Efficient Algorithms for Finding Submasses in Weighted Strings

Nikhil Bansal[1], Mark Cieliebak[2], and Zsuzsanna Lipták[3]

[1] IBM Research, T.J. Watson Research Center*
[2] ETH Zurich, Institute of Theoretical Computer Science** *and*
Center for Web Research, Department of Computer Science, University of Chile
[3] Universität Bielefeld, AG Genominformatik, Technische Fakultät***

Abstract. We study the Submass Finding Problem: Given a string s over a weighted alphabet, i.e., an alphabet Σ with a weight function $\mu : \Sigma \to \mathbb{N}$, decide for an input mass M whether s has a substring whose weights sum up to M. If M is indeed a submass, then we want to find one or all occurrences of such substrings. We present efficient algorithms for both the decision and the search problem. Furthermore, our approach allows us to compute efficiently the number of different submasses of s. The main idea of our algorithms is to define appropriate polynomials such that we can determine the solution for the Submass Finding Problem from the coefficients of the product of these polynomials. We obtain very efficient running times by using Fast Fourier Transform to compute this product. Our main algorithm for the decision problem runs in time $\mathcal{O}(\mu_s \log \mu_s)$, where μ_s is the total mass of string s. Employing standard methods for compressing sparse polynomials, this runtime can be viewed as $\mathcal{O}(\sigma(s) \log^2 \sigma(s))$, where $\sigma(s)$ denotes the number of different submasses of s. In this case, the runtime is independent of the size of the individual masses of characters.

1 Introduction

Over the past few years, interest in the area of weighted strings has received increasing attention. A *weighted string* is defined over an alphabet $\Sigma = \{a_1, \ldots, a_{|\Sigma|}\}$ with a weight function $\mu : \Sigma \to \mathbb{N}$, which assigns a specific weight (or mass) to each character of the alphabet. The weight of a string s is just the sum of the weights of all characters in s.
Several applications from bioinformatics can be formalized as problems on strings over a weighted alphabet; most notably, mass spectrometry experiments, which constitute an experimentally very efficient and therefore promising alternative method of protein identification and de-novo peptide sequencing, and are also increasingly being used for DNA.
For our purposes, proteins are strings over the 20-letter amino acid alphabet. The molecular masses of the amino acids are known up to high

* P.O. Box 218, Yorktown Heights, NY 10598, `nikhil@us.ibm.com`
** Clausiusstr. 49, CH-8092 Zurich, `cieliebak@inf.ethz.ch`
*** Postfach 10 01 31, D-33592 Bielefeld, `zsuzsa@cebitec.uni-bielefeld.de`

precision. In order to enforce that the masses be positive integers, we assume that non-integer masses have been scaled. One of the main applications of protein mass spectrometry is database lookup. Here, a protein is cut into substrings (*digestion*), the molecular mass of the substrings is determined, and the list of masses is compared to a protein database [1–3].
In the following, we skip some of the proofs due to space limitations.

1.1 Definitions and Problem Statements

Before we define the problems we are going to solve in this paper, we first need to fix some notation for weighted strings. Given a finite alphabet Σ and a mass function $\mu : \Sigma \to \mathbb{N}$, where $\mathbb{N}$ denotes the set of positive integers (excluding 0). We denote by $\mu_{\max} = \max \mu(\Sigma)$, the largest mass of a single character. For a string $s = s_1 \dots s_n$ over Σ, define $\mu(s) := \sum_{i=1}^{n} \mu(s_i)$. We denote the length n of s by $|s|$. We call $M > 0$ a *submass* of s if there exists a substring t of s with mass M, or, equivalently, if there is a pair of indices (i, j) such that $\mu(s_i \dots s_j) = M$. We call such a pair (i, j) a *witness* of M in s, and we denote the number of witnesses of M in s by $\kappa(M) = \kappa(M, s)$. Note that $\kappa(M) \leq n$. We want to solve the following problems:

Submass Query Problem Fix a string s over Σ. Let $|s| = n$.
INPUT: k masses $M_1, \dots, M_k \in \mathbb{N}$.
OUTPUT: A subset $I \subseteq \{1, \dots, k\}$ such that $i \in I$ if and only if M_i is a submass of s.

