Algorithmic Complexity of Protein Identification: Combinatorics of Weighted Strings

TECHNICAL REPORT no. 361, ETH Zurich, Dept. of Computer Science

Mark Cieliebak[*], Thomas Erlebach[†], Zsuzsanna Lipták[*], Jens Stoye[‡], Emo Welzl[*]

August 13, 2001

Abstract

We investigate a problem from computational biology: Given a constant–size alphabet $\mathcal{A}$ with a weight function $\mu : \mathcal{A} \to \mathbb{R}^+$, find an efficient data structure and query algorithm solving the following problem: For a weight $M \in \mathbb{R}^+$ and a string σ over $\mathcal{A}$, decide whether σ contains a substring with weight M (One–String Mass Finding Problem). If the answer is **yes**, then we may in addition require a witness, i.e. indices $i \leq j$ such that the substring beginning at position i and ending at position j has weight M. We allow preprocessing of the string, and measure efficiency in two parameters: storage space required for the preprocessed data, and running time of the query algorithm for given M. We are interested in data structures and algorithms requiring subquadratic storage space and sublinear query time, where we measure the input size as the length of the input string. We present two efficient algorithms: Lookup solves the problem with $O(n)$ space and $O(\frac{n}{\log n} \cdot \log\log n)$ time; Interval solves the problem for binary alphabets with $O(n)$ space in $O(\log n)$ time. We sketch a third algorithm, Cluster, which can be adjusted for a space–time–tradeoff but for which we do not yet have a resource analysis. We introduce a function on weighted strings which is closely related to the analysis of algorithms for the One–String Mass Finding Problem: The number of different submasses of a weighted string. We present several properties of this function, including upper and lower bounds. Finally, we introduce two more general variants of the problem and sketch how algorithms may be extended for these variants.

[*]ETH Zurich, Inst. for Theor. Computer Science; {cielieba, zsuzsa, emo}@inf.ethz.ch
[†]ETH Zurich, Computer Eng. and Networks Lab.; erlebach@tik.ee.ethz.ch
[‡]Max Planck Institute for Molecular Genetics, Berlin; stoye@molgen.mpg.de

1 Introduction

In the present paper, we introduce a combinatorial problem which originates from computational biology: Given a string σ over a weighted alphabet $\mathcal{A}$, find a data structure and an algorithm which, for a given weight $M \in \mathbb{R}^+$, decides whether σ has a (contiguous) substring of weight M. If the answer is **yes**, we may in addition ask for a *witness*, i.e. two positions within σ where a substring with weight M begins and ends. The actual problem in computational biology is to find several masses $M_1, \ldots, M_m$ in a database of strings. We concentrate on the one–string problem because algorithms can be extended easily to the multiple–string problem. We formally define the other problem variants at the end of the paper and sketch how extensions may be designed. There are two simple algorithms which will solve the one–string problem: One uses linear time and no additional storage space, both measured in the length of the string; the other has logarithmic running time, but requires additional storage space which may be quadratic. We are interested in data structures and algorithms which are better than these two, i.e. which need subquadratic space *and* sublinear time.

Formulated in this way, the problem becomes a purely combinatorial and algorithmic problem: Are there data structures and algorithms which allow searching in weighted strings of size n with $o(n^2)$ additional storage space and $o(n)$ time? If so, can we find a tradeoff between space and time?

The problem differs from traditional string searching problems in one important aspect: While those look for substructures of strings (substrings, non–contiguous subsequences, particular types of substrings such as repeats, palindromes etc.), we are interested only 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 hand, we gain the additional structure of the weight function, such as its additivity. For instance, the problem of searching in $X + Y$, where X and Y are two sets of numbers, turns out to be closely related to our problem (see [Fre75] and [HPSS75]). However, we have been able to extend negative results which have been reached for that problem ([CDF90]) to show that that approach using the naïve solution without preprocessing cannot lead to an efficient algorithm for our problem. The only result related to our problem which we have found in the vast amount of literature on strings (e.g. [AG95], [Gus97], [Lot97], [RS97], [CR94]) is one which does not deal with combinatorics, but rather with language classes (see Section 5 for more details).

Our problem is positioned between the areas of string algorithms, search algorithms, and algebra. We believe that it is not only relevant for computational biology, but that it is also of theoretical interest to the field of combinatorial searching. As far as we are aware, no efficient algorithms have so far been presented for this problem.

We would like to stress at this point that, even though the source of our problem is a biological question, the results we present here are primarily of theoretical interest. The reason is twofold: First, none of the algorithms we present are really efficiently applicable in their current form. LOOKUP requires sublinear time, but the asymptotic improvement over a linear time algorithm is not very large (a factor of $\frac{\log \log n}{\log n}$); how superior it actually is over a linear–time algorithm will have to be tested in experiments. Algorithm INTERVAL is very efficient both in time and space, but it only works for alphabets of size 2, a case which never occurs in the biological setting. The second reason is that all biological data are prone to errors; in fact, there is no such thing as error–free data. Thus, all application in computational biology needs to be highly fault tolerant. Our algorithms as presented here are not fault tolerant, even though they can be adapted to become tolerant to measurement errors. However, this aspect is not included in the present paper.

1.1 Biological Motivation

Proteomics is the field which investigates the nature and function of proteins. As in molecular biology in general, large amounts of data are being accumulated at present, which presents particular computational and mathematical challenges.

Proteins are large molecules which play a fundamental role in all living organisms. They are made up of smaller molecules (amino acids) which are linked together in a certain order. 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 in humans are made up of the 20 most common amino acids. For the purposes of this paper, we will view a protein as a finite string over an alphabet of size 20.

The information about known proteins is stored in large databases, such as SWISS-PROT (nearly $100,000$ proteins) or PIR (more than $200,000$ sequences). When a protein is isolated, one would like to know whether it is already known and if so, identify it. An obvious way is to establish its primary structure: This is called *de novo protein sequencing.* However, protein sequencing, unlike DNA sequencing, is very expensive (both in time and money!). E.g. identifying *one* amino acid with Edman Degradation, one standard method for protein sequencing, takes about 45 minutes, which makes this approach unfeasible in a high–throughput context.

Therefore, methods are required which test the protein against a database without having to sequence it first. One such method—which we will investigate here—makes use of the differences in molecular weights of amino acids: The protein is broken up into smaller pieces and these pieces are then weighed[1], using a mass spectrometer. This will yield a "fingerprint" of the protein which can then be tested against the database: The goal is to find a protein in the database which has substrings matching each of the input masses.

