *Nucleic Acids Research*, 29(1):159–164, 2001.

[Qua01] J. Quackenbush. Computational analysis of microarray data. *Nature Reviews Genetics*, 2:418–427, 2001.

[Ram] J. L. Ramírez-Alfonsín. *The Diophantine Frobenius Problem*. Oxford University Press. To appear.

[Ram96] J. L. Ramírez-Alfonsín. Complexity of the Frobenius problem. *Combinatorica*, 16(1):143–147, 1996.

[RDPS+02] C. P. Rodi, B. Darnhofer-Patel, P. Stanssens, M. Zabeau, and D. van den Boom. A strategy for the rapid discovery of disease markers using the MassARRAY system. *BioTechniques*, 32:S62–S69, 2002.

[RSMpt] K. R. Rasmussen, J. Stoye, and G. Myers. Efficient q-gram filters for finding all ϵ-matches over a given length. Unpublished manuscript.

[Sal03] A. Salomaa. Counting (scattered) subwords. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 81:165–179, 2003.

[SEM98] J. Stoye, D. Evers, and F. Meyer. Rose: Generating sequence families. *Bioinformatics*, 14(2):157–163, 1998.

[seq] `http://fields.scripps.edu/sequest/`.

[Siu96] G. Siuzdak. *Mass Spectrometry for Biotechnology*. Academic Press, 1996.

[SJ01] W. Staudenmann and P. James. Interpreting peptide tandem mass-spectrometry fragmentation spectra. pages 143–165. 2001.

[SMMK84] T. Sakurai, T. Matsuo, H. Matsuda, and I. Katakuse. PAAS 3: A computer program to determine probable sequence of peptides from mass spectrometric data. *Biomedical Mass Spectrometry*, 11(8):396–399, 1984.

[SS04] T. Schmidt and J. Stoye. Quadratic time algorithms for findig common intervals in two and more sequences. In *Proc. of the Fifteenth Annual Combinatorial Pattern Matching Symposium (CPM'04), Istanbul July 5-7, 2004*, volume 3109 of *LNCS*, pages 347–358, 2004.

[TBDS90] D. Torney, C. Burks, D. Davison, and K. M. Sirotkin. Computation of d^2: A measure of sequence dissimilarity. In G. Bell and T. Marr, editors, *Computers and DNA*, pages 109–125. Addison-Wesley, 1990.

[The] The R Foundation for Statistical Computing. Available at `http://www.r-project.org/`.

[TJ97] J. A. Taylor and R. S. Johnson. Sequence database searches via de novo peptide sequencing by tandem mass spectrometry. *Rapid Comm. Mass Spec.*, 11:1067–1075, 1997.

[TJ01] J. A. Taylor and R. S. Johnson. Implementation and uses of automated de novo peptide sequencing by tandem mass spectrometry. *Anal. Chem.*, 73:2594–2604, 2001.

[Ukk92] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92:191–211, 1992.

[VA03] S. Vinga and J. Almeida. Alignment-free sequence comparison—a review. *Bioinformatics*, 19(3):513–524, 2003.

[VAM+01] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, J. D. Gocayne, P. Amanatides, R. M. Ballew, D. H. Huson, J. R. Wortman, Q. Zhang, C. Kodira, X. H. Zheng, L. Chen, M. Skupski, G. Subramanian, P. D. Thomas, J. Zhang, G. L. Gabor Miklos, C. Nelson, S. Broder, A. G. Clark, J. Nadeau, V. A. McKusick, N. Zinder, A. J. Levine, R. J. Roberts, M. Simon, C. Slayman, M. Hunkapiller, R. Bolanos, A. Delcher, I. Dew, D. Fasulo, M. Flanigan, L. Florea, A. Halpern, S. Hannenhalli, S. Kravitz, S. Levy, C. Mobarry, K. Reinert, K. Remington, J. Abu-Threideh, E. Beasley, K. Biddick, V. Bonazzi, R. Brandon, M. Cargill, I. Chandramouliswaran, R. Charlab, K. Chaturvedi, Z. Deng, V. Di Francesco, P. Dunn, K. Eilbeck, C. Evangelista, A. E. Gabrielian, W. Gan, W. Ge, F. Gong, Z. Gu, P. Guan, T. J. Heiman, M. E. Higgins, R. R. Ji, Z. Ke, K. A. Ketchum, Z. Lai, Y. Lei, Z. Li, J. Li, Y. Liang, X. Lin, F. Lu, G. V. Merkulov, N. Milshina, H. M. Moore, A. K. Naik, V. A. Narayan, B. Neelam, D. Nusskern, D. B. Rusch, S. Salzberg, W. Shao, B. Shue, J. Sun, Z. Wang, A. Wang, X. Wang, J. Wang, M. Wei, R. Wides, C. Xiao, C. Yan, A. Yao, J. Ye, M. Zhan, W. Zhang, H. Zhang, Q. Zhao, L. Zheng, F. Zhong, W. Zhong, S. Zhu, S. Zhao, D. Gilbert,