Submass Witness Problem Fix a string s over Σ. Let $|s| = n$.
INPUT: k masses $M_1, \dots, M_k \in \mathbb{N}$.
OUTPUT: A subset $I \subseteq \{1, \dots, k\}$ such that $i \in I$ if and only if M_i is a submass of s, and a set $\{(b_i, e_i) : i \in I, (b_i, e_i) \text{ is witness of } M_i\}$.

Submass All Witnesses Problem Fix a string s over Σ. Let $|s| = n$.
INPUT: k masses $M_1, \dots, M_k \in \mathbb{N}$.
OUTPUT: A subset $I \subseteq \{1, \dots, k\}$ 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 $W_i := \{(b, e) : (b, e) \text{ is witness of } M_i \text{ in } s\}$.

The three problems above can be solved by a simple algorithm, which we refer to as LINSEARCH. It 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 problem in $\Theta(kn)$ time and $\mathcal{O}(1)$ space in addition to the storage space required for the input string and the output.
Another algorithm, BINSEARCH, computes all submasses of s in a preprocessing step and stores the submasses in a sorted array, which can then be queried in time $\mathcal{O}(\log n)$ for an input mass M for the SUBMASS QUERY PROBLEM and the SUBMASS WITNESS PROBLEM. The storage space required is proportional to $\sigma(s)$, the number of different submasses of string 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 becomes $\mathcal{O}(k \log n + K)$, where $K = \sum_{i=1}^{k} \kappa(M_i)$ is the number of witnesses for the query masses. Note that any algorithm solving the SUBMASS ALL WITNESSES PROBLEM will have runtime $\Omega(K)$.

In this paper, we present a novel approach to the problems above which often outperforms the naïve algorithms. The main idea is similar to using generating functions for counting objects, which have been applied, for instance, in attacking the Coin Change Problem [4]. We apply similar ideas using finite polynomials rather than infinite ones as follows. We define appropriate polynomials such that we can determine the solution for the three problems above from the coefficients of the product of these polynomials. We will obtain very efficient running times by using Fast Fourier Transform to compute this product. More precisely, ALGORITHM 1 solves the SUBMASS QUERY PROBLEM with preprocessing time $\mathcal{O}(\mu_s \log \mu_s)$, query time $\mathcal{O}(k \log n)$ and storage space $\Theta(\sigma(s))$, where μ_s denotes the total mass of the string s. For the SUBMASS WITNESS PROBLEM, we present a Las Vegas algorithm, ALGORITHM 2, with preprocessing time $\mathcal{O}(\mu_s \log \mu_s)$, expected query time $\mathcal{O}(\mu_s \log^3 \mu_s + k \log n)$, and storage space $\Theta(\sigma(s))$. Finally, we present ALGORITHM 3, a deterministic algorithm for the SUBMASS ALL WITNESSES PROBLEM with preprocessing time $\mathcal{O}((Kn\mu_s \log \mu_s)^{\frac{1}{2}})$ and running time $\mathcal{O}((Kn\mu_s \log \mu_s)^{\frac{1}{2}})$, where K is the output size, i.e., the total number of witnesses.

Many algorithms for weighted strings, such as BINSEARCH, have a space complexity which is proportional to $\sigma(s)$, the number of submasses of s. For this reason, we define the following problem:

Number of Submasses Problem Given string s of length n, how many different submasses does s have?

This problem is of interest because we can use $\sigma(s)$ to choose between algorithms whose complexity depends on this number. It is open how the number of submasses of a given string can be computed efficiently. It can, of course, be done in $\Theta(n^2 \log \sigma(s))$ time by computing the masses of all substrings $s_i \dots s_j$, for all pairs of indices $1 \leq i \leq j \leq n$, and counting the number of different masses. We show how ALGORITHM 1 can be adapted to solve the NUMBER OF SUBMASSES PROBLEM in time $\mathcal{O}(\mu_s \log \mu_s)$, outperforming the naïve algorithm for small values of μ_s.

Throughout the paper, we present our runtimes as a function of μ_s, the total mass of the string s. However, we can use the technique of Cole and Hariharan [5] in a straightforward way to give a Las Vegas algorithm with expected running time in terms of the number of distinct submasses. This transformation loses a factor of at most $O(\log n)$ in the running time.

