PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A

rsta.royalsocietypublishing.org

Research

**Cite this article:** Amir A, Butman A, Porat E. 2014 On the relationship between histogram indexing and block-mass indexing. *Phil. Trans. R. Soc. A* **372**: 20130132.
http://dx.doi.org/10.1098/rsta.2013.0132

One contribution of 11 to a Theo Murphy Meeting Issue 'Storage and indexing of massive data'.

**Subject Areas:**
theory of computing, algorithmic information theory

**Keywords:**
reduction, complexity, classes of indexing problems

**Author for correspondence:**
Amihood Amir
e-mail: amir@cs.biu.ac.il

# On the relationship between histogram indexing and block-mass indexing

Amihood Amir[1,2], Ayelet Butman[3] and Ely Porat[1]

[1]Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
[2]Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA
[3]Department of Software Engineering, Holon Institute of Technology, Holon, Israel

*Histogram indexing*, also known as *jumbled pattern indexing* and *permutation indexing* is one of the important current open problems in pattern matching. It was introduced about 6 years ago and has seen active research since. Yet, to date there is no algorithm that can preprocess a text $T$ in time $o(|T|^2/\text{polylog}|T|)$ and achieve histogram indexing, even over a binary alphabet, in time independent of the text length. The pattern matching version of this problem has a simple linear-time solution. *Block-mass pattern matching* problem is a recently introduced problem, motivated by issues in mass-spectrometry. It is also an example of a pattern matching problem that has an efficient, almost linear-time solution but whose indexing version is daunting. However, for fixed finite alphabets, there has been progress made. In this paper, a strong connection between the histogram indexing problem and the block-mass pattern indexing problem is shown. The reduction we show between the two problems is amazingly simple. Its value lies in recognizing the connection between these two apparently disparate problems, rather than the complexity of the reduction. In addition, we show that for both these problems, even over unbounded alphabets, there are algorithms that preprocess a text $T$ in time $o(|T|^2/\text{polylog}|T|)$ and enable answering indexing queries in time polynomial in the query length. The contributions of this paper are twofold: (i) we introduce the idea of allowing a trade-off between the preprocessing time and query time of various

Royal Society Publishing

indexing problems that have been stumbling blocks in the literature. (ii) We take the first step in introducing a class of indexing problems that, we believe, cannot be pre-processed in time $o(|T|^2/\text{polylog}|T|)$ and enable linear-time query processing.

# 1. Introduction

## (a) Motivation

Traditional pattern matching regards the text $T$ and pattern $P$ as *sequential* strings, provided and stored in sequence (e.g. from left to right). Therefore, implicit in the conventional approximate pattern matching was the assumption that there may indeed be errors in the *content* of the data, but the *order* of the data is inviolate. However, some non-conforming problems have changed this view. Such examples range from text editing [1,2], through computational biology [3–6] and to architecture [7,8].

Motivated by these questions, a new pattern matching paradigm—*pattern matching with rearrangements*—was proposed by Amir *et al*. [9]. In this model, the pattern *content* remains intact, but the relative positions (addresses) may change. Several papers [7–13] followed the initial definition of the new paradigm. The simplest pattern matching with rearrangements problem is that of *histogram matching*, also known as *jumbled pattern matching*, *Parikh matching* or *permutation matching*. The goal of histogram matching is given a text, $T$, and pattern $P$, find all locations $i$ in the text where $T[i], \ldots, T[i+|P|-1]$ has the same alphabet symbols and the same frequency of symbols as $P$. Formally,

**Definition 1.1.**

1. Let $\Sigma$ be an alphabet, $S = S[1], \ldots, S[n]$, $S[i] \in \Sigma$. Let $\{\sigma_{i_1}, \ldots, \sigma_{i_k}\} \subseteq \Sigma$ be the set of alphabet symbols appearing in $S$. Let $\#_{\sigma_{i_j}}$ be the number of times $\sigma_{i_j}$ occurs in $S$, $j = 1, \ldots, k$. Call $\#_{\sigma_{i_j}}$ the *histogram frequency* of $\sigma_{i_j}$. Then $\{(\sigma_{i_1}, \#_{\sigma_{i_1}}), \ldots, (\sigma_{i_k}, \#_{\sigma_{i_k}})\}$ is the *histogram* of $S$.
2. The *histogram matching* problem is the following:
   *Input*: $T = T[1], \ldots, T[n]$ be a text and $P = P[1], \ldots, P[m]$ be a pattern over alphabet $\Sigma$.
   *Output*: All indices $i$ where the substring $T[i], \ldots, T[i+m-1]$ of $T$ has the same histogram as $P$.

In §2*a*, we will show that the histogram matching problem can be solved in time $O(n \log |\Sigma|)$.

We now consider another interesting, and efficiently solved, problem in the rearrangement model that was recently introduced by Ng *et al*. [14]. The problem is *block-mass indexing* (BMI).

From a biological point of view, the BMI problem arises in mass spectrometry. *Tandem mass spectrometry* (MS/MS) analyses *peptides* (short 8–30 amino acid long fragments of proteins) by generating their *spectra*.[1] The still unsolved problem in computational proteomics is to reconstruct a peptide from its spectrum: even the most advanced de novo peptide sequencing tools correctly reconstruct only 30–45 in MS/MS database searches [15]. After two decades of algorithmic developments, it seems that de novo peptide sequencing 'hits a wall' and that accurate full-length peptide reconstruction is nearly impossible due to the limited information content of MS/MS spectra. Recently, Jeong *et al*. [16] advocated the use of *gapped* (rather than full length) peptides to address the existing bottleneck in de novo peptide sequencing.

Given a string of $n$ integers $a_1, a_2 \ldots, a_n$ and $k$ integers $1 \leq i_1 < \cdots < i_k < n$, a gapped peptide is a string of $(k+1)$ integers $a_1 + \cdots + a_{i_1}, a_{i_1+1} + \cdots + a_{i_2}, \ldots, a_{i_k+1} + \cdots + a_n$. For example, if a peptide LNRVSQGK is represented as a sequence of its rounded amino acid masses 113, 114, 156,

[1]Spectra are complex objects that, for the sake of brevity, are not formally described in this paper.

99, 87, 128, 57, 128 then 113 + 114, 156 + 99 + 87, 128 + 57, 128 represents a gapped peptide 227, 342, 185, 128.

In [14], the BPM algorithm was presented. BPM leads to the modelling identification of mutated peptides and to modelling identification of fused peptides in tumour genomes.