There are two basic methods for breaking up the protein into smaller pieces: Random fragmentation (the protein will break at non–prespecified points), or the technique of using a cleavage agent (such as an enzyme, e.g. trypsin), which literally cuts the protein in certain well–defined places. The latter method is referred to by molecular biologists as *digestion*. Using digestion is algorithmically rather simple, since the breakup points are known in advance; it is thus possible to preprocess the database in an appropriate way. There is a large amount of literature on this approach ([YISGH93, HBS+93, JQCG93, PHB93, MHR93, EMYI94]), some papers dealing with different aspects and modifications of the problem, e.g. the minimum number of masses needed to identify a protein ([PHB93]), combinatoric ([PDT00]) or probabilistic ([BE01]) models for scoring the difference of two mass spectra, or approaches for a correct identification even in the presence of posttranslational modifications of the protein ([MW94, YEM95, PMDT01]). The review [YI98] as well as chapter 11 of the book [Pev00] contain more detailed introductions to this topic. For an introduction to computational biology in general, see [SM97]; for more on molecular biology, [Str88]; while [GW91] is an easy–going introduction to genetics for non–biologists.

In this paper, we deal with algorithmic questions of the random fragmentation method. Since we never make any assumptions about the probability distribution of breaking points, any algorithm for the random fragmentation method can be used for digested inputs, too. Testing by random weights is algorithmically far more complex than the digestion method, because the cutting places are not known in advance. In the long run, for the biological application, algorithms are needed which are not only efficient, but also fault tolerant: They need to be tolerant both to measurement errors ($M \pm \epsilon$;

[1] Biologists will excuse some rough simplifications.

missing or additional masses in the spectrum), and to sequencing errors. Most of the algorithms presented can be adapted easily for the first type of fault tolerance, but this needs to be treated in detail; we have not dealt with the second type of error at all so far.

1.2 Overview

The paper is organized as follows. We first introduce the problem and all necessary definitions in Section 2, where we also present some simple ideas which motivate our efficiency requirements. In Section 3, we design an algorithm (LOOKUP) that solves the problem efficiently, with linear storage space and sublinear running time. Section 4 contains an algorithm (INTERVAL) which solves the problem for alphabets of size 2 and has a very good performance. However, we do not think that it can be generalized to larger alphabets. Section 5 deals with combinatorics of weighted strings; we introduce a function on weighted strings, the number of different submasses, and present upper and lower bounds as well as some other results. In Section 6, we presents two other problem variants and discuss how algorithms for the original problem can be extended to these. Finally, in Section 7, we sketch ongoing work. In particular, we present another algorithm (CLUSTER) which has shown a good performance in experiments, but for which we do not yet have a resource analysis.

A note on vocabulary: As in all work at the boundary of two disciplines, we are unable to avoid using both sets of terms. We will thus use the terms *sequence* (biology) and *string* (mathematics) interchangeably, as well as the terms *mass* and *weight*.

2 Problem and Simple Solutions

Fix an *alphabet* $\mathcal{A}$ of size $|\mathcal{A}| = s$ and a *mass function* $\mu : \mathcal{A} \to \mathbb{R}^+$. The *mass* (or the *weight*) of a string (or a *sequence*) σ over $\mathcal{A}$ is defined as the sum of the individual masses $\mu(\sigma) := \sum_{i=1}^{n} \mu(\sigma(i))$, where $\sigma(i)$ denotes the i'th letter of σ, and $n = |\sigma|$ is the length of σ. For a mass $M \in \mathbb{R}^+$ and a string σ of length n, we say that M is a *submass* of σ if σ has a substring of mass M, i.e. if there are indices $1 \leq i \leq j \leq n$ s.t. $\mu(\sigma(i,j)) = M$, where $\sigma(i,j)$ is the substring of σ starting with $\sigma(i)$ and ending with $\sigma(j)$. For $a \in \mathcal{A}$, let us denote the multiplicity of a in σ by $|\sigma|_a := |\{i \mid \sigma(i) = a\}|$.

The ONE–STRING MASS FINDING PROBLEM is defined as follows:

> Given a string σ of length $|\sigma| = n$ and a mass M, is M a submass of σ?

Hereby, preprocessing of the string is allowed, because we are interested in repeating the query for many different M's. We are looking for data structures and query algorithms where storage space and query time are good, in a sense to be specified later.

There are two simple algorithms to solve the problem. The first one, LINSEARCH, does not take advantage of data preprocessing. It simply performs a linear search through the string: For given σ, start at position $\sigma(1)$ and add up masses until reaching the first position j s.t. $\mu(\sigma(1,j)) \geq M$. If the mass of the substring $\sigma(1,j)$ equals M, then output **yes** and stop; else start subtracting masses from the beginning of the string until the smallest index i s.t. $\mu(\sigma(i,j)) \leq M$ is reached. Repeat until finding a pair of indices (i,j) s.t. $\mu(\sigma(i,j)) = M$, or until reaching the end of the string (i.e. until the current substring is $\sigma(i,n)$ for some i, and $\mu(\sigma(i,n)) < M$). The algorithm can be visualized as shifting two pointers ℓ and r through the string, where ℓ points to the beginning of the current substring, and r to its end. LINSEARCH takes $O(n)$ time, since it looks at each letter at most twice. It requires no preprocessing, and it uses storage space $O(1)$ in addition to the space needed by the input sequence.

Example 1. Let $\mathcal{A} = \{a,b,c\}, \mu(a) = 1, \mu(b) = 2, \mu(c) = 5$. Let $\sigma = abbcabccaabb$. A linear search for mass[2] $M = 14$ would first shift r up to position 7, where the current submass first exceeds M: it now equals 18. Then the left pointer is moved to position 4, the current submass now being 13, etc. until reaching positions 5 and 9 (see figure 1).

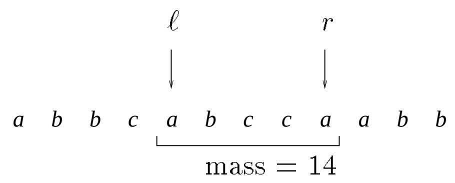


Figure 1: Example 1 – Linear search for $M = 14$

The other simple algorithm calculates and sorts all submasses of σ in a preprocessing step and stores them in a sorted array. Given a query mass M, we perform binary search for M in the array. We will refer to this algorithm as BINSEARCH. Required storage space is proportional to the

[2]Note that in general a mass can be a real number (of course, approximated by a rational number). However, we will use only integers in our examples for reasons of simplicity.

number of different submasses in σ, which we will investigate in Section 5. This number $\text{Klaus}(\sigma)$ is $O(n^2)$. The time for answering a query is $O(\log n)$.

We are looking for data structures and query algorithms for the ONE–STRING MASS FINDING PROBLEM that are better than LINSEARCH and BINSEARCH, i.e. require additional storage space $o(n^2)$, and query time $o(n)$. We will call such data structures *skinny*, and such query algorithms *speedy*. Note that we ignore time and space required during the preprocessing step, as long as these are in reasonable bounds. The reason is that we are able to use large resources for obtaining the data structure to be used, because the preprocessing will be done only once, while the query step will typically be repeated many times. We also ignore space required by the query algorithm, because this is bounded by the running time, which is sublinear for a speedy algorithm.

In this context, the question naturally arises whether a given mass M can be the weight of a string. This question can be solved with a simple Integer Linear Program.

