Theory of Computing Systems (2019) 63:506–542
https://doi.org/10.1007/s00224-018-9881-2

# Pattern Matching and Consensus Problems on Weighted Sequences and Profiles

**Tomasz Kociumaka**[1] · **Solon P. Pissis**[2] · **Jakub Radoszewski**[1]

Published online: 10 August 2018


## Abstract

We study pattern matching problems on two major representations of uncertain sequences used in molecular biology: weighted sequences (also known as position weight matrices, PWM) and profiles (scoring matrices). In the simple version, in which only the pattern or only the text is uncertain, we obtain efficient algorithms with theoretically-provable running times using a variation of the lookahead scoring technique. We also consider a general variant of the pattern matching problems in which both the pattern and the text are uncertain. Central to our solution is a special case where the sequences have equal length, called the consensus problem. We propose algorithms for the consensus problem parameterised by the number of strings that match one of the sequences. As our basic approach, a careful adaptation of the classic meet-in-the-middle algorithm for the knapsack problem is used. On the lower bound side, we prove that our dependence on the parameter is optimal up to lower-order terms conditioned on the optimality of the original algorithm for the knapsack problem. Therefore, we make an effort to keep the lower order terms of the complexities of our algorithms as small as possible.

**Keywords** Weighted sequence · Position weight matrix · Profile matching · Multichoice Knapsack

A preliminary version of this research appeared at the 27th International Symposium on Algorithms and Computation (ISAAC 2016).

✉ Jakub Radoszewski
jrad@mimuw.edu.pl

Tomasz Kociumaka
kociumaka@mimuw.edu.pl

Solon P. Pissis
solon.pissis@kcl.ac.uk

[1] Institute of Informatics, University of Warsaw, Warsaw, Poland

[2] Department of Informatics, King's College London, London, UK

## 1 Introduction

We study two well-known representations of uncertain texts: *weighted sequences* and *profiles*. A *weighted sequence* (also known as position weight matrix, PWM) for every position and every letter of the alphabet specifies the probability of occurrence of this letter at this position; see Table 1 for an example. A weighted sequence represents many different strings, each with the probability of occurrence equal to the product of probabilities of its letters at subsequent positions of the weighted sequence. Usually a threshold $\frac{1}{z}$ is specified, and one considers only strings that match the weighted sequence with probability at least $\frac{1}{z}$. A *scoring matrix* (or a profile) of length $m$ is a matrix with $m$ columns indexed by positions $1, \ldots, m$ and $\sigma$ rows corresponding to the alphabet. The *score* of a string of length $m$ is the sum of scores in the scoring matrix of the subsequent letters of the string at the respective positions. A string is said to match a scoring matrix if its matching score is above a specified threshold $Z$.

### 1.1 Weighted Pattern Matching and Profile Matching

First of all, we study the standard variants of pattern matching problems on weighted sequences and profiles, in which only the pattern or the text is an uncertain sequence. In the most popular formulation of the Weighted Pattern Matching problem, we are given a weighted sequence of length $n$, called a text, a solid (standard) string of length $m$, called a pattern, both over an alphabet of size $\sigma$, and a *threshold probability* $\frac{1}{z}$. We are asked to find all positions in the text where the fragment of length $m$ represents the pattern with probability at least $\frac{1}{z}$. Each such position is called an *occurrence* of the pattern in the text; we also say that the fragment of the text and the pattern *match*. The Weighted Pattern Matching problem can be solved in $O(\sigma n \log m)$ time via the Fast Fourier Transform [7]. The average-case complexity of the WPM problem has also been studied and a number of fast algorithms have been presented for certain values of *weight ratio* $\frac{z}{m}$ [4, 5]. An indexing variant of the problem has also been considered [1, 2, 13, 14, 16, 19]; here, one is to preprocess a weighted text to efficiently answer pattern matching queries. The most efficient index [2] for a constant-sized alphabet uses $O(nz)$ space, takes $O(nz)$ time to construct and answers queries in optimal $O(m + occ)$ time, where $occ$ is the number of occurrences reported. A more general indexing data structure, which assumes $z = O(1)$, was presented in [6]. A streaming variant of the Weighted Pattern Matching problem was considered very recently in [23].

In the classic Profile Matching problem, the pattern is an $m \times \sigma$ profile, the text is a solid string of length $n$, and our task is to find all positions in the text where the fragment of length $m$ has score at least $Z$. A naïve approach to the Profile

**Table 1** A weighted sequence $X$ of length 4 over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$

| $X[1]$ | $X[2]$ | $X[3]$ | $X[4]$ |
|---|---|---|---|
| $\pi_1^{(X)}(\mathtt{a}) = 1/2$ | $\pi_2^{(X)}(\mathtt{a}) = 1$ | $\pi_3^{(X)}(\mathtt{a}) = 3/4$ | $\pi_4^{(X)}(\mathtt{a}) = 0$ |
| $\pi_1^{(X)}(\mathtt{b}) = 1/2$ | $\pi_2^{(X)}(\mathtt{b}) = 0$ | $\pi_3^{(X)}(\mathtt{b}) = 1/4$ | $\pi_4^{(X)}(\mathtt{b}) = 1$ |

MATCHING problem works in $O(nm + m\sigma)$ time. A broad spectrum of heuristics improving this algorithm in practice is known; for a survey, see [22]. However, all these algorithms have the same worst-case running time. One of the principal heuristic techniques, coming in different flavours, is *lookahead scoring* that consists in checking if a partial match could possibly be completed by the highest scoring letters in the remaining positions of the scoring matrix and, if not, pruning the naïve search. The PROFILE MATCHING problem can also be solved in $O(\sigma n \log m)$ time via the Fast Fourier Transform [24].

**Our results** As the first result, we show how the lookahead scoring technique combined with a data structure for answering longest common extension (LCE) queries in a string can be applied to obtain simple and efficient algorithms for the standard pattern matching problems on uncertain sequences. For a weighted sequence, by $R$ we denote the size of its list representation. In the case that $\sigma = O(1)$, which often occurs in molecular biology applications, we have $R = O(n)$. In the PROFILE MATCHING problem, we set $M$ as the number of strings that match the scoring matrix with score above $Z$. In general $M \leq \sigma^m$; however, we may assume that for practical data this number is actually much smaller. We obtain the following results:

**Theorem 1.1** PROFILE MATCHING *can be solved in* $O(m\sigma + n \log M)$ *time.*

**Theorem 1.2** WEIGHTED PATTERN MATCHING *can be solved in* $O(R+n \log z)$ *time.*

### 1.2 PROFILE CONSENSUS and MULTICHOICE KNAPSACK

Along the way to our most involved contribution, we study PROFILE CONSENSUS, a consensus problem on uncertain sequences. Specifically, we are to check for the existence of a string that matches two scoring matrices, each above threshold $Z$. The PROFILE CONSENSUS problem is essentially equivalent to the well-known MULTICHOICE KNAPSACK problem (also known as the MULTIPLE CHOICE KNAPSACK problem). In this problem, we are given $n$ classes $C_1, \ldots, C_n$ of at most $\lambda$ items each—$N$ items in total—each item $c$ characterised by a value $v(c)$ and a weight $w(c)$. The goal is to select one item from each class so that the sums of values and of weights of the items are below two specified thresholds, $V$ and $W$. (In the more intuitive formulation of the problem, we require the sum of values to be *above* a specified threshold, but here we consider an equivalent variant in which both parameters are symmetric.) This problem generalises the (binary) KNAPSACK problem, in which we have $\lambda = 2$. The MULTICHOICE KNAPSACK problem is widely used in practice, but most research concerns approximation or heuristic solutions; see [17] and references therein. As far as exact solutions are concerned, the classic meet-in-the middle approach by Horowitz and Sahni [12], originally designed for the (binary) KNAPSACK problem, immediately generalises to an $O^*(\lambda^{\lceil \frac{n}{2} \rceil})$-time[1] solution for MULTICHOICE KNAPSACK.

[1] The $O^*$ notation suppresses factors polynomial with respect to the instance size, whereas the $\tilde{O}$ notation ignores factors polylogarithmic with respect to the instance size (encoded in binary).

Several important problems can be expressed as special cases of the MULTICHOICE KNAPSACK problem using folklore reductions (see [17]). This includes the SUBSET SUM problem, which, for a set of $n$ integers, asks whether there is a subset summing up to a given integer $Q$, and the $k$-SUM problem which, for $k$ classes of $\lambda$ integers, asks to choose one element from each class so that the selected integers sum up to zero. These reductions give immediate hardness results for the MULTICHOICE KNAPSACK problem and thus yield the same consequences for PROFILE CONSENSUS. For the SUBSET SUM problem, as shown in [9, 11], the existence of an $O^*(2^{\varepsilon n})$-time solution for every $\varepsilon > 0$ would violate the Exponential Time Hypothesis (ETH) [15, 20]. Moreover, the $O^*(2^{n/2})$ running time, achieved in [12], has not been improved yet despite much effort. The 3-SUM conjecture [10] and the more general $k$-SUM conjecture state that the 3-SUM and $k$-SUM problems cannot be solved in $O(\lambda^{2-\varepsilon})$ time and $O(\lambda^{\lceil \frac{k}{2} \rceil (1-\varepsilon)})$ time, respectively, for any $\varepsilon > 0$.

**Our results** In the complexities of our algorithms, the instance size of MULTICHOICE KNAPSACK is described by the number of classes $n$, the total number of items $N = |C_1| + \cdots + |C_n|$, and the maximum size of a class $\lambda = \max\{|C_1|, \ldots, |C_n|\}$. We also introduce additional parameters based on the number of solutions with *feasible* weight or value:

$$A_V = \left| \left\{ (c_1, \ldots, c_n) \,:\, c_i \in C_i \text{ for all } i = 1, \ldots, n, \sum_i v(c_i) \leq V \right\} \right|,$$

that is, the number of choices of one element from each class that satisfy the value threshold,

$$A_W = \left| \left\{ (c_1, \ldots, c_n) \,:\, c_i \in C_i \text{ for all } i = 1, \ldots, n, \sum_i w(c_i) \leq W \right\} \right|,$$

$A = \max(A_V, A_W)$, and $a = \min(A_V, A_W)$. We obtain the following result.

**Theorem 1.3** MULTICHOICE KNAPSACK *can be solved in* $O(N + \sqrt{a\lambda} \log A)$ *time.*

Note that $a \leq A \leq \lambda^n$ and thus the running time of our algorithm for MULTICHOICE KNAPSACK is bounded by $O(N + n\lambda^{(n+1)/2} \log \lambda)$. Up to lower order terms (i.e., the factor $n \log \lambda = (\lambda^{(n+1)/2})^{o(1)}$), this matches the time complexities of the fastest known solutions for both SUBSET SUM (also binary KNAPSACK) and 3-SUM. Our parameters identify a new measure of difficulty for the MULTICHOICE KNAPSACK problem. The main novel part of our algorithm for MULTICHOICE KNAPSACK is an appropriate (yet intuitive) notion of *ranks* of partial solutions.

### 1.3 WEIGHTED CONSENSUS and GENERAL WEIGHTED PATTERN MATCHING

Analogously to the PROFILE CONSENSUS problem, we define the WEIGHTED CONSENSUS problem. In the WEIGHTED CONSENSUS problem, given two weighted sequences of the same length, we are to check if there is a string that matches each of them with probability at least $\frac{1}{z}$. A routine to compare user-entered weighted

sequences with existing weighted sequences in the database is used, e.g., in JASPAR,[2] a well-known database of PWMs. Finally, we study a general variant of pattern matching on weighted sequences. In the GENERAL WEIGHTED PATTERN MATCHING (GWPM) problem, both the pattern and the text are weighted. In the most common definition of the problem (see [3, 13]), we are to find all fragments of the text that give a positive answer to the WEIGHTED CONSENSUS problem with the pattern. The authors of [3] proposed an algorithm for the GWPM problem based on the weighted prefix table that works in $O(nz^2 \log z + n\sigma)$ time. Solutions to these problems can be applied in transcriptional regulation: motif and regulatory module finding; and annotation of regulatory genomic regions.

**Our results** For a weighted sequence, by $\lambda$ let us denote the maximal number of letters with score at least $\frac{1}{z}$ at a single position (thus $\lambda \leq \min(\sigma, z)$). Our algorithm for the MULTICHOICE KNAPSACK problem (covered in Section 1.2) yields time complexities $O(R + \sqrt{z\lambda} \log z)$ and $O(n\sqrt{z\lambda} \log z)$ for WEIGHTED CONSENSUS and GWPM, respectively. Using a tailor-made solution based on the same scheme, we obtain faster procedures as specified below.

**Theorem 1.4** *The* GENERAL WEIGHTED PATTERN MATCHING *problem can be solved in* $O(n\sqrt{z\lambda}(\log\log z + \log\lambda))$ *time, and the* WEIGHTED CONSENSUS *problem can be solved in* $O(R + \sqrt{z\lambda}(\log\log z + \log\lambda))$ *time.*

In particular, we obtain the following result for the practical case of $\sigma = O(1)$.

**Corollary 1.5** GENERAL WEIGHTED PATTERN MATCHING *over a constant-sized alphabet can be solved in* $O(n\sqrt{z}\log\log z)$ *time.*

We also provide a simple reduction from MULTICHOICE KNAPSACK to WEIGHTED CONSENSUS, which lets us transfer the negative results to the GWPM problem.

**Theorem 1.6** WEIGHTED CONSENSUS *is NP-hard and cannot be solved in:*

1. $O^*(z^\varepsilon)$ *time for every* $\varepsilon > 0$*, unless the exponential time hypothesis (ETH) fails;*
2. $O^*(z^{0.5-\varepsilon})$ *time for some* $\varepsilon > 0$*, unless there is an* $O^*(2^{(0.5-\varepsilon)n})$*-time algorithm for the* SUBSET SUM *problem;*
3. $\tilde{O}(R + z^{0.5}\lambda^{0.5-\varepsilon})$ *time for some* $\varepsilon > 0$ *and for* $n = O(1)$*, unless the 3-*SUM *conjecture fails.*

For the higher-order terms, our complexities match the conditional lower bounds; therefore, in the proofs of Theorems 1.3 and 1.4 we put significant effort to keep the lower order terms of the complexities as small as possible.

Finally, we analyse the complexity of the MULTICHOICE KNAPSACK and GENERAL WEIGHTED PATTERN MATCHING problems in case of a large $\lambda$. This is

[2] http://jaspar.genereg.net

a theoretical study that shows a possibility of improvement of the complexity for instances that do not originate from the SUBSET SUM and $k$-SUM problems.

**Theorem 1.7** *For every positive integer $k = O(1)$, the* MULTICHOICE KNAPSACK *problem can be solved in $O(N + (a^{\frac{k+1}{2k+1}} + \lambda^k) \log A (\frac{\log A}{\log \lambda})^k)$ time.*

**Theorem 1.8** *If $\lambda^{2k-1} \leq z \leq \lambda^{2k+1}$ for some positive integer $k = O(1)$, then the* WEIGHTED CONSENSUS *problem can be solved in $O(R + (z^{\frac{k+1}{2k+1}} + \lambda^k) \log \lambda)$ time, and the* GWPM *problem can be solved in $O(n(z^{\frac{k+1}{2k+1}} + \lambda^k) \log \lambda)$ time.*

A preliminary version of this research appeared as [18].

### 1.4 Structure of the Paper

We start with Preliminaries, where we recall basic notions on classic strings and formalise the model of computation. The following four sections describe our algorithms: in Section 3 for PROFILE MATCHING; in Section 4 for WEIGHTED PATTERN MATCHING; in Section 5 for PROFILE CONSENSUS; and in Section 6 for WEIGHTED CONSENSUS and GWPM. In Section 7 we present conditional lower bounds for the GWPM problem based on the special cases of MULTICHOICE KNAPSACK. Finally, in Section 8 we perform a multivariate analysis of PROFILE CONSENSUS and GWPM and present improved solutions in the case that $\frac{\log a}{\log \lambda}$ is a constant other than an odd integer.

