Fast Algorithms for Subset Matching and Tree Pattern Matching*

Richard Cole† Ramesh Hariharan‡ Piotr Indyk§

Abstract

This paper describes an $O(s \log^3 s)$ time deterministic algorithm, an $O(s \frac{\log^3 s}{\log \log s})$ time randomized Las Vegas algorithm, and an $O(s \log s)$ time randomized Monte Carlo algorithm for the *Subset Matching* problem. Here, s is the sum of the sizes of the text and pattern sets. A variant of this algorithm also solves the *Interval Matching* problem in $O(n(z + \log \sigma) \log \sigma)$ time. Here n is the text length, σ the alphabet size, and z the interval size. One of the key ingredients of our deterministic solution to the Subset Matching problem is the construction of efficient *Data-Dependent Superimposed Codes*.

In conjunction with a previous result reducing *Tree Pattern Matching* to Subset Matching in linear time, this implies an $O(n \log^3 m)$ time deterministic algorithm, an $O(n \frac{\log^3 m}{\log \log m})$ time randomized Las Vegas algorithm, and an $O(n \log n)$ time randomized Monte Carlo algorithm for the Tree Pattern Matching problem, where n and m are the text and pattern sizes, respectively.

1 Introduction

The *Subset Matching* problem is defined as follows. The input consists of a text t and a pattern p, of lengths n and m, respectively. Each text location and each pattern location is a set of characters drawn from an alphabet Σ of size σ. Let s denote the total sum of the sizes of all text and pattern sets, where each such set has size at least 1, without loss of generality. Thus $s \geq m + n$. The Subset Matching problem is to find all occurrences of the pattern in the text (see Fig.1). The pattern is said to *occur* or *match* at text position i if the set $p[j]$ is a subset of the set $t[i + j - 1]$, for all j, $1 \leq j \leq m$.

Our initial motivation for studying the Subset Matching problem was the *Tree Pattern Matching* problem. In this problem, the text and the pattern are ordered, node-labelled 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 with the same label. This is an important problem

*This paper reports work whose extended abstracts have appeared earlier in the Proceedings of the 29th ACM Symposium on Theory of Computing, 1997 [3], Proceedings of the 38th IEEE Symposium on Foundations of Computer Science, 1997 [12], and Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms, 1999 [4].

†Courant Institute, New York University, cole@cs.nyu.edu. This work was supported in part by NSF grants CCR9202900, CCR9503309, CCR9800085.

‡Indian Institute of Science, Bangalore, ramesh@csa.iisc.ernet.in. This work was done in part while visiting NYU. This work was supported in part by NSF grants CCR9202900, CCR9503309, CCR9800085.

§Massachusetts Institute of Technology, indyk@theory.lcs.mit.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.

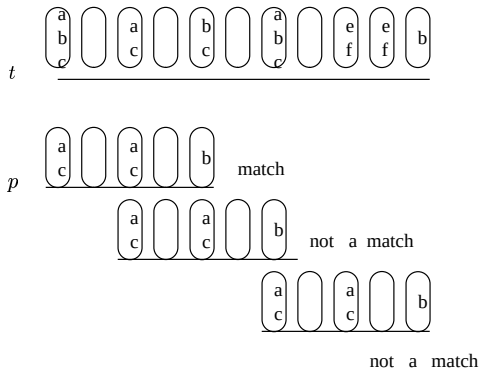

Figure 1: Example of Subset Matching.

which has a number of applications (see [11]). The best bound known for the Tree Pattern Matching problem prior to this paper was $O(nm^{.5}\log m)$, due to Dubiner, Galil and Magen [8] (also see [16],[11]). As shown in [3, 5], the Tree Pattern Matching problem can be reduced in linear time to the Subset Matching problem. If n and m are the text and pattern sizes for the Tree Pattern matching problem, this reduction produces a pattern of length $O(m)$ and a text of size $O(n)$ such that the sum of the text set sizes is $O(n)$, the sum of the pattern set sizes is $O(m)$, each text/pattern set has size $O(m)$, and the alphabet size is $|\Lambda| + O(m)$, where Λ is the labelling alphabet for the trees.

A special case of subset matching is *Interval Matching*, in which each set is an interval of integers. Here, we introduce the parameter z, which denotes the maximum set size. We can solve Interval Matching in time $O(n\log\sigma(z+\log\sigma))$. Interval Matching solves the *Bounded Difference Matching* problem. In the Bounded Difference problem the input consists of a pattern and a text, where each pattern or text entry is an integer. In a match, every pair of aligned characters differs in value by at most z, where z is an input parameter, also an integer. To reduce this to Interval Matching, the pattern entry x is replaced by the interval $[x]$, and the text entry y is replaced by the interval $[y-z, y+z]$. As observed by Indyk [12], Bounded Difference Matching can be used to find interesting patterns in time series data. This problem also arises in detecting melodic patterns in musical scores [7, 2].

Another measure capturing the notion of limited difference is the *Total Difference*; for two length m strings u and v this is defined to be $\sum_{i=1}^{m}|u(i)-v(i)|$. It is not clear whether this can be computed in subquadratic time for each alignment of a pattern with a text. However, the similar *Total Square Difference*, defined as $\sum_{i=1}^{m}(u(i)-v(i))^2$ is readily computed in $O(n\log m)$ time for all alignments of a length m pattern with a length n text (simply note that $\sum_{i=1}^{m}(u(i)-v(i))^2 = \sum_{i=1}^{m}u(i)^2 + \sum_{i=1}^{m}v(i)^2 - 2\sum_{i=1}^{m}u(i)v(i)$ [18]).

Subsequently, Indyk and others discovered additional applications for Subset Matching [14].

Our Result. Our main result in this paper is an $O(s\log^3 s)$ time deterministic algorithm, an $O(s\frac{\log^3 s}{\log\log s})$ time randomized Las Vegas algorithm, and an $O(s\log s)$ time randomized Monte Carlo algorithm, for the Subset Matching problem.

