Tree pattern matching and subset matching in deterministic $O(n \log^3 n)$-time

Richard Cole* Ramesh Hariharan† Piotr Indyk‡

January 4, 1999

Abstract

The main goal of this paper is to give an $O(n \log^3 n)$ time deterministic algorithm for the the Subset Matching problem. This immediately yields an algorithm of the same efficiency for the Tree Pattern Matching problem. We also give an $O(n \log^3 n / \log \log n)$ time randomized algorithm for these problems. Finally, we give a $O(n \log n(z + \log n))$ time deterministic algorithm for a useful specialization of the Subset Matching problem in which all sets are intervals of a given length z.

1 Introduction

The main contribution of the paper are the improved running times for deterministic algorithms for the following two problems

- *Tree Pattern Matching problem:* given two ordered trees (called the *text* and the *pattern*), find all occurrences of the pattern in the text. The pattern occurs at a particular text position if placing the pattern with root at that text position leads to a situation in which each pattern node overlaps some text node.

- *Subset Matching problem:* given a pattern string $p = p[1 : n]$ and a text string t such that each pattern and text location is a set of characters drawn from some alphabet, find all occurrences of the pattern in the text. The pattern is said to occur at text position i if the set $p[j]$ is a subset of the set $t[i + j - 1]$, for all j.

The Tree Pattern Matching problem is well studied and has many applications (see [HO82]). The fastest algorithms for this problem were given by Cole and Hariharan who showed a linear reduction from Tree Pattern Matching to Subset Matching and provided $O(n \log^3 n)$-time randomized algorithm for the latter problem [CH97] (n denotes the size of the input). Subsequently, Indyk and others discoved several other applications for Subset Matching and its special cases [Ind97, IMV98]. See Section 1.1 for more description of previous and related work.

In this paper we give an $O(n \log^3 n)$ time *deterministic* algorithm for the the Subset Matching problem. This improves an earlier bound of $O(n^{1+o(1)})$ by Indyk [Ind97] and immediately yields an algorithm of the same efficiency for the Tree Pattern Matching problem. We also give an $O(n \log^3 n / \log \log n)$ time randomized algorithm for these problems. Finally, we give an $O(n \log n(z+\log n))$ time deterministic algorithm for a useful specialization of the Subset Matching problem in which all sets are intervals of a given length z.

These improved bounds follow immediately from improved bounds for the following two problems.

1. Data Dependent Superimposed Codes. This problem was defined by Indyk [Ind97]. Basically, given a collection of sets drawn from a universe U, the problem is to find equal length binary codes for the items in U that are *separating*, in the following sense. A set is coded as the 'or' of the codes for its items, and the codes for any item e and set A, with $e \notin A$, satisfy:

> there is at least one '1' in $code(e)$ such that the bit in $code(A)$ in the same position is a zero.

For the moment, we specify the complexity using just two parameters: z, the bound on the set size, and $n = |U|$; we suppose that the number of sets and the sum of the sets sizes are both $O(n)$. Indyk gave an algorithm which produces codes of

$$\text{size } O(z \log^3 n) \text{ in time } O(nz \log^2 n).$$

We improve this to produce codes of size

*Courant Institute, NYU, cole@cs.nyu.edu. This work was supported in part by NSF grants CCR-9503309 and CCR-9800085

†Indiana Institute of Science, ramesh@csa.iisc.ernet.in. This work was done in part while visiting NYU. This work was supported by NSF grants CCR-9503309 and CCR-9800085

‡Stanford University, indyk@cs.stanford.edu. This work was supported by Stanford Graduate Fellowship and NSF Award CCR-9357849, with matching funds from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

$$O(z \log n \log\log n) \text{ in time } O(nz \log^2 n).$$

For the case when the universe is ordered and all sets are intervals are of length z, we give an *explicit* construction of superimposed codes of length $O(z + \log n)$; we call them *interval codes.* The code length can be showed to be optimal for this set family; it also *beats* the existential $O(z \log n)$ bound obtained via the natural probabilistic method. We also consider a weighted variant of the problem in which the number of sets of size $\geq k$ is at most $n/2^k$. For this scenario, we can further improve the bounds to produce codes of size

$$O(\log^2 n) \text{ in time } O(n \log^2 n).$$

(To compare these bounds to those in the previous paragraph, set $z = \log n$.)

2. A Minimum Weight Problem on 2-D arrays. This problem was implicitly considered by Cole and Hariharan and then formalized by Indyk. The problem is to rotate the rows of a sparse 2-D array so as to minimize the sum of the entries in any column (called the column *weights*). The entries are non-negative integers (in our application they are all ones or zeros). Let n be the number of non-zero entries plus the row length. The trivial linear randomized algorithm of using random shifts achieves a bound of $O(\log n / \log\log n)$ on the weights. Indyk gave a deterministic $O(n \cdot 2^{\sqrt{\log n \log\log n}})$ time algorithm that achieved a bound of $O(2^{\sqrt{\log n \log\log n}})$ on the weights. In this paper we give a deterministic $O(n \log^3 n)$ time algorithm that achieves a bound of $O(\log n)$ on the weights.

The weights in the Minimum Weight Problem correspond to the set sizes in the Subset Matching Problem. The reason reducing the weights is helpful is that the efficiency of the algorithm for Subset Matching is proportional to the bound on the weights (aside the cost of the weight reduction step).

The Subset Matching problem is translated into a string matching with "don't cares" problem, by replacing each set with its coding in a data dependent superimposed code. The string matching problem is then solved by standard methods based on convolutions.

Consequently, the overall running time for subset matching comprises the sum of three terms:

1. $O(\log n)$ times the size of the problem resulting from the data dependent coding; this coding has size $O(n \log^2 n)$, since each set has a code size of $O(\log^2 n)$. This is $O(n \log^3 n)$ time.

2. The time to compute the coding, $O(n \log^2 n)$.

3. The time to reduce the weights, $O(n \log^3 n)$.

Thus improvements in the bounds on the weights or in the size of the coding will lead to better overall running times, so long as the time to compute (2) and (3) above stay appropriately bounded. Indeed, our better bound for a randomized algorithm for Subset Matching results from achieving an $O(\log n / \log\log n)$ bound on the weights.

The algorithm for the Mimimum Weight problem is a derandomization of a very simple process (of giving each row a random length rotation). Indyk's derandomization was based on the method of conditional probabilities [AS92] and pessimistic estimators. We devise a more direct method built on a new technique we call *approximate convolution.* Our solution to this problem immediately improves Indyk's bound for the Subset Matching problem to $O(n \log^5 n)$.