We define the *length* of a mass M as $\lambda(M) := \max(\{|\tau| \mid \tau \in \mathcal{A}^*, \mu(\tau) = M\} \cup \{-1\})$. Here, $\lambda(M) = -1$ means that there is no string with mass M. Suppose that we know in advance that all query masses are short in comparison to n, i.e. there is a function $f(n)$ such that $\lambda(M) \leq f(n) = o(n)$ for all queries M. Then there are two approaches to improve BINSEARCH: First, we store all submasses of σ of length $\ell \leq f(n)$ in a sorted array. This requires storage space $O(n \cdot f(n))$, since for each position i in σ, at most $f(n)$ substrings of length $\ell \leq f(n)$ start in i. For a query, we do binary search in this array. This takes time $O(\log n)$, which is speedy. Since $f(n) = o(n)$, the algorithm is skinny, too. The second approach works as follows. Since the alphabet is of constant size, there are at most $(|\mathcal{A}|+1)^{f(n)}$ different strings of length at most $f(n)$. This is independent of σ. We store a sorted array of all masses of these strings and note for each mass whether it is a submass of σ. Given a mass M, we can perform binary search on this array. This requires storage space $O(|\mathcal{A}|^{f(n)})$ and time $O(\log |\mathcal{A}|^{f(n)}) = O(f(n))$. For $f(n)$ sufficiently small (e.g. $f(n) = c \log n$, for a small constant $c < \frac{1}{\log \mathcal{A}}$), this is both skinny and speedy.

These two approaches perform well only if $f(n)$ is small. In the rest of the paper we will deal with the more general problem where $f(n)$ is not restricted. We can combine these two approaches and use them to improve other algorithms in the sense that they will run faster on short masses.

3 An Algorithm that is Both Skinny and Speedy

In this section, we present algorithm LOOKUP that solves the ONE–STRING MASS FINDING PROBLEM using space $O(n)$ and time $O(\frac{n}{\log n} \cdot \log\log n)$. The idea is as follows. Similar to the simple linear search algorithm LINSEARCH introduced in Section 2, LOOKUP shifts two pointers along the sequence which point to the potential beginning and end of a substring with mass M. However, $c(n)$ steps of the simple algorithm are bundled into one step here. If $c(n)$ is chosen appropriately, i.e. approximately $\log n$, then this will reduce the number of steps from $O(n)$ to $O(\frac{n}{\log n})$, while each step will now require $O(\log\log n)$ time instead of constant time. We will hereby heavily exploit the fact that the alphabet has constant size.

3.1 An Example

Example 2. Let $\mathcal{A} = \{a, b, c\}, \mu(a) = 1, \mu(b) = 2, \mu(c) = 5$. Let us assume that we are looking for $M = 14$ in $\sigma = abbcabccaabb$. LINSEARCH would shift two pointers ℓ and r through the sequence, until reaching positions 5 and 9 respectively, where it would stop because the substring $\sigma(5, 9) = abcca$ has weight 14 (see Example 1). Let us assume that $c(n) = 3$. We divide the sequence σ into blocks of size $c(n)$. Now rather than shifting the two pointers letter by letter, we will shift them by a complete block at a time. In order to do this, we store, for each block, a pointer to an index I which corresponds to the substring within the block. Let us assume now that ℓ is at the beginning of the first block, and r is at the end of the second block, as indicated in Figure 2. We are interested in the possible *changes* to the current submass if we shift the two pointers at most $c(n)$ to the right. Given a list of these, we could search for $M - \sigma(\ell, r)$. For example, the current submass in Figure 2 is $\sigma(1, 6) = 13$, and we want to know whether by moving ℓ and r at most 3 positions to the right, we can achieve a change of $14 - 13 = 1$.

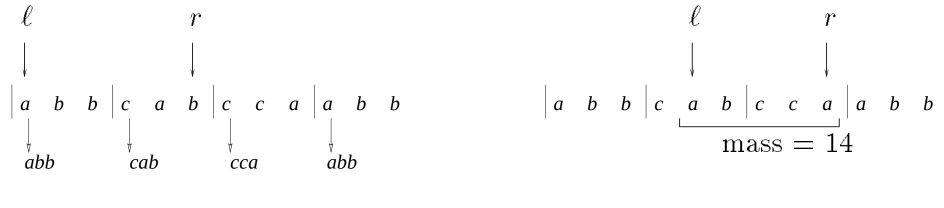


Figure 2: Example 2 – LOOKUP searching for $M = 14$

We can calculate these possible changes and store them in a $(c(n)+1) \times (c(n)+1)$ matrix T whose (i,j)–entry holds the submass change when ℓ is moved $i-1$ positions to the right, and r is moved $j-1$ positions to the right:

$$T[abb, cca] = \begin{pmatrix} 0 & 5 & 10 & 11 \\ -1 & 4 & 9 & 10 \\ -3 & 2 & 7 & 8 \\ -5 & 0 & 5 & 6 \end{pmatrix}$$

In order to be able to do fast search, we store the entries of the matrix in a sorted array: $S[abb, cca] = [-5, -3, -1, 0, 2, 4, 5, 6, 7, 8, 9, 10, 11]$. Now we can find out in time $O(\log(\text{size of array}))$ whether the difference we are looking for is there. In the present case, 1 is not in the array, which tells us that we have to move one of the two pointers to the next block.

To determine which pointer to move, we consider what the linear search algorithm LINSEARCH would do when searching for M and starting in the current positions of the left resp. right pointers. Since M is not present within these two blocks, at least one of the two pointers would reach the end of its current block. Here, we want to move the pointer which first reaches the end of its block. We can determine which pointer this is if we compare the difference $M - \sigma(\ell, r)$ with the matrix entry $T(c(n), c(n))$ corresponding to $c(n) - 1$ moves of both the left and the right pointer (in this case 7). If the difference is smaller, we move the left pointer to the next block, otherwise we move the right one. In our example, we have a difference of 1, thus we move the left pointer to the next block.

This will change the current submass by -5 (the minimum of the array), yielding $\sigma(4,6) = 13 - 5 = 8$. Thus, we now look for $M - \sigma(4,6) = 14 - 8 = 6$. The sorted array for this pair of positions is $S[cab, cca] = [-8, -6, -5, -3, -1, 0, 2, 3, 4, 5, 6, 10, 11]$, and the matrix is as follows:

$$T[cab, cca] = \begin{pmatrix} 0 & 5 & 10 & 11 \\ -5 & 0 & 5 & 6 \\ -6 & -1 & 4 & 5 \\ -8 & -3 & 2 & 3 \end{pmatrix}$$

6 is in the array: By looking in the matrix, we can see that a difference of 6 can be achieved by moving the left pointer by 1 position and the right pointer by 3 positions. The algorithm outputs positions 5 and 9 and then terminates.

3.2 Algorithm Lookup

We postpone the exact choice of the function $c(n)$ to the analysis, but assume for now that it is approximately $\log n$. For simplicity, we assume that $c(n)$ is a divisor of n.