In conjunction with the above mentioned linear time reduction from Tree Pattern Matching to Subset Matching, this leads to an $O(n\log^3 m)$ time deterministic algorithm, an $O(n\frac{\log^3 m}{\log\log m})$ time randomized Las Vegas algorithm, and an $O(n\log n)$ time randomized Monte Carlo algo-

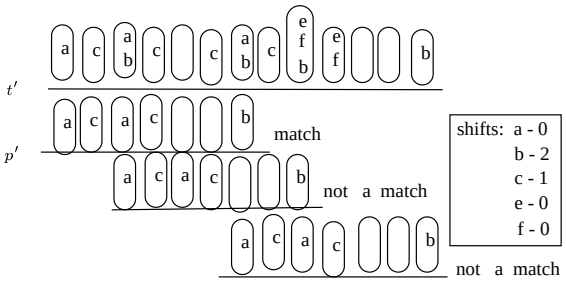

Figure 2: The set strings t', p'. Trailing empty sets in p' have not been shown.

rithm, for the Tree Pattern Matching problem. Note that the logarithmic terms for the deterministic and the Las Vegas randomized algorithms have m and not n, because each text/pattern set has $O(m)$ size and further, the length of the text can be restricted to $2m$ by the standard technique of chopping up a large text into smaller texts. This trick is applicable to the Monte Carlo algorithm as well, but does not reduce the $\log n$ factor to $\log m$ because the failure probability needs to be inverse polynomial in n and not m.

Our main result is obtained using the following key idea in conjunction with efficient solutions for two new problems which could be of independent interest as well.

The Key Idea: *Character Shifting.* Recall that each set in the text/pattern is drawn from an alphabet of size Σ. We create a new text t' and a new pattern p' of length s each (see Fig.2). For each character e in the above alphabet, a shift $shift(e)$ is chosen, where $shift(e)$ is an integer in $[0 \ldots 2s-1]$. t' and p' are created as follows: if e is in the set $t[i]$ then e is put in the set $t'[i + shift(e)]$; p' is built analogously from p. p' is said to match at location i in t' if $1 \leq i \leq |t| - |p| + 1$ and further, set $p'[j]$ is a subset of the set $t'[i+j-1]$, for all locations j in p'. It can easily be seen that the set of matches of p in t is identical to the set of matches of p' in t'. Further, note that $|t'|, |p'| = O(s)$.

Character shifting spreads out the characters more thinly. We use this fact in two different ways; the first leads to the Monte Carlo algorithm, while the second leads to the Las Vegas and the deterministic algorithms.

The Monte Carlo Algorithm. The Monte Carlo algorithm is simple and proceeds as follows. Consider any mismatch i of p in t, $1 \leq i \leq |t| - |p| + 1$. Let j be any location in p and e be a character in $p[j]$ such that $e \notin t[i+j-1]$ (such j, e exist as p mismatches at i). Then, a random choice of shifts for the characters ensures that location $t'[i + j - 1 + shift(e)]$ is empty with probability at least $1/2$ (while $p[j + shift(e)]$ contains e). Thus detecting mismatches reduces to detecting non-empty locations in the pattern which are aligned with empty locations in the text. Using a Monte Carlo algorithm for Boolean Wildcard Matching (defined in Section 2) due to Indyk [13], we can detect such locations in $O(s)$ time. To detect all mismatches, we need to repeat the above procedure $O(\log s)$ times, giving an $O(s \log s)$ time algorithm.

The Las Vegas and Deterministic Algorithms. The Las Vegas and deterministic algorithms are more elaborate and use the fact that character shifting reduces the maximum set size. The importance of reducing the maximum set size will become clear shortly. This leads to the first of the two problems mentioned above.

Set Size Minimization: The Row Shifting Problem. The aim of this problem is to

choose a set of good shifts for each character. We could use the following simple randomized Las Vegas algorithm which ensures that each set has size $O(\frac{\log s}{\log\log s})$ with inverse polynomial probability: each row independently chooses a random shift. As we will show, this is the best bound achievable on the set size. It is not clear how the above randomized algorithm can be derandomized within the time bounds we need. Our contribution here is a different deterministic algorithm which takes $O(s\log^3 s)$ time and ensures a slightly worse set size bound of $O(\log s)$.

Data-Dependent Superimposed Coding. Once the set sizes are small, our idea is to assign equal length binary codes to each character, so that the following property is satisfied: for each character e and each text set S such that $e \notin S$, one of the code bits of e is 1 and the corresponding code bit is 0 in the codes for all characters $f \in S$. Such codes are called *superimposed codes*. If we can find codes of length l satisfying the above property, then, as we will show, all occurrences of p' in t' can be found in $O(s \times l \times \log s)$ time, using standard convolution [10]. The motivation for reducing set sizes comes from the fact that l seems to depend on the maximum set size z.

Before describing this dependence, we explain the term *data-dependent*. This term is added to emphasize the fact that the above property need be satisfied only for the given sets S; this contrasts with the data-independent definition in which all subsets of size z must satisfy the above property. In fact, the data-independent version has been studied earlier. Dyachkov and Rykov [9] showed a lower bound of $\Omega(z^2\log_z \sigma)$ on the code length. They also showed a non-constructive upper bound of $O(z^2\log\sigma)$ on the code length. The best explicit construction [15] (based on Reed-Solomon codes) achieves $O(z^2\log_z^2\sigma)$. In order to the break the $\tilde{\Omega}(z^2)$ code length lower bound, we use the above "data dependent" definition.

Our contribution here is three-fold:

1. Given any arbitrary collection of sets of total size s drawn from an alphabet of size σ, we show how each character in the alphabet can be assigned an $O(z\log\sigma\min\{\log\log s, \log\log\sigma\}+ z\log s)$ length code in $O(sz\log^2\sigma(1+\log z/\log\log\sigma))$ time, deterministically.

2. Given a collection of sets of total size s drawn from an alphabet of size σ, we show how to obtain $O(\log^2 s)$ length codes in $O(s\log^2\sigma)$ deterministic time, provided the following additional property holds: the number of sets of of size at least i is $O(\frac{s}{2^i})$, for all i, $0 \le i \le \log s$. In addition, we show that the set sizes resulting from the Row Shifting procedure will indeed satisfy this property.

   Under a slightly different additional property, namely, the expected number of sets of size at least i is $O(\frac{s}{i^{\Theta(i)}})$, for all i, $0 \le i \le \frac{\log s}{\log\log s}$, we show how to obtain codes of expected length $O(\frac{\log^2 s}{\log\log s})$ in $O(s\log^2\sigma)$ expected time. Further, we show that the above mentioned randomized Las Vegas solution to the Row Shifting problem will indeed have this additional property.

3. Given a collection of intervals each of length at most z and of total length s drawn from an interval of size σ, we show how to obtain $O(z+\log\sigma)$ length codes in $O(s)$ time.