Combinatorially, the BPI problem is defined below.

**Definition 1.2.** Let $T = T[1], T[2], \ldots, T[n]$ be a text over finite alphabet $\Sigma$, where $\Sigma \subset \mathbb{N}$. The *mass* of an alphabet symbol, $\sigma$ is its value, i.e. $\text{mass}(\sigma) = \sigma$. Let $S = T[i], \ldots, T[i+1], \ldots T[j]$ be a substring of $T$. Then mass($S$) (the *mass* of substring $S$) $= \sum_{\ell=i}^{j} T[\ell]$.

Substrings $T[i], \ldots T[j]$ and $T[j+1], \ldots T[k]$ are called *consecutive*. A *block* in $T$ is a sequence of consecutive substrings. The *mass of a block* $B$ (mass($B$)) is a string comprises the masses of the consecutive substrings of $B$, respectively. Formally, if $B = S_1 \cdots S_k$, then $\text{mass}(B) = \text{mass}(S_1), \ldots, \text{mass}(S_k)$.

**Example 1.3.** Assume an alphabet of 18 symbols that represents integer masses of 20 amino acids.
Let $T = 114, 77, 57, 112, 113, 186, 57, 99, 112, 112, 186, 113, 99$.
The substrings (57, 112, 113), (186, 57) and (99, 112, 112) define a block $B$ in $T$.
$\text{mass}(B) = 282, 243, 323$.

**Definition 1.4.** The *block-mass pattern matching problem* is defined as follows:
*Input*: A text T over alphabet $\Sigma$ and a pattern $P$ over $\mathbb{N}$.
*Output*: All blocks $B$ in the text T such that $\text{mass}(B) = P$.

We now introduce the indexing model. *Pattern matching with indexing* is the problem where, given text $T$, one preprocesses it so that subsequent pattern matching queries of the type: 'Given pattern $P$, find all occurrences of $P$ in $T$'. For a fixed finite alphabet, it is known how to preprocess a text in time $O(|T|)$ such that subsequent queries can be solved in time $O(|P|)$. These results are quite sophisticated and involve using suffix trees or arrays [17–22]. It is of interest to consider both the histogram matching problem and the block-mass matching problem in an indexing model. We formally define them below.

The *histogram indexing* (*Jumbled Indexing*, *Parikh Indexing*, *Permutation Indexing*) problem is defined as follows:

**Definition 1.5.** Let $T = T[1], \ldots, T[n]$ be a text over alphabet $\Sigma$.
*Preprocess*: $T$ to enable the following queries.
*Query*: Given pattern $P = P[1], \ldots, P[m]$ over $\Sigma$, decide whether there is a substring $T[i], \ldots, T[i + m - 1]$ of $T$ that has the same histogram as $P$.

The *indexing* version of the block-mass pattern matching problem, where the text is preprocessed so that subsequent queries can be answered without the need to search the entire text, was considered by Ng *et al.* [14]. Their indexing scheme handles a single-element pattern. The problem is formally defined below.

**Definition 1.6.** Let $T = T[1], T[2], \ldots, T[n]$ be a text over finite alphabet $\Sigma = \{i_1, \ldots, i_c\}$, where $\Sigma \subset \mathbb{N}$, and $i_1 < i_2 < \cdot < i_c$.

The *BMI problem* is that of preprocessing text $T$ in a manner that allows efficient solutions queries of the form:

*Query*: Given pattern $P$ over alphabet $\mathbb{N}$, find all blocks in $T$ whose mass is equal to the mass of $P$.

If indexing is not required, and we only consider the sequential version of the histogram pattern matching problem, and of the block-mass matching problem, these problems can be efficiently solved, as we will show in §2. In this sense, they are similar to the exact matching problem that is also efficiently solved, albeit using non-trivial techniques (e.g. [23,24]). Surprisingly enough, there are various ways to solve the exact indexing problem with linear-time

text preprocessing and linear-time pattern query, but the histogram indexing problem has proved elusively hard and is one of the important open problems in pattern matching today [25–29].

Similarly, the block-mass indexing problem does not have a pleasing solution. Ng *et al.* [14] present an efficient $O(m^2 + \text{tocc})$ algorithm for seeking blocks of mass $m$, following an $O(n^{1.5})$ text preprocessing, where the text length is $n$, and tocc is the number of length-$m$ blocks of mass $m$ in the text. They also present an algorithm with a linear-time preprocessing but a slower query time, of $O(m \cdot (\sqrt{n \cdot \text{tocc}} + \text{tocc}))$. Finally, they prove that there is an algorithm with preprocessing of $O(n \log n)$ time and space where the expected query time is $O(m + \text{tocc})$. The Ng *et al.* results hold for finite alphabets, and their main attractiveness is the algorithm with fast expected query time.

The main contribution of this paper is in showing that there is a strong relationship between the histogram indexing problem and the BPI problem with a single-element pattern, where a fixed finite alphabet is considered. Proving this relationship is not too complex. However, this is a case where a very *simple* observation is *important* because it illuminates a useful relationship between apparently different notions that no one thought to juxtapose. In this specific case, this relationship is useful for a number of reasons:

1. Sometimes there is more insight about a particular problem, and such insight can help solve equivalent problems.
2. It may have seemed strange that Ng *et al.* [14] considered a seemingly simple case of the BPI problem—one where the pattern has a single element. Our reduction shows that the single-element pattern case is indeed quite challenging.
3. Most importantly, this note offers some more theoretical understanding of indexing in the rearrangement model that has proved tougher than exact indexing. We conjecture that there is a class of indexing problems of greater complexity than exact matching, where it is impossible to preprocess the text $T$ in time $o(|T|^2/\text{polylog}|T|)$ and enable linear-time query processing. The problems considered in this paper are the first two examples of such problems.

Another major contribution of this paper is a new concept in measuring complexity of this class of indexing problems. Six years of research have not produced a histogram indexing algorithm whose pre-processing time is lower than $o(|T|^2/\text{polylog}|T|)$. Having conjectured that we cannot achieve a linear-time query processing followed by a pre-processing that is significantly faster than quadratic, we lower our sights. We develop a trade-off algorithm whereby, for any given $\epsilon$, the text $T$ is preprocessed in time $O(|T|^{1+\epsilon}\ \text{polylog}|T|)$, and then query pattern $P$ is processed in time $O(|P|^{1/\epsilon})$. We show that this trade-off scheme exists for both BMI and histogram indexing over unbounded alphabets. A similar idea was independently shown by Kociumaka *et al.* [30].