1.2 Related Work

In [3], several algorithms for the SUBMASS QUERY PROBLEM were presented, including LINSEARCH, BINSEARCH, and an algorithm with $\mathcal{O}(n)$ storage space and query time $\mathcal{O}(\frac{kn}{\log n})$, using $\mathcal{O}(n)$ time and space for preprocessing. However, this is an asymptotic result only, since the constants are so large that for a 20-letter alphabet and realistic string sizes, the algorithm is not applicable. Another algorithm was presented in [3]

which solves the Submass Query Problem for binary alphabets with query time $\mathcal{O}(\log n)$ and $\mathcal{O}(n)$ space but does not produce witnesses. Edwards and Lippert [1] considered the Submass All Witnesses Problem and presented an algorithm that preprocesses the database by compressing witnesses using suffix trees. However, they work under the assumption that the queries are limited in range.
The study of weighted strings and their submasses[4] has further applications in those problems on strings over an un-weighted alphabet where the focus of interest are not substrings, but rather equivalence classes of substrings defined by multiplicity of characters. One examines objects of the form $(n_1, \ldots, n_{|\Sigma|})$ which represent all strings $s_1 \ldots s_n$ such that the cardinality of character a_i in each string is exactly n_i, for all $1 \le i \le |\Sigma|$. These objects have been referred to in recent publications variously as *compositions* [6], *compomers* [7, 8], *Parikh-vectors* [9], *multiplicity vectors* [3], and *π-patterns* [10]. A similar approach has been referred to as *Parikh-fingerprints* [11, 12]. Here, Boolean vectors are considered of the form $(b_1, \ldots, b_{|\Sigma|})$, where $b_i = 1$ if and only if a_i occurs in the string. Applications range from identifying gene clusters [12] to pattern recognition [11], alignment [6] or SNP discovery [8].

2 Searching for Submasses Using Polynomials

In this section, we introduce the main idea of our algorithms, the encoding of submasses via polynomials. We first prove some crucial properties, and then discuss algorithmic questions.
Let $s = s_1 \ldots s_n$. In the rest of the paper, we denote by μ_s the total mass of the string s, and the empty string by ε. Define, for $0 \le i \le n$, $p_i := \sum_{j=1}^{i} \mu(s_j) = \mu(s_1 \ldots s_i)$, the i'th prefix mass of s. In particular, $p_0 = \mu(\varepsilon) = 0$. We define two polynomials

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

$$Q_s(x) := \sum_{i=0}^{n-1} x^{\mu_s - p_i} = x^{\mu_s} + x^{\mu_s - \mu(s_1)} + \ldots + x^{\mu_s - \mu(s_1 \ldots s_{n-1})} \quad (2)$$

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

$$C_s(x) := P_s(x) \cdot Q_s(x) = \sum_{m=0}^{2\mu_s} c_m x^m. \quad (3)$$

Since any submass of s with witness (i, j) can be written as a difference of two prefix masses, namely as $p_j - p_{i-1}$, we obtain the following

Lemma 1. *Let $P_s(x), Q_s(x)$ and $C_s(x)$ from Equations* (1) *through* (3)*. 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.*

Lemma 1 immediately implies the following facts. For a proposition $\mathcal{P}$, we denote by $[\mathcal{P}]$ the Boolean function which equals 1 if $\mathcal{P}$ is true, and 0 otherwise. Then $\sum_{m=\mu_s+1}^{2\mu_s} [c_m \neq 0] = \sigma(s)$, the number of submasses of 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.

[4] Note that we use the expressions "weight" and "mass" synomymously, hence "weighted string" but "submass."

2.1 Algorithm and Analysis

The algorithm simply consists of computing $C_s(x)$.

ALGORITHM 1

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 SUBMASS QUERY PROBLEM: Search for each query mass M_i for $1 \leq i \leq k$, and return `yes` if found, `no` otherwise.
   (b) For the NUMBER OF SUBMASSES PROBLEM: Return size of array.

The polynomial $C_s(x)$ can be computed with Fast Fourier Transform (FFT)[13], which runs in time $\mathcal{O}(\mu_s \log \mu_s)$, since $\deg C_s = 2\mu_s$. As mentioned in the Introduction, we can employ methods from [5] for sparse polynomials and reduce $\deg C_s$ to $\mathcal{O}(\sigma(s))$, the number of non-zero coefficients. However, for the rest of this paper, we will refer to the running time as proportional to $\mu_s \log \mu_s$.