The claimed bounds on the Subset Matching problem now follow from our above mentioned results on the Row Shifting problem and the Data-Dependent Superimposed Coding problem. The Row Shifting algorithm takes $O(s\log^3 s)$ deterministic time and ensures that the parameter z going into the Superimposed Coding problem is $O(\log s)$. The Superimposed Coding algorithm then takes $O(s\log^2\sigma)$ deterministic time and produces codes of length $O(\log^2 s)$ (see (2) above).

Finally, the convolution process takes $O(s \times \log^2 s \times \log s)$ deterministic time. The overall time is thus $O(s \log^3 s)$, as claimed. The randomized Las Vegas variant of the Row Shifting algorithm takes $O(s)$ randomized time and ensures that the parameter z going into the Superimposed Coding problem is $O(\frac{\log s}{\log \log s})$, with inverse polynomial probability. The overall expected time then becomes $O(s \times \frac{\log^3 s}{\log \log s})$, as claimed [1].

Finally, for the Interval Matching problem the Superimposed Coding for intervals above leads to an overall $O(|t| \times (z + \log \sigma) \log(|p|(z + \log \sigma)))$ time algorithm, where $|t|, |p|$ are the text and pattern lengths, respectively.

Roadmap. Section 2 sets up some preliminaries. Section 3 gives a simple Monte Carlo randomized algorithm. Section 4 gives an outline of the Las Vegas randomized and deterministic algorithms for Subset Matching. The above outline will lead to the Row Shifting and Superimposed Coding problems. We give randomized and deterministic algorithms for the Row Shifting problem in Sections 5 and 6, respectively. Superimposed Coding is described in Section 7.

2 Preliminaries

Let t denote the given text, p the given pattern, and $\Sigma = \{1 \ldots \sigma\}$ the alphabet set. Each text/pattern location is a subset of this alphabet set. Let s denote the sum of the sizes of the text and pattern sets. Without loss of generality, we assume that each text/pattern set is non-empty; otherwise, we can add special characters to empty sets without affecting the outcome. An outcome of this assumption is that the time complexity of the algorithm becomes a function of s alone as s dominates the text/pattern lengths. We also assume $\sigma \leq s + 1$; if not, then the alphabet characters appearing in the sets can be renamed in the range $[0, s-1]$ by means of a sort, which takes $O(s \log s)$ time, and the value s is then used to represent all characters not appearing in any of the sets.

Definition of aligned. A pattern p is aligned with a text t at location i if $p[1]$ is aligned with $t[i]$.

Convolution. We will use the following standard lemma [10, 13] for *boolean wildcard matching.* In boolean wildcard matching the text, x, and pattern, y, are bit strings. In a match every 1 in the pattern must be aligned with a 1 in the text. There is no constraint on the text characters aligned with a 0 in the pattern. Thus for each position i in x at which pattern y does not wildcard match there exists an index j such that $y[j] = 1$ and $x[i + j - 1] = 0$.

Lemma 2.1 *Given two bit strings x, y, $|y| \leq |x|$, all locations i in x for which the aligned pattern y does not wildcard match can be determined in $O(|x| \log |y|)$ deterministic time and in $O(|x|)$ Monte Carlo randomized time*[2].

Definition of set codes. Suppose equal length binary codes are assigned to each element of Σ. Then, for $S \subseteq \Sigma$, the *set code* for S is defined as the bitwise-or of the codes assigned to the various elements of S.

Definition of separation. Given a set $S \subseteq \Sigma$ and an element $e \in \Sigma, e \notin S$, a code assigned to elements of Σ is said to *satisfy* (or *separate*) (S, e) if some bit in the code for e is 1 while the corresponding bit in the set code for S is 0.

[1] A simpler but weaker randomized Las Vegas algorithm for the Subset Matching problem which achieves $O(s \times \log^3 s)$ time with inverse polynomial probability appears in the extended abstract [3].

[2] This algorithm detects a mismatch with constant probability and a match with surety.

3 The Monte Carlo Subset Matching Algorithm

We perform character shifting to obtain t', p' from t, p, respectively, as described in the introduction. For each character $e \in \Sigma$, $shift(e)$ is chosen uniformly at random from the range $0 \ldots 2s-1$. The shifts for different characters are chosen independently (in fact, pairwise independence is all we need, though we do not use this fact). Recall from the introduction that p' matches t' at location i if and only if p matches t at location i. The following additional property is easily seen; it arises from choosing shifts randomly and enables efficient detection of mismatches.

Property. Consider any mismatch i of p in t, $1 \leq i \leq |t| - |p| + 1$. Let j be any location in p and e be a character in $p[j]$ such that $e \notin t[i+j-1]$ (such j, e exist because p mismatches at i). Then, location $t'[i+j-1+shift(e)]$ is empty with probability at least $1/2$ (while $p[j+shift(e)]$ contains e).

Next, we convert t' to a new text t'' by replacing each non-empty set by 1 and each empty set by 0. p'' is obtained from p' analogously. From the above property, it follows that if p mismatches t at location i (i.e., there exist j, e such that $e \in p[j], e \notin t[i+j-1]$) then, with probability at least $1/2$, p'' will mismatch t'' in the boolean wildcard sense at location i (i.e., the 1 in p'' at location $j + shift(e)$ is aligned with a 0 in t'' at location $i + j - 1 + shift(e)$). Further, it is easily seen that if p matches t at location i, then p'' will match t'' at location i in the boolean wildcard sense.

Next, using Lemma 2.1, we can solve the boolean wildcard matching problem on t'', p'' in $O(s)$ time. The resulting set of reported matches will include all real matches of p in t; in addition, any particular mismatch of p in t will be reported as a match with constant probability $\alpha \leq \frac{1}{2}$. Repeating this procedure $O(\log s)$ times and taking the intersection of the set of matches reported in the various trials ensures that all $O(s)$ mismatches are detected, with inverse polynomial failure probability. The total time taken is $O(s \log s)$.

4 The Las Vegas/Deterministic Subset Matching Algorithm Outline

Our effort will be centered around the following claim.

Lemma 4.1 *Suppose each element of Σ is given a binary code of length l with the following property: for each text set S and each element $e \in \Sigma, e \notin S$, the above code separates (S, e). Then the Subset Matching problem can be solved in $O(sl \log(sl))$ time.*

Proof. Obtain a string x from t by replacing each set by it associated code (defined above). Similarly, obtain string y from p. Note that $|x|, |y| \leq sl$. Further, by virtue of the separation property, note that p matches t starting at location i, $1 \leq i \leq |t| - |p| + 1$, if and only if y matches x at i. The lemma follows from Lemma 2.1. □

The next aim is to minimize l. We will describe a method to obtain short codes which will satisfy all the separation constraints; however, the length of these codes will be proportional to the size of the largest text set. The size of the largest text set can be decreased using the following key transformation.