## 2. Sequential solutions

The most well-known problems for which there is no known efficient indexing solution is *Approximate Indexing*. Can one preprocess a text and then allow queries that seek all text substrings that are at most edit-distance $k$ to the query pattern $P$? Much work has been done on this problem (e.g. [31–33]) and yet no efficient indexing algorithm has been found. This may not be surprising, as there are no known linear-time sequential solutions to the problem [34–37].

In this section, we show that this is not the case for either the histogram matching problem or the block-mass matching problem. Both have efficient sequential solutions.

### (a) Sequential histogram matching

The following simple sliding window algorithm solves the histogram matching problem in time $O(|T| \log |\Sigma|)$, where $\Sigma$ is the set of distinct alphabet symbols in the pattern. The algorithm considers a window of length $m$ that slides over the text. The window is initialized to $T[1], \ldots, T[m]$. A slide to the right changes window $T[i], \ldots, T[i+m-1]$ to window

$T[i+1],\ldots,T[i+m]$. For every window, we keep the *window frequency* of every symbol in the window, i.e. the number of times the symbol occurs in the window, and the *histogram equality counter*, which keeps count of the number of symbols in the window whose window frequency is not equal to their histogram frequency. For every window $T[i],\ldots,T[i+m-1]$ whose histogram equality counter is 0, we have a histogram match at location $i$.

We initialize the first window in time $O(m)$ as follows.

**Sliding window Algorithm Initialization:**

1. Compute the window frequency of every alphabet symbol $\sigma$ in window $T[1],\ldots,T[m]$.
2. Write in the histogram equality counter $HEC$ the number of symbols whose window frequency does not equal their histogram frequency.

**end Initialization**

Every window slide requires a constant time update:

**Window Slide** from $T[i],\ldots,T[i+m-1]$ to $T[i+1],\ldots,T[i+m]$:

1. Decrement by 1 the window frequency of $T[i]$
2. Increment by 1 the window frequency of $T[i+m]$
3. Update $HEC$ as necessary to reflect the above changes.

**end Initialization**

## (b) Block-mass pattern matching

We provide a $O(n \log n)$ time convolutions-based algorithm for the sequential version of the block-mass patter matching problem.

Consider the following reduction:
Let $u:\mathbb{N}\rightarrow 10^*$ be the function that maps a natural number $n$ to a binary string $10^{n-1}$, where $a^i$ means the symbol $a$ repeated $i$ times.

**Example 2.1.** $u(5)=10\,000$, $u(2)=10$, $u(7)=1\,000\,000$ and $u(1)=1$.

For string $T$, construct bit array $B_T=B_T[1],\ldots,B_T[\mathrm{mass}(T)]$, where $B_T$ is the concatenation of $u(T[1]),\ldots,u(T[n])$.

**Definition 2.2.** Let $A=A[1],\ldots,A[n]$ and $B=B[1],\ldots,B[m]$ be binary arrays. We say that $B$ *point matches* $A$ *at location* $i$, if $A[i+\ell-1]=1$ for all $\ell$ such that $B[\ell]=1$.

**Example 2.3.** Let $A=00100100001010100100$ and $B=10001$. Then $B$ point matches in locations 11 and 15 of $A$ because those are the only two locations where *every* 1 in $B$ is aligned with a 1 in $A$. Note that there is no problem with the fact that when $B$ is positioned at location 11 of $A$, $A[13]=1$ whereas $B[3]=0$. We do not care if 1's in $A$ are aligned with 0's in $B$. It is only the other direction that is problematic.

Our reduction is now complete. It is obvious that

**Lemma 2.4.** *$P$ block-mass matches $T$ at location $i$ iff there is a point matching of $B_P$ in $B_T$ at location* $(\sum_{\ell=1}^{i-1} T[\ell])+1$.

Point matching can be effectively computed by *convolutions*.

### (i) The finite alphabet case

Convolutions are used for filtering in signal processing and other applications. A convolution uses two initial functions, $t$ and $p$, to produce a third function $t \otimes p$. We formally define a discrete convolution.

**Definition 2.5.** Let $T$ be a function whose domain is $\{1, \ldots, n\}$ and $P$ a function whose domain is $\{1, \ldots, m\}$. We may view $T$ and $P$ as arrays of numbers, whose lengths are $n$ and $m$, respectively. The *discrete convolution of T and P* is the polynomial multiplication $T \otimes P$, where

$$(T \otimes P)[j] = \sum_{i=1}^{m} T[j+i-1]P[i], \quad j = 1, \ldots, n-m+1.$$

In the general case, the convolution can be computed by using the fast Fourier transform (FFT) [38] on $T$ and $P^{\mathrm{R}}$, the reverse of $P$. This can be done in time $O(n \log m)$, in a computational model with word size $O(\log m)$.

*Important Property*: The crucial property contributing to the usefulness of convolutions is the following. For every fixed location $j_0$ in $T$, we are, in essence, overlaying $P$ on $T$, starting at $j_1$, i.e. $P[1]$ corresponds to $T[j_1]$, $P[2]$ to $T[j_1+1], \ldots, P[i]$ to $T[j_1+i-1], \ldots, P[m]$ to $T[j_1+m-1]$. We multiply each element of $P$ by its corresponding element of $T$ and add all $m$ resulting products. This is the convolution's value at location $j_1$.

Clearly, computing the convolution's value for every text location $j$, can be done in time $O(nm)$. The fortunate property of convolutions over algebraically close fields is that they can be computed *for all n* text locations in time $O(n \log m)$ using the FFT.

This property of convolutions can be used to efficiently compute relations of patterns in texts. This is generally done via *linear reductions* to convolutions. In the definition below $\mathbb{N}$ represents the natural numbers and $\mathbb{R}$ represents the real numbers.

**Definition 2.6.** Let $P$ be a pattern of length $m$ and $T$ a text of length $n$ over some alphabet $\Sigma$. Let $R(S_1, S_2)$ be a relation on strings of length $m$ over $\Sigma$. We say that the relation $R$ *holds between P and location j of T* if $R(P[1] \cdots P[m], T[j]T[j+1] \cdots T[j+m-1])$.