The improvement for the Data Dependent Coding problem gains a smaller factor of $\Theta(\log^2 n)$. It relies on two steps. The first step is to formulate Indyk's algorithm in terms of a constraint satisfaction problem and accelerate it by adaptively changing a parameter that had been left fixed previously. The second is to take advantage of the distribution of set sizes (weights) yielded by the solution to the Minimum Weight problem.

Finally, for the case when all sets are intervals of a given length z, an aplication of the interval codes yield an algorithm for Subset Matching with running time $O(n \log n (z + \log n))$. This problem has been considered in [Ind97] with application to searching in financial data.

In the remainder of the introduction, we describe the various problems in more detail and review past work.

1.1 Background and Related Work

The Tree Pattern Matching problem. The text and the pattern are ordered, binary trees and all occurrences of the pattern in the text are sought. Here, the pattern occurs at a particular text position if placing the pattern with root at that text position leads to a situation in which each pattern node overlaps some text node.

This problem is well studied and has many applications (see [HO82]). Actually, in these applications, the tree need not be binary and the edges may be labelled; however, as shown in [DGM94], this general problem can be converted to a problem on binary trees with unlabelled edges but with a blow-up in size proportional to the logarithm of the size of the pattern. Subsequently, it was shown that there need be no blow-up in problem size [CH97].

The naive algorithm for tree pattern matching takes time $O(nm)$, where n is the text size and m is the pattern size. Hoffman and O'Donnell [HO82] gave another algorithm with the same worst case bound. This algorithm decomposes the pattern into strings, each string representing a root-to-leaf path. It then finds all occurrences of each of these strings in the text tree. The first $o(nm)$ algorithm was obtained by Kosaraju [Kos89] who first noticed the connection of the tree pattern matching problem to the problem of string matching with don't-cares and the problem of convolving two strings. Kosaraju's algorithm takes $O(nm^{.75}\log m)$ time. Dubiner, Galil and Magen [DGM94] improved Kosaraju's algorithm by discovering and exploiting periodicities in paths in the pattern. They obtained a bound of $O(nm^{.5}\log m)$.

A significant improvement in the running times was obtained by Cole and Hariharan; they gave a linear time reduction of Tree Pattern Matching to the Subset Matching problem, which they introduced [CH97]. They also found an $O(n\log^3 n)$ time randomized algorithm for Subset Matching, and an $O(nm^{.5})$ time deterministic algorithm. Indyk sharply improved the deterministic bound, giving an algorithm with running time $O(nm^{\sqrt{\frac{\log\log n}{\log n}}(1+o(1))})$.

The Subset Matching Problem. This problem was defined in [CH97]. The goal is to find all occurrences of a pattern string p of length m in a text string t of length n, where each pattern and text location is a set of characters drawn from some alphabet. The pattern is said to occur at text position i if the set $p[j]$ is a subset of the set $t[i+j-1]$, for all j, $1 \leq j \leq m$. The special case when all sets are intervals of a given length has also been considered (see [Ind97]), with application to searching in financial data.

We use the following notation: $\mathcal{T} = \cup_i t[i]$, $\mathcal{P} = \cup_i p[i]$, $\mathcal{A} = \mathcal{T} \cup \mathcal{P}$, $s = \sum_i |t[i]| + \sum_j |p[j]|$. As pointed out in [CH97], without loss of generality one can assume that $\mathcal{A} = \mathcal{T} = \mathcal{P}$, $n \leq 2m$ and $s \leq 6m$ (otherwise the problem can be linearly reduced to several subset matching subproblems satisfying these properties). Therefore, in the sequel we consider only t's and p's satisfying the above conditions, unless mentioned otherwise.

Indyk discovered a deterministic algorithm for Subset Matching [Ind97] with a running time of $O(nm^{\sqrt{\frac{\log\log n}{\log n}}(1+o(1))})$. This stemmed from the introduction of the notion of "data dependent" superimposed codes, efficient algorithms for their construction and their application to Subset Matching.

2 Preliminaries

In this section we introduce basic notions and definitions used later in the paper.

By $[d]$ we denote the set $\{0,\ldots d-1\}$. Let $v, w \in \{0,1\}^d$ be two vectors. As there is a natural correspondence between vectors and subsets of $[d]$, we will often describe the vectors using set terminology. Specifically, we say that v *contains* w if, for each $i = 1\ldots d$, $w_i \leq v_i$, where v_i is the ith coordinate of v and w_i is the ith coordinate of w, or in other words $w \subseteq v$.

Vector and matrix notation. $|a|$ denotes $\sum_i |a[i]|$ (we call it the *norm* of a). For a matrix $A[0\ldots k-1, 0\ldots n-1]$, $A[\cdot, i]$ ($A[i,\cdot]$ resp.) denotes the ith column (row resp.) of A. Finally, $|A|$ is defined to be the vector $[|A[\cdot,0]|,\ldots,|A[\cdot,n-1]|]$. Note that $||A||$ is equal to $|v_A|$ where $v_A = |A|$; thus $||A||$ denotes the sum of elements of A.

Superimposed Codes. There exist several variants of superimposed codes. In this paper we use the definition from [DR83].

DEFINITION 1. *An $N \times M$ 0-1 matrix A is called a* superimposed (z,M)-code *(or (z,M)-code) of length N if the boolean sum of any z columns of A does not contain any other column. We refer to A's columns (which we denote by $A[0]\ldots A[M-1]$) as* codewords.

In this paper we are interested in the situation when M and z are given and the goal is to minimize N. Let $N_{\min}(z,M)$ denote the minimum length of any (z,M)-code. Dyachkov and Rykov [DR83] showed that $N_{\min}(z,M) = \Omega(z^2\log_z M)$ (similar bounds were also obtained by Erdos, Frankl and Furedi [EFF85]). They also obtained an upper bound of $O(z^2\log M)$; it was based on a probabilistic argument and therefore was non-constructive. The best explicit construction [KS64] (based on Reed-Solomon codes) achieves $N = O(z^2\log_z^2 M)$.

In order to the break the $\tilde{\Omega}(z^2)$ code length lower bound, we use the following "data dependent" definition of superimposed codes [Ind97], which generalizes Definition 1.