Preprocessing: Given σ of length n, first compute $c(n)$. Next, build a table T of size $|\mathcal{A}|^{c(n)} \times |\mathcal{A}|^{c(n)}$. Each row resp. column of T will be indexed by a string from $\mathcal{A}^{c(n)}$. For $I, J \in \mathcal{A}^{c(n)}$, the table entry $T[I, J]$ contains the matrix and the sorted array as described above. The matrix contains all differences $\mu(\text{prefix}(J)) - \mu(\text{prefix}(I))$, including the empty prefixes. Note that the table T depends only on n and $\mathcal{A}$, and not on the sequence σ itself. Next, divide σ into blocks of length $c(n)$. For each block, store a pointer to an index I that we will use to look up table T. Each such index I represents one string from $\mathcal{A}^{c(n)}$.

Query Algorithm: Given M, set $\ell := 1$ and $r := 0$. Repeat the following steps until M has been found or $r > n$:

1. Say ℓ is set to the beginning of the i'th block and r to the end of the $(j-1)$'th block. Then look in the sorted array in $T(I, J)$ where the pointer of block i resp. j points to index I resp. J. Find whether $M - \sigma(\ell, r)$ is in the array with binary search.

2. If $M - \sigma(\ell, r)$ is in the array, search for an entry (k, l) in the matrix $T(I, J)$ which equals $M - \sigma(\ell, r)$ by exhaustive search[3], and return **yes**, along with the witness $i' := i \cdot c(n) + k, j' := j \cdot c(n) + (l - 1)$, since $\sigma(i', j')$ has mass M.

3. If $M - \sigma(\ell, r)$ is not in the array and if $M - \sigma(\ell, r)$ is less than the matrix entry at position $(c(n), c(n))$, then increment ℓ by $c(n)$ and set $\sigma(\ell, r) := \sigma(\ell, r) + \min(\text{array})$; otherwise, increment r by $c(n)$ and set $\sigma(\ell, r) := \sigma(\ell, r) + \max(\text{array})$.

Analysis: First we derive formulas for space and time, and then we show how to choose $c(n)$. The space needed for storing table T is:

$$\begin{aligned}
&\text{number of entries} \cdot (\text{size of matrix} + \text{size of sorted array}) \\
&= |\mathcal{A}|^{2c(n)} \cdot ((c(n)+1)^2 + (c(n)+1)^2) \\
&= O(|\mathcal{A}|^{2c(n)} \cdot c(n)^2).
\end{aligned}$$

[3] Alternatively, we could have stored (k, l) during the preprocessing in the sorted array.

Space needed for storing the pointer at each block is:

$$\begin{aligned}
&\text{number of blocks} \cdot \log(\text{number of elements in } \mathcal{A}^{c(n)}) \\
&= \frac{n}{c(n)} \cdot \log(|\mathcal{A}|^{c(n)}) \\
&= O(n).
\end{aligned}$$

For the query time, observe that after each iteration (consisting of Steps 1 to 3), either ℓ or r is advanced to the next block. As each of the pointers can advance at most $\frac{n}{c(n)}$ times, there can be at most $2\frac{n}{c(n)}$ iterations. Each iteration except the last one takes time $O(\log c(n)^2) + O(1)$. The last iteration may take time $O(c(n)^2)$.

In total, the algorithm requires storage space $O(n + |\mathcal{A}|^{2c(n)} \cdot c(n)^2)$ and time $O(\frac{n}{c(n)} \log c(n) + \frac{n}{c(n)} + c(n)^2)$. If we choose $c(n) := \frac{\log_{|\mathcal{A}|} n}{4}$, then we obtain $|\mathcal{A}|^{c(n)} = n^{\frac{1}{4}}$. This yields a storage space of $O(n + n^{\frac{1}{2}} \cdot \log^2 n) = O(n)$ and time $O(\frac{n}{\log n} \log \log n)$, which is both skinny and speedy. Other choices of $c(n)$ do not asymptotically improve time and speed at the same time.

Theorem 1. *Algorithm* LOOKUP *solves the* ONE–STRING MASS FINDING PROBLEM *with storage space $O(n)$ and query time $O(\frac{n}{\log n} \log \log n)$.*

4 A Speedier Algorithm for Binary Alphabets

In this section, we present algorithm INTERVAL which solves the ONE–STRING MASS FINDING PROBLEM for an alphabet of size 2. It runs in time $O(\log n)$ and uses space $O(n)$. The algorithm *decides* whether a given mass is a submass of σ, but does not return a witness.

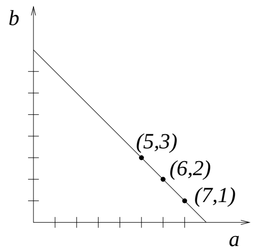
Let σ be a string over $\mathcal{A} := \{a, b\}$ and fix $k \leq n$. Observe that, when sliding a window of size k over σ, then in one step, the multiplicities of a and b within the window change at most by one. We represent substrings of σ by points in the $\mathbb{Z} \times \mathbb{Z}$ lattice, where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

$$S_k := \{(i, j) \in \mathbb{Z}\times\mathbb{Z} \mid i{+}j = k, \text{ there is a substring } \tau \text{ of } \sigma : |\tau|_a = i, |\tau|_b = j\}.$$

All points in S_k will lie on a line (a diagonal), and moreover, they will form an interval on this line, cp. Example 3. Each such interval has two extremal points.

Assume for a moment that we know the multiplicities of a and b in M, i.e. i and j s.t. $M = i \cdot \mu(a) + j \cdot \mu(b)$, and they are unique. Then, we

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



can check whether $(i, j) \in S_{i+j}$, which takes $O(1)$ time if we have stored the extremal points of all S_k during the preprocessing phase. This would require space linear in n. If, in addition, i and j are known to be unique, then this algorithm would decide wether M is a submass of σ. Unfortunately, we do not know the multiplicities of a and b in M. We define $d := \mu(b) - \mu(a)$ (w.l.o.g., we assume that $\mu(a) < \mu(b)$) and use the residue[4] of M mod d to look up a table. The table, generated during the preprocessing phase, contains representations of all submasses of σ.

Let $M_k := \{\mu(\tau) \mid \tau \text{ is a } k\text{-length substring of } \sigma\}$. Observe that consecutive elements of M_k (when sorted) differ by exactly d. Therefore, we can write $M_k = \{c_k + \ell \cdot d \mid \ell = 0, \ldots, n_k - 1\}$, where $c_k = \min M_k$ and $n_k = |M_k|$. Furthermore, $M_k = \{r_k + \ell \cdot d \mid \ell = a_k, \ldots, b_k\}$, where $r_k = (c_k \bmod d)$, $a_k = \lfloor \frac{c_k}{d} \rfloor$ and $b_k = a_k + n_k - 1$. This says that all submasses of the same length have the same residue modulo d.

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

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

[4]We define $x \bmod c$, for real $x, c \in \mathbb{R}^+$, by $(x \bmod c) := r$, where r is the unique real number $0 \le r < c$ s.t. $x = g \cdot c + r$ and $g \in \mathbb{N}$.

Observe that $r_k = (k \cdot \mu(a) \bmod d)$. Thus, we may have the same residue modulo d for different values of k. Instead of storing a_k and b_k for each r_k 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.