## 2 Preliminaries

Let $\Sigma = \{1, \ldots, \sigma\}$ be an alphabet. A *string* $S$ over $\Sigma$ is a finite sequence of letters from $\Sigma$. By $\Sigma^m$ we denote the set of strings of length $m$ over $\Sigma$. We denote the length of $S$ by $|S|$ and, for $1 \leq i \leq |S|$, the $i$-th letter of $S$ by $S[i]$. By $S[i\,..\,j]$ we denote the string $S[i] \cdots S[j]$ called a *factor* of $S$ (if $i > j$, then the factor is an empty string). A factor is called a *prefix* if $i = 1$ and a *suffix* if $j = |S|$. For two strings $S$ and $T$, we denote their concatenation by $S \cdot T$ ($ST$ in short).

For a string $S$ of length $n$, by $\mathrm{LCE}(i, j) = lcp(S[i\,..\,n], S[j\,..\,n])$ we denote the length of the longest common prefix of suffixes $S[i\,..\,n]$ and $S[j\,..\,n]$. This value lets us easily determine the longest common prefix $lcp(S[i\,..\,i'], S[j\,..\,j'])$ of any two factors starting at positions $i$ and $j$, respectively. The following fact specifies a well-known efficient data structure answering LCE queries; see [8] for details.

**Fact 2.1** Let $S$ be a string of length $n$ over an integer alphabet of size $\sigma = n^{O(1)}$. After $O(n)$-time preprocessing, in $O(1)$ time one can compute $\mathrm{LCE}(i, j)$ for any indices $i, j$.

The *Hamming distance* between two strings $X$ and $Y$ of the same length, denoted by $d_H(X, Y)$, is the number of positions where the strings differ.

### 2.1 Model of Computations

For problems on weighted sequences, we assume the word-RAM model with word size $w = \Omega(\log n + \log z)$ and integer alphabet of size $\sigma = n^{O(1)}$. We consider the log-probability model of representations of weighted sequences, that is, we assume that probabilities in the weighted sequences and the threshold probability $\frac{1}{z}$ are all of the form $c^{\frac{p}{2^{dw}}}$, where $c$ and $d$ are constants and $p$ is an integer that fits in a constant number of machine words. Additionally, the probability 0 has a special representation. The only operations on probabilities in our algorithms are multiplications and divisions, which can be performed exactly in $O(1)$ time in this model. Our solutions to the MULTICHOICE KNAPSACK problem only assume the word-RAM model with word size $w = \Omega(\log S + \log a)$, where $S$ is the sum of integers in the input instance; this does not affect the $O^*$ running time.

## 3 PROFILE MATCHING

In the PROFILE MATCHING problem, we consider a *scoring matrix* (a profile) $P$ of size $m \times \sigma$. For $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, \sigma\}$, we denote the integer score of the letter $j$ at the position $i$ by $P[i, j]$. The *matching score* of a string $S$ of length $m$ with the matrix $P$ is

$$\text{Score}(S, P) = \sum_{i=1}^{m} P[i, S[i]].$$

If $\text{Score}(S, P) \geq Z$ for an integer *threshold* $Z$, then we say that the string $S$ *matches the matrix* $P$ *above threshold* $Z$. We denote the family of strings $S$ that match $P$ above threshold $Z$ by $\mathbf{M}_Z(P)$.

For a string $T$ and a scoring matrix $P$, we say that $P$ *occurs in* $T$ *at position* $i$ *with threshold* $Z$ if $T[i\,..\,i + m - 1]$ matches $P$ above threshold $Z$. Then $Occ_Z(P, T)$ is the set of all positions where $P$ occurs in $T$. These notions let us define PROFILE MATCHING:

PROFILE MATCHING PROBLEM

**Input:** A string $T$ of length $n$, a scoring matrix $P$ of size $m \times \sigma$, and a threshold $Z$.

**Output:** The set $Occ_Z(P, T)$.

**Parameters:** $M = |\mathbf{M}_Z(P)|$.

### 3.1 Solution to PROFILE MATCHING

For a scoring matrix $P$, the *heavy string* of $P$, denoted $\mathbf{H}(P)$, is constructed by choosing at each position the heaviest letter, that is, the letter with the maximum

score (breaking ties arbitrarily). In other words, $\mathbf{H}(P)$ is a string that matches $P$ with the maximum score.

**Observation 3.1** If $\text{Score}(S, P) \geq Z$ for a string $S$ of length $m$ and an $m \times \sigma$ scoring matrix $P$, then $d_H(\mathbf{H}(P), S) \leq \lfloor \log |\mathbf{M}_Z(P)| \rfloor$.

*Proof* Let $d = d_H(\mathbf{H}(P), S)$. We can construct $2^d$ strings of length $|S|$ that match $P$ with a score above $Z$ by taking either of the letters $S[j]$ or $\mathbf{H}(P)[j]$ at each position $j$ such that $S[j] \neq \mathbf{H}(P)[j]$. Hence, $2^d \leq |\mathbf{M}_Z(P)|$, which concludes the proof. □

Our solution for the PROFILE MATCHING problem works as follows. We first construct $P' = \mathbf{H}(P)$ and the data structure for *lcp*-queries in $P'T$. Let the variable $s$ store the matching score of $P'$. In the $p$-th step, we calculate the matching score of $T[p\,..\,p+m-1]$ by iterating through subsequent mismatches between $P'$ and $T[p\,..\,p+m-1]$ and making adequate updates in the matching score $s$. The mismatches are found using *lcp*-queries: If $P'[i]$ is aligned against $T[j]$, we compute $\Delta = lcp(P'[i\,..\,m], T[j\,..\,n])$. Then, $P'[i\,..\,i+\Delta-1] = T[j\,..\,j+\Delta-1]$, but $P'[i+\Delta] \neq T[j+\Delta]$ yields a mismatch (assuming $i+\Delta \leq m$ and $j+\Delta \leq n$). To locate the next mismatch, we need to repeat the procedure above with $i$ and $j$ increased by $\Delta+1$. This process terminates when the score drops below $Z$ or when all the mismatches have been found. In the end, we include $p$ in $Occ_Z(P, T)$ if the final matching score is above $Z$. A pseudocode is given in the ProfileMatching($P$, $T$, $Z$) procedure.

**Procedure** ProfileMatching($P$, $T$, $Z$)

```
m := |P|; n := |T|; Occ := ∅;
P' := H(P);
Construct the data structure for lcp-queries in P'T;
s := Σ_{j=1}^{m} P[j, P'[j]];
for p := 1 to n − m + 1 do
    s' := s; i := 1; j := p;
    while s' ≥ Z and i ≤ m do
        Δ := lcp(P'[i..m], T[j..n]);
        i := i + Δ + 1; j := j + Δ + 1;
        if i ≤ m + 1 then
            s' := s' + P[i − 1, T[j − 1]] − P[i − 1, P'[i − 1]];
    if s' ≥ Z then insert p to Occ;
return Occ;
```

We obtain the following result.

**Theorem 1.1** PROFILE MATCHING *can be solved in* $O(m\sigma + n \log M)$ *time.*

*Proof* Let us bound the time complexity of the presented algorithm. The heavy string $P'$ can be computed in $O(m\sigma)$ time. The data structure for *lcp*-queries in $P'T$ can be

constructed in $O(n+m)$ time by Fact 2.1. Finally, for each position $p$ in the text $T$ we will consider at most $\lfloor \log M \rfloor + 1$ mismatches between $P'$ and $T$, as afterwards the score $s'$ drops below $Z$ due to Observation 3.1. □

## 4 Weighted Pattern Matching

A *weighted sequence* $X = X[1]\cdots X[n]$ of length $|X| = n$ over alphabet $\Sigma$ is a sequence of sets of pairs of the form $X[i] = \{(j,\ \pi_i^{(X)}(j)) \ :\ j \in \Sigma\}$. Here, $\pi_i^{(X)}(j)$ is the occurrence probability of the letter $j$ at the position $i \in \{1,\ldots,n\}$. These values are non-negative and sum up to 1 for a given $i$. For all our algorithms, it is sufficient that the probabilities sum up to *at most* 1 for each position. Also, the algorithms sometimes produce auxiliary weighted sequences with sum of probabilities being smaller than 1 on some positions.

We denote the maximum number of letters occurring at a single position of the weighted sequence (with non-zero probability) by $\lambda$ and the total size of the representation of a weighted sequence by $R$. The standard representation consists of $n$ lists with up to $\lambda$ elements each, so $R = O(n\lambda)$. However, the lists can be shorter in general. Also, if the threshold probability $\frac{1}{z}$ is specified, at each position of a weighted sequence it suffices to store letters with probability at least $\frac{1}{z}$, and clearly there are at most $z$ such letters for each position. This reduction can be performed in linear time, so we shall always assume that $\lambda \le z$. Moreover, the assumption that $\Sigma$ is an integer alphabet of size $\sigma = n^{O(1)}$ lets us assume without loss of generality that the entries $(j, \pi_i^{(X)}(j))$ in the lists representing $X[i]$ are ordered by increasing $j$: if this is not the case, we can simultaneously sort these lists in linear time.

The *probability of matching* of a string $S$ with a weighted sequence $X$, $|S| = |X| = n$, is

$$\mathbf{P}(S, X) = \prod_{i=1}^{n} \pi_i^{(X)}(S[i]).$$

We say that a string $S$ *matches a weighted sequence* $X$ *with probability at least* $\frac{1}{z}$, denoted by $S \approx_{\frac{1}{z}} X$, if $\mathbf{P}(S, X) \ge \frac{1}{z}$. We also denote $\mathbf{M}_z(X) = \{S \in \Sigma^n : \mathbf{P}(S, X) \ge \frac{1}{z}\}$.

Given a weighted sequence $T$, by $T[i\,..\,j]$ we denote a weighted sequence, called a *factor* of $T$, equal to $T[i]\cdots T[j]$ (if $i > j$, then the factor is empty). We say that a string $P$ of length $m$ *occurs* in $T$ at position $i$ if $P$ matches the factor $T[i\,..\,i+m-1]$. The set of positions where $P$ occurs in $T$ is denoted by $Occ_{\frac{1}{z}}(P, T)$.

Weighted Pattern Matching Problem

**Input:** A string $P$ of length $m$, a weighted sequence $T$ of length $n$ with at most $\lambda$ letters at each position and $R$ in total, and a threshold $\frac{1}{z}$.

**Output:** The set $Occ_{\frac{1}{z}}(P, T)$.

### 4.1 Weighted Sequences versus Profiles

As shown below, profiles and weighted sequences are essentially equivalent objects.

**Fact 4.1** 1. Given a weighted sequence $X$ of length $n$ over an alphabet of size $\sigma$ and a probability $\frac{1}{z}$, one can construct in $O(n\sigma)$ time an $n \times \sigma$ profile $P$ and a threshold $Z$ such that $\mathbf{M}_Z(P) = \mathbf{M}_z(X)$.
2. Given an $m \times \sigma$ profile $P$ and a threshold $Z$, one can construct in $O(m\sigma)$ time a weighted sequence $X$ and a probability $\frac{1}{z}$ such that $\mathbf{M}_z(X) = \mathbf{M}_Z(P)$.

*Proof* Given a weighted sequence $X$, one can construct an equivalent profile $P$ setting $P[i, s] = -\log \pi_i^{(X)}(s)$ for each position $i$ and character $s$. If $\pi_i^{(X)}(s) = 0$, we set $P[i, s] = \infty$ (which can be replaced by a sufficiently large finite value after we fix the threshold $Z$). The profile $P$ satisfies $\mathbf{M}_Z(P) = \mathbf{M}_z(X)$ for $Z = \log z$.

To construct an inverse mapping, we need to normalise the scores first. For this, we construct a normalised profile $P'$ setting $P'[i, s] := P[i, s] + \log(\sum_{s\in\Sigma} 2^{-P[i,s]})$. As a result, we have $\mathbf{M}_Z(P) = \mathbf{M}_{Z'}(P')$ for $Z' = Z + \sum_{i=1}^{m} \log(\sum_{s\in\Sigma} 2^{-P[i,s]})$. Now, we can build an equivalent weighted sequence $X$ by setting $\pi_i^{(X)}(s) = 2^{-P'[i,s]}$. Note that

$$\sum_{s\in\Sigma} \pi_i^{(X)}(s) = \sum_{s\in\Sigma} 2^{-P[i,s]-\log(\sum_{s'\in\Sigma} 2^{-P[i,s']})} = \left(\sum_{s\in\Sigma} 2^{-P[i,s]}\right) \cdot 2^{-\log(\sum_{s\in\Sigma} 2^{-P[i,s]})} = 1$$

holds as required. Moreover, $\mathbf{M}_z(X) = \mathbf{M}_{Z'}(P') = \mathbf{M}_Z(P)$ for $z = 2^{Z'}$. □

In the light of Fact 4.1, it may seem that the results for profiles and weighted sequences should coincide. However, we use different parameters to study the complexity of the algorithmic problems in these models: for profiles this is the number $|\mathbf{M}_Z(P)|$ of matching strings, while for weighted sequence this is the inverse $z$ of the threshold probability $\frac{1}{z}$. These parameters are related by the following observation:

**Observation 4.2** A weighted sequence $X$ satisfies $|\mathbf{M}_z(X)| \leq z$ for every threshold.

However, the bound $|\mathbf{M}_z(X)| \leq z$ is not tight in general, which gives more power to algorithms parameterised by $z$. Moreover, $z$ is a part of the input (as opposed to $|\mathbf{M}_Z(P)|$ for profiles). Furthermore, it is natural to consider a common threshold probability $\frac{1}{z}$ for multiple weighted sequences, e.g., factors of a weighted text $T$ as in Weighted Pattern Matching.

A more technical difference lies in the representation of profiles and weighted sequences, which we have chosen consistently with the literature. A profile is stored as a dense $m \times \sigma$ matrix, while in a weighted sequence of the same length we do not explicitly keep entries with $\pi_i^{(X)}(s) = 0$, so the input size $R$ can be smaller than $m \cdot \sigma$. This allows for faster algorithms—because reading the input takes less time—but at the same time poses some challenges—because $\pi_i^{(X)}(s)$ cannot be accessed in constant time, unless $\sigma = O(1)$ or we allow randomisation. This is illustrated below in case of the Weighted Pattern Matching problem and also in Section 6.

### 4.2 Solution to Weighted Pattern Matching

The approach from our solution to Profile Matching can be used for Weighted Pattern Matching. In a natural way, we extend the notion of a heavy string to weighted sequences. This lets us restate Observation 3.1 in the language of probabilities instead of scores.

**Observation 4.3** If a string $P$ matches a weighted sequence $X$ of the same length with probability at least $\frac{1}{z}$, then $d_H(\mathbf{H}(X), P) \leq \lfloor \log z \rfloor$.

Comparing to the solution to Profile Matching, we compute the heavy string of the text instead of the pattern. An auxiliary variable $\alpha$ stores the matching probability between a factor of $\mathbf{H}(T)$ and the corresponding factor of $T$; it is updated when we move to the next position of the text. The rest of the algorithm is basically the same as previously; see the pseudocode of WeightedPatternMatching($P$, $T$, $\frac{1}{z}$).

**Procedure** WeightedPatternMatching($P$, $T$, $\frac{1}{z}$)

$m := |P|$; $n := |T|$; $Occ := \emptyset$;
$T' := \mathbf{H}(T)$;
Construct the data structure for *lcp*-queries in $PT'$;
$\alpha := \prod_{j=1}^{m} \pi_j^{(T)}(T'[j])$;
**for** $p := 1$ **to** $n - m + 1$ **do**
  $\alpha' := \alpha$; $i := 1$; $j := p$;
  **while** $\alpha' \geq \frac{1}{z}$ **and** $i \leq m$ **do**
    $\Delta := lcp(P[i\,..\,m], T'[j\,..\,n])$;
    $i := i + \Delta + 1$; $j := j + \Delta + 1$;
    **if** $i \leq m + 1$ **then**
      $\alpha' := \alpha' \cdot \pi_{j-1}^{(T)}(P[i-1]) \,/\, \pi_{j-1}^{(T)}(T'[j-1])$;
  **if** $\alpha' \geq \frac{1}{z}$ **then** insert $p$ to $Occ$;
  **if** $p \leq n - m$ **then**
    $\alpha := \alpha \cdot \pi_{p+m}^{(T)}(T'[p+m]) \,/\, \pi_p^{(T)}(T'[p])$;
**return** $Occ$;

**Implementation for large alphabets** The algorithm above takes $O(n \log z)$ time for $\sigma = O(1)$. In the general case, we need to efficiently implement the following operations on the weighted sequence:

- finding the letter with the maximum probability at a given position,
- computing the probability of a given letter at a given position.