Character Shifting. We start by applying character shifting with "good" shifts to obtain a new pattern p' and a new text t' in which all the sets have size $O(\log s)$. p' and t' are formed from p and t, respectively, as described in Section 1.

Determining Good Shifts. Our aim is to choose a set of shifts which will reduce the maximum text set size to $O(\log s)$ (deterministically) and $O(\frac{\log s}{\log\log s})$ (randomized, with inverse polynomial probability). The time taken will be $O(s\log^3\sigma)$ and $O(s)$, respectively.

We formulate the problem of determining good shifts as follows. Consider a $\sigma \times s$ boolean matrix A such that $A[e,j]$ is 1 if and only if element $e \in \Sigma$ occurs in text set $t[j]$. Clearly, A has exactly s ones. This matrix is not specified explicitly; rather, only a list of the positions having a one is given. The aim is to rotate, i.e., circularly shift, the rows of A (different rows may be shifted by different amounts) so that the maximum number of ones in any column becomes "small". We call this the *Row Rotation* problem and describe our solutions to this problem in Sections 5 and 6, respectively. Note that the above description involves circular shifts in the range $0 \ldots s-1$, while the introduction talked of linear shifts in the range $0 \ldots 2s-1$. It is easily seen that any solution to the Row Rotation problem can be unfolded into a solution for the Row Shifting problem in such a way that each set size only reduces.

Superimposed Coding. Having obtained a good set of shifts, we proceed to the coding. This is described in Section 7. Starting with the t', p' obtained from the deterministic Row Rotation algorithm, this section obtains a set of codes of length $l = O(\log^2 s)$ in $O(s\log^2\sigma)$ time. From Lemma 4.1, an $O(s\log^3 s)$ time deterministic algorithm for Subset Matching follows (note that while the lemma is stated for t, p, it is actually applied on t', p'; since $|t'|, |p'| = O(s)$, the time bounds in the lemma continue to hold).

In addition, starting with the t', p' obtained from the randomized Row Rotation algorithm, Section 7 also obtains a set of codes of expected length $l = O(\frac{\log^2 s}{\log\log s})$ in $O(s\log^2\sigma)$ expected time. From Lemma 4.1, an $O(s\frac{\log^3 s}{\log\log s})$ expected time randomized algorithm for Subset Matching follows.

5 The Row Rotation Problem: A Randomized Algorithm

We sketch a randomized $O(s)$ time algorithm which determines row shifts such that each column has $O(\frac{\log s}{\log\log s})$ ones, with inverse polynomial probability.

The algorithm is the obvious one; each row is given a random circular shift between 1 and s. Consider any particular column. It is easily seen that the expected number of ones in this column is $O(1)$; further, using the Chernoff bound [6], it can be shown that the number of ones in this column is $O(\frac{\log s}{\log\log s})$, with inverse polynomial probability. Finally, using the union bound, the number of ones in each column is $O(\frac{\log s}{\log\log s})$, with inverse polynomial probability.

Further, using the Chernoff bound [6], we get that the expected number of columns having more than i ones is $O(\frac{s}{i^{\Theta(i)}})$, $i = O(\frac{\log s}{\log\log s})$. This observation will be needed in the superimposed coding phase, to obtain efficient codes. This yields:

Lemma 5.1 *The above randomized algorithm determines a circular shift for each row of A in $O(s)$ time with the property that every column has $O(\frac{\log s}{\log\log s})$ ones. Further, the expected number of columns having at least i ones is $O(\frac{s}{i^{\Theta(i)}})$, $i = O(\frac{\log s}{\log\log s})$.*

5.1 A Lower Bound of $\Omega(\frac{\log s}{\log \log s})$

We give an example array A having $r = \Theta(\frac{\log s}{\log \log s})$ rows, s columns, and $O(s)$ ones, such that any collection of shifts applied to the rows will produce some column with $\Omega(\frac{\log s}{\log \log s})$ ones. Thus, the bound given by the randomized algorithm is tight.

The above claim will hold for all values of s satisfying the following property: s is the product of r primes $p_1 \ldots p_r$, where each $p_i \geq \frac{\log s}{\log \log s}$. There exist infinitely many s with this property.

The array A is set up as follows. Each row corresponds to one of the prime factors p_i of s; the ones in this row are distributed at equal intervals of p_i each, starting with a one in the first column.

Next, consider any arbitrary collection of rows shifts, $0 \leq s_1, \ldots, s_r \leq s-1$. We claim that some column x will have all entries equal to one. For such a column x to exist, x must simultaneously satisfy the following conditions:

$$\forall i \in 1 \ldots r,\ x \equiv s_i (\text{mod } p_i)$$

The existence of a unique x satisfying all these congruences is guaranteed by the Chinese Remainder Theorem [17], since $p_1 p_2 \ldots p_r = s$. Finally, we note that the number of ones in all the rows together is given by $\Theta(\sum_i s/p_i) = \Theta(s)$.

6 The Row Rotation Problem: A Deterministic Algorithm

In this section, we give a deterministic $O(s \log^3 s)$ time algorithm which determines row shifts such that each column has $O(\log s)$ ones. We do not know how to obtain the sharper bound given by the randomized algorithm within this time bound.

Our algorithm has the following overall framework. At each step, the rows of A are partitioned in *megarows*. Initially, each row is a megarow. The general step considers two megarows and determines a "good" relative shift of one entire megarow with respect to the other. This shift is applied to the rows of the second megarow and then the two sets of rows are placed in a single combined megarow. Note that no shifting happens within either of the two megarows in this step; relative shifts within each megarow have been determined and frozen already. The procedure ends when all rows come together into a single megarow. Two issues needs further description: which two megarows are chosen at each instant and how a good relative shift between megarows is determined.

6.1 Shifting Megarows

First, we describe how a good relative shift between two megarows is determined. We will then bound the time complexity of this procedure and also quantify the goodness of this shift. Our algorithm needs the following definitions.

Definition of Vector Sum and Product. Given vectors v, w of length s, $v + w$ is defined as a vector u such that $u[i] = v[i] + w[i]$, $1 \leq i \leq s$. $v \times w$ is defined analogously. $v.w$ is the usual dot-product and is defined as $\sum_{i=1}^{s} (v \times w)[i]$.