Theorem 1. ALGORITHM 1 *solves the* SUBMASS QUERY PROBLEM *in time* $\mathcal{O}(\mu_s \log \mu_s + k \log n)$*, or in time* $\mathcal{O}(\mu_s \log \mu_s + k)$*, depending on the storage method.* ALGORITHM 1 *solves the* NUMBER OF SUBMASSES PROBLEM *in time* $\mathcal{O}(\mu_s \log \mu_s)$.

Proof. The preprocessing step takes time $\mathcal{O}(\mu_s \log \mu_s)$. The query time for the SUBMASS QUERY PROBLEM is $\mathcal{O}(k \cdot \log \sigma(s)) = \mathcal{O}(k \log n)$. Instead of using a sorted array, we can instead store the submasses in an array of size μ_s (which can be hashed to $\mathcal{O}(\sigma(s))$ size) and allow for direct access in constant time, thus reducing query time to $\mathcal{O}(k)$. □

Along the same lines, for the NUMBER OF SUBMASSES PROBLEM, our algorithm allows computation of $\sigma(s)$ in $\mathcal{O}(\mu_s \log \mu_s) = \mathcal{O}(n \cdot \mu_{\max} \log(n \cdot \mu_{\max}))$ time. The naïve solution of generating all submasses requires $\Theta(n^2 \log n)$ time and $\Theta(\sigma(s))$ 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 $\mu_{\max} = o(\frac{n}{\log n})$.

3 A Las Vegas Algorithm for Finding Witnesses

We now describe how to find a witness for each submass of the string s in time $\mathcal{O}(\mu_s \text{ polylog}(\mu_s))$.
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 this, we simply do a binary search amongst the prefix masses $p_1, \ldots, p_{j-1}$ for $p_j - M$. 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 (2), set

$$R_s(x) := \sum_{i=1}^{n} i \cdot x^{p_i} \quad \text{and} \tag{4}$$

$$F_s(x) := R_s(x) \cdot Q_s(x) = \sum_{m=0}^{2\mu_s} f_m x^m. \tag{5}$$

In the following lemma, we use the definition of c_m from (3).

Lemma 2. *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 $c_m = 1$, by Lemma 1, $m - \mu_s$ has exactly one witness (i_0, j_0). Thus, $f_m = j_0$. $\square$

3.1 The Algorithm

We first run a procedure which uses random subsets to try and find witnesses for the query masses. 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. For all query masses which are in this set, we find the beginning positions with binary search within the prefix masses, as described above, to find the witness in time $\mathcal{O}(\log n)$. For any remaining query masses, we run LinSearch. In the following, let $[x^i]A(x)$ denote[5] the coefficient a_i of x^i of the polynomial $A(x) = \sum_j a_j x^j$.

Algorithm 2
1. Compute $C_s(x)$ from Equation (3), and check which of the queries are submasses of s.
2. Procedure try-for-witness
   (i) For a from 1 to $2\log n$, do:
      (ii) Let $b = 2^{-a/2}$. Repeat $24 \log n$ 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 $F_{I_1,I_2}(x) = R_{I_1}(x) \cdot Q_{I_2}(x)$.

[5] Incidentally, our two different uses of "[ ]" are both standard, for generating functions and logical expressions, respectively. Since there is no danger of confusion, we have chosen to use both rather than introduce new ones.

   - Let $c_i = [x^i]C_{I_1,I_2}(x)$ and $f_i = [x^i]R_{I_1,I_2}(x)$.
   - For $i > \mu_s$, if $c_i = 1$ and if i has not yet been successful, then store the pair $(i - \mu_s, f_i)$. Mark i as successful.
3. For all submasses amongst the queries M_ℓ, $1 \leq \ell \leq k$, if an ending position was found in this way, find the beginning position with binary search amongst the prefix masses.
4. If there is a submass M_ℓ for which no witness was found, find one using LINSEARCH.