We say that $R$ is *linearly reduced* to convolutions if there exist a natural number $c$, a constant time computable function $f : \mathbb{N}^c \to \{0, 1\}$, and linear-time functions $\ell_1^m, \ldots, \ell_c^m$ and $r_1^n, \ldots, r_c^n$, $\forall n, m \in \mathbb{N}$, where $\ell_i^m : \Sigma^m \to \mathbb{R}^m$, $r_i^n : \Sigma^n \to \mathbb{R}^n$, $i = 1, \ldots, c$ such that $R$ holds between $P$ and location $j$ in $T$ iff $f(\ell_1^m(P) \otimes r_1^n(T)[j], \ell_2^m(P) \otimes r_2^n(T)[j], \ldots, \ell_c^m(P) \otimes r_c^n(T)[j]) = 1$.

Let $R$ be a relation that is linearly reduced to convolutions. It follows immediately from the definition that, using the FFT to compute the $c$ convolutions, it is possible to find all locations $j$ in $T$ where relation $R$ holds in time $O(n \log m)$.

**Lemma 2.7.** *Let T be a binary array of length n and P a binary array of length m. The point matching of P in T can be computed in time* $O(n \log m)$.

*Proof.* $\Sigma = \{0, 1\}$ and $R$ is the point matching relation. The locations where $R$ holds between $P$ and $T$ are the locations $j$ where $T[j+i-1] \geq P[i]$, $i = 1, \ldots, m$. This can be computed in time $O(n \log m)$ by the following simple reduction to convolutions.

Let $\ell_1 = \chi_1$, $r_1 = \chi_{\overline{1}}$, where

$$\chi_1(x) = \begin{cases} 1, & \text{if } x = 1; \\ 0, & \text{otherwise} \end{cases}$$

and

$$\chi_{\overline{1}}(x) = \begin{cases} 1, & \text{if } x \neq 1; \\ 0, & \text{otherwise} \end{cases}$$

and where we extend the definition of the functions $\chi_\sigma$ to a strings in the usual manner, i.e. for $S = S[1]S[2] \cdots S[n]$, $\chi_\sigma(S) = \chi_\sigma(S[1])\chi_\sigma(S[2]) \cdots \chi_\sigma(S[n])$.

Let

$$f(x) = \begin{cases} 1, & \text{if } x = 0; \\ 0, & \text{otherwise.} \end{cases}$$

Then for every text location $j$, $f(\ell_1(P) \otimes r_1(T)[j]) = 1$ iff there is a point matching of $P$ at location $j$ of $T$. ■

*Time*: For a finite alphabet $\Sigma$, $|B_T| = O(|T|)$, since $|B_T| \leq \sigma_{\max}|T|$, where $\sigma_{\max} \geq \sigma$, $\forall \sigma \in \Sigma$. We also have that $\sum_{\ell=1}^{m} P[\ell] \leq \text{mass}(T)$. We conclude that we can solve the block-mass pattern matching problem over finite alphabets in time $O(n \log n)$.

### (ii) The infinite alphabet case

In the infinite alphabet case, our reduction to bit-vectors $B_T$ and $B_P$ may be too large, even exponential in the input size. However, these vectors are quite sparse, with the vast majority of entries being 0. Therefore, we can overcome the exponential blowup in size by encoding the arrays as sets of the indices of the bit vector locations whose value is 1. The size of these sets is proportional to the original arrays.

The problem is that we are confronted with the problem of finding the convolution vector $W$ of the two vectors $B_T, B_P$, when the two vectors are not given explicitly. While in the regular fast convolution the running time is $O(|B_T| \log |B_P|)$, the aim here is to compute $W$ in time proportional to the number of non-zero entries in $W$. This problem was posed by Muthukrishnan [39].

Cole & Hariharan [40] proposed the following algorithm:

**Theorem 2.8.** *Let $V_1$ and $V_2$ be two sparse arrays, where $n_i$ is the number of non-zero entries in $V_i$, $i = 1, 2$. The convolution of $V_1$ and $V_2$ can be computed in time $O(w \log^2 n_2)$, by a Las Vegas randomized algorithm with failure probability that is inverse polynomial in $n_2$, where $w$ is the number of non-zero entries in the convolution vector.*

This result, although randomized, is very fast in practice. However, our situation is even better. We are assuming a static database $T$ where we can afford some preprocessing that subsequently will allow fast queries. In this case, there is a fast deterministic algorithm [41], which preprocesses the $V_1$ array in time $O(n_1^2)$, where $n_1$ is the number of non-zero entries of $V_1$, and subsequently achieves the following time for the sparse convolution:

**Theorem 2.9.** *Let $V_1$ and $V_2$ be two sparse arrays, where $n_i$ is the number of non-zero entries in $V_i$, $i = 1, 2$. The convolution of $V_1$ and $V_2$ can be computed in time $O(w \log^3 n_2)$, where $w$ is the number of non-zero entries in the convolution vector, following a $O(n_1^2)$ preprocessing of $V_1$.*

The above results mean that the block-mass pattern matching problem over alphabet $\mathbb{N}$ can be solved in the following time.

*Time*: Let $T$ be a text of length $n$ and $P$ a pattern of length $m$ over alphabet $\mathbb{N}$. The block-mass pattern matching problem can be solved in time $O((n + m) \log^2 m) = O(n \log^2 m)$ by a Las Vegas randomized algorithm with failure probability that is inverse polynomial in $m$. If $T$ is preprocessed in time $n^2$, then subsequent block-mass pattern matching queries can be solved deterministically in time $O(n \log^3 m)$.

Ng *et al.* [14] consider the simpler problem of indexing a string for block-mass matching of length-1 patterns. We will be considering the indexing version of this special case. Note, however, that block-mass pattern matching of length-1 patterns does not need the convolutions mechanism. It is easy to compute a 'sliding window' over $T$, where we compute the sum of all the numbers in the window and compare it to (the now single number in) $P$. Move the window by subtracting numbers on the left and adding numbers to the right. This algorithm gives a linear-time solution for the block-mass pattern matching problem with $P$ of length 1.

## 3. Indexing

As previously mentioned, the histogram indexing problem has proved elusively hard [25–29]. No known solution exists that answers pattern queries in time linear in the pattern length and preprocessing whose time is $o(n^2/\text{polylog } n)$, where $n$ is the text length.