Megarows as Vectors. Recall that we seek to have $O(s/2^i)$ columns with i or more ones. This motivates us to form a vector v of weights for each megarow as follows: suppose there are x ones in the ith position in the shifted rows forming the megarow; then the weight stored in $v[i]$ is 2^x if $x > 0$, and 0 otherwise. The weight, $wt(v)$, of the megarow corresponding to v is

$\sum_i v[i]$. Vector v is called the *weight megarow vector* or *weight vector* for short. Our goal is to limit the weight of the final megarow to $O(s)$. To achieve this, we will limit the weight of the megarows at hand at each stage of the algorithm to $O(s)$ also.

Let v, w, respectively, be the weight vectors for two megarows being combined. Our aim is to find a good circular shift of w relative to v. The same shift will be used for the corresponding megarows.

One technical issue before we proceed is that of determining and representing a weight megarow vector. We represent such a vector as a list of non-zero entries. It is easy to see that this representation can be computed in time proportional to the weight of the corresponding megarow.

The Key Primitive. Consider a particular shift w' of w. Suppose we apply this shift to the megarow corresponding to w and then combine the megarows corresponding to v and w into one megarow. The vector u corresponding to this new megarow will have the following properties:

1. $u[i] = v[i] \times w'[i]$, for i such that both $v[i]$ and $w'[i]$ are non-zero,
2. $u[i] = v[i]$, for i such that $w'[i] = 0$.
3. $u[i] = w'[i]$, for i such that $v[i] = 0$.

We would like to choose the shift of w defining w' so as to minimize the weight of u. A natural approach is to use convolutions. To this end, let $\tilde{v}$ be the vector v with all entries of 0 replaced by $1 = 2^0$, and all other entries unchanged. Let $\bar{v}$ be the vector v with all non-zero entries replaced by 0 and all zero entries replaced by 1. $\tilde{w}'$ and $\bar{w}'$ are defined analogously. Then the weight of u is given by $\tilde{v} \cdot \tilde{w}' - \bar{v} \cdot \bar{w}'$. This quantity can be calculated for all possible shifts w' of w in $O(s \log s)$ time using two convolutions (or polynomial multiplications, see [1]).

The problem with the algorithm sketched in the previous paragraph is that it is too expensive, time-wise. We will be able to afford only $O(f(\max\{wt(v), wt(w)\}) \times \log s)$ time and not $O(s \log s)$ time (note that $\max\{wt(v), wt(w)\}$ could be much smaller than s), where $f()$ is a function to be described later with the property that $f(x) \leq s$ for all x.

Our Algorithm: Sparse Convolutions. The key idea is to shrink v, w down to two smaller vectors v_1, w_1, respectively, of size $f(\max\{wt(v), wt(w)\})$ each, and then perform the above procedure of finding the shift which gives the least dot-product on these smaller vectors. These smaller vectors will be represented explicitly, and not as a list of non-zero entries.

v_1 is defined as follows. Partition v into disjoint blocks of size $\Theta(s/f(\max\{wt(v), wt(w)\}))$ each. v_1 has one entry for each such block; the value of this entry is the sum of the entries in this block. w_1 is defined analogously. The time taken to determine v_1 and w_1 from v, w is easily seen to be $O(wt(v) + wt(w))$.

We now find a shift w_1' of w_1 which minimizes the dot-product $v_1.w_1'$. This shift corresponds to a shift of w relative to v in the obvious way (i.e., by multiplying the shift obtained by the block length). The time taken for this procedure is $O(f(\max\{wt(v), wt(w)\}) \log s)$.

The total time is $O(wt(v) + wt(w) + f(\max\{wt(v), wt(w)\}) \log s)$.

Increase in Weight. Recall that u is the weight vector corresponding to the megarow obtained by shifting and combining the two megarows in question. A key factor in the analysis involves showing that $wt(u)$ is not too much more than $wt(v) + wt(w)$. We show:

Lemma 6.1 $wt(v) + wt(w) \leq wt(u) \leq (wt(v) + wt(w))(1 + \frac{wt(v) \times wt(w)}{(wt(v)+wt(w))f(\max\{wt(v),wt(w)\})})$.

Proof. The first inequality is evident from the three properties above which define u. We prove the second inequality.

The sum of $v_1.w_1'$ over all shifts w_1' of w_1 is exactly $wt(v) \times wt(w)$. Since v_1, w_1 have size $f(\max\{wt(v), wt(w)\})$ each, it follows from an averaging argument that the w_1' resulting from one of these $f(\max\{wt(v), wt(w)\})$ shifts is such that $v_1.w_1'$ is at most $\frac{wt(v)\times wt(w)}{f(\max\{wt(v),wt(w)\})}$. Going from v_1, w_1' to v, w' (w' is the corresponding shift for w), it is easily seen that $v.w'$ is also at most $\frac{wt(v)\times wt(w)}{f(\max\{wt(v),wt(w)\})}$. ¿From the three properties above which define u, it follows that $wt(u)$ is at most

$$wt(v) + wt(w) + v.w' \leq wt(v) + wt(w) + \frac{wt(v) \times wt(w)}{f(\max\{wt(v), wt(w)\})}.$$

The lemma follows. □

6.2 Pairing Megarows and Analysis

The order is which megarows are paired depends on the weights of the associated vectors. Note that at the very beginning, the sum of the megarow weights is $2s$ (because there are only s ones in A). We will show that a bound of $O(s)$ on the sum of the weights of the current magarows holds not just at the beginning, but throughout the algorithm, due to Lemma 6.1 and the choice of the function $f()$. In particular, it will hold at the end, implying that the final megarow (which holds all the rows of A) can have at most $O(\log s)$ ones in any column.

We classify the megarow-vector weights into categories $[2^i, 2^{i+1})$, $0 < i = O(\log s)$. The pairings are now performed in phases, with several pairings being performed in each phase. Consider a particular phase and consider the current lowest non-empty category, $[2^i, 2^{i+1})$, say. If this category has at least two megarows then we pair the megarows in this category (leaving out one megarow, possibly) and combine the megarows in each pairing within this phase. The unpaired megarow, if any, is put on hold. If there is already another megarow on hold, necessarily from a lower index category, then the two megarows on hold are combined. ¿From Lemma 6.1, it follows that the new megarow which results from combining two paired megarows in the same category will be in a strictly higher category. It follows that the number of phases will be $\Theta(\log s)$.

Defining $f()$. Before proceeding with the analysis, we define the function f: $f(x) = \min\{\lceil x \log s \rceil, s\}$.