DEFINITION 2. *Let $\mathcal{S} = S_0\ldots S_{K-1}$ be a collection of subsets of $[M]$. An $N \times M$ 0-1 matrix A is called a superimposed $\mathcal{S}$-code if for any $S \in \mathcal{S}$ the boolean sum of columns with indices in S does not contain any other column.*

We need the following additional notation: $||S||$ denotes the representation size of a set S (in words). For example if S is a family of sets of numbers, $||S||$ is the sum of cardinalities of all sets in S.

There are two issues in the construction of superimposed codes: the first is to find short codes, and the second is to construct them quickly. Indyk gave an algorithm for constructing superimposed codes of size $O(z\log K\log^2 M)$ in time $O(||S||z\log^2 M)$ [Ind97].

3 Overview of the algorithm

The basic technique used by the Subset Matching algorithm is provided in the following theorem:

THEOREM 3.1. *[Ind97] The subset matching problem for strings t and p such that $|t[i]| \leq z$ for all $i = 0 \dots n-1$ can be solved in time $O(nN(n,z)\log n + T(n,z))$, where $T(n,z)$ is is the time needed to compute a $\{t[0], \dots, t[n-1]\}$-superimposed code of length $N(n,z)$.*

The above theorem gives a fast subset matching algorithm provided that we can ensure that:

- Each set of the text has small cardinality.
- The superimposed code can be found quickly.

Reducing set size. Our goal is to generate strings t'' and p'' such that all sets in both t'' and p'' have small cardinality and p'' matches t'' at position i if and only if p matches t at the same position. The general idea is as follows. Define 0-1 matrices $T[0 \dots k-1, 0 \dots n-1]$ and $P[0 \dots k-1, 0 \dots n-1]$ such that $T[a,i] = 1$ if $t[i] = a$ and $P[a,i] = 1$ if $p[i] = a$. It is easy to see that p matches t at position i if and only if

(3.1)$\forall_{j=0 \dots n-1} \forall_a P[a,j] \leq T[a, i+j]$ or equivalently
(3.2)$\forall_a \forall_{j=0 \dots n-1} P[a,j] \leq T[a, i+j]$

Given a vector X of length n and i, $0 \leq i \leq n-1$ we define $X+i$ to be the following vector of length n: $X[j] = (X+i)[(j+i) \bmod n]$, for $0 \leq j \leq n-1$. We say that $X+i$ is the result of *shifting* X by i positions. Cole and Hariharan [CH97] observed that conditions (1) and (2) remain invariant when the corresponding rows of both T and P are shifted by the same number of positions. Therefore, by applying suitable shifts to both T and P, it is possible to reduce the number of ones in each column of T (we refer later to this quantity as to the *weight* of that column). This motivates the following Minimum Weight Problem:

DEFINITION 3. **(Minimum Weight Problem):** *[Ind97] Given positive integers b, d and a binary matrix $T[0 \dots k-1, 0 \dots n-1]$ containing at most b ones, find (if possible) k positive integers $l_0 \dots l_{k-1} \in [n]$ such that the matrix T' obtained by shifting the ith row of T by l_i positions has the property that each of its columns contains at most d ones.*

MWP(n,k,b,d) denotes the class of Minimum Weight Problems with parameters n, k, b and d.

THEOREM 3.2. *The $MWP(n,k,n,O(\log n))$ problem can be solved in deterministic $O(n \log^3 n)$ time.*

In fact, our algorithm gives an even stronger guarantee. Specifically, for any $x \geq 0$ we define a function

$$w(x) = \begin{cases} 2^x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

and extend it to arrays by defining $w(a)[i] = w(a[i])$. Then our algorithm is guaranteed to output T' such that $|w(|T'|)| = O(n)$.

Construction of superimposed codes. We give two algorithms for constructing short superimposed codes. The first one generates codes of length only a factor of $\log\log n$ larger than the probabilistic bound. More specifically, we show how to construct superimposed codes of size $O(z \log M(\min\{\log\log ||S||, \log\log M\} - \log\log d) + z\log ||S||)$ in time $O(||S|| \frac{d}{\log d} z \log^2 M)$, where d is a free parameter, $2 \leq d \leq M$.

The second algorithm works under the assumption that $|w(|T'|)| = O(n)$ and (by calling the first procedure several times) produces codes which are optimal. This is due to the fact that the weight bound implies that the number of sets of size $\geq k$ is at most $O(n/2^k)$, and for this class of sets we can apply the above construction according to set size, which leads to the improved bounds stated earlier.

Finally, in Section 6 we show that for the family of sets containing all intervals of $[M]$ of length z one can find codes of length $O(z + \log M)$. This leads to an $O(n \log m(z + \log M))$-time deterministic algorithm for subset matching where all sets are intervals of $[M]$ of length z.

4 The minimum Weight Problem

Let $a[0 \dots n-1]$ and $b[0 \dots m-1]$ be arrays of reals. $a \circ b$ is defined as follows: $(a \circ b)[s] = a \cdot (b+s)$ where $\cdot$ is a *dot product*. It is well-known that this operation can be performed in $O(n \log n)$ time [AHU74].

For a 2-row array, a simple approach to finding optimal shifts is to let $a[i] = T[0,i]$ and $b[i] = T[1,i]$ and compute $a \circ b$; the value s minimizing $(a \circ b)[s]$ provides the optimal shift of row 1 of T with respect to row 0. More generally, one can imagine an algorithm based on pairing the rows, for each pair determining the best relative shift among the two rows, then combining the pairs of rows into single rows and iterating. We ignore the details of this approach for it is too expensive; it will have a running time of $\Theta(nk \log n)$, and for our application $k = \Theta(n)$, where k is the number of columns.

We will follow the framework described in the previous paragraph, but instead of computing the best i we compute a "good enough" i, using a method we call *approximate convolution*. The running time of this method is, up to log terms, proportional to the number of ones in the two rows being compared.

We combine rows by adding their aligned entries. Our goal is to find a good relative shift for each pair of rows before combining them. More specifically, if the two rows to be combined have weights w_1 and w_2, respectively, we seek to keep the weight of the combined rows bounded by $(w_1+w_2)(1+1/\log n)$. For if the total initial weight of the rows was n, on combining the rows pairwise, over a series of $\log n$ iterations, we obtain a single row of weight $O(n)$. Shifts on combined rows are applied to each of their constituent rows. On applying all these shifts, the weight bound implies there are at most $O(\log n)$ items per column. In fact, the parameters in the actual algorithm need to vary, depending on the weights of the rows being combined.