3.2 Analysis

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

Lemma 3. *(1) For a query mass M with $\kappa(M) = \kappa$, and $a = \lfloor \log_2 \kappa \rfloor$. Consider the step 2.iii of* ALGORITHM 2*. The probability that the coefficient $c_{M+\mu_s}$ of $C_{I_1,I_2}(x)$ for a (as defined above) is not 1 is at most $\frac{7}{8}$. (2) 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 $1/n$.*

Theorem 2. ALGORITHM 2 *solves the* SUBMASS WITNESS PROBLEM *in expected time $\mathcal{O}(\mu_s \log^3 \mu_s + k \log n)$.*

Proof. Denote the number of distinct submasses amongst the query masses by k'. By Lemma 3, the probability that the algorithm finds a witness for each of the $k' = O(n^2)$ submasses is at least $1 - 1/n$. In this case, the expected running time is the time for running the procedure for finding witness ending positions, plus the time for finding the k' witnesses:

$$\mathcal{O}(\underbrace{\mu_s \log \mu_s}_{\text{Step 1.}} + \underbrace{2 \log n}_{\text{Step 2.i}} \cdot \underbrace{24 \log n}_{\text{Step 2.ii}} \cdot \underbrace{\mu_s \log \mu_s}_{\text{Steps 2.iii}}) + k \cdot \mathcal{O}(\log n)$$

In the case when the algorithm does not output all witnesses, we simply run LINSEARCH search for all the submasses in time $O(kn)$. However, since the probability of this event is at most $1/n$, the excepted time in this case is at most $O(k)$. This implies the required bound on the running time. ◻

4 A Deterministic Algorithm for Finding All Witnesses

Recall that, given the string s of length n and k query masses $M_1, \ldots, M_k$, we are able to solve the SUBMASS ALL WITNESSES PROBLEM in $\Theta(k \cdot n)$ time and $\mathcal{O}(1)$ space with LINSEARCH, or in $\Theta(n^2 \log n + k \log n)$ time and $\Theta(n^2)$ space with BINSEARCH. Thus, the two naïve algorithms yield a runtime of $\Theta(\min(kn, (n^2 + k) \log n))$.
Our goal here is to give an algorithm which outperforms the bound above, provided certain conditions hold. Clearly, in general it is impossible to

beat the bound $\min(kn, n^2)$ because that might be the size of the output, K, the total number of witnesses to be returned. Our goal will be to produce something good if $K << kn$.
Now consider two strings s and t. We are interested in submasses of $s \cdot t$ with a witness which spans or touches the border between s and t. More precisely, we refer to a witness (i, j) of m 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 (1). 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 4. *For two strings s, t, and the polynomial*

$$D_{s,t}(x) := (x^0 + P_{s^r}(x)) \cdot (x^0 + P_t(x)) = \sum_{m=0}^{\mu(s)+\mu(t)} d_m x^m, \qquad (6)$$

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

4.1 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 substings to actually find the witnesses. The crucial observation is given in Lemma 5. We now describe the details.
We divide the string s into g substrings of approximately equal length: $s = t_1 \cdot t_2 \cdots t_g$ (where we will choose g later), and denote by $M_{i,j} = \sum_{m=i+1}^{j-1} \mu(t_m)$. In particular, if $j \leq i + 1$, then $M_{i,j} = 0$.
In order to have a good choice for g, we need to know the total size of the output, $K = \sum_{\ell=1}^{k} \kappa(M_\ell)$. This we can do by computing $C_s(x)$ and then adding up the coefficients c_{M_ℓ} for $1 \leq \ell \leq k$. 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 breaking when the first pointer moves past the first position of the second string.

ALGORITHM 3
1. Preprocesssing step:
   (a) Compute μ_s and $C_s(x)$ as defined in (3), and compute $K = \sum_{\ell=1}^{k} c_{M_\ell}$. 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 (6).

2. Query step: For each $1 \leq \ell \leq k$,
   (a) For each i such that $[x^{M_\ell+\mu(t_i)}]C_{t_i}(x) \neq 0$, run LinSearch on t_i for M_ℓ and return all witnesses.
   (b) For each pair (i,j) such that $[x^{M_\ell-M_{i,j}}]D_{t_i,t_j}(x) \neq 0$, run LinSearch on $t_i \cdot t_j$ for submass $M_\ell - M_{i,j}$ and return all border-spanning witnesses.
   (c) If M_ℓ was not a submass of any of the above strings, return `no`.

4.2 Analysis

The following lemma shows the correctness of Algorithm 3.