The situation is not much better for the BMI problem. However, research there evolved in a different direction. Ng *et al.* [14] considered the special case of indexing a string for block-mass matching of length-1 patterns. Their algorithm assumes a finite fixed alphabet. Its bottleneck is computing the intersection of two sets. Assume we have an algorithm `Set Intersection` whose running time is $O(t_i)$. Ng *et al.* show:

**Theorem 3.1.** *Let $T$ be a text over fixed finite alphabet $\Sigma \subset \mathbb{N}$. Then it is possible to preprocess $T$ in time $O(n \log n)$ such that subsequent block-mass matching queries for patterns $P = P[1]$ can be answered in time $O(P[1] \cdot t_i + \text{tocc})$, where* tocc *is the number of blocks in $T$ with mass $P$, and $O(P[1])$ is the length of a block with mass $P$.*

In the attempt to circumvent the problem posed by set intersection, the following two solutions were provided:

**Theorem 3.2.** *Let $T$ be a text over fixed finite alphabet $\Sigma \subset \mathbb{N}$. Then it is possible to preprocess $T$ in time $O(n^{1.5})$ such that subsequent block-mass matching queries for patterns $P = P[1]$ can be answered in time $O(P[1]^2 + \text{tocc})$, where* tocc *is the number of blocks in $T$ with mass $P$, and $O(P[1])$ is the length of a block with mass $P$.*

**Theorem 3.3.** *Let $T$ be a text over fixed finite alphabet $\Sigma \subset \mathbb{N}$. Then it is possible to preprocess $T$ in time $O(n \log n)$ such that subsequent block-mass matching queries for patterns $P = P[1]$ can be answered in expected time $O(P[1] + \text{tocc})$, where* tocc *is the number of blocks in $T$ with mass $P$.*

## 4. The reductions

In this section, we begin study of a class of indexing problem where, we conjecture, no linear-time query can be done with $o(|T|^2/\text{polylog}|T|)$ pre-processing time. We start with the BMI problem and the histogram indexing problem. We prove the following two lemmas.

**Lemma 4.1.** *The histogram indexing problem over alphabet $\Sigma = \{1, \ldots, k\}$ is polynomially reducible to the BMI problem with length-1 pattern, over an alphabet of size $k$. In fact, the reduction is in time $O(n \log n)$, where $n$ is the input size.*

*Proof.* Given text $T$ of length $n$ over $\Sigma = \{1, \ldots, k\}$, construct text $T^{(j)}$ of length $2n + 1$ where for $i = 1, \ldots, n$:

$$T^{(j)}[2i] = j^{T[i]-1},$$

$$T^{(j)}[2i-1] = j^k$$

and

$$T^{(j)}[2n+1] = j^k.$$

**Example 4.2.** Let $T = 1, 1, 2, 1, 3, 2, 1, 2, 3$. Take $j = 4$.
$T^{(4)} = 64, 1, 64, 1, 64, 4, 64, 1, 64, 16, 64, 4, 64, 1, 64, 4, 64, 16, 64$.

Clearly, the length of $T^{(j)}$ is $2n$ words. The size of each word is $k\times$ the word size of $T$, but $k$ is a constant, thus the construction is linear.

The intuition behind this reduction is as follows. Assume that $j > m$, where $m$ is the pattern length. The $j^k$ padding guarantees that any pattern of length $m$, regardless of the histogram, will have total block mass less than all patterns of length $m + 1$ and more than all patterns of length $m - 1$. In addition, because of the encoding of symbol $i$ to mass $j^{i-1}$ it is impossible for any combination of symbols smaller than $i$ to achieve a mass larger than or even equal to that of $i$. This forces a uniqueness in the mapping of histograms to block masses that enables the reduction. We proceed with the details.

Construct $T^{(2^\ell)}$, $\ell = 2, \ldots, \log n$.

If we can preprocess each of the $T^{(j)}$'s in time $o(n^2)$ and subsequently solve block-mass queries of length 1 in time $O(f(n))$, then we can solve histogram queries in time $O(m^{k+1} + f(n))$. When presented with a pattern of length $m$, simply seek, in $T^{(j)}$, for the smallest $j$ such that $j \geq m$, a block with mass

$$\sum_{i=1}^{k} k_i j^{i-1} + (m+1)j^k$$

for any input pattern with $k_i$ $i$'s, $i = 1, \ldots, k$. The mass is bounded by $2m^{k+1}$, and it exists iff there is a substring with the input histogram.

**Example 4.3.** In the above example, consider a histogram of a pattern of length 3 that has one each of the elements 1, 2 and 3. $j = 4 > 3 = m$, so $T^{(4)}$ is the string we use for the block mass search. We seek locations whose mass is $1 + 4 + 16 + 4 \times 64 = 277$. This occurs at positions 5,7,9 and 13. Note that if only three pads are used, even if the symbol in between is the largest (16), the total sum is $16 \times 2 + 3 \times 64 = 214 < 277$. If five pads are used, even if the symbol between them is the smallest (1), the total sum is $3 + 5 \times 64 = 323 > 277$. Note also that no combination of two elements less than 16 add up to 16 (since $4 + 4 = 8 < 16$. This is true for every element. None can be replaced by a combination of smaller symbols.

■

Because of this complexity relation, we can infer for histogram indexing, algorithms that are derived for BMI.

**Corollary 4.4.** *Histogram indexing over a fixed finite alphabet can be solved in* $O(m^{2k+2} + \text{tocc})$ *time for seeking permutations of patterns of length* $m$*, following an* $O(n^{1.5} \log n)$ *text preprocessing, where the text length is* $n$*, and* tocc *is the number of length-*$m$ *substrings in the text that are permutations of the pattern.*

*Proof.* This is a direct result of lemma 4.1 and corollary 5.2. ■

Note that binary alphabet is a special case. We can construct $T^{(j)}$ as follows: For $i = 1, \ldots, n$:

$$T^{(j)}[2i] = T[i],$$

$$T^{(j)}[2i-1] = j$$

and

$$T^{(j)}[2n+1] = j.$$

Thus, the query time for a pattern of length $m$ is $O(m^4)$, with a $O(n^{1.5} \log n)$ text preprocessing.

In §5*b*, we will show a direct trade-off algorithm for histogram indexing, which applies also to unbounded alphabets.

To complete the complexity relation between the two problems, we show that the converse is also true.

**Lemma 4.5.** *The BMI problem with length-1 pattern over fixed finite alphabet* $\{a_1, \ldots, a_k\}$*, where* $a_i \in \mathbb{N}$*,* $i = 1, \ldots, k$*, is linearly reducible to the histogram indexing problem over a ternary alphabet, resulting in a query time with a multiplicative factor of the pattern length.*

*Proof.* Given text $T$ over fixed finite alphabet $\{a_1, \ldots, a_k\}$, where $a_i \in \mathbb{N}$, $i = 1, \ldots, k$, construct a text $T'$ as follows.