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	$[0,5]$

4.1 Algorithm Interval

In the preprocessing phase, we calculate the r_k's, a_k's, and b_k's as above. We then sort the r_k's, thus obtaining a sorted array $q_1, \ldots, q_m$, where $m \leq n$ (since different S_k's may have the same residue). For each q_l, we compute a list of interval endpoints which represents the union of all intervals $[a_k, b_k]$ with $r_k = q_l$. 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 contained in σ,

1. decompose $M = g \cdot d + r$, where $r = (M \bmod d)$ and $g \in \mathbb{N}$;
2. find index $l \in \{1, \ldots, m\}$ such that $r = q_l$, using binary search; if no such index can be found, then M is not a submass of σ, 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 σ iff 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 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 2. *Algorithm* INTERVAL *solves the* ONE–STRING MASS FINDING 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 Weighted Strings

A question closely related to the analysis of algorithms for the ONE–STRING MASS FINDING PROBLEM is the following: Given σ, how many different submasses does σ have? We define a function Klaus which counts the number of different submasses of a string:

$$\mathrm{Klaus}(\sigma) := |\{\mu(\sigma(i,j)) \mid 1 \leq i \leq j \leq |\sigma|\}|.$$

For example, the storage space required by BINSEARCH is proportional to $\mathrm{Klaus}(\sigma)$. An obvious upper bound on $\mathrm{Klaus}(\sigma)$ for $|\sigma| = n$ is the number of different substrings of σ, which is at most $\frac{n(n+1)}{2}$. However, different substrings may have the same mass, e.g. if the multiplicities of their letters coincide (i.e. if $\forall a \in \mathcal{A} : |\sigma|_a = |\tau|_a$, then $\mu(\sigma) = \mu(\tau)$). In addition, different letters may yield the same sum, e.g. if $\mu(a) = 2$, and $\mu(b) = 3$, then $\mu(aaa) = \mu(bb)$. We define the UNIQUE DECOMPOSITION PROPERTY:

> A mass function μ has the UNIQUE DECOMPOSITION PROPERTY (UDP) if, for all strings σ and τ:
>
> $$\mu(\sigma) = \mu(\tau) \iff \forall a \in \mathcal{A} : |\sigma|_a = |\tau|_a.$$

This just means that the masses are linearly independent over the integers. With the UDP, a mass M has at most one decomposition $M = \sum_{a\in\mathcal{A}} \nu(a)\cdot\mu(a)$ with $\nu(a) \in \mathbb{N}$. For strings σ over alphabet $\mathcal{A} = \{a_1, \ldots, a_s\}$, we define the *multiplicity vector* $\mathrm{mult}(\sigma) = (|\sigma|_{a_1}, \ldots, |\sigma|_{a_s})$. With the UDP, we have $\mu(\sigma) = \mu(\tau) \Leftrightarrow \mathrm{mult}(\sigma) = \mathrm{mult}(\tau)$.

For the rest of this section, we assume that μ has the UDP.

The question of the value of $\mathrm{Klaus}(\sigma)$ for a given string σ is equivalent to the question of how many different multiplicity vectors (also called *Parikh–vectors*, see [ABB97]) the set $L_\sigma := \{\tau \mid \tau \text{ is substring of } \sigma\}$ has. If we denote by $\mathcal{A}^{\oplus}$ the free commutative monoid over $\mathcal{A}$, then any language $L \subseteq \mathcal{A}^*$ induces a subset $L^{\oplus}$ of $\mathcal{A}^{\oplus}$, namely $L^{\oplus} := \{\prod_{a\in\mathcal{A}} a^{|\tau|_a} \mid \tau \in L\}$. Since the mapping

$$\prod_{a\in\mathcal{A}} a^{|\tau|_a} \mapsto \sum_{a\in\mathcal{A}} |\tau|_a \cdot \mu(a)$$

is a bijection, we have $\mathrm{Klaus}(\sigma) = |L_\sigma^{\oplus}|$. We are not aware that $|L_\sigma^{\oplus}|$ has been characterized in the literature.

In the following, we present strings for which the number of different submasses is linear resp. quadratic in their length, and we prove a tight upper bound and a lower bound (which is not tight) on Klaus. We denote by $[P]$ the characteristic value of a proposition P, i.e. $[P] = 1$ if P is true, and $[P] = 0$ otherwise. As usual, $s = |\mathcal{A}|$.

Lemma 3. *There are strings $\sigma, \tau \in \mathcal{A}^n$ s.t.* $\mathrm{Klaus}(\sigma) = \Theta(n)$ *and* $\mathrm{Klaus}(\tau) = \Theta(n^2)$. *In particular, for $k, r, n_1, \ldots, n_s \in \mathbb{N}$ s.t. $r \leq s$ and $\sum_{i=1}^{s} n_i = n$,*

1. $\mathrm{Klaus}((a_1 \ldots a_s)^k) = (k-1)\cdot(s^2+1-s) + \frac{1}{2}(s^2+s)$,
2. $\mathrm{Klaus}((a_1 \ldots a_s)^k a_1 \ldots a_r) =$
   $(k-1)\cdot(s^2+1-s) + \frac{1}{2}(s^2+s) + r(s-1) + [r=s]$,
3. $\mathrm{Klaus}(a_1^{n_1} a_2^{n_2} \ldots a_s^{n_s}) = n + \sum_{1\leq i<j\leq s} n_i \cdot n_j$.

Proof.

1. Let $\sigma := (a_1 \ldots a_s)^k$. Note that substrings of σ have length $m\cdot s+p$, where $m = 1, \ldots, k-1$, and $p = 1, \ldots, s$. Now for fixed $0 \leq m \leq k-2$, there is exactly one subsum with length $m\cdot s+s$, since all substrings of this length have weight $\sum_{i=1}^{s}(m+1)\mu(a_i)$; while for each $p = 1, \ldots, s-1$, there are s different subsums, namely one for each $1 \leq j \leq s$: $\sum_{i=1}^{s} m\cdot\mu(a_i) + \sum_{i=1}^{p}\mu(a_{(j+i) \bmod s})$. Putting this together yields $(k-1)(1+(s-1)s)$. Finally, for $m = k-1$, the string $\sigma = (a_1 \ldots a_s)^m a_1 \ldots a_s$ has s different subsums with length $m\cdot s+1$, $s-1$ with length $m\cdot s+2$ and so on, yielding $\sum_{i=s}^{1} i = \frac{1}{2}(s^2+s)$ subsums. Since the alphabet size s is constant and $k = \frac{n}{s}$, this expression is linear in n: $\mathrm{Klaus}(\sigma) = (k-1)\cdot(s^2+1-s) + \frac{1}{2}(s^2+s) = \Theta(n(s-1)) + O(1) = \Theta(n)$.

2. This is an easy extension of *1.*, noting that each of the last r letters will contribute $s-1$ new subsums ending in this letter: For a_j, $1 \leq j \leq r$, $\sigma(i, k\cdot s+j)$ will be new for all $i = 1, \ldots, j-1, j+1, \ldots, s$. In addition, if $r = s$, then the substring $\sigma(s, n) = a_s(a_1 \ldots a_s)^k$ will also be new.