Bounding Megarow-vector Weights. Next, we show that the sum of the megarow weights at the end of each phase is $O(s)$. It immediately follows that the total time taken within a phase is $O(s \log^2 s)$ (one $\log s$ factor comes from the definition of $f()$ and another from the overhead for sparse convolution), which gives $O(s \log^3 s)$ time overall. Further, as stated earlier, this also implies that the final megarow obtained has $O(\log s)$ ones in each column.

The megarow-vector weights are bounded by the following lemma.

Lemma 6.2 *Consider a phase in which all but at most one of the megarows which are combined belong to category $[2^j, 2^{j+1})$. Let O and N denote the sum of the megarow-vector weights at the beginning and the end of this phase, respectively. Then $\frac{N}{O} \leq (1 + \max\{\frac{1}{\log s}, \frac{2^{j+1}}{s}\})$.*

Proof. The proof uses Lemma 6.1 and the definition of $f()$.

Consider two megarows that are paired in this phase. Let the corresponding vectors be v, w, respectively. Without loss of generality, let $wt(v) < wt(w)$. Note that w must be in category $[2^j, 2^{j+1})$; v could be in a smaller category. Then, by Lemma 6.1 and the definition of $f()$:

$$
\begin{aligned}
wt(u) &\leq (wt(v)+wt(w))\left(1+\frac{wt(v)\times wt(w)}{(wt(v)+wt(w))f(\max\{wt(v),wt(w)\})}\right)\\
&\leq (wt(v)+wt(w))\left(1+\frac{wt(w)}{f(wt(w))}\right)\\
&\leq (wt(v)+wt(w))\left(1+\max\{\frac{1}{\log s},\frac{wt(w)}{s}\}\right)\\
&\leq (wt(v)+wt(w))\left(1+\max\{\frac{1}{\log s},\frac{2^{j+1}}{s}\}\right)
\end{aligned}
$$

The lemma now follows by applying the above to all megarow pairings in this phase. □

Corollary 6.3 *The sum of the megarow-vector weights at the end of each phase is $O(s)$.*

Proof. The sum of the megarow-vector weights at the very beginning is $2s$, as there are only s ones in A. We denote this sum by I. Next, consider a phase in which all but at most one of the megarows which are combined belong to category $[2^j, 2^{j+1})$. Then, by Lemma 6.2, the sum of the megarow-vector weights at the end of this phase is at most I times the following quantity:

$$\Pi_{i=1}^{j+1}(1+\max\{\frac{1}{\log s},\frac{2^i}{s}\})$$

If $2^j \leq s$, then the above quantity is easily seen to be $O(1)$. Otherwise, if $2^j > s$, consider the most recent previous phase in which the megarows which are paired belong to category $[2^l, 2^{l+1})$, $2^l \leq s$. In the above product, the terms up to and including this phase multiply to $O(1)$. Therefore, beyond this phase, at most $O(1)$ megarows survive (because the total sum of megarow weights is $O(s)$ and all but possibly one megarow has weight at least s). Terms corresponding to the subsequent $O(1)$ phases then contribute a further $O(1)$ factor. Thus, the above quantity is $O(1)$; the corollary follows. □

This leads to the following result, the second part of which also follows immediately from the fact the final megarow-vector weight is $O(s)$.

Lemma 6.4 *The above deterministic algorithm determines a circular shift for each row of A in $O(s\log^3 s)$ time with the property that every column has $O(\log s)$ ones. Further, the number of columns having having more than i ones is $O(\frac{s}{2^i})$, $i = O(\log s)$.*

7 Constructing Superimposed Codes

Recall that $\Sigma = \{1 \ldots \sigma\}$. The aim of this section is to assign each element $e \in \Sigma$ an equal length code so that for each text set S and each element $e \in \Sigma, e \notin S$, (S, e) is separated (or satisfied, as defined in Section 2). We refer to the (S, e) pairs as *constraints* to be satisfied. Recall from Sections 1 and 2 that z denotes the maximum text set size, s is the sum of the text and pattern set sizes, and that $s \geq \sigma + 1$.

The starting point of our algorithm is the following procedure, whose details will appear later.

Procedure *Encode*$(\mathcal{C}, k)$. Let k be any integer greater than 0 and let $\mathcal{C}$ be any given sub-collection of constraints. Let $|\mathcal{C}|$ denote the quantity $\sum_{(S,e)\in\mathcal{C}} |S|$; we call this quantity the *size* of $\mathcal{C}$. This procedure assigns codes to the elements involved in these constraints[3] in such a way that the total size of constraints not satisfied by this code is less than $\frac{|\mathcal{C}|}{2^k}$. These codes have length $O(zk)$ and the procedure takes $O(|\mathcal{C}|z\log\sigma(1+\log z/\log\log\sigma))$ time.

The problem with applying the above procedure directly to the set of all constraints is that the constraint size could be large, i.e., $\Omega(s\sigma)$. Thus to obtain our time bound of $O(sz\log^2\sigma)$, we need to apply procedure *Encode*() on smaller, carefully chosen subcollections of constraints.

Let d be any parameter between 2 and σ. First, we show how to choose these sub-collections to obtain codes of length $O(z\log(sd)\log_d\sigma)$. Subsequently, we will modify this procedure to obtain codes of length $O(z\log\sigma\log\log_d\sigma + z\log(sd))$. Setting d to a constant in this expression leads to codes of length $O(z\log\sigma\log\log s + z\log s)$. Then, we will show how the code length can be reduced further under additional conditions.

7.1 Getting Codes of length $O(z\log(sd)\log\sigma)$

We now define $\log_d\sigma$ sub-collections of constraints and invoke procedure *Encode*() on each such sub-collection. We need the following definition.

Definitions. Consider the d-ary representations of the numbers $1\ldots\sigma$. For $e\in\Sigma$, define $e^*_{(j,d)} = e \bmod d^j$, i.e., $e^*_{(j,d)}$ is the number obtained by taking just the least significant j digits in the d-ary representation of e. Similarly, define $S^*_{(j,d)}$ for a set $S\subseteq\Sigma$ as $\{e^*_{(j,d)}|e\in S\}$.

Subcollection $\mathcal{C}_j$. The jth sub-collection, $0\le j<\log_d\sigma$ has constraints of the form $(e^*_{(j,d)}, S^*_{(j,d)})$, for all (S,e) such that $e^*_{(j,d)}\notin S^*_{(j,d)}$, but $e^*_{(j-1,d)}\in S^*_{(j-1,d)}$.