First, construct $T_u$ where

$$T_u[i] = \text{the unary representation of } T[i], \quad i = 1, \ldots, n.$$

Because the alphabet is fixed and finite, $|T_u| = a_{\max}|T| = O(|T|)$, where $a_{\max}$ is the largest number in $\{a_1, \ldots, a_k\}$.

Now construct $T'$ by replacing every $T_u[i]$ by the substring

$$c\, a^{|T_u[i]|}\, c\, b^{|T_u[i]|},$$

where $\sigma^x$ means $\sigma$ concatenated to itself $x$ times. Clearly, $|T'| = O(|T_u|) = O(|T|)$.

**Example 4.6.** Assume $T = 4, 4, 1, 2, 3$.
Then $T_u = 1111, 1111, 1, 11, 111$.
$T' = c\ aaaa\ c\ bbbb\ c\ aaaa\ c\ bbbb\ c\ a\ c\ b\ c\ aa\ c\ bb\ c\ aaa\ c\ bbb$.

Suppose we want to answer a BMI query of pattern $P[1] = m$. Consider the pattern $P_j = a^m b^m c^{2j}$, where $1 \le j \le m$. We claim that any location in $T$ that histogram matches $P_j$ has a block-mass $m$. For the justifications consider the following cases:

1. It cannot be the case that a solution to the histogram query starts with an $a$ and ends with a $b$, or starts with a $b$ and ends with an $a$, because then the number of $c$'s must be odd and that contradicts the query.
2. It cannot be the case that a solution to the histogram query starts and ends with a $c$ because then the number of $c$'s must be odd and that contradicts the query.
3. If the solution to the histogram query starts with an $a$ and ends with an $a$, then the number of $b$'s is exactly equal to $m$ and this correctly answers our BMI query in the affirmative. A similar argument holds for a solution to the histogram query that starts and ends with a $b$.
4. If the solution to the histogram query starts with a $ca$ and ends in an $a$, then the number of $b$'s is exactly equal to $m$ and this correctly answers our BMI query in the affirmative. A similar argument holds for a solution to the histogram query that starts with a $cb$ and ends with a $b$.
5. Finally, if the solution to the histogram query starts with a $ca$ and ends in a $b$, then the number of $a$'s is exactly $m$ and therefore this is an affirmative answer to the BMI query. A similar argument holds for a solution to the histogram query that starts with a $cb$ and ends with a $a$.

Note that while the solutions to the histogram queries exactly denote solutions to the BMI queries, it is still the case that up to $m$ consecutive solutions to the histogram query may indicate the same BMI query solution. It is easy to extract the correct solution from these responses, but it introduces a multiplicative factor of $m$ to our query response time.

Therefore, if we can preprocess $T'$ in time $o(n^2)$ and subsequently solve histogram indexing queries in time $O(f(m))$, then we can solve the BMI query in time $O(m^2 f(m))$ by simply querying all $m$ patterns: $P_1, \ldots, P_m$. ■

Putting the above two lemmas together we get:

**Theorem 4.7.** *The histogram indexing problem over fixed finite alphabet is equivalent to the BMI problem with length-1 pattern over a fixed finite alphabet, up to a multiplicative factor $m^k$ slowdown in query processing time.*

## 5. The trade-off algorithms

We have seen that to date there is no known algorithm for either the histogram indexing or the BMI problem that answers pattern queries in time linear in the pattern length following preprocessing whose time is $o(n^2/\text{polylog } n)$, where $n$ is the text length. Ng *et al.* [14] suggested reducing the preprocessing time on the BMI problem by increasing the query time. The query time is still dependent only on the pattern size and not the text length, but it is no longer linear in the pattern size. A similar relaxation was exploited by Kociumaka *et al.* [30] for the histogram indexing problem. In this section, we show trade-off algorithms for these two indexing problems that work for unbounded alphabets. The similarity between the two trade-off algorithms further strengthens our belief that these problems have similar complexities.

## (a) Alphabet independent block-mass indexing

In this section, we provide an algorithm for BMI that works for infinite alphabets. The algorithm has a trade-off between the preprocessing time and the query time. For a given $\epsilon$, if the processing time is $O(n^{1+\epsilon}\log n)$, then the query time, for pattern $P[1]$, is $P[1]^{1/\epsilon}$. Let $T$ be a text of length $n$ over alphabet $\mathbb{N}$.

The following algorithm preprocesses the data:

**Preprocessing Algorithm:**

1. For every $1 \le i \le j \le n$ such that $|j-i| \le n^{\epsilon}$ compute $s_{i,j} = \sum_{\ell=i}^{j} T[\ell]$.
2. Arrange the $s_{ij}$'s in an ordered data structure $TT$, where for every value $x$ there is a list of all pairs $(i,j)$ such that $s_{ij} = x$.

**end Algorithm**

**Preprocessing Time:** $O(n^{1+\epsilon}\log n)$.

Upon the arrival of query pattern $P[1]$, invoke the following algorithm:

**Query Processing Algorithm:**

1. If $P[1] \le n^{\epsilon}$ check if $P[1]$ is in $TT$, and if so, output $P[1]$'s list of pairs $(i,j)$.
2. If $P[1] > n^{\epsilon}$ then run a sliding window algorithm on the text $T$ seeking the sum $P[1]$.

**end Algorithm**

**Query Processing time:** For $P[1] \le n^{\epsilon}$ the time is $O(\log n)$. For $P[1] > n^{\epsilon}$, the time is $O(n)$, however, $n = (n^{\epsilon})^{1/\epsilon} > P[1]^{1/\epsilon}$.

Conclude:

**Theorem 5.1.** *Let $T$ be a text over $\mathbb{N}$, $\epsilon \in (0,1]$. Then it is possible to preprocess $T$ in time $O(n^{1+\epsilon}\log n)$ such that subsequent block-mass matching queries for patterns $P = P[1]$ can be answered in time $O(P[1]^{1/\epsilon} + \text{tocc})$, where tocc is the number of blocks in $T$ with mass $P$, and $O(P[1])$ is the length of a block with mass $P$.*

**Corollary 5.2.** *Let $T$ be a text over $\mathbb{N}$. Then it is possible to preprocess $T$ in time $O(n^{1.5}\log n)$ such that subsequent block-mass matching queries for patterns $P = P[1]$ can be answered in time $O(P[1]^2 + \text{tocc})$, where tocc is the number of blocks in $T$ with mass $P$, and $O(P[1])$ is the length of a block with mass $P$.*

## (b) Alphabet independent histogram indexing

We present a histogram indexing algorithm with a trade-off between the pre-processing and query processing time. Assume we have a text $T$ of length $n$.