3. First consider submasses which start and end with the same letter a_i. For fixed $1 \leq i \leq s$, there are n_i different submasses of this type, yielding

$\sum_{i=1}^{s} n_i = n$ different submasses. All other submasses start with some letter a_i and end with a different letter a_j, where $i < j$. For each pair i, j, there are $n_i \cdot n_j$ different choices of the beginning and ending position which generate different submasses. Now, if we choose all n_i roughly equal, i.e. $n_i = \lfloor \frac{n}{s} \rfloor$ or $n_i = \lfloor \frac{n}{s} \rfloor + 1$, this yields $\sum_{i=1}^{s} n_i + \sum_{1 \leq i < j \leq s} n_i n_j \approx n + \binom{s}{2} \cdot (\frac{n}{s})^2 \approx n + \frac{1}{2} n^2 = \Theta(n^2)$. □

Lemma 4. *Let $n \in \mathbb{N}, x \in \mathcal{A}$ and $\sigma \in \mathcal{A}^n$.*

1. *If σ does not contain letter x, then* $\mathrm{Klaus}(\sigma x) = \mathrm{Klaus}(\sigma) + (n+1)$.
2. *If σ contains letter x, then*
   $\mathrm{Klaus}(\sigma x) \leq \mathrm{Klaus}(\sigma) + n - |\sigma|_x + [\sigma(n) = x]$.

Proof.

1. Obvious.

2. There are $\mathrm{Klaus}(\sigma)$ different submasses starting and ending within σ. Furthermore, there are n substrings of σx starting within σ and ending in x. For each index $1 \leq i \leq n-1$ s.t. $\sigma(i) = x$, we have $\mathrm{mult}(\sigma(i,n)) = \mathrm{mult}(\sigma(i+1,n)x)$. Thus, none of these substrings generates a new submass. There are $|\sigma|_x$ such substrings if $\sigma(n) \neq x$, and $|\sigma|_x - 1$ otherwise. □

Lemma 5. *Let $n \in \mathbb{N}$ and fix $0 \leq n_1, \ldots, n_s \in \mathbb{N}$ s.t. $\sum_{i=1}^{s} n_i = n$. Then,*

$$\mathrm{Klaus}(a_1^{n_1} \ldots a_s^{n_s}) = \max\{\mathrm{Klaus}(\tau) \mid \forall i = 1, \ldots, s : |\tau|_{a_i} = n_i\}.$$

Proof. By induction on n: For $n = 1$, the claim is obvious. Choose $\sigma \in \mathcal{A}^{n+1}$ and denote by $n_i := |\sigma|_{a_i}$ for $i = 1, \ldots, s$. Up to relabeling (which leaves Klaus invariant), we may assume that the last letter of σ is a_s. If $n_s = 1$,

$$\begin{aligned}
\mathrm{Klaus}(\sigma) &= \mathrm{Klaus}(\sigma') + (n+1) && \text{by Lemma 4} \\
&\leq \mathrm{Klaus}(a_1^{n_1} \ldots a_{s-1}^{n_{s-1}}) + (n+1) && \text{by the induction hypothesis} \\
&= \mathrm{Klaus}(a_1^{n_1} \ldots a_s^{n_s}).
\end{aligned}$$

Otherwise, $n_s > 1$, and

$$\begin{aligned}
\mathrm{Klaus}(\sigma) &\leq \mathrm{Klaus}(\sigma') + n - |\sigma'|_{a_s} + 1 && \text{by Lemma 4} \\
&\leq \mathrm{Klaus}(a_1^{n_1} \ldots a_s^{n_s - 1}) + n - (n_s - 1) + 1 && \text{by the ind.hypo.} \\
&= n + \sum_{1 \leq i < j < s} n_i n_j + \sum_{i=1}^{s-1} n_i (n_s - 1) + \sum_{i=1}^{s-1} n_i + 1 && \text{by Lemma 3} \\
&= (n+1) + \sum_{1 \leq i < j \leq s} n_i n_j \;=\; \mathrm{Klaus}(a_1^{n_1} \ldots a_s^{n_s}). && \text{by Lemma 3}
\end{aligned}$$

□

Theorem 6 (Tight Upper Bound on Klaus**).** *Let $\sigma \in \mathcal{A}^n$. Then* $\text{Klaus}(\sigma) \leq n + \sum_{1\leq i<j\leq s} n_i n_j$*, where* $n_i = \lfloor \frac{n}{s} \rfloor$ *or* $n_i = \lfloor \frac{n}{s} \rfloor + 1$ *for* $i = 1, \ldots, s$. *In particular, if n is a multiple of s, then* $\text{Klaus}(\sigma) \leq \frac{s-1}{2s} n^2 + n$. *This bound is tight.*

Proof. Let $\sigma \in \mathcal{A}^n$. Denote by $y_i := |\sigma|_{a_i}$ for $i = 1, \ldots, s$. Then, by Lemma 5, $\text{Klaus}(\sigma) \leq \text{Klaus}(a_1^{y_1} \ldots a_s^{y_s}) = n + \sum_{1\leq i<j\leq s} y_i y_j$. Let $f(x_1, \ldots, x_s) := \sum_{1\leq i<j\leq s} x_i x_j$. Function f reaches its maximum on the set $\{(x_1, \ldots, x_s) \mid \sum_{i=1}^{s} x_i = n\}$ if all values are approximately equal, i.e. if they all equal $\lfloor \frac{n}{s} \rfloor$ or $\lfloor \frac{n}{s} \rfloor + 1$. Moreover, since $\text{Klaus}(a_1^{n_1} \ldots a_s^{n_s}) = n + \sum_{i<j} n_i n_j$, this bound is tight. If n is a multiple of s, then $n_i = \frac{n}{s}$ for all i, and thus:

$$\max\{\text{Klaus}(\sigma) \mid |\sigma| = n\} = n + \binom{s}{2}(\frac{n}{s})^2 = \frac{s-1}{2s}n^2 + n.$$

□

For a lower bound, we define the index of the first occurrence of a letter $a \in \mathcal{A}$ in a string σ as $\text{Pos}_a(\sigma) := \min(\{i \mid \sigma(i) = a\} \cup \{|\sigma| + 1\})$.

Lemma 7 (Lower Bound on Klaus**).** *Let $n \in \mathbb{N}$ and $\sigma \in \mathcal{A}^n$. Then* $\text{Klaus}(\sigma) \geq \sum_{a\in\mathcal{A}} |\sigma|_a \cdot \text{Pos}_a(\sigma)$.