For a weighted sequence in the standard list representation, we can compute the maximum-probability letter at each position in $O(R)$ time which lets us perform the former operation in $O(1)$ time. We also explicitly store the probabilities of the heaviest letters so that $\pi_j^{(T)}(T'[j])$ can be retrieved in constant time for any index $j$.

To implement the latter operation for an arbitrary character, we store each $T[j]$ in a weight-balanced binary tree [21], with the weight of $(s, \pi_j^{(T)}(s))$ equal to $\pi_j^{(T)}(s)$. As a result, any $\pi_j^{(T)}(s)$ can be retrieved in $O(-\log \pi_j^{(T)}(s)) = O(\log z)$ time. During the course of the $p$-th step of the algorithm, $\alpha'$ is a product of some probabilities including all the retrieved probabilities $\pi_j^{(T)}(s)$ with $s \neq T'[j]$. The **while** loop is executed only when $\alpha' \geq \frac{1}{z}$, so the product of these probabilities (excluding the one retrieved in the final iteration) is at least $\frac{1}{z}$. Consequently, the overall retrieval time in the $p$-th step is $O(\log z)$.

This way, we can implement the algorithm in $O(R + n \log z)$ time.

**Theorem 1.2** WEIGHTED PATTERN MATCHING *can be solved in* $O(R+n \log z)$ *time.*

*Remark 4.4* In the same complexity one can solve GWPM with a solid text.

## 5 PROFILE CONSENSUS and MULTICHOICE KNAPSACK

Let us start with a precise statement of the MULTICHOICE KNAPSACK problem.

MULTICHOICE KNAPSACK PROBLEM

**Input:** A set $\mathbf{C}$ of $N$ items partitioned into $n$ disjoint classes $C_i$, each of size at most $\lambda$, two integers $v(c)$ and $w(c)$ for each item $c \in \mathbf{C}$, and two thresholds $V$ and $W$.

**Question:** Does there exist a *choice* $S$ (a set $S \subseteq \mathbf{C}$ such that $|S \cap C_i| = 1$ for each $i$) satisfying both $\sum_{c \in S} v(c) \leq V$ and $\sum_{c \in S} w(c) \leq W$?

**Parameters:** $A_V$ and $A_W$: the number of choices $S$ satisfying $\sum_{c \in S} v(c) \leq V$ and $\sum_{c \in S} w(c) \leq W$, respectively; as well as $A = \max(A_V, A_W)$ and $a = \min(A_V, A_W)$.

For a fixed instance of MULTICHOICE KNAPSACK, we say that $S$ is a *partial choice* if $|S \cap C_i| \leq 1$ for each class. The set $D = \{i : |S \cap C_i| = 1\}$ is called its *domain*. For a partial choice $S$, we define $v(S) = \sum_{c \in S} v(c)$ and $w(S) = \sum_{c \in S} w(c)$.

### 5.1 PROFILE CONSENSUS versus MULTICHOICE KNAPSACK

As shown below, PROFILE CONSENSUS and MULTICHOICE KNAPSACK are essentially equivalent problems.

**Fact 5.1** 1. Consider an instance of PROFILE CONSENSUS with two $m \times \sigma$ profiles $P$, $Q$ and a common threshold $Z$. In $O(m\sigma)$ time one can construct an equivalent instance of MULTICHOICE KNAPSACK with $m$ classes of $\sigma$ items each, $A_V = |\mathbf{M}_Z(P)|$, and $A_W = |\mathbf{M}_Z(Q)|$.

2. Consider an instance of MULTICHOICE KNAPSACK with $n$ classes of at most $\lambda$ items each. In $O(n\lambda)$ time one can construct an equivalent instance of PROFILE

CONSENSUS with two $n \times \lambda$ profiles $P$, $Q$ and a common threshold $Z$ such that $|\mathbf{M}_Z(P)| = A_V$ and $|\mathbf{M}_Z(Q)| = A_W$.

*Proof* Given an instance $(P, Q, Z)$ of the PROFILE CONSENSUS problem, we construct an equivalent instance of MULTICHOICE KNAPSACK with $m$ classes of $\sigma$ items each, denoted $c_{i,j}$ for $1 \leq i \leq m$ and $1 \leq j \leq \sigma$, each with value $v(c_{i,j}) = -P[i, j]$ and weight $w(c_{i,j}) = -Q[i, j]$. We set both thresholds to $V = W = -Z$. It is straightforward to verify that the constructed instance satisfies the required conditions.

This construction is easily reversible if $V = W$ and the size of each class is $\lambda$. In general, we add dummy items (with infinite or very large weight and value), decrease the weight of each item by $\frac{1}{n}(W - V)$, and decrease the weight threshold to $V$. □

The only technical difference between MULTICHOICE KNAPSACK and PROFILE CONSENSUS is that the profiles are stored as dense $m \times \sigma$ matrices while the classes in MULTICHOICE KNAPSACK can be of different size so the input size $N$ can be smaller than the number of classes $n$ times the bound $\lambda$ on the class size.

Below, we formulate our results in the more established language of MULTICHOICE KNAPSACK.

### 5.2 Overview of the Solution

The classic $O(2^{n/2})$-time solution to the KNAPSACK problem [12] is based on a meet-in-the-middle approach. The set $D = \{1, \ldots, n\}$ is partitioned into two domains $D_1, D_2$ of size roughly $n/2$, and for each $D_i$, all partial choices $S$ are generated and ordered by $v(S)$. This reduces the problem to an instance of MULTICHOICE KNAPSACK with two classes, which is solved using a folklore linear-time solution (described for completeness in Section 5.5).

The meet-in-the-middle approach to KNAPSACK generalises directly to a solution to MULTICHOICE KNAPSACK. The partition may be chosen as to balance the number of partial choices in each domain, and so the worst-case time complexity is $O(\sqrt{Q\lambda})$, where $Q = \prod_{i=1}^{n} |C_i|$ is the number of choices.

Our aim in this section is to replace $Q$ with the parameter $a$ (which never exceeds $Q$). The overall running time is going to be $O(N + \sqrt{a\lambda} \log A)$.

Again, we will partition the set of classes into two groups, for each group we will generate a subset of all partial choices, and then we will check if two partial choices can be joined into a feasible solution. However, several questions arise with this approach in order to obtain the desired complexity:

(1) How to partition the set of classes?
(2) In what order should the partial choices be generated?
(3) How many partial choices should be generated, given that the value of the parameter $a$ is not known in advance?

As for question (1), we consider all partitions of the form $D = \{1, \ldots, j\} \cup \{j + 1, \ldots, n\}$ for $1 \leq j \leq n$. This results in an extra $O(n)$ factor in the time complexity.

However, in Section 5.7 we introduce preprocessing which reduces the general case to the case when $n = O\left(\frac{\log A}{\log \lambda}\right)$.

A natural idea to deal with question (2) is to consider only partial choices with small values $v(S)$ or $w(S)$. This is close to our actual solution, which is based on a notion of *ranks* of partial choices that we introduce in Section 5.3.

Finally, to tackle question (3), we generate the partial choices batch-wise until either a solution is found or we can certify that it does not exist. The idea of this step is presented also in Section 5.3, while the generation procedure is detailed in Section 5.4. While dealing with these issues, a careful implementation is required to avoid several further extra factors in the running time.

In the end, we show that the number of partial choices that need to be generated is indeed $O(\sqrt{a\lambda})$. Our final solution to MULTICHOICE KNAPSACK is presented in Section 5.6 without the instance size reduction and in Section 5.8 using the reduction.

### 5.3 Ranks of Partial Choices

For a partial choice $S$, we define $\mathrm{rank}_v(S)$ as the number of partial choices $S'$ with the same domain for which $v(S') \leq v(S)$. We symmetrically define $\mathrm{rank}_w(S)$. For simplicity, if $c \in C_i$, we denote $\mathrm{rank}_v(c) = \mathrm{rank}_v(\{c\})$ and $\mathrm{rank}_w(c) = \mathrm{rank}_w(\{c\})$. Ranks are introduced as an analogue of match probabilities in weighted sequences. Probabilities are multiplicative, while for ranks we have submultiplicativity:

**Fact 5.2** If $S = S_1 \cup S_2$ is a decomposition of a partial choice $S$ into two disjoint subsets, then $\mathrm{rank}_v(S_1)\,\mathrm{rank}_v(S_2) \leq \mathrm{rank}_v(S)$ (and same for $\mathrm{rank}_w$).

*Proof* Let $D_1$ and $D_2$ be the domains of $S_1$ and $S_2$, respectively. For every partial choices $S'_1$ over $D_1$ and $S'_2$ over $D_2$ such that $v(S'_1) \leq v(S_1)$ and $v(S'_2) \leq v(S_2)$, we have $v(S'_1 \cup S'_2) = v(S'_1) + v(S'_2) \leq v(S)$. Hence, $S'_1 \cup S'_2$ must be counted while determining $\mathrm{rank}_v(S)$. □

For $0 \leq j \leq n$, let $\mathbf{L}_j$ be the list of partial choices with domain $\{1, \ldots, j\}$ ordered by value $v(S)$, and for $\ell > 0$ let $\mathbf{L}_j[\ell]$ be the $\ell$-th element of $\mathbf{L}_j$. Analogously, for $1 \leq j \leq n+1$, we define $\mathbf{R}_j$ as the list of partial choices over $\{j, \ldots, n\}$ ordered by $v(S)$, and for $r > 0$, $\mathbf{R}_j[r]$ as the $r$-th element of $\mathbf{R}_j$. If any of the partial choices $\mathbf{L}_j[\ell]$, $\mathbf{R}_j[r]$ does not exist, we assume that its value is $\infty$.

The following two observations yield a decomposition of each choice into a single item and two partial solutions of a small rank. Observe that we do not need to know $A_V$ in order to check if the ranks are sufficiently large.

**Lemma 5.3** *Let $\ell$ and $r$ be positive integers such that $v(\mathbf{L}_j[\ell]) + v(\mathbf{R}_{j+1}[r]) > V$ for each $0 \leq j \leq n$. For every choice $S$ with $v(S) \leq V$, there is an index $j \in \{1, \ldots, n\}$ and a decomposition $S = L \cup \{c\} \cup R$ such that $v(L) < v(\mathbf{L}_{j-1}[\ell])$, $c \in C_j$, and $v(R) < v(\mathbf{R}_{j+1}[r])$.*

*Proof* Let $S = \{c_1, \ldots, c_n\}$ with $c_i \in C_i$ and, for $0 \le i \le n$, let $S_i = \{c_1, \ldots, c_i\}$. If $v(S_{n-1}) < v(\mathbf{L}_{n-1}[\ell])$, we set $L = S_{n-1}$, $c = c_n$, and $R = \emptyset$, satisfying the claimed conditions.

Otherwise, we define $j$ as the smallest index $i$ such that $v(S_i) \ge v(\mathbf{L}_i[\ell])$, and we set $L = S_{j-1}, c = c_j$, and $R = S \setminus S_j$. The definition of $j$ implies $v(L) < v(\mathbf{L}_{j-1}[\ell])$ and $v(L \cup \{c\}) \ge v(\mathbf{L}_j[\ell])$. Moreover, we have $v(L \cup \{c\}) + v(R) = v(S) \le V < v(\mathbf{L}_j[\ell]) + v(\mathbf{R}_{j+1}[r])$, and thus $v(R) < v(\mathbf{R}_{j+1}[r])$. □

**Fact 5.4** Let $\ell, r > 0$. If $v(\mathbf{L}_j[\ell]) + v(\mathbf{R}_{j+1}[r]) \le V$ for some $j \in \{0, \ldots, n\}$, then $\ell \cdot r \le A_V$.

*Proof* Let $L$ and $R$ be the $\ell$-th and $r$-th entry in $\mathbf{L}_j$ and $\mathbf{R}_{j+1}$, respectively. Note that $v(L \cup R) \le V$ implies $\mathrm{rank}_v(L \cup R) \le A_V$ by definition of $A_V$. Moreover, $\mathrm{rank}_v(L) \ge \ell$ and $\mathrm{rank}_v(R) \ge r$ (the equalities may be sharp due to draws). Now, Fact 5.2 yields the claimed bound. □

### 5.4 Generating Partial Choices of Small Rank

Note that $\mathbf{L}_j$ can be obtained by interleaving $|C_j|$ copies of $\mathbf{L}_{j-1}$, where each copy corresponds to extending the choices from $\mathbf{L}_{j-1}$ with a different item. If we were to construct $\mathbf{L}_j$ having access to the whole $\mathbf{L}_{j-1}$, we could apply the following standard procedure. For each $c \in C_j$, we maintain an *iterator* on $\mathbf{L}_{j-1}$ pointing to the first element $S$ on $\mathbf{L}_{j-1}$ for which $S \cup \{c\}$ has not yet been added to $\mathbf{L}_j$. The associated *value* is $v(S \cup \{c\})$. All iterators initially point at the first element of $\mathbf{L}_{j-1}$. Then the next element to append to $\mathbf{L}_j$ is always $S \cup \{c\}$ corresponding to the iterator with minimum value. Having processed this partial choice, we advance the iterator (or remove it, once it has already scanned the whole $\mathbf{L}_{j-1}$). This process can be implemented using a binary heap $H_j$ as a priority queue, so that initialisation requires $O(|C_j|)$ time and outputting a single element takes $O(\log |C_j|)$ time. Each partial choice $S \in \mathbf{L}_j$ is stored in $O(1)$ space using a pointer to a partial choice $S' \in \mathbf{L}_{j-1}$ such that $S = S' \cup \{c\}$ for some $c \in C_j$.

For $r \ge 0$, let $\mathbf{L}_j^{(i)}$ be the prefix of $\mathbf{L}_j$ of length $\min(i, |\mathbf{L}_j|)$ and $\mathbf{R}_j^{(i)}$ be the prefix of $\mathbf{R}_j$ of length $\min(i, |\mathbf{R}_j|)$. A technical transformation of the procedure stated above leads to an online algorithm that constructs the prefixes $\mathbf{L}_j^{(i)}$ and $\mathbf{R}_j^{(i)}$, as shown in the following lemma. Along with each reported partial choice $S$, the algorithm also computes $w(S)$.

**Lemma 5.5** *After $O(N)$-time initialisation, one can compute $\boldsymbol{L}_1[i], \ldots, \boldsymbol{L}_n[i]$ knowing $\boldsymbol{L}_1^{(i-1)}, \ldots, \boldsymbol{L}_n^{(i-1)}$ in $O(n \log \lambda)$ time. Symmetrically, one can construct $\boldsymbol{R}_1[i], \ldots, \boldsymbol{R}_n[i]$ from $\boldsymbol{R}_1^{(i-1)}, \ldots, \boldsymbol{R}_n^{(i-1)}$ in the same time complexity.*

*Proof* Our online algorithm is going to use the same approach as the offline computation of lists $\mathbf{L}_j^{(i)}$. The order of computations will be different, though.

At each step, for $j = 1$ to $n$ we shall extend lists $\mathbf{L}_j^{(i-1)}$ with a single element (unless the whole $\mathbf{L}_j$ has already been generated) from the top of the heap $H_j$. We keep an invariant that each iterator in $H_j$ always points to an element that is already in $\mathbf{L}_{j-1}^{(i-1)}$ or to $\mathbf{L}_{j-1}[i]$: the first element that has not been yet added to $\mathbf{L}_{j-1}$, which is represented by the top of the heap $H_{j-1}$.

We initialise the heaps as follows: we introduce $H_0$ which represents the empty choice $\emptyset$ with $v(\emptyset) = 0$. Next, for $j = 1, \ldots, n$ we build the heap $H_j$ representing $|C_j|$ iterators initially pointing to the top of $H_{j-1}$. The initialisation takes $O(N)$ time in total since a binary heap can be constructed in time linear in its size.

At each step, the lists $\mathbf{L}_j^{(i-1)}$ are extended for consecutive values $j$ from 1 to $n$. Since $\mathbf{L}_{j-1}^{(i-1)}$ is extended before $\mathbf{L}_j^{(i-1)}$, by the invariant, all iterators in $H_j$ point to the elements of $\mathbf{L}_{j-1}^{(i)}$ while we compute $\mathbf{L}_j[i]$. We take the top of $H_j$ and move it to $\mathbf{L}_j^{(i)}$. Next, we advance the corresponding iterator and update its position in the heap $H_j$. After this operation, the iterator might point to the top of $H_{j-1}$. If $H_{j-1}$ is empty, this means that the whole list $\mathbf{L}_{j-1}$ has already been generated and traversed by the iterator. In this case, we remove the iterator.