S. Baumhueter, G. Spier, C. Carter, A. Cravchik, T. Woodage, F. Ali, H. An, A. Awe, D. Baldwin, H. Baden, M. Barnstead, I. Barrow, K. Beeson, D. Busam, A. Carver, A. Center, M. L. Cheng, L. Curry, S. Danaher, L. Davenport, R. Desilets, S. Dietz, K. Dodson, L. Doup, S. Ferriera, N. Garg, A. Gluecksmann, B. Hart, J. Haynes, C. Haynes, C. Heiner, S. Hladun, D. Hostin, J. Houck, T. Howland, C. Ibegwam, J. Johnson, F. Kalush, L. Kline, S. Koduru, A. Love, F. Mann, D. May, S. McCawley, T. McIntosh, I. McMullen, M. Moy, L. Moy, B. Murphy, K. Nelson, C. Pfannkoch, E. Pratts, V. Puri, H. Qureshi, M. Reardon, R. Rodriguez, Y. H. Rogers, D. Romblad, B. Ruhfel, R. Scott, C. Sitter, M. Smallwood, E. Stewart, R. Strong, E. Suh, R. Thomas, N. N. Tint, S. Tse, C. Vech, G. Wang, J. Wetter, S. Williams, M. Williams, S. Windsor, E. Winn-Deen, K. Wolfe, J. Zaveri, K. Zaveri, J. F. Abril, R. Guigo, M. J. Campbell, K. V. Sjolander, B. Karlak, A. Kejariwal, H. Mi, B. Lazareva, T. Hatton, A. Narechania, K. Diemer, A. Muruganujan, N. Guo, S. Sato, V. Bafna, S. Istrail, R. Lippert, R. Schwartz, B. Walenz, S. Yooseph, D. Allen, A. Basu, J. Baxendale, L. Blick, M. Caminha, J. Carnes-Stine, P. Caulk, Y. H. Chiang, M. Coyne, C. Dahlke, A. Mays, M. Dombroski, M. Donnelly, D. Ely, S. Esparham, C. Fosler, H. Gire, S. Glanowski, K. Glasser, A. Glodek, M. Gorokhov, K. Graham, B. Gropman, M. Harris, J. Heil, S. Henderson, J. Hoover, D. Jennings, C. Jordan, J. Jordan, J. Kasha, L. Kagan, C. Kraft, A. Levitsky, M. Lewis, X. Liu, J. Lopez, D. Ma, W. Majoros, J. McDaniel, S. Murphy, M. Newman, T. Nguyen, N. Nguyen, M. Nodell, S. Pan, J. Peck, M. Peterson, W. Rowe, R. Sanders, J. Scott, M. Simpson, T. Smith, A. Sprague, T. Stockwell, R. Turner, E. Venter, M. Wang, M. Wen, D. Wu, M. Wu, A. Xia, A. Zandieh, and X. Zhu. The sequence of the human genome. *Science*, 291:1304–1351, 2001.

[WDRP02] N. Wicker, D. Dembele, W. Raffelsberger, and O. Poch. Density of points clustering, application to transcriptomic data analysis. *Nucleic Acids Research*, 30(18):3992–4000, 2002.

[WDYF85] C. Whitehouse, R. Dreyer, M. Yamashita, and J. Fenn. Electrospray interface for liquid chromatographs and mass spectrometers. *Analytical Chemistry*, 57:675–679, 1985.

[Wil90] H. Wilf. *generatingfunctionology*. Academic Press, 1990.

[YHR01] K. Yeung, D. Haynor, and W. Ruzzo. Validating clustering for gene expression data. *Bioinformatics*, 17(4):309–318, 2001.

[YI98] J. R. Yates III. Database searching using mass spectrometry data. *Electrophoresis*, 19(6):893–900, 1998.

[YIEM95] J. R. Yates III, J. K. Eng, and A. L. McCormack. Mining genomes: Correlating tandem mass-spectra of modified and unmodified peptides to sequences in nucleotide databases. *Anal. Chem.*, 67(18):3202–3210, 1995.

[YIEMS95] J. R. Yates III, J. K. Eng, A. L. McCormack, and D. Schieltz. Method to correlate tandem mass spectra of modified peptides to amino acid sequences in the protein database. *Anal. Chem.*, 67(8):1426–1436, 1995.

[Zim03] J. Zimmermann. Suitability Comparison of String Distance Measures for EST Clustering. Master's thesis, ETH Zurich, Dept. of Computer Science, Sept. 2003.