Proof. Let $x = \sigma(n)$. If x does not occur in $\sigma(1, n-1)$, then appending x to $\sigma(1, n-1)$ generates n new strings, i.e. $\text{Klaus}(\sigma) = \text{Klaus}(\sigma(1, n-1)) + n = \text{Klaus}(\sigma(1, n-1)) + \text{Pos}_x(\sigma(1, n-1))$. On the other hand, if x occurs in $\sigma(1, n-1)$, then it generates at least $\text{Pos}_x(\sigma(1, n-1))$ new strings, namely those starting in positions $i = 1, \ldots, \text{Pos}_x(\sigma(1, n-1))$ and ending in $\sigma(n) = x$. Thus, in both cases we obtain $\text{Klaus}(\sigma) \geq \text{Klaus}(\sigma(1, n-1)) + \text{Pos}_x(\sigma(1, n-1))$. Applying this n times, we obtain

$$\text{Klaus}(\sigma) \geq \sum_{i=1}^{n} \text{Pos}_{\sigma(i)}(\sigma(1, i-1)).$$

Let $i \in \{1, \ldots, n\}$. If $\sigma(i)$ occurs in $\sigma(1, i-1)$, then $\text{Pos}_{\sigma(i)}(\sigma(1, i-1)) = \text{Pos}_{\sigma(i)}(\sigma)$. In the sum above, this happens $|\sigma|_{\sigma(i)} - 1$ times. For the first occurrence of letter $\sigma(i)$ in σ, we count $\text{Pos}_{\sigma(i)}(\sigma)$, too. Thus, $\sum_{i=1}^{n} \text{Pos}_{\sigma(i)}(\sigma(1, i-1)) = \sum_{a\in\mathcal{A}} \text{Pos}_a(\sigma) \cdot |\sigma|_a$. □

For fixed multiplicities $n_1, \ldots, n_s$, the sum $\sum_{a\in\mathcal{A}} \text{Pos}_a(\sigma) \cdot |\sigma|_a$ is minimized over all strings with these multiplicities if all different letters occurring in σ are positioned in the first positions of σ, ordered ascending according to their multiplicities. In particular, if each letter occurs exactly k times, we obtain $\text{Klaus}(\sigma) \geq k \sum_{i=1}^{s} i = \frac{k}{2}(s^2+s)$.

This lower bound, however, is not tight: For instance, for $s = 4$ and equal multiplicities $k = 3$, the lower bound is 30, while the minimum value of Klaus is 36, e.g. for string $\sigma = abcdabcdabcd$.

On the other hand, from Lemma 3 we know that $\text{Klaus}((a_1 \dots a_s)^k) = (k-1) \cdot (s^2 + 1 - s) + \frac{1}{2}(s^2 + s)$, thus these strings miss the lower bound of Lemma 7 only by a factor of 2. However, we know that they do not minimize Klaus, since e.g. $\text{Klaus}((abcde)^4) = 78$, while $\text{Klaus}((abcde)^2(aedcb)^2) = 75$.

6 Problem Variants

The Multiple–String Mass Finding Problem is defined as follows:

> Given k strings $\sigma_1, \dots, \sigma_k$ and a mass $M \in \mathbb{R}^+$, return a list $i_1, \dots, i_r$ of those strings σ_{i_j} which have M as a submass.

An algorithm Ψ for the One–String Mass Finding Problem can be extended to an algorithm for the Multiple–String Mass Finding Problem by running Ψ on each string σ_i one by one. Required storage space and time simply sum up.

Alternatively, we can define a new string $\sigma := \sigma_1 \omega \sigma_2 \omega \dots \omega \sigma_k$, where ω is a new letter with mass $\mu(\omega) := \max_{i=1}^{k}\{\mu(\sigma_i)\} + 1$. Before applying Ψ to σ, we check whether $M \geq \mu(\omega)$. If so, then M cannot be a submass of any of the strings, and we are done. Otherwise, we know that whenever Ψ finds mass M in σ, then it is a submass of σ_i for some index i. If algorithm Ψ can output *all* positions of M in σ, this solves the Multiple–String Mass Finding Problem. If Ψ only *decides* whether M is a submass of σ (i.e. it outputs only **yes** or **no**), we use a kind of "binary tree search" BinTreeSearch to find all σ_i with submass M as follows. First, we run Ψ on σ as described above. If it outputs **no**, then no string σ_i has submass M, and we are done. Otherwise, we divide σ into two new strings $\sigma_l := \sigma_1 \omega \dots \omega \sigma_{\lfloor \frac{k}{2} \rfloor}$ and $\sigma_r := \sigma_{\lfloor \frac{k}{2} \rfloor + 1} \omega \dots \omega \sigma_k$ and run Ψ on both strings separately. We repeat the division step until the new strings cover exactly one σ_i, in which case the answer of Ψ determines whether σ_i has a submass M. Analysis of BinTreeSearch depends heavily on storage space and query time required by Ψ. For instance, if algorithm Ψ requires storage space linear in the length of the string, then the storage space of BinTreeSearch is $O((\log k) \cdot \sum_{i=1}^{k} |\sigma_i|)$. Query time of BinTreeSearch is output sensitive (i.e. it depends on the number of strings with submass M), in contrast to the simple idea of applying Ψ to each string one by one.

Given a specific algorithm for the ONE–STRING MASS FINDING PROBLEM, there might be even better ways to extend it to the MULTIPLE–STRING MASS FINDING PROBLEM: E.g. for BINSEARCH, we can use *one* sorted array to store all submasses of all strings. For each submass x we store the set of indices I_x of all those strings which have a submass x. Given mass M, we perform binary search in the array and output all indices stored in I_M. Required storage space remains unchanged, but the running time becomes $O(\log \sum_{i=1}^{k} |\sigma_i| + |I_M|)$, where $|I_M| \leq k$ is the size of the output. A similar idea applies to LOOKUP, which we can improve by storing only *one* table T (e.g. with index length $\max_{i=1}^{k}\{n_i\}$) and use it for all runs of the algorithm. However, this does not decrease the asymptotic space required, which still remains linear.

Finally, we define a third problem variant, the MULTIPLE–STRING MULTIPLE–MASS FINDING PROBLEM:

> Given k strings $\sigma_1, \ldots, \sigma_k$, m masses $M_1, \ldots, M_m \in \mathbb{R}^+$, and a threshold $1 \leq t \leq m$, return a list $i_1, \ldots, i_r$ of those strings σ_{i_j} which have at least t of the masses as submasses.

In the setting of our application in computational biology, this will be a more realistic formulation, since typically, one breaks a given protein in several pieces and wants to find the protein in the database which contains all, or at least many, of these pieces. Obviously, the MULTIPLE–STRING MULTIPLE–MASS FINDING PROBLEM can be solved by applying algorithms for the MULTIPLE–STRING MASS FINDING PROBLEM m times. We are investigating the question whether concurrently searching for m masses can be performed more efficiently.

7 Ongoing Work

Currently, we are working on another algorithm called CLUSTER, which solves the ONE–STRING MASS FINDING PROBLEM. The main idea is this: Given M, imagine that some oracle gave us a set of positions C with the following property: If M is a submass of σ, then there is $i \in C$ s.t. i *covers* M, i.e. there is a substring of σ that starts in i and has mass M. Then we would only have to check whether one of the positions in C covers M. But this can be done fast, as long as we have stored in advance, in each position $i = 1, \ldots, n$, the mass of the suffix $\sigma(i, n)$: We can now do binary search on

$j = i, \ldots, n$ where in each step, we compare $\sigma(i,j) = \sigma(i,n) - \sigma(j+1,n)$ with M and change j accordingly. This takes time $O(\log(n-i)) = O(\log n)$ for each element $i \in C$, i.e. $O(|C| \log n)$ in total.