This way we indeed simulate the previous offline solution. A single phase makes $O(1)$ operations on each heap $H_j$. The running time is bounded by $O(\sum_j \log |C_j|) = O(n \log \lambda)$ at each step of the algorithm. □

### 5.5 MULTICHOICE KNAPSACK for $n = 2$ Classes

Let us recall the final processing of the meet-in-the-middle solution to the KNAPSACK problem [12]. We formulate it in terms of MULTICHOICE KNAPSACK with two classes.

An item $c \in C_j$ is *irrelevant* if there is another item $c' \in C_j$ that *dominates* $c$, i.e., such that $v(c) \geq v(c')$ and $w(c) \geq w(c')$. Observe that removing an irrelevant item leads to an equivalent instance of the MULTICHOICE KNAPSACK problem, and it may only decrease the parameters $A_V$ and $A_W$.

**Lemma 5.6** *The* MULTICHOICE KNAPSACK *problem can be solved in* $O(N)$ *time if* $n = 2$ *and the elements* $c$ *of* $C_1$ *and* $C_2$ *are sorted by* $v(c)$.

*Proof* Since the items of $C_1$ and $C_2$ are sorted by $v(c)$, a single scan through these items lets us remove all irrelevant elements. Next, for each $c_1 \in C_1$ we compute $c_2 \in C_2$ such that $v(c_2) \leq V - v(c_1)$ but otherwise $v(c_2)$ is largest possible. As we have removed irrelevant elements from $C_2$, this item also minimises $w(c_2)$ among all elements satisfying $v(c_2) \leq V - v(c_1)$. Hence, if there is a feasible solution containing $c_1$, then $\{c_1, c_2\}$ is feasible. If we process elements $c_1$ by non-decreasing values $v(c_1)$, the values $v(c_2)$ do not increase, and thus the items $c_2$ can be computed in $O(N)$ time in total. □

### 5.6 MULTICHOICE KNAPSACK Parameterised by $a$

Combining the procedures of Lemmas 5.5 and 5.6 with the combinatorial results of Section 5.3, we obtain the first algorithm for MULTICHOICE KNAPSACK parameterised by $a$.

**Proposition 5.7** MULTICHOICE KNAPSACK *can be solved in* $O(n(\lambda + \sqrt{a\lambda})\log\lambda)$ *time.*

*Proof* Below, we give an algorithm working in $O(n(\lambda + \sqrt{A_V\lambda})\log\lambda)$ time. The final solution runs it in parallel on the original instance and on the instance with $v$ and $V$ swapped with $w$ and $W$, waiting until at least one of them terminates.

We increment an integer $r$ starting from 1, maintaining $\ell = \lceil \frac{r}{\lambda} \rceil$ and the lists $\mathbf{L}_j^{(\ell)}$ and $\mathbf{R}_{j+1}^{(r)}$ for $0 \le j \le n$, as long as $v(\mathbf{L}_j[\ell]) + v(\mathbf{R}_{j+1}[r]) \le V$ for some $j$ (or until all the lists have been completely generated). By Fact 5.4, we stop at $r = O(\sqrt{A_V\lambda})$ and due to Lemma 5.5, the process takes $O(n\sqrt{A_V\lambda}\log\lambda)$ time.

According to Lemma 5.3, every feasible solution $S$ admits a decomposition $S = L \cup \{c\} \cup R$ with $L \in \mathbf{L}_{j-1}^{(\ell)}$, $c \in C_j$, and $R \in \mathbf{R}_{j+1}^{(r)}$ for some index $j$. We consider all possibilities for $j$. For each of them we will reduce searching for $S$ to an instance of the MULTICHOICE KNAPSACK problem with 2 classes of $O(\sqrt{A_V\lambda})$ items. By Lemma 5.6, these instances can be solved in $O(n\sqrt{A_V\lambda})$ time in total.

The items of the $j$-th instance are going to belong to classes $\mathbf{L}_{j-1}^{(\ell)} \odot C_j$ and $\mathbf{R}_{j+1}^{(r)}$, where $\mathbf{L}_{j-1}^{(\ell)} \odot C_j = \{L \cup \{c\} : L \in \mathbf{L}_{j-1}^{(\ell)}, c \in C_j\}$. The set $\mathbf{L}_{j-1}^{(\ell)} \odot C_j$ is constructed by merging $|C_j| \le \lambda$ sorted lists, each of size $\ell = O(1 + \sqrt{A_V/\lambda})$. This takes $O((\lambda + \sqrt{A_V\lambda})\log\lambda)$ time, which results in $O(n(\lambda + \sqrt{A_V\lambda})\log\lambda)$ time over all indices $j$.

Clearly, each feasible solution of the constructed instances represents a feasible solution of the initial instance, and by Lemma 5.3, every feasible solution of the initial instance has its counterpart in one of the constructed instances. □

### 5.7 Preprocessing to Reduce Instance Size

In order to improve the running time for MULTICHOICE KNAPSACK, we develop two reductions and run them as preprocessing to the procedure of Proposition 5.7. First, we observe that items $c$ with $\mathrm{rank}_v(c) > A_V$ or $\mathrm{rank}_w(c) > A_W$ cannot belong to any feasible solution. Moreover their removal results in $\lambda \le a$, which lets us hide the $O(n\lambda\log\lambda)$ term in the running time. Our second reduction decreases the number of classes $n$ to $O\left(\frac{\log A}{\log\lambda}\right)$. For this, we repeatedly remove irrelevant items (as defined in Section 5.5) and merge small classes into their Cartesian product (so that the class sizes are more balanced).

For each class $C_i$, let $v_{\min}(i) = \min\{v(c) : c \in C_i\}$. Also, let $V_{\min} = \sum_{i=1}^{n} v_{\min}(i)$; note that $V_{\min}$ is the smallest possible value $v(S)$ of a choice $S$. We symmetrically define $w_{\min}(i)$ and $W_{\min}$.

**Lemma 5.8** *Given an instance $I$ of the* MULTICHOICE KNAPSACK *problem, one can compute in $O(N)$ time an equivalent instance $I'$ with $N' \leq N$, $n' = n$, $A'_V = A_V$, $A'_W = A_W$, and $\lambda' \leq \min(\lambda, a)$.*

*Proof* From each class $C_i$ we remove all items $c$ such that $V_{\min} + v(c) - v_{\min}(i) > V$ or $W_{\min} + w(c) - w_{\min}(i) > W$. Afterwards, for each item $c \in C_i$ one can obtain a choice $S$ such that $c \in S$ and $v(S) \leq V$ (or $w(S) \leq W$) by choosing the elements with the minimal value (minimal weight, respectively) in all the remaining classes. □

Our second preprocessing consists of several steps. First, we quickly reduce the number of classes to $n = O(\log A)$.

**Lemma 5.9** *Given an instance $I$ of the* MULTICHOICE KNAPSACK *problem, one can compute in linear time an equivalent instance $I'$ with $N' \leq N$, $A'_V \leq A_V$, $A'_W \leq A_W$, $\lambda' \leq \lambda$, and $n' \leq 2 \log A$.*

*Proof* Observe that if a class $C_i$ contains an item $c$ for which both $v(c) = v_{\min}(i)$ and $w(c) = w_{\min}(i)$, then we can greedily include it in the solution $S$. Hence, we can remove such a class, setting $V := V - v_{\min}(i)$ and $W := W - w_{\min}(i)$. We execute this reduction rule exhaustively, which clearly takes $O(N)$ time in total and may only decrease the parameters $A_V$ and $A_W$. After the reduction, the minima $v_{\min}(i)$ and $w_{\min}(i)$ must be attained by distinct items of every class $C_i$.

We shall prove that now we can either find out that $A \geq 2^{n/2}$ or that we are dealing with a NO-instance. To decide which case holds, let us define $\Delta_V(i)$ as the difference between the second smallest value in the multiset $\{v(c) : c \in C_i\}$ and $v_{\min}(i)$. We set $\Delta_V^{\text{mid}}$ as the sum of the $\lceil \frac{n}{2} \rceil$ smallest values $\Delta_V(i)$ for $1 \leq i \leq n$; we define $\Delta_W^{\text{mid}}$ analogously.

*Claim* If $V_{\min} + \Delta_V^{\text{mid}} \leq V$, then $A_V \geq 2^{n/2}$; if $W_{\min} + \Delta_W^{\text{mid}} \leq W$, then $A_W \geq 2^{n/2}$; otherwise, we are dealing with a NO-instance.

*Proof* First, assume that $V_{\min} + \Delta_V^{\text{mid}} \leq V$. This means that there is a choice $S$ with $v(S) \leq V$ containing at least $\frac{n}{2}$ items $c$ such that $\text{rank}_v(c) \geq 2$. Hence, Fact 5.2 yields $\text{rank}_v(S) \geq 2^{\lceil n/2 \rceil}$ and consequently $A_V \geq 2^{n/2}$, as claimed. Symmetrically, if $W_{\min} + \Delta_W^{\text{mid}} \leq W$, then $A_W \geq 2^{n/2}$.

Now, suppose that there is a feasible solution $S$. As no class contains a single item minimising both $v(c)$ and $w(c)$, there are at least $\lceil \frac{n}{2} \rceil$ classes for which $S$ contains an item not minimising $v(c)$, or at least $\lceil \frac{n}{2} \rceil$ classes for which $S$ contains an item not minimising $w(c)$. Without loss of generality, we assume that the former holds. Let $D$ be the set of at least $\lceil \frac{n}{2} \rceil$ classes $i$ satisfying the condition. If $c \in C_i$ does not minimise $v(c)$, then $v(c) \geq v_{\min}(i) + \Delta_V(i)$. Consequently, $V \geq v(S) = V_{\min} +$

$\sum_{i \in D} \Delta_V(i)$. However, observe that $\sum_{i \in D} \Delta_V(i) \geq \Delta_V^{\text{mid}}$, so $V \geq V_{\min} + \Delta_V^{\text{mid}}$, as claimed. □

The conditions from the claim can be verified in $O(N)$ time using a linear-time selection algorithm to compute $\Delta_V^{\text{mid}}$ and $\Delta_W^{\text{mid}}$. If any of the first two conditions holds, we return the instance obtained using our reduction. Otherwise, we output a dummy NO-instance. □

In the improved reduction we use two basic steps. The first one is expressed in the following lemma.

**Lemma 5.10** *Consider a class of items in an instance of the* MULTICHOICE KNAPSACK *problem. In linear time, we can remove some irrelevant items from the class so that the resulting class $C$ satisfies* $\max(\text{rank}_v(c), \text{rank}_w(c)) > \frac{1}{3}|C|$ *for each item* $c \in C$.

*Proof* First, note that using a linear-time selection algorithm, we can determine for each item $c$ whether $\text{rank}_v(c) \leq \frac{1}{3}|C|$ and whether $\text{rank}_w(c) \leq \frac{1}{3}|C|$. If there is no item satisfying both conditions, we keep $C$ unaltered. Otherwise, we have an item which dominates at least $|C| - \text{rank}_v(c) - \text{rank}_w(c) \geq \frac{1}{3}|C|$ other items. We scan through all items in $C$ and remove those dominated by $c$. Next, we repeat the algorithm. The running time of a single phase is clearly linear, and since $|C|$ decreases geometrically, the total running time is also linear. □

The second reduction step decreases the number of classes by replacing two distinct classes $C_i$, $C_j$ with their Cartesian product $C_i \times C_j$, assuming that the value (weight) of a pair $(c_i, c_j)$ is the sum of values (weights) of $c_i$ and $c_j$. This clearly leads to an equivalent instance of the MULTICHOICE KNAPSACK problem, does not alter the parameters $A_V$, $A_W$, and decreases $n$. On the other hand, $N$ and $\lambda$ may increase; the latter happens only if $|C_i| \cdot |C_j| > \lambda$.

These two reduction rules let us implement our preprocessing procedure.

**Lemma 5.11** *Given an instance $I$ of the* MULTICHOICE KNAPSACK *problem, one can compute in $O(N + \lambda \log A)$ time an equivalent instance $I'$ with $A'_V \leq A_V$, $A'_W \leq A_W$, $\lambda' \leq \lambda$, and $n' = O\left(\frac{\log A}{\log \lambda}\right)$.*

*Proof* First, we apply Lemma 5.9 to make sure that $n \leq 2 \log A$ and $N = O(\lambda \log A)$. We may now assume that $\lambda \geq 3^6$, as otherwise we already have $n = O\left(\frac{\log A}{\log \lambda}\right)$.

Throughout the algorithm, whenever there are two distinct classes of size at most $\sqrt{\lambda}$, we shall replace them with their Cartesian product. This may happen only $n - 1$ times, and a single execution takes $O(\lambda)$ time, so the total running time needed for this part is $O(\lambda \log A)$.

Furthermore, for every class that we get in the input instance or obtain as a Cartesian product, we apply Lemma 5.10. The total running time spent on this is also $O(\lambda \log A)$.

Having exhaustively applied these reduction rules, we are guaranteed that we have $\max(\text{rank}_v(c), \text{rank}_w(c)) > \frac{1}{3}\sqrt{\lambda} \geq \lambda^{\frac{1}{3}}$ for items $c$ from all but one class. Without loss of generality, we assume that the classes satisfying this condition are $C_1, \ldots, C_k$.

Recall that $v_{\min}(i)$ and $w_{\min}(i)$ are defined as minimum values and weights of items in class $C_i$ and that $V_{\min}$ and $W_{\min}$ are their sums over all classes. For $1 \leq i \leq k$, we define $\Delta_V(i)$ as the difference between the $\left\lceil \lambda^{\frac{1}{3}} \right\rceil$-th smallest value in the multiset $\{v(c) : c \in C_i\}$ and $v_{\min}(i)$. Next, we define $\Delta_V^{\text{mid}}$ as the sum of the $\lceil \frac{k}{2} \rceil$ smallest values $\Delta_V(i)$. Symmetrically, we define $\Delta_W(i)$ and $\Delta_W^{\text{mid}}$. We shall prove a claim analogous to that in the proof of Lemma 5.9.

*Claim* If $V_{\min} + \Delta_V^{\text{mid}} \leq V$, then $A_V \geq \lambda^{\frac{1}{6}k}$; if $W_{\min} + \Delta_W^{\text{mid}} \leq W$, then $A_W \geq \lambda^{\frac{1}{6}k}$; otherwise, we are dealing with a NO-instance.

*Proof* First, suppose that $V_{\min} + \Delta_V^{\text{mid}} \leq V$. This means that there is a choice $S$ with $v(S) \leq V$ which contains at least $\frac{k}{2}$ items $c$ with $\text{rank}_v(c) \geq \lambda^{\frac{1}{3}}$. By Fact 5.2, the rank of this choice is at least $\lambda^{\frac{1}{6}k}$, so $A_V \geq \lambda^{\frac{1}{6}k}$, as claimed. The proof of the second case is analogous.

Now, suppose that there is a feasible solution $S = \{c_1, \ldots, c_n\}$. For $1 \leq i \leq k$, we have $\text{rank}_v(c_i) \geq \lambda^{\frac{1}{3}}$ or $\text{rank}_w(c_i) \geq \lambda^{\frac{1}{3}}$. Consequently, $\text{rank}_v(c_i) \geq \lambda^{\frac{1}{3}}$ holds for at least $\lceil \frac{k}{2} \rceil$ classes or $\text{rank}_w(c_i) \geq \lambda^{\frac{1}{3}}$ holds for at least $\lceil \frac{k}{2} \rceil$ classes. Without loss of generality, we assume that the former holds. Let $D$ be the set of (at least $\lceil \frac{k}{2} \rceil$) classes $i$ satisfying the condition. For each $i \in D$, we clearly have $v(c_i) \geq v_{\min}(i) + \Delta_V(i)$, while for each $i \notin D$, we have $v(c_i) \geq v_{\min}(i)$. Consequently, $V \geq v(S) \geq V_{\min} + \sum_{i \in D} \Delta_V(i) \geq V_{\min} + \Delta_V^{\text{mid}}$. Hence, $V \geq V_{\min} + \Delta_V^{\text{mid}}$, which concludes the proof. □

The condition from the claim can be verified using a linear-time selection algorithm: first, we apply it for each class to compute $\Delta_V(i)$ and $\Delta_W(i)$, and then, globally, to determine $\Delta_V^{\text{mid}}$ and $\Delta_W^{\text{mid}}$. If one of the first two conditions holds, we return the instance obtained through the reduction. It satisfies $A \geq \lambda^{\frac{1}{6}k}$, i.e., $n \leq 1 + k \leq 1 + 6\frac{\log A}{\log \lambda}$. Otherwise, we construct a dummy NO-instance. □

### 5.8 Main Result

We apply the preprocessing of the previous section to arrive at our final algorithm.