[ZLH04] J. Zimmermann, Zs. Lipták, and S. Hazelhurst. A method for evaluating the quality of string dissimilarity measures and clustering algorithms for EST clustering. In *Proc. of IEEE Fourth Symposium on Bioinformatics and Bioengineering (BIBE'04)*, pages 301–309, 2004.

Appendix: List of Publications

Publications in Refereed Journals

- Nikhil Bansal, Mark Cieliebak, and Zsuzsanna Lipták: Finding Submasses in Weighted Strings with Fast Fourier Transform. Discrete Applied Mathematics, Special Issue on Computational Biology, to appear.

- Jonas Grossmann, Franz Felix Roos, Mark Cieliebak, Riko Jacob, Zsuzsanna Lipták, Lucas K. Mathis, Matthias Müller, Peter Widmayer, Wilhelm Gruissem, and Sacha Baginsky: AuDeNS - A tool for automated peptide de novo sequencing. Journal of Proteome Research, to appear.

- Mark Cieliebak, Thomas Erlebach, Zsuzsanna Lipták, Jens Stoye, and Emo Welzl: Algorithmic Complexity of Protein Identification: Combinatorics of Weighted Strings. Discete Applied Mathematics, Special Issue on Combinatorics of Searching, Sorting, and Coding, Vol. 137/1: 27-46 (2004).

Publications in Refereed Conference Proceedings

- Sebastian Böcker and Zsuzsanna Lipták: The Money Changing Problem revisited: Computing the Frobenius number in time $O(ka_1)$. Proceedings of the Eleventh International Computing and Combinatorics Conference (COCOON'05): 965-974 (2005).

- Sebastian Böcker and Zsuzsanna Lipták: Efficient Mass Decomposition. Proceedings of the ACM Symposium on Applied Computing (ACM-SAC'05): 151-157 (2005).

- Nikhil Bansal, Mark Cieliebak, and Zsuzsanna Lipták: Efficient Algorithms for Finding Submasses in Weighted Strings. Proceedings of the Fifteenth Annual Combinatorial Pattern Matching Symposium (CPM'04): 194-204 (2004).

- Judith Zimmermann, Zsuzsanna Lipták, and Scott Hazelhurst: A Method for Evaluating the Quality of String Dissimilarity Measures and Clustering Algorithms for EST Clustering. Proceedings of IEEE Fourth Symposium on Bioinformatics and Bioengineering (BIBE'04): 301-309 (2004).

- Mark Cieliebak, Thomas Erlebach, Zsuzsanna Lipták, Jens Stoye, and Emo Welzl: Algorithmic Complexity of Protein Identification: Searching in Weighted Strings. Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science (TCS'02): 143-156 (2002).

- Zsuzsanna Lipták and Arfst Nickelsen: Broadcasting in Complete Networks with Dynamic Edge Faults. Proceedings of the 4th International Conference on Principles of Distributed Systems (OPODIS'00), Studia Informatica Universalis: 123-142 (2000).

Technical Reports

- Sebastian Böcker and Zsuzsanna Lipták: The Money Changing Problem revisited: Computing the Frobenius number in time $O(ka_1)$. Technical Report no. 2004-02, Technical Faculty, Bielefeld University (June 2004).
- Scott Hazelhurst, Zsuzsanna Lipták, Judith Zimmermann: A Comparative Study of Biological Distances for EST Clustering. Technical Report TR-Wits-CS-2003-3, School of Computer Science, University of the Witwatersrand, Johannesburg, South Africa (May 2003).
- Sacha Baginsky, Mark Cieliebak, Wilhelm Gruissem, Torsten Kleffmann, Zsuzsanna Lipták, Matthias Müller, Paolo Penna: AuDeNS - A tool for de novo peptide sequencing. Technical Report no. 383, Dept. of Computer Science, ETH Zurich (Oct. 2002).
- Mark Cieliebak, Thomas Erlebach, Zsuzsanna Lipták, Jens Stoye, Emo Welzl: Algorithmic Complexity of Protein Identification: Combinatorics of Weighted Strings. Technical Report no. 361, Dept. of Computer Science, ETH Zurich (Aug. 2001).

Posters

- Sebastian Böcker, Michael Kaltenbach, Zsuzsanna Lipták: Algorithms for Interpreting Mass Spectrometry Data. German Conference on Bioinformatics (GCB'04), Bielefeld, Germany (2004).

Diplom Thesis (German Masters)

Zsuzsanna Lipták: Zur algebraischen Charakterisierung regulärer Termsprachen (On the Algebraic Characterization of Regular Term Languages, in German). Freie Universität Berlin, Fachbereich Mathematik (Dec. 1998).