It is now easily seen that, for each j, $0\le j<\log_d\sigma$, the total size of constraints in $\mathcal{C}_j$ involving S is $|S|d$. Therefore, the total size of constraints in $\mathcal{C}_j$ over all sets S is at most sd. Further, for each j, these constraints can be explicitly determined in $O(sd)$ time. Using *Encode*$(\mathcal{C}_j, \log(sd))$, in time $O(sdz\log\sigma(1+\log/\log\log\sigma))$ time, we obtain codes of length $O(z\log(sd))$, satisfying *all* constraints in $\mathcal{C}_j$ (note that the second argument to *Encode*() has value $\log sd$ here). This procedure is performed for each j, $0\le j<\log_d\sigma$. The total time taken is thus $O(sdz\log\sigma\log_d\sigma(1+\log/\log\log\sigma))$.

Putting the codes together. The following lemma now shows how the final code for e can be obtained from the codes for $e^*_{(1,d)}, e^*_{(2,d)}, \ldots, e^*_{(\log_d\sigma,d)}$ obtained above in successive calls to *Encode*(), so that all the original constraints are satisfied.

Lemma 7.1 *The following coding for the elements of Σ satisfies all the original constraints: for each $e\in\Sigma$, code(e) is the code obtained by concatenating the codes for $e^*_{(1,d)}, e^*_{(2,d)}, \ldots, e^*_{(\log_d\sigma,d)}$.*

Proof. Consider any constraint (S,e), $e\notin S$. We show that this constraint is satisfied by the above coding.

Since $e\notin S$, there must exist a j, $0\le j<\log_d\sigma$ such that $e^*_{(j-1,d)}\in S^*_{(j-1,d)}$ but $e^*_{(j,d)}\notin S^*_{(j,d)}$. Then *Encode*$(\mathcal{C}_j, log(sd))$ returns codes satisfying $(e^*_{(j,d)}, S^*_{(j,d)})$. Therefore, it must be the case that the code for $e^*_{(j,d)}$ has a 1 at some position while the codes for $f^*_{(j,d)}$, $f\in S$, have 0's at this position. The lemma follows. □.

[3] An element is involved in constraint (S,e) if it either belongs to S or is the same as e.

Analysis. The total time taken is $O(sdz \log \sigma \log_d \sigma (1 + \log z / \log\log \sigma))$, as stated above. The total code length is $O(z \log(sd) \log_d \sigma)$. Setting d to a constant, we obtain code length $O(z \log s \log \sigma)$ and running time $O(sz \log^2 \sigma (1 + \log z / \log\log \sigma))$.

7.2 Getting Codes of length $O(z(\log \sigma \log\log_d \sigma + \log(sd)))$.

Recall that the parameter d can be any number between 2 and σ. Varying this parameter will be instrumental in achieving the reduction in code length described in this section.

The main idea is to call $Encode(\mathcal{C}_j, 2\log d + 1)$ instead of $Encode(\mathcal{C}_j, \log(sd))$, for each j. Consequently, the codes returned by this procedure will be smaller, i.e., of length $O(z \log d)$ instead of $O(z \log(sd))$; however, they could leave constraints of total size up to $\frac{|\mathcal{C}_j|}{2d^2}$ unsatisfied, for each j. At this point, to satisfy these remaining constraints, the parameter d is replaced by d^2 and the remaining unsatisfied constraints with respect to this new parameter are determined, as described below.

Definitions. The previous section defined the sub-collections $\mathcal{C}_j$, assuming a fixed value of d. We introduce new notation to parameterize these sub-collections by d, i.e., we replace $\mathcal{C}_j$ by $\mathcal{C}_j^d$, $0 \leq j < \log_d \sigma$.

Overview of Algorithm Phases. The algorithm now proceeds in phases. The parameter d is squared in each successive phase (i.e., in the ith phase, the d^{2^i}-ary representations for elements of Σ are considered, where i varies from 0 to $\log\log_d \sigma$). For brevity, we shall denote d^{2^i} by d_i.

In the ith phase, we call $Encode()$ on sub-collections of constraints (to be described shortly) which are subsets, respectively, of the collections $\mathcal{C}_1^{d_i} \ldots \mathcal{C}_{\log_{d_i} \sigma}^{d_i}$. These calls result in a code being assigned to each of $e^*_{(1,d_i)}, e^*_{(2,d_i)}, \ldots, e^*_{(\log_{d_i} \sigma, d_i)}$, for each $e \in \Sigma$. The second parameter in all these calls is set to $2\log d_i + 1$; therefore, each call returns with less than a $\frac{1}{2d_i^2}$ fraction (by size) of the constraints unsatisfied.

The total size of constraints involved in each call to $Encode()$ in phase 0 is $O(sd)$, as before. As we will show later, the total size of constraints involved in each call to $Encode()$ in phase $i > 0$ will be at most $\frac{sd}{d^0 d^1 d^2 d^4 d^8 \ldots d^{2^i - 1}}$. So, at the beginning of the last phase, this quantity will be less than

$$\frac{sd}{d^1 d^2 d^4 d^{2^{\log\log_d \sigma - 1}}} = \frac{sd^2}{\sigma} \leq sd.$$

A call to $Encode()$ with these constraints and the second parameter set to $\log(sd)$ will now satisfy these constraints as well. So the total number of rounds is $\log\log_d \sigma$.

The final code for each $e \in \Sigma$ will be the concatenation of the codes obtained above for

$$e^*_{(1,d)} \ldots e^*_{(\log_d \sigma, d)}, e^*_{(1,d^2)} \ldots e^*_{(\log_{d^2} \sigma, d^2)}, \cdots, e^*_{(1, d^{2^{\log\log_d \sigma}})} \ldots e^*_{(\log_{(d^{2^{\log\log_d \sigma}})} \sigma, d^{2^{\log\log_d \sigma}})}$$

.

Time Complexity and Code Length. ¿From the above, it follows that the total time taken for one call to $Encode()$ within phase i, $0 \leq i < \log\log_d \sigma$, is

$$O(\frac{sd}{d^0 d^1 d^2 d^4 \ldots d^{2^i - 1}} \times z \log \sigma (1 + \log z / \log\log \sigma)) = O(\frac{sdz \log \sigma (1 + \log z / \log\log \sigma)}{d^{2^i - 1}}).$$

Summing over all $\log_{d^{2^i}} \sigma$ calls within this phase gives

$$O(\frac{sdz \log^2 \sigma (1 + \log z / \log\log \sigma)}{d^{2^i - 1} 2^i \log d}).$$

Summed over all phases, $i < \log\log_d \sigma$, this gives $O(sz \log^2 \sigma(1 + \log z/\log\log\sigma)\frac{d}{\log d})$ time.