**Theorem 1.3** MULTICHOICE KNAPSACK *can be solved in* $O(N + \sqrt{a\lambda} \log A)$ *time.*

*Proof* Before running the algorithm of Proposition 5.7, we apply the reductions of Lemmas 5.8 and 5.11. With this order of reductions, we already have $\lambda \leq a$ during the execution of Lemma 5.11, so the $O(\lambda \log A)$ term is dominated by $O(\sqrt{a\lambda} \log A)$. □

## 6 Weighted Consensus and General Weighted Pattern Matching

The Weighted Consensus problem is formally defined as follows.

Weighted Consensus Problem

**Input:** Two weighted sequences $X$ and $Y$ of length $n$ with at most $\lambda$ letters at each position and $R$ in total, and a threshold probability $\frac{1}{z}$.

**Output:** A string $S$ such that $S \approx_{\frac{1}{z}} X$ and $S \approx_{\frac{1}{z}} Y$ or NONE if no such string exists.

Due to Facts 4.1 and 5.1, the Weighted Consensus problem is essentially equivalent to Multichoice Knapsack. The only difference is that we study Multichoice Knapsack with respect to unknown parameters $a$ and $A$, whereas in Weighted Consensus we know the parameter $z$. By Observation 4.2, these values for equivalent instances satisfy $a \leq A \leq z$, so Theorem 1.3 immediately yields:

**Proposition 6.1** Weighted Consensus *can be solved in* $O(R + \sqrt{z\lambda}\log z)$ *time.*

In Sections 6.2 and 6.3 we show that the $O(\log z)$ term can be reduced to $O(\log\lambda + \log\log z)$. Such an improvement is possible because the bound $a \leq A \leq z$ is not tight in general.

If two weighted sequences admit a consensus, we write $X \approx_{\frac{1}{z}} Y$ and say that $X$ *matches* $Y$ *with probability at least* $\frac{1}{z}$. With this definition of a match, we extend the notion of an occurrence and the notation $Occ_{\frac{1}{z}}(P, T)$ to arbitrary weighted sequences.

General Weighted Pattern Matching (GWPM) Problem

**Input:** Two weighted sequences $P$ and $T$ of length $m$ and $n$, respectively, with at most $\lambda$ letters at each position and $R$ in total, and a threshold probability $\frac{1}{z}$.

**Output:** The set $Occ_{\frac{1}{z}}(P, T)$.

In the case of the GWPM problem, it is more useful to provide an *oracle* that finds witness strings that correspond to the respective occurrences of the pattern. Such an oracle, given $i \in Occ_{\frac{1}{z}}(P, T)$, computes a string that matches both $P$ and $T[i\,..\,i+m-1]$.

### 6.1 Reduction to Weighted Consensus on Short Sequences

The GWPM problem clearly can be reduced to $n + m - 1$ instances of Weighted Consensus. This leads to a naïve $O(nR + n\sqrt{z\lambda}\log z)$-time algorithm. In this subsection, we remove the first term in this complexity.

Our solution applies the tools developed in Section 4 for Weighted Pattern Matching and uses an observation that is a consequence of Observation 4.3.

**Observation 6.2** If $X$ and $Y$ are weighted sequences that match with threshold $\frac{1}{z}$, then $d_H(\mathbf{H}(X), \mathbf{H}(Y)) \leq 2\lfloor \log z \rfloor$. Moreover, there exists a consensus string $S$ such that $S[i] = \mathbf{H}(X)[i] = \mathbf{H}(Y)[i]$ unless $\mathbf{H}(X)[i] \neq \mathbf{H}(Y)[i]$.

*Proof* The fact that $X \approx_{\frac{1}{z}} Y$ means that there exists a string $P$ such that $P \approx_{\frac{1}{z}} X$ and $P \approx_{\frac{1}{z}} Y$. Let the set $A_1$ represent the positions of mismatches between $\mathbf{H}(X)$ and $P$ and the set $A_2$ represent the positions of mismatches between $\mathbf{H}(Y)$ and $P$. By Observation 4.3, $|A_1|, |A_2| \leq \lfloor \log z \rfloor$. Let $A$ be the set of mismatches between $\mathbf{H}(X)$ and $\mathbf{H}(Y)$. We have $A \subseteq A_1 \cup A_2$ and thus $|A| \leq 2\lfloor \log z \rfloor$. Finally, observe that for each $i \in A \setminus (A_1 \cup A_2)$ we may replace $P[i]$ with $\mathbf{H}(X)[i] = \mathbf{H}(Y)[i]$ to obtain a string $S$ such that $S \approx_{\frac{1}{z}} X$ and $S \approx_{\frac{1}{z}} Y$ and $S[i] = \mathbf{H}(X)[i] = \mathbf{H}(Y)[i]$ unless $i \in A$. □

The algorithm starts by computing $P' = \mathbf{H}(P)$ and $T' = \mathbf{H}(T)$ and the data structure for *lcp*-queries in $P'T'$. We try to match $P$ against every factor $T[p\,..\,p+m-1]$ of the text. Following Observation 6.2, we check if $d_H(T'[p\,..\,p+m-1], P') \leq 2\lfloor \log z \rfloor$. If not, then we know that no match is possible. Otherwise, let $D$ be the set of positions of mismatches between $T'[p\,..\,p+m-1]$ and $P'$. Assume that we store $\alpha = \prod_{j=1}^{m} \pi^{(T)}_{p+j-1}(T'[p+j-1])$ and $\beta = \prod_{j=1}^{m} \pi^{(P)}_{j}(P'[j])$. Now, we only need to check what happens at the positions in $D$. If $D = \emptyset$, it suffices to check if $\alpha \geq \frac{1}{z}$ and $\beta \geq \frac{1}{z}$.

Otherwise, we construct two weighted sequences $X$ and $Y$ by selecting only the positions from $D$ in $T[p\,..\,p+m-1]$ and in $P$. In $O(|D|)$ time we can compute $\alpha' = \prod_{j \notin D} \pi^{(T)}_{p+j-1}(T'[p+j-1])$ and $\beta' = \prod_{j \notin D} \pi^{(P)}_{j}(P'[j])$. We multiply the probabilities of all letters at the first position in $X$ by $\alpha'$ and in $Y$ by $\beta'$. It is clear that $X \approx_{\frac{1}{z}} Y$ if and only if $T[p\,..\,p+m-1] \approx_{\frac{1}{z}} P$.

Thus, we reduced the GWPM problem to at most $n-m+1$ instances of the problem of Weighted Consensus for sequences of length $O(\log z)$. If we memorise the solutions to all those instances together with the underlying sets of mismatches $D$, we can also implement the oracle for the GWPM problem with $O(m)$-time queries. We obtain the following reduction.

**Lemma 6.3** *The* GWPM *problem and the computation of its oracle can be reduced in* $O(R + (n-m+1)\log z)$ *time to at most* $n-m+1$ *instances of the* Weighted Consensus *problem for weighted sequences of length* $O(\log z)$.

By Proposition 6.1, each of the resulting instances of Weighted Consensus can be solved in $O(\lambda \log z + \sqrt{z\lambda}\log z) = O(\sqrt{z\lambda}\log z)$ time (due to $z \geq \lambda$).

**Proposition 6.4** GWPM *problem can be solved in* $O(n\sqrt{z\lambda}\log z)$ *time. An oracle for the* GWPM *problem using* $O(n \log z)$ *space and supporting queries in* $O(m)$ *time can be computed within the same time complexity.*

In the remainder of this section, we design a tailor-made solution which lets us improve the $O(\log z)$ factors in Propositions 6.1 and 6.4 to $O(\log\log z + \log\lambda)$.

### 6.2 Reduction to Short Dissimilar Weighted Consensus

Let us notice that in the previous section we actually reduced GWPM to instances of Weighted Consensus that satisfy an additional *dissimilarity* requirement, as stated in the following problem.

Short Dissimilar Weighted Consensus (SDWC) Problem

**Input:** A threshold probability $\frac{1}{z}$ and two weighted sequences $X$ and $Y$ of length $n \leq 2\lfloor \log z \rfloor$ with at most $\lambda \leq z$ letters at each position and such that $\mathbf{H}(X)$ and $\mathbf{H}(Y)$ are *dissimilar*, i.e., $\mathbf{H}(X)[i] \neq \mathbf{H}(Y)[i]$ for each position $i$. The letters at each position in $X$ and $Y$ are sorted by non-increasing probability.

**Output:** A string $S$ such that $S \approx_{\frac{1}{z}} X$ and $S \approx_{\frac{1}{z}} Y$ or NONE if no such string exists.

In the SDWC problem, we further require an ordering of letters according to their probabilities. This assumption is trivial if $\sigma = O(1)$; otherwise, we use the preprocessing of Section 5.7 to expedite sorting. The following result refines Lemma 6.3.

**Lemma 6.5** *The* GWPM *problem and the computation of its oracle can be reduced in* $O(R + (n - m + 1)\lambda \log z)$ *time to at most* $n - m + 1$ *instances of* SDWC.

*Proof* The reduction of Section 6.1 in $O(R + (n - m + 1)\log z)$ time results in $n - m + 1$ dissimilar instances of length at most $2 \log z$. However, the characters are not ordered by non-increasing probabilities. Before we sort them, we apply Lemma 5.11 in order to reduce the length to $O(\frac{\log z}{\log \lambda})$; this takes $O(\lambda \log z)$ time. Note that both removing irrelevant characters and merging two positions into their Cartesian product preserves the property that the probabilities at each position sum up to at most one, so the resulting instance of Multichoice Knapsack can be interpreted back as an instance of Weighted Consensus. Finally, we sort the probabilities in $O(\lambda \log \lambda)$ time per position, i.e., in $O(\lambda \log z)$ time per instance of SDWC. □

### 6.3 Solving Short Dissimilar Weighted Consensus

#### 6.3.1 Overview

We follow the same general meet-in-the-middle scheme as the algorithm for Multichoice Knapsack presented in Proposition 5.7. The latter relies on Lemma 5.3, whose analogue in terms of weighted sequences and probabilities is much simpler.

**Observation 6.6** Consider weighted sequences $X$ and $Y$ of length $n$ and $z, z_\ell, z_r \in \mathbb{R}_+$ such that $z_\ell \cdot z_r \geq z$. Any $S \in \mathbf{M}_z(X) \cap \mathbf{M}_z(Y)$ admits a decomposition $S = L \cdot c \cdot R$, where:

– $\mathbf{P}(L, X[1..|L|]) \geq \frac{1}{z_\ell}$,

– $c$ is a single letter,
– $\mathbf{P}(R, X[n-|R|+1\,..\,n]) \geq \frac{1}{z_r}$.

Motivated by this formulation, we employ a notion of $\frac{1}{z}$-*solid prefixes* of a weighted sequence $X$—strings $S$ such that $S \approx_{\frac{1}{z}} X[1\,..\,|S|]$—and a symmetric notion of $\frac{1}{z}$-*solid suffixes*. By Observation 4.2, the number of $\frac{1}{z}$-solid prefixes of weighted sequence $X$ of length $n$ is at most $nz$. A direct application of the approach of Proposition 5.7, using solid prefixes and suffixes as partial choices, would result in generating up to $nz_\ell$ solid prefixes and $nz_r$ solid suffixes of $X$. Recall that, in case of SDWC, $n = O(\log z)$.

However, $\frac{1}{z}$-solid prefixes have more structure than prefix partial choices of rank at most $z$. We exploit this structure by introducing a notion of *light* $\frac{1}{z}$-*solid prefixes*, that is, $\frac{1}{z}$-solid prefixes that end with a non-heavy letter in $X$, that are the key ingredient in our solution. We show that the number of light $\frac{1}{z}$-solid prefixes of $X$ is at most $z$. Our algorithm for SDWC applies this fact to limit the number of generated $\frac{1}{z_\ell}$-solid prefixes and $\frac{1}{z_r}$-solid suffixes to $z_\ell$ and $z_r$, respectively.

The following subsections correspond to subsequent subsections of Section 5:

– In Section 6.3.2 (corresponds to Section 5.3) we show the $O(z)$ bound on the number of light $\frac{1}{z}$-solid prefixes (or sufixes) and prove a decomposition property for them that is similar to Observation 6.6 (but more complex).
– Section 6.3.3 (corresponds to Section 5.4) contains an algorithm for generating light $\frac{1}{z'}$-solid prefixes of $X$ that are simultaneously $\frac{1}{z}$-solid prefixes of $Y$. Intuitively, light solid prefixes of a given length $k \leq n$ can be obtained from light solid prefixes of any length smaller than $k$ by extending them with any character. This gives $O(n\lambda)$ lists of solid prefixes to be merged by probabilities which multiplies the complexity by $O(\log(n\lambda)) = O(\log\log z + \log\lambda)$.
– Section 6.3.4 (corresponds to Section 5.5) shows how to compute a solution based on sorted lists of common solid prefixes and suffixes of lengths summing up to $n$.
– Section 6.3.5 (corresponds to Section 5.6) implements the meet-in-the-middle approach. Because of the more complicated decomposition property this part of the algorithm is the most complex. It consists of $O(\log n) = O(\log\log z)$ phases.

#### 6.3.2 Combinatorics of Light Solid Prefixes (Counterpart of Section 5.3)

We define a *light* $\frac{1}{z}$-*solid prefix* of a weighted sequence $X$ as a $\frac{1}{z}$-solid prefix $S$ of length $k$ such that $k = 0$ or $S[k] \neq \mathbf{H}(X)[k]$.

We say that a string $P$ is a *maximal* $\frac{1}{z}$-*solid prefix* of a weighted sequence $X$ if $P$ is a $\frac{1}{z}$-solid prefix of $X$ and no string $P' = Ps$, for $s \in \Sigma$, is a $\frac{1}{z}$-solid prefix of $X$. Maximal solid prefixes have following simple property, originally due to Amir et al. [1].

**Fact 6.7** ([1]) A weighted sequence has at most $z$ *maximal* $\frac{1}{z}$-solid prefixes, that is, $\frac{1}{z}$-solid prefixes which cannot be extended to any longer $\frac{1}{z}$-solid prefix.

Fact 6.7 lets us bound the number of light solid prefixes.

**Fact 6.8** A weighted sequence has at most $z$ different light $\frac{1}{z}$-solid prefixes.

*Proof* We show a pair of inverse mappings between the set of maximal $\frac{1}{z}$-solid prefixes of a weighted sequence $X$ and the set of light $\frac{1}{z}$-solid prefixes of $X$. If $P$ is a maximal $\frac{1}{z}$-solid prefix of $X$, then we obtain a light $\frac{1}{z}$-solid prefix by removing all trailing letters of $P$ that are heavy letters at the corresponding positions in $X$. For the inverse mapping, we extend each light $\frac{1}{z}$-solid prefix by heavy letters as long as the prefix is $\frac{1}{z}$-solid. □

With this notion and its symmetric counterpart, *light $\frac{1}{z}$-solid suffixes*, we can state a stronger version of Observation 6.6. Note that this is where the dissimilarity is crucial.

**Lemma 6.9** *Consider an instance $(X, Y, \frac{1}{z})$ of the* SDWC *problem, and let $z_\ell, z_r \geq 1$ be real numbers such that $z_\ell \cdot z_r \geq z$. If $X \approx_{\frac{1}{z}} Y$, then every consensus string $S$ can be decomposed into $S = L \cdot c \cdot C \cdot R$ such that the following conditions hold for some $U, V \in \{X, Y\}$:*

- *$L$ is a light $\frac{1}{z_\ell}$-solid prefix of $U$,*
- *$c$ is a single letter,*
- *all letters of $C$ are heavy in $V$,*
- *$R$ is a light $\frac{1}{z_r}$-solid suffix of $V$.*

*Proof* We set $L$ as the longest proper prefix of $S$ which is a $\frac{1}{z_\ell}$-solid prefix of both $X$ and $Y$, and we define $k := |L|$. Note that $L$ is a light $\frac{1}{z_\ell}$-solid prefix of $X$ or $Y$, because $\mathbf{H}(X)$ and $\mathbf{H}(Y)$ are dissimilar. If $k = n - 1$, we conclude the proof setting $c = S[n]$ and $C = R$ to empty strings.