The formal description of the algorithm is as follows. The algorithm proceeds in a sequence of steps. After each step the rows of T are partitioned into *megarows*. Each megarow r of size s is an $h \times n$ matrix induced by a sequence $i_0 \ldots i_{t-1}$ of row indices and a sequence $t_0 \ldots t_{h-1}$ of *shifts* such that $r[u,v] = T[i_u, v - t_v \mod n]$. Note that each row of T is a trivial megarow; therefore in the sequel we use the term row to denote both a row and a megarow. During each step two rows r_1 and r_2 are selected and *merged* (we describe both the selection and merging procedure in detail shortly). The new row replaces old rows r_1 and r_2 and the algorithm continues to the next step.

The details of the algorithm follow. First, we partition the rows of T into classes $C_0 \ldots C_{\log n}$ such that the ith class C_i contains rows r for which $|w(r)| \in [2^i, 2^{i+1})$. Then we apply the following procedure:

procedure *REDUCE*
for $i = 1 \ldots \log n - 2$
  while $|C_i| > 1$ do
    choose two rows r_1 and r_2 from C_i
    create a row r by merging r_1 and r_2
    (described below)
    add r to the C_j with $|w(r)| \in [2^j, 2^{j+1})$
  endwhile
  if $C_i \neq \emptyset$, add $r \in C_i$ to C_{i+1}
endfor

The goal of the merging procedure is to find a shift t such that $|w(r_1 + (r_2+t))|$ is small; this value can be found using a convolution. Unfortunately, the number of potential values of t is large (i.e. $O(n)$). Therefore, we resort to a technique we call *approximate convolution*, which follows. A parameter λ is needed. For $n/(\log^{[j]} n)^2 \leq |w(r)| < n/(\log^{[j+1]} n)^2$, $\lambda = (\log^{[j+1]})^2$, for $j \geq 1$, and for $|w(r)| < n/\log^2 n$, $\lambda = \log n$. Recall $r_1 \in C_i$ and $r_2 \in C_i$. Set $n_i = \min\{2^i\lambda, n\} = 2^i\lambda$. The integer interval $[n]$ is split into n_i equal intervals $I_0 \ldots I_{n_i-1}$. Define $a_1[j] = \sum_{i\in I_j} w(|r_1|)[i]$ and $a_2[j] = \sum_{i \in I_j} w(|r_2|)[i]$. By using convolutions we find the shift s minimizing the value $|a_1 \cdot (a_2+s)|$. Finally, the value of t is defined to be $t = s\frac{n}{n_i}$.

After we find this value, we form a new row r from r_1 and r_2 as follows. Let $i_0^1 \ldots i_{h_0-1}^1$ and $t_0^1 \ldots t_{h_0-1}^1$ denote the sequences inducing r_1; similarily, let $i_0^2 \ldots i_{h_1-1}^2$ and $t_0^2 \ldots t_{h_1-1}^2$ denote the sequences inducing r_2. The new row r of size h_1+h_2 is induced by sequences $i_0^1 \ldots i_{h_1-1}^1, i_0^2 \ldots i_{h_2-1}^2$ and $t_0^1, \ldots, t_{h_1-1}^1, t_0^2 + t, \ldots, t_{h_2-1}^2 + t$.

Correctness. The correctness of this procedure is proven as follows. First we give the guarantees for the approximate convolution subroutine.

LEMMA 4.1. *Let a_1 and a_2 be two arrays of length n_i; also let $w_1 = |a_1|$ and $w_2 = |a_2|$. Then there exists an s such that*
$$|a_1 \cdot (a_2+s)| \leq \frac{w_1 w_2}{n_i}.$$

Proof. Choose s uniformly at random from $[n_i]$. Then for any $i \in [n_i]$:
$$E_s\left[(a_1 \cdot (a_2+s))[i]\right] = a_1[i] \cdot \frac{w_2}{n_i}$$
Therefore (by linearity of expectation)
$$E_s\left[|a_1 \cdot (a_2+s)|\right] = \frac{w_1 w_2}{n_i}.$$
Thus there exists an s for which $|a_1 \cdot (a_2+s)| \leq \frac{w_1 w_2}{n_i}$.

LEMMA 4.2. *The row r satisfies the inequality*
$$|w(|r|)| \leq \frac{w_1 w_2}{n_i} + w_1 + w_2.$$

Proof. Let $r_2' = r_2 + t$, where t is the shift computed by the algorithm. Define $S = \{i | w(|r_1|)[i] \cdot w(|r_2'|)[i] = 0\}$ and $\overline{S} = [n] - S$. Then (by the definition of w)
$$\begin{aligned} |w(|r|)| \;&=\; \sum_{i \in S} (w(|r_1|)[i] + w(|r_2'|)[i]) + \\ &\quad\; \sum_{i \in \overline{S}} w(|r_1|)[i] \cdot w(|r_2'|)[i] \end{aligned}$$
which is bounded by
$$\begin{aligned} & w_1 + w_2 + \sum_{i \in \overline{S}} w(|r_1|)[i] \cdot w(|r_2'|)[i] \\ = \;& w_1 + w_2 + \sum_{i \in [n]} w(|r_1|)[i] \cdot w(|r_2'|)[i] \end{aligned}$$

$$
\begin{aligned}
&= w_1 + w_2 + \sum_{j\in[n_i]} \sum_{i\in I_j} w(|r_1|)[i]\cdot w(|r_2'|)[i] \\
&\le w_1 + w_2 + \\
&\quad \sum_{j\in[n_i]}\left[\left(\sum_{i\in I_j} w(|r_1|)[i]\right)\left(\sum_{i\in I_j} w(|r_2'|)[i]\right)\right] \\
&= w_1 + w_2 + \sum_{j\in[n_i]} a_1[j]\cdot(a_2+s)[j] \\
&= w_1 + w_2 + |a_1\cdot(a_2+s)| \\
&\le w_1 + w_2 + \frac{w_1w_2}{n_i}
\end{aligned}
$$

The next lemma shows that the weight of the row obtained as a result of the procedure is $O(n)$.