Consider patterns of length $\ell$. Using a sliding window of length $\ell$, we can scan the text and update the histogram for every index $i$ by deleting element $T[i-1]$ and adding element $T[i+\ell]-1$. This update takes constant time for a finite fixed alphabet $\Sigma$, and $\log n$ time for unbounded alphabets, since we can keep the histogram in a balanced sort tree, for example. Thus, the time to prepare all histograms of length $\ell$ is $O(n\log|\Sigma|)$, where $\Sigma$ is the text alphabet. The necessary space is $O(n)$. We can index the text indices by histograms in various ways. One possible way is by sorting the symbols in every histogram and keeping the sorted lists in a trie format. For every tree leaf, we have a list of vectors, representing the values of the histograms with these symbols. This list of vectors can also be sorted and kept in a balanced sorted tree, where every leaf is a list of indices having the histogram represented by this leaf. Thus, processing a query of length $\ell$ takes time $O(\ell\log|\Sigma|)$. We call the data structure described above, the *histogram index data structure for length $\ell$* and denote it by $HT_{\ell}$.

We are now ready for our trade-off algorithm.

**Preprocessing Algorithm:**

1. For every $\ell = 1, \ldots, n^{\epsilon}$ compute the histogram index data structure $HT_{\ell}$.

**end Algorithm**

**Preprocessing Time:** $O(n^{1+\epsilon} \log |\Sigma|)$.

Upon the arrival of query pattern $P$, invoke the following algorithm:

**Query Processing Algorithm:**

1. If $|P| \leq n^{\epsilon}$ check if $P$'s histogram is in $HT_{|P|}$, and if so, output the list of indices with $P$'s histogram.
2. If $|P| > n^{\epsilon}$ then run a sliding window algorithm on the text $T$ seeking the histogram of $P$.

**end Algorithm**

*Query Processing time*: For $|P| \leq n^{\epsilon}$ the time is $O(\log |\Sigma|)$. For $|P| > n^{\epsilon}$, the time is $O(n)$, however, $n = (n^{\epsilon})^{1/\epsilon} > |P|^{1/\epsilon}$.

## 6. Conclusion and open problem

We have proved the equivalence of two important and challenging indexing problems in the rearrangement model—histogram indexing and BMI.

The key open problem, then, is whether these problems can indeed be solved with a $o(|T|\ \text{polylog}|T|)$ text indexing time and $O(|P|)$ query time. We conjecture that this is not the case. We believe that they are both members of a class of indexing problems of equivalent complexity.

A less sweeping interesting problem is showing that these two problems are equivalent without query-time degradation.

## References


1. Amir A, Lewenstein M, Porat E. 2002 Approximate swapped matching. *Info. Process. Lett.* **83**, 33–39. (doi:10.1016/S0020-0190(01)00302-7)
2. Amir A, Cole R, Hariharan R, Lewenstein M, Porat E. 2003 Overlap matching. *Info. Comput.* **181**, 57–74. (doi:10.1016/S0890-5401(02)00035-4)
3. Christie DA. 1996 Sorting by block-interchanges. *Inform. Process. Lett.* **60**, 165–169. (doi:10.1016/S0020-0190(96)00155-X)
4. Berman P, Hannenhalli S. 1996 Fast sorting by reversal. In *Proc. 8th Annual Symp. on Combinatorial Pattern Matching* (*CPM*), vol. 1075 (eds DS Hirschberg, EW Myers). Lecture Notes in Computer Science, pp. 168–185. Berlin, Germany: Springer.
5. Carpara A. 1997 Sorting by reversals is difficult. In *Proc. 1st Annual Int. Conf. on Research in Computational Biology* (*RECOMB*), pp. 75–83. New York, NY: ACM Press.
6. Bafna V, Pevzner PA. 1998 Sorting by transpositions. *SIAM J. on Disc. Math.* **11**, 221–240. (doi:10.1137/S089548019528280X)
7. Amir A, Aumann Y, Kapah O, Levy A, Porat E. 2008 Approximate string matching with address bit errors. In *Proc. 19th Symp. on Combinatorial Pattern Matching* (*CPM*), Lecture Notes in Computer Science, pp. 118–129. Berlin, Germany: Springer.
8. Amir A, Eisenberg E, Keller O, Levy A, Porat E. 2011 Approximate string matching with stuck address bits. *Theor. Comput. Sci.* **412**, 3537–3544. (doi:10.1016/j.tcs.2011.02.044)
9. Amir A, Aumann Y, Benson G, Levy A, Lipsky O, Porat E, Skiena S, Vishne U. 2006 Pattern matching with address errors: rearrangement distances. In *Proc. 17th ACM-SIAM Symposium on Discrete Algorithms* (*SODA*), pp. 1221–1229. New York, NY: ACM Press.