Otherwise, we have $\mathbf{P}(S[1\,..\,k+1], V[1\,..\,k+1]) < \frac{1}{z_\ell}$ for $V = X$ or $V = Y$. Since $\mathbf{P}(S, V) \geq \frac{1}{z}$ and $z_\ell \cdot z_r \geq z$, this implies $\mathbf{P}(S[k+2\,..\,n], V[k+2\,..\,n]) \geq \frac{1}{z_r}$, i.e., that $S[k+2\,..\,n]$ is a $\frac{1}{z_r}$-solid suffix of $V$. We set $c = S[k+1]$, $C$ as the longest prefix of $S[k+2\,..\,n]$ composed of letters heavy in $V$, and $R$ as the remaining suffix of $S[k+2\,..\,n]$. Then $R$ is clearly a light $\frac{1}{z_r}$-solid suffix of $V$. □

### 6.3.3 Generating Solid Prefixes (Counterpart of Section 5.4)

We say that a string $P$ is a common $\frac{1}{z}$-solid prefix (suffix) of weighted sequences $X$ and $Y$ if it is a $\frac{1}{z}$-solid prefix (suffix) of both $X$ and $Y$. Let $(X, Y, \frac{1}{z})$ be an instance of the SDWC problem. A *standard representation* of a common $\frac{1}{z}$-solid prefix $P$ of length $k$ of $X$ and $Y$ is a triple $(P, p_1, p_2)$ such that $p_1$ and $p_2$ are the probabilities $p_1 = \mathbf{P}(P, X[1\,..\,k])$ and $p_2 = \mathbf{P}(P, Y[1\,..\,k])$.

If $\sigma$ is constant, the string $P$ can be directly represented using $O(\log z)$ bits due to $|P| = O(\log z)$. Otherwise, $P$ is written using variable-length encoding so that

a letter that occurs at a given position with probability $p$ in $X$ has a representation that consists of $O(\log \frac{1}{p})$ bits. For every position $i$, the encoding can be constructed by assigning subsequent integer identifiers to letters according non-increasing order of $\pi_i^{(X)}(c)$. Note that an instance of SDWC problem provides us with the desired sorted order of letters. This lets us store a $\frac{1}{z}$-solid prefix using $O(\log z)$ bits: we concatenate the variable-length representations of its letters and we store a bit mask of size $O(\log z)$ that stores the delimiters between the representations of single letters.

In either case, our assumptions on the model of computations imply that the standard representation takes constant space. Moreover, constant time is sufficient to extend a common $\frac{1}{z}$-solid prefix by a given letter. An analogous representation can be used also to store common $\frac{1}{z}$-solid suffixes.

The following observation describes longer light solid prefixes in terms of shorter ones.

**Observation 6.10** Let $P$ be a non-empty light $\frac{1}{z}$-solid prefix of $X$. If one removes its last letter and then removes all the trailing letters which are heavy at the respective positions in $X$, then a shorter light $\frac{1}{z}$-solid prefix of $X$ is obtained.

We build upon Observation 6.10 to derive an efficient algorithm for generating light solid prefixes.

**Lemma 6.11** *Let $(X, Y, \frac{1}{z})$ be an instance of the* SDWC *problem and let $z' \leq z$. The standard representations of all common $\frac{1}{z}$-solid prefixes of $X$ and $Y$ being light $\frac{1}{z'}$-solid prefixes of $X$, sorted first by their length and then by the probabilities in $X$, can be generated in $O(z'(\log\log z + \log\lambda) + \log^2 z)$ time.*

*Proof* For $k \in \{0, \ldots, n\}$, let $\mathbf{B}_k$ be a list of the requested solid prefixes of length $k$ sorted by their probabilities $p_1$ in $X$. Fact 6.8 guarantees that $\sum_{k=0}^{n} |\mathbf{B}_k| \leq z'$.

We compute the lists $\mathbf{B}_k$ for subsequent lengths $k$. We start with $\mathbf{B}_0$ containing the empty string with its probabilities $p_1 = p_2 = 1$. To compute $\mathbf{B}_k$ for $k > 0$, we use Observation 6.10. For a given $i \in \{0, \ldots, k-1\}$, we iterate over all elements $(P, p_1, p_2)$ of $\mathbf{B}_i$ ordered by the non-increasing probabilities $p_1$ and try to extend each of them by the heavy letters in $X$ at positions $i+1, \ldots, k-1$ and by the letter $s$ at position $k$. We process the letters $s$ ordered by $\pi_k^{(X)}(s)$, ignoring the first one ($\mathbf{H}(X)[k]$) and stopping as soon as we do not get a $\frac{1}{z'}$-solid prefix of $X$.

More precisely, with $X' = \mathbf{H}(X)$, we compute

$$p_1' := p_1 \cdot \prod_{j=i+1}^{k-1} \pi_j^{(X)}(X'[j]) \cdot \pi_k^{(X)}(s) \quad \text{and} \quad p_2' := p_2 \cdot \prod_{j=i+1}^{k-1} \pi_j^{(Y)}(X'[j]) \cdot \pi_k^{(Y)}(s),$$

check if $p_1' \geq \frac{1}{z'}$ and $p_2' \geq \frac{1}{z}$, and, if so, insert $(P \cdot X'[i+1\,..\,k-1] \cdot s, p_1', p_2')$ at the beginning of a new list $L_{i,s}$, indexed both by the letter $s$ and by the length $i$ of the shorter light $\frac{1}{z'}$-solid prefix. When we encounter an element $(P, p_1, p_2)$ of $\mathbf{B}_i$ and a letter $s$ for which $p_1' < \frac{1}{z'}$, we proceed to the next element of $\mathbf{B}_i$. If this happens for the heaviest letter $s \neq \mathbf{H}(X)[k]$, we stop considering the current list $\mathbf{B}_i$

and proceed to $\mathbf{B}_{i-1}$. The final step consists in merging all the $k\lambda$ lists $L_{i,s}$ in the order of probabilities in $X$; the result is $\mathbf{B}_k$.

Let us analyse the time complexity of the $k$-th step of the algorithm. If an element $(P, p_1, p_2)$ and letter $s$ that we consider satisfy $p'_1 \geq \frac{1}{z'}$, this accounts for a new light $\frac{1}{z'}$-solid prefix of $X$. Hence, in total (over all steps) we consider $O(z')$ such elements. Note that some of these elements may be discarded due to the condition on $p'_2$.

For each inspected element $(P, p_1, p_2)$, we also consider at most one letter $s$ for which $p'_1$ is not sufficiently large. If this is not the only letter considered for this element, such a candidate can be charged to the previously considered letter. The opposite situation may happen once for each list $\mathbf{B}_i$, which may give $O(k)$ additional operations in the $k$-th step, $O(\log^2 z)$ in total.

Thanks to the order in which the lists are considered, we can store products of probabilities $\prod_{j=i+1}^{k-1} \pi_j^{(X)}(X'[j])$, $\prod_{j=i+1}^{k-1} \pi_j^{(Y)}(X'[j])$ and factors $X'[i+1\,..\,k-1]$ so that the representation of each subsequent light $\frac{1}{z'}$-solid prefix of length $k$ is computed in $O(1)$ time. Finally, the merging step in the $k$-th phase takes $O(|\mathbf{B}_k| \log(k\lambda)) = O(|\mathbf{B}_k|(\log\log z + \log\lambda))$ time if a binary heap of $O(k\lambda)$ elements is used.

The time complexity of the whole algorithm is

$$O\left(\log^2 z + \sum_{k=1}^{n} |\mathbf{B}_k|(\log\log z + \log\lambda)\right).$$

By the already mentioned Fact 6.8, this is $O(\log^2 z + z'(\log\log z + \log\lambda))$. □

#### 6.3.4 Merging Solid Prefixes with Suffixes (Counterpart of Section 5.5)

Next, we provide an analogue of Lemma 5.6.

**Lemma 6.12** *Let $L$ and $R$ be lists containing, for some $k \in \{0, \ldots, n\}$, standard representations of common $\frac{1}{z}$-solid prefixes of length $k$ and common $\frac{1}{z}$-solid suffixes of length $n-k$ of $X$ and $Y$, respectively. If the elements of the lists are sorted according to non-decreasing probabilities in $X$ and $Y$, respectively, one can check in $O(|L| + |R|)$ time whether the concatenation of any $\frac{1}{z}$-solid prefix from $L$ and $\frac{1}{z}$-solid suffix from $R$ yields a consensus string $S$ for $X$ and $Y$.*

*Proof* First, we filter out dominated elements of the lists, i.e., elements $(P, p_1, p_2)$ such that there exists another element $(P', p'_1, p'_2)$ with $p'_1 \geq p_1$ and $p'_2 \geq p_2$. This can be done in linear time. After this operation, the list $R$ is ordered according to non-increasing probabilities in $X$, so we reverse the list so that now both both lists are ordered with respect to the non-decreasing probabilities in $X$.

For every element $(P, p_1, p_2)$ of $L$, we compute the leftmost element $(P', p'_1, p'_2)$ of $R$ such that $p_1 p'_1 \geq \frac{1}{z}$. This element maximises $p'_2$ among all elements satisfying the latter condition. Hence, it suffices to check if $p_2 p'_2 \geq \frac{1}{z}$, and if so, report the result $S = P P'$. As the lists are ordered by $p_1$ and $p'_1$, respectively, all such elements can be computed in $O(|L| + |R|)$ total time. □

### 6.3.5 Merge-in-the-Middle Implementation (Counterpart of Section 5.6)

In this section, we solve the SDWC problem based on Lemma 6.9. We generate all candidates for $L \cdot c$ and $R$ using Lemma 6.11, and we apply a divide-and-conquer procedure to fill this with $C$. Our procedure works for fixed $U, V \in \{X, Y\}$; the algorithm repeats it for all four choices.

Let $\mathbf{L}_i$ denote a list of all common $\frac{1}{z}$-solid prefixes of $X$ and $Y$ obtained by extending a light $\frac{\sqrt{\lambda}}{\sqrt{z}}$-solid prefix of $U$ of length $i-1$ by a single letter $s$ at position $i$, and let $\mathbf{R}_i$ denote a list of all common $\frac{1}{z}$-solid suffixes of $X$ and $Y$ of length $n-i+1$ that are light $\frac{1}{\sqrt{z\lambda}}$-solid suffixes of $V$. We assume that the lists $\mathbf{L}_i$ and $\mathbf{R}_i$ are sorted according to the probabilities in $U$ and $V$, respectively. We assume that $\mathbf{L}_{n+1} = \emptyset$, whereas $\mathbf{R}_{n+1}$ contains only a representation of an empty string.

The following lemma shows how to compute the lists $\mathbf{L}_i$ and $\mathbf{R}_i$ and bounds their total size. In case of $\sigma = O(1)$ it is a direct consequence of Lemma 6.11. Otherwise, one needs to exercise caution when computing the lists $\mathbf{L}_i$.

**Lemma 6.13** *The total size of lists $\mathbf{L}_i$ and $\mathbf{R}_i$ for $i \in \{1, \ldots, n+1\}$ is $O(\sqrt{z\lambda})$; they can be computed in $O(\sqrt{z\lambda}(\log\log z + \log\lambda))$ time.*

*Proof* $O(\sqrt{z\lambda}(\log\log z + \log\lambda))$-time computation of the lists $\mathbf{R}_i$ is directly due to Lemma 6.11. As for the lists $\mathbf{L}_i$, we first compute in $O\left(\frac{\sqrt{z}}{\sqrt{\lambda}}(\log\log z + \log\lambda)\right)$ time the lists of all light $\frac{\sqrt{\lambda}}{\sqrt{z}}$-solid prefixes of $U$, sorted by the lengths of strings and then by the probabilities in $U$, again using Lemma 6.11. Then for each length $i-1$ and for each letter $s$ at the $i$-th position, we extend all these prefixes by a single letter. This way we obtain $\lambda$ lists for a given $i-1$ that can be merged according to the probabilities in $U$ to form the list $\mathbf{L}_i$. Generation of the auxiliary lists takes $O\left(\frac{\sqrt{z}}{\sqrt{\lambda}} \cdot \lambda\right) = O(\sqrt{z\lambda})$ time in total, and merging them using a binary heap takes $O(\sqrt{z\lambda}\log\lambda)$ time. This way we obtain an $O(\sqrt{z\lambda}(\log\log z + \log\lambda))$-time algorithm. □

Let $\mathbf{L}^*_{a,b}$ be a list of common $\frac{1}{z}$-solid prefixes of $X$ and $Y$ of length $b$ obtained by taking a common $\frac{1}{z}$-solid prefix from $\mathbf{L}_i$ for some $i \in \{a, \ldots, b\}$ and extending it by $b-i$ letters that are heavy at the respective positions in $V$. Similarly, $\mathbf{R}^*_{a,b}$ is a list of common $\frac{1}{z}$-solid suffixes of length $n-a+1$ obtained by taking a common $\frac{1}{z}$-solid suffix from $\mathbf{R}_i$ for some $i \in \{a, \ldots, b\}$ and prepending it by $i-a$ letters that are heavy in $V$. Again, we assume that each of the lists $\mathbf{L}^*_{a,b}$ and $\mathbf{R}^*_{a,b}$ is sorted according to the probabilities in $U$ and $V$, respectively.

A *basic interval* is an interval $[a, b]$ represented by its endpoints $1 \le a \le b \le n+1$ such that $2^j$ divides $a-1$ and $b = \min(n+1, a+2^j-1)$ for some integer $j$ called the *layer* of the interval. For every $j = 0, \ldots, \lceil\log(n+1)\rceil$, there are $\Theta\left(\frac{n}{2^j}\right)$ basic intervals in the $j$-th layer and they are pairwise disjoint.

*Example 6.14* For $n = 7$, the basic intervals are $[1,1], \ldots, [8,8], [1,2], [3,4], [5,6], [7,8], [1,4], [5,8], [1,8]$.

**Lemma 6.15** *The total size of the lists $\mathbf{L}^*_{a,b}$ and $\mathbf{R}^*_{a,b}$ for all basic intervals $[a,b]$ is $O(\sqrt{z\lambda}\log\log z)$ and they can all be constructed in $O(\sqrt{z\lambda}(\log\log z+\log\lambda))$ time.*

*Proof* We compute all the lists $\mathbf{L}^*_{a,b}$ and $\mathbf{R}^*_{a,b}$ for basic intervals $[a,b]$ of subsequent layers $j=0,\ldots,\lceil\log(n+1)\rceil$. For $j=0$, we have $\mathbf{L}^*_{a,a}=\mathbf{L}_a$ and $\mathbf{R}^*_{a,a}=\mathbf{R}_a$. All these lists can be computed in $O(\sqrt{z\lambda}(\log\log z+\log\lambda))$ time via Lemma 6.13.

Suppose that we wish to compute $\mathbf{L}^*_{a,b}$ for $a<b$ at layer $j$ (the computation of $\mathbf{R}^*_{a,b}$ is symmetric). Take $c=a+2^{j-1}-1$. Let us iterate through all the elements $(P,p_1,p_2)$ of the list $\mathbf{L}^*_{a,c}$, extend each string $P$ by $\mathbf{H}(V)[c+1\,..\,b]$, and multiply the probabilities $p_1$ and $p_2$ by

$$\prod_{i=c+1}^{b}\pi_i^{(X)}(\mathbf{H}(V)[i]) \quad\text{and}\quad \prod_{i=c+1}^{b}\pi_i^{(Y)}(\mathbf{H}(V)[i]),$$

respectively. If a common $\frac{1}{z}$-solid prefix is obtained, it is inserted at the end of an auxiliary list $L$. The resulting list $L$ is merged with $\mathbf{L}^*_{c+1,b}$ according to the probabilities in $U$; the result is $\mathbf{L}^*_{a,b}$.