LEMMA 4.3. *The value of* $|w(|r|)|$ *obtained by merging rows* r_1 *and* r_2 *from* C_i *satisfies*

$$|w(|r|)| \le (|w(|r_1|)| + |w(|r_2|)|)\left(1+\frac{1}{\lambda}\right)$$

Proof. let $w_1 = |w(|r_1|)|$, $w_2 = |w(|r_2|)|$. After the merge the weight w of the new row r is bounded by:

$$
\begin{aligned}
w &\le w_1 + w_2 + \frac{w_1w_2}{n_i} \\
&\le w_1 + w_2 + \frac{w_1+w_2}{2}\frac{\max\{w_1,w_2\}}{2^i\lambda} \\
&\le (w_1+w_2)(1+1/\lambda)
\end{aligned}
$$

It is helpful to record how many items contribute to a row; we define this to be the *base weight* of the row: $bw(r)$ is the sum of the weights of the entries in row r.

LEMMA 4.4. *The ratio of the base weight and the weight of each final row obtained by running the procedure* REDUCE *is* $O(1)$.

Proof. Partition the iterations of procedure *REDUCE* into phases based on the value of λ. There are at most $\log^{[i]} n$ phases with $\lambda = (\log^{[i]} n)^2$, for $i > 1$, and there are at most $\log n$ phases with $\lambda = \log n$ $(i = 1)$. Thus, by Lemma 4.3, the ratio of the weight of a final vector to its base weight is bounded by

$$(1+\frac{1}{\log n})^{\log n}\prod_{i>1}\left(1+\frac{1}{(\log^{[i]} n)^2}\right)^{\log^{[i]} n}$$

which is $O(1)$.

Thus, at the end of this process we have $O(1)$ rows each of weight at most n. They are then repeatedly combined pairwise by a direct convolution, until one row remains; by Lemma 4.2, this row has weight at most $0(n)$. This implies that in each row, there are at most $\log n$ items in any column. Hence we obtain columns of size $O(\log n)$, and indeed of total weight $O(n)$.

We turn to the running time. Processing C_i takes time $O(\lambda n \log n)$, for $O(\lambda n)$ is the total size of the arrays to be convolved, and the $\log n$ factor accounts for the time to compute convolution. Summing over all i gives a time bound of $(O(n\log^3 n)$ (recall that $\lambda \le \log n$).

5 Constructing superimposed codes

We use the notation 1_y (for $y \in [b]$) to denote a b-bit vector containing 1 in its yth position and zeros elsewhere.

5.1 The intuition. The superimposed codes are built from a series $h_1, \cdots, h_k$ of hash functions as follows. Suppose h_i has range $[b_i]$; then the code for item a is simply the concatenation $1_{h_1(a)}1_{h_2(a)}\cdots 1_{h_k(a)}$. $h_1, \cdots h_k$ provide a separating code, if for each set C and each $a \notin C$, there is some hash function h_i such that $h_i(a) \ne h_i(c)$ for all $c \in C$ (we write $h_i(a) \not\subseteq h_i(C)$).

The separating property for superimposed codes can be expressed as $O(M|S|)$ constraints of the form (a, C) where a is an item and $C \in S$ is a set, with $a \notin C$. We will devise a collection H of hash functions such that for each constraint, for at least half of the hash functions $h \in H$, $h(a) \not\subseteq h(C)$, i.e. h separates a and C. (Recall that $h(C) = \bigvee_{c\in C} c$.) Thus in time $O(|H|M||S||)$ we could find $O(\log M + \log|S|)$ hash functions which separate all the constraints (assuming each hash function can be evaluated in $O(1)$ time on a single item).

Unfortunately, $\Theta(M) = \Theta(||S||) = n$ in our application, which makes this procedure unacceptably slow. Indyk observed that the computation could be structured to run much faster [Ind97], at the cost of using more hash functions. We can view his approach as follows. Each item is considered to be a d-ary number $a = a_1a_2\cdots a_t$ $(t = \log_d M)$, and $0 \le a_i < d$, where d is a free parameter. His algorithm performs t iterations; in each iteration, one more digit in all the items is considered. Thus in the first iteration a constraint (a, C), where $C = (b, c, \cdots, f)$ is reduced to (a_1, C_1) where $C_1 = (b_1, c_1, \cdots, f_1)$. The constraints considered in this first iteration are those (a_1, C_1) such that $a_1 \notin C_1$. Note that there are at most $d||S||$ of them. Now, in time $O(|H|d||S||)$ we can find $O(\log||S|| + \log d)$ hash functions that separate all these constraints. At this point, we introduce the second digit and the constraints take the form (a_1a_2, C_2), where $C_2 = (b_1b_2, c_1c_2, \cdots, f_1f_2)$. All constraints that were satisfied in the first iteration are dropped. Again, there are at most $d||S||$ constraints at hand. Over the h

iterations, this process yields $O(\frac{\log M}{\log d}(\log ||S|| + \log d))$ hash functions in time $O(|H|d||S||\frac{\log M}{\log d})$. Indyk chose H with $|H| = O(z \log M)$, and with each hash function having an $O(z \log M)$ range.

Our innovation is find a way to exploit the trade-off between the running time and the number of hash functions due to the parameter d. The basic idea is that after we already chose a few hash functions, the number of constraints decreases, which allows us to choose larger value of d without time penalty. More specifically, instead of eliminating all the constraints at each of the iterations, we only eliminate an appropriate fraction (all but $1/2d^2$). At this point, we can afford to square d (i.e. to double the number of bits in each digit). We then attack the remaining constraints using these longer digits. In effect, we are creating a 2-level iteration. Thus in the first of the outer iterations, we define the collections of constraints (a_1, C_1), (a_1a_2, C_2), $\cdots$ exactly as in the previous paragraph. For each of these h collections, we find $2\log d + 1$ hash functions, each of which eliminates at least half of the remaining constraints. Now, in the second of the outer iterations, we use a new coding $a = (a_1a_2)(a_3a_4)\cdots(a_{h-1}a_h) = a'_1a'_2\cdots a'_{h/2}$. We again form constraints (a'_1, C'_1), $(a'_1a'_2, C'_2)$, $\cdots$. For each digit, there are now at most $d^2||S||/2d^2 + d||S||/2d^2 \le (d||S||)/d$ constraints remaining. In this iteration, we seek to eliminate all but $1/2(d^2)^2$ of the constraints.

After $\log\log_d d||S||$ iterations all the constraints will have been eliminated. Perhaps before this, after $\log\log_d M$ iterations, $d = M$; at this point, in one final iteration, all the remaining constraints can be satisfied by means of a further $\log ||S||$ hash functions.