But how do we find such sets C? First calculate all submasses, and for each submass x, store all positions i which cover x, and then sort the submasses by size. Next, cluster the submasses in disjoint intervals $I_1, \ldots, I_m$, and for each I_k, store a set C_k of candidate positions with the property that, for any $x \in I_k$, x is a submass of σ if and only if there is an $i \in C_k$ covering x. Such a set always exists, since in the worst case, we have to store one position for each submass in I_k. The interesting question is how to cluster the submasses. We will return to this shortly.

First, to the query algorithm: Given M, we can first find k s.t. $M \in I_k$ with binary search on the intervals. Now we check, for each $i \in C_k$, whether i covers M. If the answer is **yes**, the algorithm returns a substring starting in i with weight M; if for all $i \in C_k$ the answer is **no**, then, by construction, we know that M is not a submass of σ.

Now some complexity considerations: Say there are m intervals, and $c = \max_{i=1,\ldots,m} |C_k|$. Storage space needed for the preprocessed data is, first, $O(m)$ for storing the intervals, since storage of one interval requires constant space. Note that in the preprocessing phase, we have to calculate all submasses of σ; however, once the preprocessing is complete, we only store the intervals. Storing the candidate positions requires $O(m \cdot c)$ space, and storing the masses of the suffixes at each position $i = 1, \ldots, n$ requires $O(n)$ space. The running time of the query algorithm is $O(\log m)$ for finding the right interval for M, and $O(c \cdot \log n)$ for checking all candidate positions. Altogether, this yields storage space $O(m \cdot c + n)$ and running time $O(\log m + c \cdot \log n)$.

In our present implementation, we decide for a size for c before the start of the preprocessing phase. After having computed and sorted all submasses, we build up the intervals and candidate sets in the following way: Let us say the current interval being built is I_k. If the next subsum x in the list of all subsums is already covered by an $i \in C_k$, then add x to I_k. Otherwise, choose a position i which covers x. If $|C_k \cup \{i\}| \leq c$, then add i to C_k and x to I_k, otherwise start a new interval $I_{k+1} = [M]$ and a new candidate set $C_{k+1} = \{i\}$.

For Cluster to be both skinny and speedy, it would need to have storage space $o(n^2)$ and running time $o(n)$. Obviously, m, the number of intervals, depends on the choice of c, the maximal size of the candidate sets. If we succeeded in defining the intervals and their candidate sets in such a way that, say, $c = O(\sqrt{n})$ and $m = O(n)$, this would meet the requirements. From simple implementation experiments, we have the impression that CLUSTER performs well, but have not yet been able to prove this.

For CLUSTER, as well as many other algorithms, storage space and the query time for a string σ depend on Klaus(σ). Therefore, we are interested in properties of function Klaus, e.g. its minimal value, or its expected value. Furthermore, we are looking for efficient algorithms to compute Klaus(σ).

As mentioned in the Introduction, for an algorithm to be of any practical value in computational biology, it needs to be fault tolerant. We are currently in the process of modifying the algorithms introduced in this paper to be tolerant against measurement errors. The next step will be to check their practical impacts.

In the long run, we are interested in the tradeoff between time and space for the mass finding problem. Both LOOKUP and CLUSTER allow for a tradeoff between time and space. We would like to find further algorithms that can be parametrized to allow for adjustment of this tradeoff.

8 Acknowledgments

We would like to thank Juraj Hromkovic who read an earlier version of this paper and made many helpful suggestions. We also thank Volker Diekert for the pointer to Parikh-vectors.

References

[ABB97] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. Context–free languages and pushdown automata. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages, vol. 1*, chapter 3, pages 111–174. Springer, 1997.

[AG95] Alberto Apostolico and Zvi Galil, editors. *Combinatorial Algorithms on Words*. Springer, 1995.

[BE01] V. Bafna and N. Edwards. SCOPE: A probabilistic model for scoring tandem mass spectra against a peptide database. *Bioinformatics*, 17(Supplement 1):13–21, 2001.

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

[CR94] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, New York, NY, 1994.

[EMYI94] J. Eng, A. McCormack, and J. R. Yates III. An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database. *J. Amer. Soc. Mass Spect.*, 5:976–989, 1994.

[Fre75] Michael 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)*, pages 240–244, 1975.

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

[GW91] Larry Gonick and Mark Wheelis. *The Cartoon Guide to Genetics.* Harper Perennial, updated edition, 1991.

[HBS+93] W. J. Henzel, T. M. Billeci, J. T. Stults, S. C. Wong, C. Grimley, and C. Watanabe. Identifying proteins from two-dimensional gels by molecular mass searching of peptide fragments in protein sequence databases. *Proc. Natl. Acad. Sci. USA*, 90(11):5011–5015, 1993.

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

[JQCG93] P. James, M. Quadroni, E. Carafoli, and G. Gonnet. Protein identification by mass profile fingerprinting. *Biochem. Biophys. Res. Commun.*, 195(1):58–64, 1993.

[Lot97] M. Lothaire. *Combinatorics on Words.* Cambridge University Press, second edition, 1997.

[MHR93] M. Mann, P. Højrup, and P. Roepstorff. Use of mass spectrometric molecular weight information to identify proteins in sequence databases. *Biol. Mass Spectrom.*, 22(6):338–345, 1993.

[MW94] M. Mann and M. Wilm. Error-tolerant identification of peptides in sequence databases by peptide sequence tags. *Anal. Chem.*, 66(24):4390–4399, 1994.

[PDT00] P. A. Pevzner, V. Dančík, and C. L. Tang. Mutation-tolerant protein identification by mass spectrometry. *J. Comp. Biol.*, 7(6):777–787, 2000.

[Pev00] Pavel A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach.* MIT Press, 2000.

[PHB93] D. J. C. Pappin, P. Højrup, and A. J. Bleasby. Rapid identification of proteins by peptide-mass fingerprinting. *Curr. Biol.*, 3(6):327–332, 1993.

[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.

[RS97] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages*, volume 1-3. Springer, 1997.

[SM97] João Setubal and João Meidanis. *Introduction to Computational Molecular Biology.* PWS Boston, 1997.

[Str88] Lubert Stryer. *Biochemistry.* Freeman, 1988.

[YEM95] J. R. Yates, J. K. Eng, and A. L. McCormack. Mining genomes: Correlating tanden mass-spectra of modified and unmodified peptides to sequences in nucleotide databases. *Anal. Chem.*, 67(18):3202–3210, 1995.

[YI98] J. R. Yates III. Database searching using mass spectrometry data. *Electrophoresis*, 19(6):893–900, 1998.

[YISGH93] J. R. Yates III, S. Speicher, P. R. Griffin, and T. Hunkapillar. Peptide mass maps: A highly informative approach to protein identification. *Anal. Biochem.*, 214:397–408, 1993.