Thus, we can compute $\mathbf{L}^*_{a,b}$ in time proportional to the sum of lengths of $\mathbf{L}^*_{a,c}$ and $\mathbf{L}^*_{c+1,b}$. (Note that the necessary products of probabilities can be computed in $O(n)=O(\log z)$ total time.) For every $j=1,\ldots,\lceil\log n\rceil$, the total length of the lists from the $j$-th layer does not exceed the total length of the lists from the $(j-1)$-th layer. By Lemma 6.13, the lists at the 0-th layer have size $O(\sqrt{z\lambda})$. The conclusion follows from the fact that $\log n=O(\log\log z)$. □

Finally, we are ready to apply a divide-and-conquer approach to solve the SDWC problem:

**Lemma 6.16** *The* SDWC *problem can be solved in $O(\sqrt{z\lambda}(\log\log z+\log\lambda))$ time.*

*Proof* The algorithm goes along Lemma 6.9, considering all choices of $U$ and $V$. For each of them, we proceed as follows.

First, we compute the lists $\mathbf{L}_i$, $\mathbf{R}_i$ for all $i=1,\ldots,n$ and $\mathbf{L}^*_{a,b}$, $\mathbf{R}^*_{a,b}$ for all basic intervals. By Lemmas 6.13 and 6.15, this takes $O(\sqrt{z\lambda}(\log\log z+\log\lambda))$ time.

Note that, in order to find out if there is a feasible solution, it suffices to attempt joining a common $\frac{1}{z}$-solid prefix from $\mathbf{L}_j$ with a common $\frac{1}{z}$-solid suffix from $\mathbf{R}_k$ for some indices $1\le j<k\le n+1$ by heavy letters of $V$ at positions $j+1,\ldots,k-1$. We use a recursive routine to find such a pair of indices $j,k\in[a,b]$ which has positive length and therefore can be decomposed into two basic subintervals $[a,c]$ and $[c+1,b]$. Then either $j\le c<k$, or both indices $j,k$ belong to the same interval $[a,c]$ or $[c+1,b]$. To check the first case, we apply the algorithm of Lemma 6.12 to $L=\mathbf{L}^*_{a,c}$ and $R=\mathbf{R}^*_{c+1,b}$. The remaining two cases are solved by recursive calls for the subintervals. The recursive routine is called first for the basic interval $[1,n+1]$.

The computations performed by the routine for the basic intervals at the $j$-th level take at most the time proportional to the total size of lists $\mathbf{L}^*_{a,b}$, $\mathbf{R}^*_{a,b}$ at the $(j-1)$-th level. Lemma 6.15 shows that the total size of the lists at all levels is $O(\sqrt{z\lambda}\log\log z)$. Consequently, the whole recursive procedure works

in $O(\sqrt{z\lambda}\log\log z)$ time. Together with the computation of the lists, this gives $O(\sqrt{z\lambda}(\log\log z + \log\lambda))$ time in total. □

Lemma 6.16 combined with Lemma 6.5 provides an efficient solution for GENERAL WEIGHTED PATTERN MATCHING. It also gives a solution to WEIGHTED CONSENSUS (which is a special case of GWPM with $n = m$). Note that $\lambda\log z = O(\sqrt{z\lambda}\log z)$ due to $z \geq \lambda$.

**Theorem 1.4** *The* GENERAL WEIGHTED PATTERN MATCHING *problem can be solved in* $O(n\sqrt{z\lambda}(\log\log z+\log\lambda))$ *time, and the* WEIGHTED CONSENSUS *problem can be solved in* $O(R + \sqrt{z\lambda}(\log\log z + \log\lambda))$ *time.*

## 7 Conditional Hardness of GWPM

The following reduction from MULTICHOICE KNAPSACK to WEIGHTED CONSENSUS immediately yields that any significant improvement in the dependence on $z$ and $\lambda$ in the running time of our algorithm would lead to breaking long-standing barriers for special cases of MULTICHOICE KNAPSACK.

**Lemma 7.1** *Given an instance $I$ of the* MULTICHOICE KNAPSACK *problem with $n$ classes $C_1, \ldots, C_n$ of maximum size $\lambda$, in linear time one can construct an equivalent instance of the* WEIGHTED CONSENSUS *problem with $z = O(\prod_{i=1}^{n} |C_i|)$ and sequences of length $O(n)$ over alphabet of size $\lambda$.*

*Proof* We construct a pair of weighted sequences $X, Y$ of length $n$ over alphabet $\Sigma = \{1, \ldots, \lambda\}$. Let $C_i = \{c_{i,1}, \ldots, c_{i,|C_i|}\}$. Intuitively, choosing letter $j$ at position $i$ will correspond to taking $c_{i,j}$ to the solution $S$.

Without loss of generality, we assume that weights and values are non-negative. Otherwise, we may subtract $v_{\min}(i)$ from $v(c_{i,j})$ and $w_{\min}(i)$ from $w(c_{i,j})$ for each item $c_{i,j}$, as well $V_{\min}$ from $V$ and $W_{\min}$ from $W$.

We set $M$ to the smallest power of two such that $M \geq \max(n,\ V,\ W)$. For $j \in \{1, \ldots, |C_i|\}$, we set:

$$p_i^{(X)}(j) = -\frac{\lceil M\log|C_i|\rceil + v(c_{i,j})}{M}, \quad p_i^{(Y)}(j) = -\frac{\lceil M\log|C_i|\rceil + w(c_{i,j})}{M}.$$

We then define $\log\pi_i^{(X)}(j) = p_i^{(X)}(j)$ and $\log\pi_i^{(Y)}(j) = p_i^{(Y)}(j)$ for $j \in \Sigma$. Moreover, we set

$$\log z_X = \frac{1}{M}\left(V + \sum_{i=1}^{n}\lceil M\log|C_i|\rceil\right), \quad \log z_Y = \frac{1}{M}\left(W + \sum_{i=1}^{n}\lceil M\log|C_i|\rceil\right).$$

The following claim holds.

*Claim* $\sum_{j=1}^{|C_i|}\pi_i^{(X)}(j) \leq 1$, $\sum_{j=1}^{|C_i|}\pi_i^{(Y)}(j) \leq 1$, and $\max(z_X, z_Y) \leq 4\prod_{i=1}^{n}|C_i|$.

*Proof* As for the first inequality, we have:

$$\sum_{j=1}^{|C_i|} \pi_i^{(X)}(j) \;=\; \sum_{j=1}^{|C_i|} 2^{-\lceil M \log |C_i| \rceil / M} 2^{-v(c_{i,j})/M} \;\le\; \sum_{j=1}^{|C_i|} 2^{-\log |C_i|} \;=\; \sum_{j=1}^{|C_i|} \frac{1}{|C_i|} \le 1.$$

The second inequality is analogous. Finally, by the choice of $M$, we have

$$\max(z_X, z_Y) \;\le\; 2^{\frac{1}{M}(\max(V,W)+n)} \prod_{i=1}^{n} |C_i| \;\le\; 4 \prod_{i=1}^{n} |C_i|.$$

□

This way, for a string $P$ of length $n$, we have

$$\log \mathbf{P}(P, X) = -\frac{1}{M} \left( \sum_{i=1}^{n} \lceil M \log |C_i| \rceil + \sum_{i=1}^{n} v(c_{i,P[i]}) \right) \ge -\log z_X$$
$$\iff \sum_{i=1}^{n} v(c_{i,P[i]}) \le V,$$

$$\log \mathbf{P}(P, Y) = -\frac{1}{M} \left( \sum_{i=1}^{n} \lceil M \log |C_i| \rceil + \sum_{i=1}^{n} w(c_{i,P[i]}) \right) \ge -\log z_Y$$
$$\iff \sum_{i=1}^{n} w(c_{i,P[i]}) \le W.$$

Thus, $P$ is a solution to the constructed instance of the WEIGHTED CONSENSUS problem with two threshold probabilities, $\frac{1}{z_X}$ and $\frac{1}{z_Y}$, if and only if $S = \{c_{i,j} : P[i] = j\}$ is a solution to the underlying instance of the MULTICHOICE KNAPSACK problem. To have a single threshold $z = \max(z_X, z_Y)$, we append an additional position $n+1$ with symbol 1 only, with $p_{n+1}^{(X)}(1) = 0$ and $p_{n+1}^{(Y)}(1) = \log z_Y - \log z_X$ provided that $z_X \ge z_Y$, and symmetrically otherwise.

If one wants to make sure that the probabilities at each position sum up to exactly one, two further letters can be introduced, one of which gathers the remaining probability in $X$ and has probability 0 in $Y$, and the other gathers the remaining probability in $Y$, and has probability 0 in $X$. □

For completeness, let us recall the folklore reductions that show that SUBSET SUM and 3-SUM are special cases of MULTICHOICE KNAPSACK. To express an instance of SUBSET SUM with integers $a_1, \ldots, a_n$ and threshold $R$ as an instance of MULTICHOICE KNAPSACK, we introduce $n$ classes of two items each, which correspond to taking and omitting the respective elements. The first item has value $a_i$ and weight $-a_i$, while for the other these are both 0. The thresholds are $V = R$ and $W = -R$.

Similarly, given an instance of 3-SUM with classes $a_{1,1}, \ldots, a_{1,\lambda}$, $a_{2,1}, \ldots, a_{2,\lambda}$, and $a_{3,1}, \ldots, a_{3,\lambda}$, we can create an instance of MULTICHOICE KNAPSACK with the same three classes of items with values $a_{i,j}$ and weights $-a_{i,j}$. The thresholds are $V = W = 0$.

**Theorem 1.6** WEIGHTED CONSENSUS *is NP-hard and cannot be solved in:*

1. $O^*(z^{\varepsilon})$ *time for every* $\varepsilon > 0$*, unless the exponential time hypothesis (ETH) fails;*
2. $O^*(z^{0.5-\varepsilon})$ *time for some* $\varepsilon > 0$*, unless there is an* $O^*(2^{(0.5-\varepsilon)n})$*-time algorithm for the* SUBSET SUM *problem;*
3. $\tilde{O}(R + z^{0.5}\lambda^{0.5-\varepsilon})$ *time for some* $\varepsilon > 0$ *and for* $n = O(1)$*, unless the 3-*SUM *conjecture fails.*

*Proof* We use Lemma 7.1 to derive algorithms for the MULTICHOICE KNAPSACK problem based on hypothetical solutions for WEIGHTED CONSENSUS. SUBSET SUM is a special case of MULTICHOICE KNAPSACK with $\lambda = 2$, i.e., $\prod_i |C_i| = 2^n$. Hence, an $O^*(z^{o(1)})$-time solution for WEIGHTED CONSENSUS would yield an $O^*(2^{o(n)})$-time algorithm for SUBSET SUM, which contradicts ETH by the results of Etscheid et al. [9] and Gurari [11]. Similarly, an $O^*(z^{0.5-\varepsilon})$-time solution for WEIGHTED CONSENSUS would yield an $O^*(2^{(0.5-\varepsilon)n})$-time algorithm for SUBSET SUM. Moreover, 3-SUM is a special case of MULTICHOICE KNAPSACK with $n = 3$ and $\prod_i |C_i| = \lambda^3$. Hence, an $\tilde{O}(R + z^{0.5}\lambda^{0.5-\varepsilon})$-time solution for WEIGHTED CONSENSUS with $n = O(1)$ yields an $\tilde{O}(\lambda + \lambda^{1.5+0.5-\varepsilon}) = \tilde{O}(\lambda^{2-\varepsilon})$-time algorithm for 3-SUM. □

Nevertheless, it might still be possible to improve the dependence on $n$ in the GWPM problem. For example, one may hope to achieve $\tilde{O}(nz^{0.5-\varepsilon} + z^{0.5})$ time for $\lambda = O(1)$.

## 8 Multivariate Analysis of MULTICHOICE KNAPSACK and GWPM

In Section 5, we gave an $O(N + a^{0.5}\lambda^{0.5} \log A)$-time algorithm for the MULTICHOICE KNAPSACK problem. Improvement of either exponent to $0.5 - \varepsilon$ would result in a breakthrough for the SUBSET SUM and 3-SUM problems, respectively. Nevertheless, this does not refute the existence of faster algorithms for some particular values $(a, \lambda)$ other than those emerging from instances of SUBSET SUM or 3-SUM. Indeed, in this section we show an algorithm that is superior if $\frac{\log a}{\log \lambda}$ is a constant other than an odd integer. We also argue that it is optimal (up to lower order terms) for every constant $\frac{\log a}{\log \lambda}$ unless the $k$-SUM conjecture fails.

We analyse the running times of algorithms for the MULTICHOICE KNAPSACK problem expressed as $O(n^{O(1)} \cdot T(a, \lambda))$ for some function $T$ monotone with respect to both arguments. The algorithm of Theorem 1.3 proves that achieving $T(a, \lambda) = \sqrt{a\lambda}$ is possible. On the other hand, if we assume that SUBSET SUM does not admit an $O^*(2^{(0.5-\varepsilon)n})$-time solution, then we immediately get that we cannot have $T(a, 2) = O(a^{0.5-\varepsilon})$ for any $\varepsilon \geq 0$. Similarly, the 3-SUM conjecture implies that $T(\lambda^3, \lambda) = O(\lambda^{2-\varepsilon})$ is impossible. While this already refutes the possibility of having $T(a, \lambda) = O(a^{0.5}\lambda^{0.5-\varepsilon})$ across all arguments $(a, \lambda)$, such a bound may still hold for some special cases covering an infinite number of arguments. For example, we may potentially achieve $T(a, \lambda) = O((a\lambda)^{0.5-\varepsilon}) = O(\lambda^{1.5-\varepsilon})$ for $a = \lambda^2$.

Before we prove that this is indeed possible, let us see the consequences of the conjectured hardness of 3-SUM and, in general, $(2k-1)$-SUM. For a non-negative integer $k$, the $(2k-1)$-SUM conjecture refutes $T(\lambda^{2k-1}, \lambda) = O(\lambda^{k-\varepsilon})$. By monotonicity of $T$ with respect to the first argument, we conclude that $T(\lambda^c, \lambda) = O(\lambda^{k-\varepsilon})$ is impossible for $c \geq 2k-1$. On the other hand, monotonicity with respect to the second argument shows that $T(\lambda^c, \lambda) = O(\lambda^{c\frac{k}{2k-1}-\varepsilon})$ is impossible for $c \leq 2k-1$. The lower bounds following from $(2k-1)$-SUM and $(2k+1)$-SUM turn out to meet at $c = 2k-1+\frac{1}{k+1}$; see Figure 1.

Consequently, we have some room between the lower and the upper bound of $\sqrt{a\lambda}$. In the aforementioned case of $a = \lambda^2$, the upper bound is $\lambda^{\frac{3}{2}}$, compared to the lower bound of $\lambda^{\frac{4}{3}-\varepsilon}$. Below, we show that the upper bound can be improved to meet the lower bound. More precisely, we show an algorithm whose running time is $O(N + (a^{\frac{k+1}{2k+1}} + \lambda^k)\log\lambda \cdot n^k)$ for every positive integer $k$. Note that $a^{\frac{k+1}{2k+1}} + \lambda^k = \lambda^{c\frac{k+1}{2k+1}} + \lambda^k$, so for $2k-1 \leq c \leq 2k+1$ the running time indeed matches the lower bounds up to the $n^k$ term.

Due to Lemma 5.11, the extra $n^k$ term reduces to $O((\frac{\log A}{\log\lambda})^k)$. Finally, we study the complexity of the GWPM problem.