Lemma 5. *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. 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 1 contributes exactly one 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 $e' = |t_i| + \sum_{i'<j}|t_{i'}|$). Moreover, (b',e') is a border-spanning submass of $t_i \cdot t_j$. Thus, by Lemma 4, (b',e') contributes exactly one to $[x^{M-M_{i,j}}]D_{t_i,t_j}(x)$. □

Using FFT for computing the polynomials, the preprocessing step of Algorithm 3 has runtime $\mathcal{O}(g\mu_s \log \mu_s)$. The query time is $\mathcal{O}(g\mu_s \log \mu_s + K\frac{n}{g})$. With $g = \lceil (\frac{Kn}{\mu_s \log \mu_s})^{\frac{1}{2}} \rceil$, we obtain the following

Theorem 3. Algorithm 3 *solves the* Submass All Witnesses Problem *in time $\mathcal{O}((Kn\mu_s \log \mu_s)^{\frac{1}{2}})$, where μ_s is the mass of the string, and 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, so $\bar{\kappa} = K/k$. Then the runtime is $(k\bar{\kappa}n\mu_s \log \mu_s)^{1/2}$. Note that the running time of the combination of the naïve algorithms for the submass all witnesses problem is $O(\min(kn, n^2 \log n))$. Thus, our algorithm beats the running time of the naïve algorithms above if $\bar{\kappa}\mu_s \log \mu_s = o(kn)$ and $(\bar{\kappa}k\mu_s \log \mu_s) = o(n^3 \log^2 n)$.

5 Discussion

In this paper we gave algorithms for several variants of finding substrings with particular submasses in a given weighted string. Our algorithms are most interesting when the masses of the individual characters are small compared to the length of the string, or more generally, when the number of different possible submasses is small compared to n^2.

Most of our algorithms have running time complexity dependent (up to polylog factors) on the number of different submasses in the given weighted string. While this may not be the best possible running time, it seems that improving this significantly will be hard. For example, consider the problem of finding the number of different submasses $\sigma(s)$. Our algorithm for this problem has runtime $\mathcal{O}(\sigma(s)\log\sigma(s))$. It is not hard to see that the easier problem of deciding whether $\sigma(s)$ is exactly equal to $n(n+1)/2$ or not is at least as hard as the 4-Sum problem. The 4-Sum problem is conjectured to have a run time complexity of $\Omega(n^2)$ [14, 15] and is one of the major problems in computational geometry. So, it is unlikely that even the number of different submasses can be determined in time $o(n^2)$ in the general case.

References

1. Edwards, N., Lippert, R.: Generating peptide candidates from amino-acid sequence databases for protein identification via mass spectrometry. In: Proc. of 2^{nd} WABI. LNCS (2002) 68–81
2. Lu, B., Chen, T.: A suffix tree approach to the interpretation of tandem mass spectra: Applications to peptides of non-specific digestion and post-translational modifications. Bioinformatics Suppl. 2 (ECCB) (2003) II113–II121
3. Cieliebak, M., Erlebach, T., Lipták, Z., Stoye, J., Welzl, E.: Algorithmic complexity of protein identification: Combinatorics of weighted strings. DAM (2004) 27–46
4. Wilf, H.: generatingfunctionology. Academic Press (1990)
5. Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proc. of 34^{th} STOC. (2002)
6. Benson, G.: Composition alignment. In: Proc. of 3^{rd} WABI. LNCS (2003) 447–461
7. Böcker, S.: Sequencing from compomers: Using mass spectrometry for DNA de-novo sequencing of 200+ nt. In: Proc. of 3^{rd} WABI. LNCS (2003) 476–497
8. Böcker, S.: SNP and mutation discovery using base-specific cleavage and MALDI-TOF mass spectrometry. Bioinformatics, Suppl. 1 (ISMB) (2003) i44–i53
9. Salomaa, A.: Counting (scattered) subwords. EATCS **81** (2003) 165–179
10. Eres, R., Landau, G.M., Parida, L.: A combinatorial approach to automatic discovery of cluster-patterns. In: Proc. of 3^{rd} WABI. LNCS (2003) 139–150
11. Apostolico, A., Landau, G., Satta, G.: Efficient text fingerprinting via Parikh mapping. J. of Discrete Algorithms (to appear)
12. Didier, G.: Common intervals of two sequences. In: Proc. of 3^{rd} WABI. LNCS (2003) 17–24
13. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation **19(90)** (1965) 297–301
14. Demaine, E.D., Mitchell, J.S.B., O'Rourke, J.: The open problems project. http://cs.smith.edu/ orourke/TOPP/ (2004)
15. Erickson, J.: Lower bounds for linear satisfiability problems. In: Proc. of 6^{th} SODA. (1995) 388–395