Again, we choose $|H| = O(z \log M)$ (actually, we choose $|H| = O(z^2 \log^2 M)$, but by searching for the hash function in two steps rather than by exhaustive search, we need test only $O(z \log M)$ hash functions). Further, we are able to reduce the range of the hash functions to $O(z)$. See Corollaries 5.1 and 5.2 for details.

This yields the size bound $O(z \log M(\min\{\log\log ||S||, \log\log M\} - \log\log d) + z \log ||S||)$ for the coding (Lemma 5.1) and a running time of $O(z \log M d||S||\frac{\log M}{\log d})$ (Lemma 5.2).

A further gain in efficiency arises for the constrained weight case, in which the number of sets of size at least k is at most $O(n/2^k)$. Our approach is simple: we compute codes separately for the sets of size r, $2^j \le r < 2^{j+1}$. For these sets, we choose $d = 2^{2^{j-1}}$. Note that $||S|| \le (2^{j+1}n/2^{2^j})$. It follows that the overall size bound is reduced to $O(\log n \log M + \log^2 n)$ and the running time becomes $O(n \log^2 M)$.

5.2 Formal description.

Let PRIME(r) denote the set of all prime numbers less than r. We will make use of the following family of hash functions. Define $\mathcal{H}_{\mathcal{FKS}}(u, r, v)$ to be the set of functions $h : [u] \to [v]$ of the form $h(x) = bx \ mod \ p \ mod \ v$, for all $p \in$ PRIME(r) and $b \in [p]$. In our application, $M = u$. This family has the following property (see [FKS84]):

Definition 4. *A family $\mathcal{H}$ of functions $h : X \to Y$ is called p-colliding if for any $x, y \in X$ such that $x \ne y$*

$$\Pr_{h \in \mathcal{H}}(h(x) = h(y)) \le p.$$

Fact 5.1. *There exists c such that for any u, z the family $\mathcal{H}_{\mathcal{FKS}}(u, cz \log u, cz)$ is $1/z$-colliding.*

Constraint satisfaction. We start by formulating the problem of constructing short superimposed codes as a constraint satisfaction problem (CSP). We say that a function $h : [u] \to [v]$ *satisfies* a pair (a, S) (where $S \subset [u]$ and $a \in [u] - S$) if $h(a) \notin h(S)$. Similarly, we say that a family $\mathcal{H}$ of hash functions satisfies a set C of pairs as above if every pair in C is satisfied by some function from $\mathcal{H}$. Moreover, we define the *set of constraints* generated by a given set S (denoted by $C(S)$) to be the set of all pairs (a, S) such that $a \in [u] - S$; for a family of sets $\mathcal{S}$ we define $C(\mathcal{S})$ to be the union of $C(S)$ for all $S \in \mathcal{S}$. The following claim shows the relation between the CSP and superimposed coding.

Claim 1. *Let $\mathcal{S}$ be a family of subsets of $[u]$ and let $h_i : [u] \to [v]$ for $i = 0 \ldots p-1$ be a family of hash functions satisfying $C(\mathcal{S})$. Let A be the set of all codewords $A[j]$, $j \in [u]$, of the form*

$$1_{h_0(j)} \ldots 1_{h_{p-1}(j)}.$$

Then A is an $\mathcal{S}$-superimposed code.

In order to solve CSP we will make use of the following basic property of hash functions.

Claim 2. *Let $\mathcal{S}$ be a family of z-subsets of $[u]$ and let $\mathcal{H}$ be a $1/2z$-colliding family of functions $h : [u] \to [v]$. Then there exists $h \in \mathcal{H}$ satisfying at least $1/2$ of the constraints in $C(\mathcal{S})$.*

For any $a \in [M]$, let $r^d(a)$ denote the sequence of numbers from $[d]$ which is the d-ary representation of a. Let $\lambda = \log_d M$ denote the length of this sequence. Also, for any such sequence s and $j \le \lambda$ let $s_{|j}$ denote the sequence comprising the first j elements of s. Both operators extend to sets in a usual way. For any constraint (a, S) we define its (i, d)-restriction $(a, S)^d_{|i}$ as $(r^d(a)_{|i}, r^d(S)_{|i})$. Finally, for any $S \in \mathcal{S}$ and $i \le \lambda$

we define the sets

$$C_i^d(S) = (a,S)^d_{|i} : a \in [M]-S, r^d(a)_{|i-1} \in r^d(S)_{|i-1}, r^d(a)_{|i} \notin r^d(S)_{|i}\}.$$

$C_i^d(\mathcal{S})$ is defined as the union of $C_i^d(S)$ for $S \in \mathcal{S}$. For any set $C \subset C_i^d(\mathcal{S})$ we define $||C||$ to be the sum of cardinalities of all sets S such that $(a,S) \in C$. We note that $|C_i^d(\mathcal{S})| \leq d||\mathcal{S}||$.

Claim 3. *Let $h_1 \dots h_\lambda$ be functions such that h_i satisfies a constraint $(a,S)^d_i$. Then the set*

$$\{h_1 \circ r^d_{|1}, h_2 \circ r^d_{|2} \dots h_\lambda \circ r^d_{|\lambda}\}$$

satisfies (a,S).

Thus it is sufficient to find hash function families which satisfy each of the sets $C_i^d(\mathcal{S})$ (which can be done fast as the sets are small). In order to present the algorithm, we need assume existence of a procedure HALFSAT(C), which returns a hash function satisfying at least half of the constraints in C. Clearly, such a procedure can be implemented in time $O(|\mathcal{H}||C|)$, where $\mathcal{H}$ is the set of hash functions to choose from. The reason we assume an abstract procedure is due to the fact that for the hash families we use this procedure can be implemented faster than this bound; we denote this faster time by $T_{HS}(s)$, where s is the number of constraints. We also use HALFSAT$^l(C)$ to denote a set of l functions satisfying all but $1/2^l$ constraints in C; clearly HALFSAT^l can be implemented in time $l \cdot T_{HS}$.