### 8.1 Algorithm for MULTICHOICE KNAPSACK

Let us start by discussing the bottleneck of the algorithm of Theorem 1.3 for large $\lambda$. The problem is that the size of the classes does not let us partition every choice $S$ into a prefix $L$ and a suffix $R$ with ranks both $O(\sqrt{A_V})$. Lemma 5.3 leaves us with

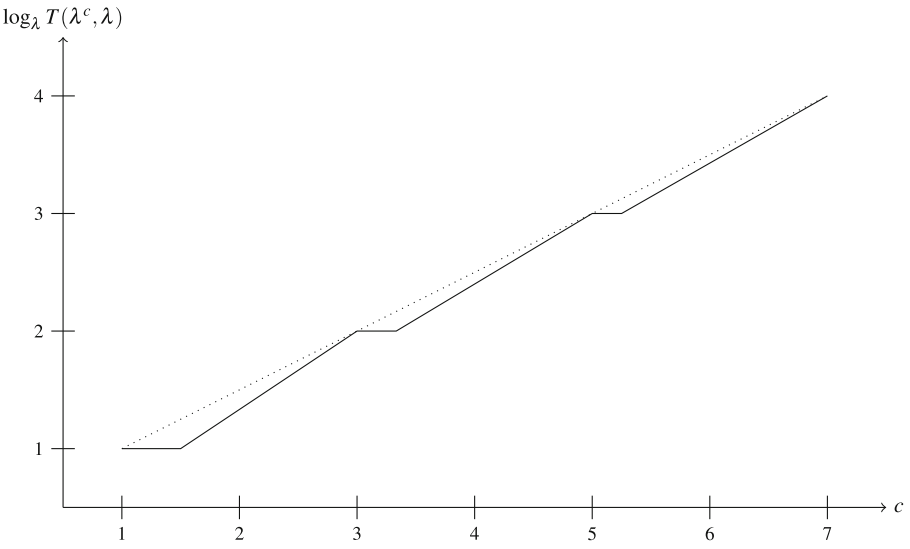


**Fig. 1** Illustration of the upper bound (dotted) and lower bound (solid) on $\log_\lambda T(\lambda^c, \lambda)$

an extra letter $c$ between $L$ and $R$, and in the algorithm we append it to the prefix (while generating $\mathbf{L}_{j-1}^{(\ell)} \odot C_j$).

We provide a workaround based on reordering of classes. Our goal is to make sure that items with large rank appear only in a few leftmost classes. For this, we guess the classes of the $k$ items with largest rank (in a feasible solution) and move them to the front. Since this depends on the sought feasible solution, we shall actually verify all $\binom{n}{k}$ possibilities.

Now, our solution considers two cases: For $j > k$, the reordering lets us assume $\mathrm{rank}_v(c) < \ell^{\frac{1}{k}}$, so we do not need to consider all items from $C_j$. For $j \leq k$, on the other hand, we exploit the fact that $|\mathbf{L}_{j-1}^{(\ell)} \odot C_j| \leq \lambda^j$, which at most $\lambda^k$.

The underlying combinatorial foundation is formalised as a variant of Lemma 5.3:

**Lemma 8.1** *Let $\ell$ and $r$ be positive integers such that $v(\boldsymbol{L}_j[\ell]) + v(\boldsymbol{R}_{j+1}[r]) > V$ for every $0 \leq j \leq n$. Let $k \in \{1, \ldots, n\}$ and suppose that $S$ is a choice with $v(S) \leq V$ such that $\mathrm{rank}_v(S \cap C_i) \geq \mathrm{rank}_v(S \cap C_j)$ for $i \leq k < j$. There is an index $j \in \{1, \ldots, n\}$ and a decomposition $S = L \cup \{c\} \cup R$ such that $L \in \boldsymbol{L}_{j-1}^{(\ell)}$, $R \in \boldsymbol{R}_{j+1}^{(r)}$, $c \in C_j$, and either $\mathrm{rank}_v(c) < \ell^{\frac{1}{k}}$ or $j \leq k$.*

*Proof* We claim that the decomposition constructed in the proof of Lemma 5.3 satisfies the extra condition on $\mathrm{rank}_v(c)$ if $j > k$. Let $S = \{c_1, \ldots, c_n\}$ and $S_i = \{c_1, \ldots, c_i\}$. Obviously $\mathrm{rank}_v(c_i) \geq 1$ for $k < i < j$ and, by the extra assumption, $\mathrm{rank}_v(c_i) \geq \mathrm{rank}_v(c)$ for $1 \leq i \leq k$. Hence, Fact 5.2 yields $\mathrm{rank}_v(S_{j-1}) \geq \mathrm{rank}_v(c)^k$. Simultaneously, we have $v(S_{j-1}) < v(\mathbf{L}_{j-1}[\ell])$, so $\mathrm{rank}_v(S_{j-1}) < \ell$. Combining these inequalities, we immediately get the claimed bound. □

**Theorem 1.7** *For every positive integer $k = O(1)$, the* Multichoice Knapsack *problem can be solved in $O(N + (a^{\frac{k+1}{2k+1}} + \lambda^k) \log A (\frac{\log A}{\log \lambda})^k)$ time.*

*Proof* As in the proof of Theorem 1.3, we actually provide an algorithm whose running time depends on $A_V$ rather than $a$. Moreover, Lemmas 5.8 and 5.11 let us assume that $n = O(\frac{\log A}{\log \lambda})$.

We first guess the $k$ positions where items with largest ranks $\mathrm{rank}_v$ are present in the solution $S$ and move these positions to the front. This gives $\binom{n}{k} = O((\frac{\log A}{\log \lambda})^k)$ possible selections. For each of them, we proceed as follows.

We increment an integer $r$ starting from 1, maintaining $\ell = \lceil r^{\frac{k}{k+1}} \rceil$ and all the lists $\mathbf{L}_j^{(\ell)}$ and $\mathbf{R}_{j+1}^{(r)}$ for $0 \leq j \leq n$, as long as $v(\mathbf{L}_j[\ell]) + v(\mathbf{R}_{j+1}[r]) \leq V$ for some $j$. By Fact 5.4, we stop with $r = O(A_V^{\frac{k+1}{2k+1}})$ and thus the total time of this phase is $O(A_V^{\frac{k+1}{2k+1}} \log A)$ due to the online procedure of Lemma 5.5.

By Lemma 8.1, every feasible solution $S$ for some $j$ admits a decomposition $S = L \cup \{c\} \cup R$, where $L \in \mathbf{L}_{j-1}^{(\ell)}$, $R \in \mathbf{R}_{j+1}^{(r)}$, $c \in C_j$, and either $\mathrm{rank}_v(c) < \ell^{\frac{1}{k}}$ or $j \leq k$; we consider all possibilities for $j$. For each of them, we shall reduce searching for $S$

to an instance of the MULTICHOICE KNAPSACK problem with $N' = O(A_V^{\frac{k+1}{2k+1}} + \lambda^k)$ and $n' = 2$. By Lemma 5.6, these instances can be solved in $O((A_V^{\frac{k+1}{2k+1}} + \lambda^k)\frac{\log A}{\log \lambda})$ time in total.

For $j \le k$, the items of the $j$-th instance are going to belong to classes $\mathbf{L}_{j-1}^{(\ell)} \odot C_j$ and $\mathbf{R}_{j+1}^{(r)}$. The set $\mathbf{L}_{j-1}^{(\ell)} \odot C_j$ can be sorted by merging $|C_j|$ sorted lists of size at most $\lambda^{j-1}$ each, i.e., in $O(\lambda^k \log \lambda)$ time. On the other hand, for $j > k$, we take $\{L \cup \{c\} : L \in \mathbf{L}_{j-1}^{(\ell)}, c \in C_j, \mathrm{rank}_v(c) \le \ell^{\frac{1}{k}}\}$ and $\mathbf{R}_{j+1}^{(r)}$. The former set can be constructed by merging at most $\min(\ell^{\frac{1}{k}}, \lambda) = \min(O(r^{\frac{1}{k+1}}), \lambda)$ sorted lists of size $\ell = O(r^{\frac{k}{k+1}})$ each, i.e., in $O(r \log \lambda) = O(A_V^{\frac{k+1}{2k+1}} \log \lambda)$ time.

Summing up over all indices $j$, this gives $O((A_V^{\frac{k+1}{2k+1}} + \lambda^k) \log A)$ time for a single selection of the $k$ positions with largest ranks, and $O((A_V^{\frac{k+1}{2k+1}} + \lambda^k) \log A (\frac{\log A}{\log \lambda})^k)$ in total.

Clearly, each solution of the constructed instances represents a solution of the initial instance, and by Lemma 8.1, every feasible solution of the initial instance has its counterpart in one of the constructed instances.

Before we conclude the proof, we need to note that the optimal $k$ does not need to be known in advance. To deal with this issue, we try consecutive integers $k$ and stop the procedure if Fact 5.4 yields that $A_V > \lambda^{2k+1}$, i.e., if $r$ is incremented beyond $\lambda^{k+1}$. If the same happens for the other instance of the algorithm (operating on $\mathrm{rank}_w$ instead of $\mathrm{rank}_v$), we conclude that $a > \lambda^{2k+1}$, and thus we shall better use larger $k$. The running time until this point is $O(\lambda^{k+1} \log \lambda (\frac{\log A}{\log \lambda})^k)$ due to Lemma 5.5. On the other hand, if $r \le \lambda^{k+1}$, the algorithm behaves as if $a \le \lambda^{2k+1}$, i.e., runs in $O(\lambda^{k+1} \log \lambda (\frac{\log A}{\log \lambda})^k)$ time. This workaround (considering all smaller values $k$) adds extra $O(\lambda^k \log \lambda (\frac{\log A}{\log \lambda})^{k-1})$ to the time complexity for the optimal value $k$, which is less than the upper bound on the running time we have for this value $k$. □

### 8.2 Algorithm for GENERAL WEIGHTED PATTERN MATCHING

If we are to bound the complexity in terms of $A$ only, the running time becomes

$$O(N + (A^{\frac{k+1}{2k+1}} + \lambda^k) \log A (\tfrac{\log A}{\log \lambda})^k).$$

Assumptions that $A \le \lambda^{2k+1}$ and $k = O(1)$ let us get rid of the $(\frac{\log A}{\log \lambda})^k$ term, which can be bounded by $(2k+1)^k = O(1)$.

**Corollary 8.2** *Let $k = O(1)$ be a positive integer such that $A \le \lambda^{2k+1}$. The* MULTICHOICE KNAPSACK *problem can be solved in $O(N + (A^{\frac{k+1}{2k+1}} + \lambda^k) \log \lambda)$ time.*

This leads to the following result for GENERAL WEIGHTED PATTERN MATCHING:

**Theorem 1.8** *If* $\lambda^{2k-1} \leq z \leq \lambda^{2k+1}$ *for some positive integer* $k = O(1)$*, then the* WEIGHTED CONSENSUS *problem can be solved in* $O(R + (z^{\frac{k+1}{2k+1}} + \lambda^k) \log \lambda)$ *time, and the* GWPM *problem can be solved in* $O(n(z^{\frac{k+1}{2k+1}} + \lambda^k) \log \lambda)$ *time.*

As we noted at the beginning of this section, Lemma 7.1 implies that any improvement of the dependence of the running time on $z$ or $\lambda$ by $z^{\varepsilon}$ (equivalently, by $\lambda^{\varepsilon}$) wound contradict the $k$-SUM conjecture.

**Acknowledgments** This work was supported by the "Algorithms for text processing with errors and uncertainties" project carried out within the HOMING programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.



## References

1. Amir, A., Chencinski, E., Iliopoulos, C.S., Kopelowitz, T., Zhang, H.: Property matching and weighted matching. Theor. Comput. Sci. **395**(2-3), 298–310 (2008). https://doi.org/10.1016/j.tcs.2008.01.006
2. Barton, C., Kociumaka, T., Pissis, S.P., Radoszewski, J.: Efficient index for weighted sequences. In: Grossi, R., Lewenstein, M. (eds.) Combinatorial Pattern Matching, CPM 2016, LIPIcs, vol. 54, pp. 4:1–4:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. Dagstuhl, Germany (2016). https://doi.org/10.4230/LIPIcs.CPM.2016.4
3. Barton, C., Liu, C., Pissis, S.P.: Linear-time computation of prefix table for weighted strings & applications. Theor. Comput. Sci. **656**, 160–172 (2016). https://doi.org/10.1016/j.tcs.2016.04.029
4. Barton, C., Liu, C., Pissis, S.P.: On-line pattern matching on uncertain sequences and applications. In: Chan, T.H., Li, M., Wang, L. (eds.) Combinatorial optimization and applications, COCOA 2016, LNCS, vol. 10043, pp. 547–562. Springer, Berlin (2016). https://doi.org/10.1007/978-3-319-48749-6_40
5. Barton, C., Liu, C., Pissis, S.P..: Fast average-case pattern matching on weighted sequences. To appear in the International Journal of Foundations of Computer Science (2017)
6. Biswas, S., Patil, M., Thankachan, S.V., Shah, R.: Probabilistic threshold indexing for uncertain strings. In: E. Pitoura, S. Maabout, G. Koutrika, A. Marian, L. Tanca, I. Manolescu, K. Stefanidis (eds.) 19th International Conference on Extending Database Technology, EDBT 2016, pp. 401–412. OpenProceedings.org. https://doi.org/10.5441/002/edbt.2016.37 (2016)
7. Christodoulakis, M., Iliopoulos, C.S., Mouchard, L., Tsichlas, K.: Pattern matching on weighted sequences. In: Algorithms and Computational Methods for Biochemical and Evolutionary Networks, CompBioNets 2004, KCL publications (2004)
8. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. Cambridge University Press, Cambridge (2007). https://doi.org/10.1017/cbo9780511546853
9. Etscheid, M., Kratsch, S., Mnich, M., Röglin, H.: Polynomial kernels for weighted problems. In: G.F. Italiano, G. Pighizzini, D. Sannella (eds.) Mathematical Foundations of Computer Science, MFCS 2015, Part II, LNCS, vol. 9235, pp. 287–298. Springer. https://doi.org/10.1007/978-3-662-48054-0_24 (2015)
10. Gajentaan, A., Overmars, M.H.: On a class of $O(n^2)$ problems in computational geometry. Comput. Geom. **5**, 165–185 (1995). https://doi.org/10.1016/0925-7721(95)00022-2
11. Gurari, E.M.: Introduction to the theory of computation. Computer Science Press (1989)

12. Horowitz, E., Sahni, S.: Computing partitions with applications to the knapsack problem. J. ACM, **21**(2), 277–292 (1974). https://doi.org/10.1145/321812.321823
13. Iliopoulos, C.S., Makris, C., Panagis, Y., Perdikuri, K., Theodoridis, E., Tsakalidis, A.K.: The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. Fundamenta Informaticae **71**(2-3), 259–277 (2006). http://content.iospress.com/articles/fundamenta-informaticae/fi71-2-3-07
14. Iliopoulos, C.S., Rahman, M.S.: Faster index for property matching. Inf. Process. Lett. **105**(6), 218–223 (2008). https://doi.org/10.1016/j.ipl.2007.09.004
15. Impagliazzo, R., Paturi, R.: On the complexity of $k$-SAT. J. Comput. Syst. Sci. **62**(2), 367–375 (2001). https://doi.org/10.1006/jcss.2000.1727
16. Juan, M.T., Liu, J.J., Wang, Y.L.: Errata for "Faster index for property matching". Inf. Process. Lett. **109**(18), 1027–1029 (2009). https://doi.org/10.1016/j.ipl.2009.06.009
17. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack problems. Springer. https://doi.org/10.1007/978-3-540-24777-7 (2004)
18. Kociumaka, T., Pissis, S.P., Radoszewski, J.: Pattern matching and consensus problems on weighted sequences and profiles. In: S. Hong (ed.) Algorithms and Computation, ISAAC 2016, LIPIcs, vol. 64, pp. 46:1–46:12. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ISAAC.2016.46 (2016)
19. Kopelowitz, T.: The property suffix tree with dynamic properties. Theor. Comput. Sci. **638**, 44–51 (2016). https://doi.org/10.1016/j.tcs.2016.02.033
20. Lokshtanov, D., Marx, D., Saurabh, S.: Lower bounds based on the Exponential Time Hypothesis. Bulletin of the EATCS **105**, 41–72 (2011). http://bulletin.eatcs.org/index.php/beatcs/article/view/92
21. Mehlhorn, K.: Nearly optimal binary search trees. Acta Inform. **5**, 287–295 (1975). https://doi.org/10.1007/BF00264563
22. Pizzi, C., Ukkonen, E.: Fast profile matching algorithms - A survey. Theor. Comput. Sci. **395**(2-3), 137–157 (2008). https://doi.org/10.1016/j.tcs.2008.01.015
23. Radoszewski, J., Starikovskaya, T.A.: Streaming $k$-mismatch with error correcting and applications. In: A. Bilgin, M.W. Marcellin, J. Serra-Sagristà, J.A. Storer (eds.) Data Compression Conference, DCC 2017, pp. 290–299. IEEE (2017). https://doi.org/10.1109/DCC.2017.14
24. Rajasekaran, S., Jin, X., Spouge, J.L.: The efficient computation of position-specific match scores with the fast Fourier transform. J. Comput. Biol. **9**(1), 23–33 (2002). https://doi.org/10.1089/10665270252833172