10. Amir A. 2006 Asynchronous pattern matching. In *Proc. 17th Symp. on Combinatorial Pattern Matching* (*CPM*). Lecture Notes in Computer Science, no. 4009, pp. 1–10. Berlin, Germany: Springer (invited talk).
11. Amir A, Aumann A, Indyk P, Levy A, Porat E. 2007 Efficient computations of $\ell_1$ and $\ell_\infty$ rearrangement distances. In *Proc. 14th Symp. on String Processing and Information Retrieval* (*SPIRE*). Lecture Notes in Computer Science, no. 4726, pp. 39–49. Berlin, Germany: Springer.
12. Amir A, Hartman T, Kapah O, Levy A, Porat E. 2007 On the cost of interchange rearrangement in strings. In *Proc. 16th Annual European Symp. on Algorithms* (*ESA*) (eds Lars Arge, Emo Welzl). Lecture Notes in Computer Science, no. 4698, pp. 99–110. Berlin, Germany: Springer.
13. Kapah O, Landau GM, Levy A, Oz N. 2008 Interchange rearrangement: the element-cost model. In *Proc. 15th Symp. on String Processing and Information Retrieval* (*SPIRE*). Lecture Notes in Computer Science, no. 5280, pp. 224–235. Berlin, Germany: Springer.
14. Ng J, Amir A, Pevzner PA. 2011 Blocked pattern matching problem and its applications in proteomics. In *Proc. 15th Annual Int. Conf. on Research in Computational Molecular Biology* (*RECOMB*), pp. 298–319. Vol. 6577, Lecture Notes in Computer Science. Berlin, Germany: Springer.
15. Frank AM, Pevzner PA. 2005 Pepnovo: De novo peptide sequencing via probabilistic network modeling. *Anal. Chem.* **77**, 964–973. (doi:10.1021/ac048788h)
16. Jeong K, Bandeira N, Kim S, Pevzner PA. 2011 Gapped spectral dictionaries and their applications for database searches of tandem mass spectra. *Mol. Cell Proteomics* **10**, M110.002220. (doi:10.1074/mcp.M110.002220)
17. Weiner P. 1973 Linear pattern matching algorithm. In *Proc. 14 IEEE Symp. on Switching and Automata Theory* (*SWAT*), pp. 1–11. Washington, DC: IEEE Computer Society.
18. McCreight EM. 1976 A space-economical suffix tree construction algorithm. *J. ACM* **23**, 262–272. (doi:10.1145/321941.321946)
19. Manber U, Myers G. 1990 Suffix arrays: a new method for on-line string searches. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms* (*SODA*), pp. 319–327.
20. Ukkonen E. 1995 On-line construction of suffix trees. *Algorithmica* **14**, 249–260. (doi:10.1007/BF01206331)
21. Farach M. 1997 Optimal suffix tree construction with large alphabets. In *Proc. 38th IEEE Symp. on Foundations of Computer Science* (*FOCS*), pp. 137–143. Washington, DC: IEEE Computer Society.
22. Kärkkäinen J, Sanders P. 2003 Simple linear work suffix array construction. In *Proc. 30th Int. Colloquium on Automata, Languages and Programming* (*ICALP 03*), Lecture Notes in Computer Science, no. 2719, pp. 943–955. Berlin, Germany: Springer.
23. Knuth DE, Morris JH, Pratt VR. 1977 Fast pattern matching in strings. *SIAM J. Comp.* **6**, 323–350. (doi:10.1137/0206024)
24. Boyer RS, Moore JS. 1977 A fast string searching algorithm. *Comm. ACM* **20**, 762–772. (doi:10.1145/359842.359859)
25. Cicalese F, Fici G, Lipták Zs. 2009 Searching for jumbled patterns in strings. In *Proc. Prague Stringology Conference* (*PSC*), pp. 105–117. Prague, Czeck Republic: Prague Stringology Club.
26. Burcsi P, Cicalese F, Fici G, Lipták Zs. 2010 Algorithms for jumbled pattern matching in strings. In *Proc. 5th Int. Conf. FUN with Algorithms* (*FUN*), pp. 89–101. Vol. 6099. Lecture Notes in Computer Science. Berlin, Germany: Springer.
27. Moosa TM, Rahman MS. 2012 Sub-quadratic time and linear size data structures for permutation matching in binary strings. *J. Disc. Algorithms* **10**, 5–9. (doi:10.1016/j.jda.2011.08.003)
28. Burcsi P, Cicalese F, Fici G, Lipták Zs. 2012 On approximate jumbled pattern matching. *Theory Comput. Syst.* **50**, 35–51. (doi:10.1007/s00224-011-9344-5)
29. Cicalese F, Laber E, Weimann O, Yuster R. 2012 Near linear time construction of an approximate index for all maximum consecutive sub-sums of a sequence. In *Proc. 23rd Annual Symp. on Combinatorial Pattern Matching*, pp. 149–158. Vol. 7354. Lecture Notes in Computer Science. Berlin, Germany: Springer.
30. Kociumaka T, Radoszewski J, Rytter W. 2013 Efficient indexes for jumbled pattern matching with constant-sized alphabet. In *Proc. 22nd Annual European Symp. on Algorithms* (*ESA*), pp. 625–636. Vol. 8125. Lecture Notes in Computer Science. Berlin, Germany: Springer.
31. Amir A, Keselman D, Landau G, Lewenstein M, Lewenstein N, Rodeh M. 2000 Indexing and dictionary matching with one error. *J. Algorithms* **37**, 309–325 (Preliminary version appeared in WADS 99.).

32. Cole R, Gottlieb L, Lewenstein M. 2004 Dictionary matching and indexing with errors and don't cares. In *Proc. 36th annual ACM Symp. on the Theory of Computing* (*STOC*), pp. 91–100. New York, NY: ACM Press.
33. Maaß MG, Nowak J. 2005 Text indexing with errors. In *Proc. 16th Symp. on Combinatorial Pattern Matching* (*CPM*). Lecture Notes in Computer Science, no. 3537, pp. 21–32. Berlin, Germany: Springer.
34. Landau GM, Vishkin U. 1989 Fast parallel and serial approximate string matching. *J. Algorithms* **10**, 157–169. (doi:10.1016/0196-6774(89)90010-2)
35. Abrahamson K. 1987 Generalized string matching. *SIAM J. Comp.* **16**, 1039–1051. (doi:10.1137/0216067)
36. Cole R, Hariharan R. 1998 Approximate string matching: a faster simpler algorithm. In *Proc. 9th ACM-SIAM Symp. on Discrete Algorithms* (*SODA*), pp. 463–472. New York, NY: ACM.
37. Amir A, Lewenstein M, Porat E. 2004 Faster algorithms for string matching with *k* mismatches. *J. Algorithms* **50**, 257–275. (doi:10.1016/S0196-6774(03)00097-X)
38. Cormen TH, Leiserson CE, Rivest RL. 1992 *Introduction to algorithms*. Cambridge, MA: MIT Press.
39. Muthukrishnan S. 1995 New results and open problems related to non-standard stringology. In *Proc. 6th Combinatorial Pattern Matching Conference*. Lecture Notes in Computer Science, no. 937, pp. 298–317. Berlin, Germany: Springer.
40. Cole R, Hariharan R. 2002 Verifying candidate matches in sparse and wildcard matching. In *Proc. 34st Annual Symp. on the Theory of Computing* (*STOC*), pp. 592–601. New York, NY: ACM.
41. Amir A, Kapah O, Porat E. 2007 Deterministic length reduction: Fast convolution in sparse data and applications. In *Proc. 18th Annual Symp. on Combinatorial Pattern Matching* (*CPM*), pp. 183–194. Vol. 4580. Lecture Notes in Computer Science, Berlin, Germany: Springer.