The procedure is as follows.

procedure *SCODE*$(\mathcal{S}, d)$
$i = 1$
$d_1 = d$
$H^1 = \emptyset$
$t_1 = \log_{d_1} M$; $l_1 = \log(2d_1^2)$
generate $C_j^{d_1} = C_j^{d_1}(\mathcal{S})$ for $j = 1 \dots t_1$; $C_1 = \cup_j C_j^{d_1}$
while $C_i \neq \emptyset$ and $d_i \leq M$
  compute $H_j^i = \text{HALFSAT}^{l_i}(C_j^{d_i})$ for $j = 1 \dots t_i$
  compute $H^{i+1} = H^i \cup \cup_{j=1\dots t_i}\{h \circ r^d_{|j} \mid h \in H^i_j\}$
  $d_{i+1} = d_i^2$
  find the sets $C_j^{d_{i+1}} = \{C_{2j}^{d_i}$ is not satisfied by $H^i_{2j}\} \cup$ $\{C_{2j-1}^{d_i}$ is not satisfied by $H^i_{2j-1}\}$, for $j = 1 \dots t_i$
  $C_{i+1} = \cup_j C_j^{d_{i+1}}$
  $t_{i+1} = \log_{d_{i+1}} M$; $l_{i+1} = \log(2d_{i+1}^2)$
  $i = i + 1$
endwhile
find $O(\log||\mathcal{S}||)$ functions satisfying C_i and add to H^i
return $H = H^i$

Below we show the bounds on the running time of the algorithm and length of the generated codes.

The code length. The following three claims follow directly from the description of the algorithm:

Claim 4. *The value of d_i is $d^{2^{i-1}}$.*

Claim 5. *At each step $|C_{i+1}| \leq |C_i|/d_i \leq d^2||\mathcal{S}||/d^{2^i}$.*

Claim 6. *The number of iterations performed by the procedure is at most $s = \min\{\log\log||\mathcal{S}||, \log\log M\} - \log\log d$.*

The length of the code is estimated in the following lemma.

Lemma 5.1. *The cardinality of the hash function set returned by the procedure is*

$$O(s \log M + \log||\mathcal{S}||)$$

Proof. At each of s steps we create $t_i \cdot l_i$ new hash functions. Therefore, the total number is bounded by

$$|H| = \sum_{i=1}^{s} t_i \cdot l_i = \sum_{i=1}^{s} \frac{\log M}{\log d_i}(2\log d_i + 1)$$

In the last step we add at most $\log||\mathcal{S}||$.

The running time. In order to estimate the complexity of the above procedure, we need to consider an implementation of the procedure *HALFSAT*. Consider a family $\mathcal{H} = \mathcal{H}_{\mathcal{FKS}}(u,r,v)$ where $r = cz\log u$ and $v = c'z$, where c and c' are sufficiently large constants. The following facts are well known (see for example [FKS84]).

Fact 5.2. *For any $\alpha > 1$ there exists c such that for any $x \in [u]$ the fraction of $p \in PRIME(r)$ which divide x is less than $1/\alpha z$.*

Corollary 5.1. *For c as above and any set of constraints C, there exists $p \in PRIME(r)$ such that the fraction of constraints $(a,S) \in C$ such that p divides $a - s$ for some $s \in S$ is at most $1/\alpha$. Such a prime can be found in $O(z\log u)||C||$ time.*

Fact 5.3. *Consider any $x, y \in [u]$ and $p \in PRIME(r)$ such that p does not divide $x - y$. Then*

$$\Pr_b[bx \bmod p \bmod v = by \bmod p \bmod v] \leq 1/\alpha z$$

for a sufficient c' depending on α.

COROLLARY 5.2. *For any set of constraints C there exists $p \in PRIME(r)$ and $b \in [p]$ such that the fraction of constraints $(a, S) \in C$ for which $ba \bmod p \bmod v = bs \bmod p \bmod v$ for some $s \in S$ is at most $2/\alpha$. Such a pair of numbers can be found in $O(z \log u)||C||$ time.*

LEMMA 5.2. *The procedure* SCODE *can be implemented to run in time $O(||\mathcal{S}||z \log M \frac{d}{\log d} \log M)$.*

Proof. Consider one iteration of the outer loop and consider the l_i successive applications of *HALFSAT*. Each successive application of *HALFSAT* works on a constraint set of at most half the size. Thus the overall running time of the l_i iterations of *HALFSAT* is proportional to the running time on the first iteration of *HALFSAT*. The constraint set has the following sizes in succesive iterations of the outer loop: $O(d||S||)$, $O(d||S||/d)$, $O(d||S||/d^2)$, $\cdots$. Thus the overall running time is $O(z \log M \frac{\log M}{\log d} \cdot d||S||)$.

After the loop is finished we know that the size of the constraint set is at most $||S||$, and so the running time of the last step lies within the above bound.

Special set families. As we mentioned before, we can obtain better results for set families $\mathcal{S}$ for which the corresponding matrix T has bounded weight (i.e. the value of $|w(|T|)|$ is upper bounded by (say) n). To this end, split $\mathcal{S}$ into disjoint sets $\mathcal{S}^0 \dots \mathcal{S}^\mu$, $\mu = \log\log|\mathcal{S}|$, such that $\mathcal{S}^j$ contains sets $S \in \mathcal{S}$ of cardinality in $[2^j, 2^{j+1})$. As $|w(|T|)| \le n$, we know that $|\mathcal{S}^j| \le \frac{n}{2^{2^j}}$. Define $d^{(j)} = 2^{2^{j-1}}$. For all $j = 0 \dots \mu$ we invoke the procedure *SCODE* with parameters $\mathcal{S}^j$ and $d^{(j)}$; note that the range of hash functions used in the jth step is $O(2^j)$, as all the sets in $\mathcal{S}$ have size upper bounded by 2^{j+1}. The union of all sets H_j obtained in this way forms the final output of the algorithm. We call this procedure *BOUNDED-SCODE*.

LEMMA 5.3. *The procedure runs in time $O(n \log^2 M)$.*

Proof. The jth invocation of *SCODE* takes time $O(2^j \log^2 M \frac{2^j \cdot n}{2^{2^j}} \cdot \frac{2^{2^{j-1}}}{2^{j-1}}) = O(n \log^2 M \frac{2^j}{2^{2^{j-1}}})$. Summing over j yields the claimed bound.

LEMMA 5.4. *The procedure generates code of length $O(\log M \log n + \log^2 n)$.*

Proof. The jth invocation of *SCODE* generates $O(\log M(\log\log(\frac{n \cdot 2^j}{2^{2^j}}) - \log\log 2^{2^{j-1}}) + \log(2^j \cdot n/2^{2^j}))$ hash functions, each of size $O(2^j)$. On summing over j, we obtain the claimed bound.

6 Interval codes

For this problem, we set $u = M$. We show that for a special family of sets containing all intervals of $[u]$ of length z, one can find codes which are more efficient than the ones obtained using the general procedure. More specifically, let $\mathcal{I}$ be a family of sets $[x \dots x+z-1]$ for all $x \in [u-z]$. We will construct $\mathcal{I}$-codes of length $O(z + \log u)$, which by elementary arguments can be easily seen to be optimal. This leads to a $O(n \log m(z + \log u))$-time deterministic algorithm for subset matching where all sets are intervals of length z. This problem has applications to searching in financial data (see [Ind97]).

The construction proceeds as follows. Let $\mathcal{F} = F_0 \dots F_{u-1}$ be a sequence of u binary vectors of length λ such that no vectors are contained in any other. Clearly, such a family exists and is easily constructible for $\lambda = O(\log u)$; for example, the set of characteristic vectors of all $p/2$-subsets of $[p]$ satisfies the inclusion requirement and has cardinality $\binom{p}{p/2}$ which is at least u for $p = 2\log u$. Moreover, define two functions $h_1, h_2 : [u] \to [2z]$ as $h_1(x) = x \bmod 2z$ and $h_2(x) = z + x \bmod 2z$. For $b \in \{1, 2\}$ the function h_b splits the domain $[u]$ into maximal intervals (call them $I_0^b \dots I_t^b$) such that h_b is increasing on each interval. For any x define $I^b(x)$ to be the (unique) index of the interval I_i^b containing x. The codeword $C(x)$ for $x \in [u]$ is then defined as

$$C(x) = 1_{h_1(x)} 1_{h_2(x)} F_{I^1(x)} F_{I^2(x)}.$$

The length of such a code is clearly as stated. Therefore it is sufficient to prove the correctness of such a code.

THEOREM 6.1. *C is an $\mathcal{I}$-code.*

Proof. We need to show that for any $x, y \in [u]$ we have $C(y) \subset C([x \dots x+z-1])$ iff $y \in [x \dots x+z-1]$. The "if" part is clear, thus we concentrate on proving the "only if" part. Let $I = [x \dots x+z-1]$. Observe first that for either $b = 1$ or $b = 2$ the interval I is completely contained in one of intervals I_i^b. As $y \notin I$, one of the following two cases hold:

- $j = I^b(y) \neq i$: in this case F_j is not contained in F_i so $C(x)$ is not contained in $C(I)$,
- $I^b(y) = i$ but $h_b(y) \notin h_b(I)$: in this case again $C(x)$ is not contained in $C(I)$.

References

[Abr87] K. Abrahamson, "Generalized string matching", *SIAM Journal of Computing* (1987), vol.16, no.6, p. 1039-1051.

[AHU74] A. Aho, J. Hopcroft, J. Ullman, Design and Analysis of Algorithms, Addison-Wesley, 1974.

[AS92] N. Alon, J. Spencer, *The probabilistic method*, Wiley, 1992.

[CH97] R. Cole, R. Hariharan, "Tree Pattern Matching and Subset Matching in Randomized $O(n \log^3 m)$ Time", *Proc. STOC'97*, pp. 66-75.

[DGM94] M. Dubiner, Z. Galil, E. Magen, "Faster tree pattern matching" *Journal of the ACM* (1994) vol.41, no.2, p. 205-213.

[DR83] A. G. Dyachkov, V. V. Rykov, "A Survey of Superimposed Code Theory", *Problems of Control and Information Theory* (1983), vol.12, no.4 (English translation).

[EFF85] P. Erdos, P. Frankl, Z. Furedi, "Families of finite sets in which no set is covered by the union of r others", *Israel J. Math.* (1985), vol.51, no.1-2, p. 79-89.

[FO95] C. Faloutsos, D. W. Oard, "A Survey of Information Retrieval and Filtering Methods", *Technical Report CS-TR-3514*, Dept. of Computer Science, Univ. of Maryland, August 1995.

[Fal92] C. Faloutsos, "Signature Files", in W. B. Frakes, R. Baeza-Yates, *Information Retrieval - Data Structures and Algorithms*, Prentice Hall, New Jersey, 1992.

[FB92] W. B. Frakes, R. Baeza-Yates, *Information Retrieval - Data Structures and Algorithms*, Prentice Hall, New Jersey, 1992.

[FKS84] M. L. Fredman, J. Komlos, E. Szemeredi, "Storing a Sparse Table with $O(1)$ Worst Case Access Time", *Journal of the ACM*(1984), vol. 31, no. 3, p. 538-544.

[FP74] M.J. Fisher, M.S. Paterson. String matching and other products. *Complexity of Computation*, SIAM-AMS proceedings, ed. R.M. Karp, 1974, pp. 113–125.

[HO82] C. M. Hoffman, M.J. O'Donnell, "Pattern Matching in Trees", *Journal of the ACM* (1982), p. 68-95.

[Ind97] P. Indyk, "Deterministic Superimposed Coding with Application to Pattern Matching", *Proc. FOCS'97*.

[IMV98] P. Indyk, R. Motwani, S. Venkatasubramanian, "Geometric Matching Under Noise: Combinatorial Bounds and Algorithms", these proceedings.

[Knu73] D. Knuth, *The Art of Computer Programming, vol. 3, Sorting and Searching*, Addison Wesley, 1973.

[Kos89] S. R. Kosaraju, "Efficient tree pattern matching", *Proc. FOCS'89*, p. 178-83.

[KS64] W. H. Kautz, R. C. Singleton, "Nonrandom Binary Superimposed Codes", *IEEE Transactions on Information Theory* (1964), vol. 10, p. 363-377.

[Lov75] L. Lovász, "On the ratio of optimal integral and fractional covers", *Discrete Mathematics* (1975), vol. 13, p. 383-390.

[Moo48] C. Mooers, "Application of Random Codes to the Gathering of Statistical Information", Bulletin 31, Zator Co., Cambridge, Mass. Based on M.S. thesis, MIT, January 1948.

[MP94] S. Muthukrishnan, K. Palem, "Non-standard Stringology: Algorithms and Complexity", in *Proc. STOC'94*, p. 770-779.

[MR95] S. Muthukrishan, H. Ramesh, "String Matching Under a General Matching Relation", *Information and Computation* (1995) vol. 122, no.1, p. 140-8.

[Mut95] S. Muthukrishnan, "New Results and Open Problems Related to Non-Standard Stringology", *Proc. 6th Annual Symposium on Combinatorial Pattern Matching* (1995), p. 298-317.