The final phase, with $i = \log\log_d \sigma$, takes time $O(sdz \log\sigma(1+\log z/\log\log\sigma)) = O(sz\log^2\sigma(1+\log z/\log\log\sigma)\frac{d}{\log d})$ as $d = O(\sigma)$. Thus the time for all the phases is also $O(sz\log^2\sigma(1+\log z/\log\log\sigma)\frac{d}{\log d})$.

We do a similar analysis for code lengths. Each call to *Encode*() in round $i, 0 \leq i < \log\log_d\sigma$, returns codes of length $O(z\log d^{2^i}) = O(z2^i \log d)$ (recall that the parameter k in these calls is set to $2\log d^{2^i} + 1$); summing this over all $\log_{d^{2^i}} \sigma$ calls within a phase gives

$$O(\frac{z2^i \log d \log\sigma}{2^i \log d}) = O(z\log\sigma).$$

Summing over all phases $i, 0 \leq i < \log\log_d\sigma$, gives $O(z\log\sigma\log\log_d\sigma)$. The codes obtained in the $\log\log_d\sigma$th phase (recall, the second parameter to *Encode*() is set to $\log(sd)$ for this final phase) sum up to $O(z\log(sd))$. Therefore, the total code length is

$$O(z(\log\sigma\log\log_d\sigma + \log(sd))).$$

The following issues remain to be addressed. Consider phase i and recall that d_i denotes d^{2^i} for brevity. So $d^{2^{i+1}} = {d_i}^2$. Let $\mathcal{A}_1 \ldots \mathcal{A}_{\log_{d_i}\sigma}$ be sub-collections of constraints which are subsets of $\mathcal{C}_1^{d_i} \ldots \mathcal{C}_{\log_{d_i}\sigma}^{d_i}$, respectively, and on which calls to *Encode*() are made in this phase. Let $\mathcal{B}_1 \ldots \mathcal{B}_{\log_{d_i^2}\sigma}$ be sub-collections of constraints which are subsets of $\mathcal{C}_1^{d_i^2} \ldots \mathcal{C}_{\log_{d_i^2}}\sigma$, respectively, and on which calls to *Encode*() are made in phase $i+1$.

1. We need to show how $\mathcal{B}_1 \ldots \mathcal{B}_{\log_{d_i^2}\sigma}$ are obtained from $\mathcal{A}_1 \ldots \mathcal{A}_{\log_{d_i}\sigma}$.
2. We need to show that the size of any $\mathcal{B}_j$ is at most $\frac{sd}{d^0 d^1 d^2 d^4 d^8 \ldots d^{2^i}}$, given that the size of any $\mathcal{A}_j$ is at most $\frac{sd}{d^0 d^1 d^2 d^4 d^8 \ldots d^{2^{i-1}}}$.
3. We need to show that the final codes assigned to elements $e \in \Sigma$ satisfy all the original constraints.

We address each of these, in turn.

Obtaining $\mathcal{B}_j$'s from $\mathcal{A}_j$'s. Consider any constraint $(e^*_{(j,d_i)}, S^*_{(j,d_i)})$ for any j, $1 \leq j \leq \log_{d_i}\sigma$. If this constraint is satisfied in the ith phase, then nothing further needs to be done for this constraint. Otherwise, this constraint generates up to d_i new constraints in the $i+1$th phase, as follows.

Recall that $e^*_{(j,d_i)}$ denotes the first j digits in the d_i-ary representation of e, where j runs from 1 to $\log_{d_i}\sigma$. If j is odd, then let $g_1 \ldots g_{d_i-1}$ denote those elements in Σ whose first j digits in the d_i-ary representations are identical to those of e but whose $j+1$th digit is different from that of e. Then $(e^*_{(j,d_i)}, S^*_{(j,d_i)})$ contributes the following new constraints to $\mathcal{B}_{\lceil \frac{j}{2} \rceil}$:

$$\left((g_1)^*_{(\lceil\frac{j}{2}\rceil,d_i^2)}, S^*_{(\lceil\frac{j}{2}\rceil,d_i^2)}\right), \ldots, \left((g_{d_i-1})^*_{(\lceil\frac{j}{2}\rceil,d_i^2)}, S^*_{(\lceil\frac{j}{2}\rceil,d_i^2)}\right), \left(e^*_{(\lceil\frac{j}{2}\rceil,d_i^2)}, S^*_{(\lceil\frac{j}{2}\rceil,d_i^2)}\right).$$

And if j is even, then $(e^*_{(j,d_i)}, S^*_{(j,d_i)})$ contributes one new constraint

$$\left(e^*_{(\lceil\frac{j}{2}\rceil,d_i^2)}, S^*_{(\lceil\frac{j}{2}\rceil,d_i^2)}\right)$$

to $\mathcal{B}_{\lceil \frac{j}{2} \rceil}$.

Size of $\mathcal{B}_{\lceil \frac{j}{2} \rceil}$. ¿From the above description, each unsatisfied constraint in $\mathcal{A}_j$ and in $\mathcal{A}_{j-1}$ (assuming j is even, otherwise take $\mathcal{A}_j$ and $\mathcal{A}_{j+1}$) contributes at most d_i new constraints to $\mathcal{B}_{\lceil \frac{j}{2} \rceil}$. The total size of unsatisfied constraints in A_j and A_{j-1} together after $Encode(*, 2\log d_i + 1)$ is less than $\frac{|\mathcal{A}_j|}{2{d_i}^2} + \frac{|\mathcal{A}_{j-1}|}{2{d_i}^2}$. Therefore,

$$|\mathcal{B}_{\lceil \frac{j}{2} \rceil}| \leq d_i(\frac{|\mathcal{A}_j|}{2{d_i}^2} + \frac{|\mathcal{A}_{j-1}|}{2{d_i}^2}) = \frac{|\mathcal{A}_j| + |\mathcal{A}_{j-1}|}{2d_i}.$$

Then, if $|\mathcal{A}_j|, |\mathcal{A}_{j-1}| \leq \frac{sd}{d^0 d^1 d^2 d^4 d^8 \ldots d^{2^{i-1}}}$, then $|\mathcal{B}_{\lceil \frac{j}{2} \rceil}| \leq \frac{sd}{d^0 d^1 d^2 d^4 d^8 \ldots d^{2^i}}$, as required.

All Constraints are Satisfied. Consider any constraint (S, e). In the first phase, there must exist a j such that $(e^*_{(j,d)}, S^*_{(j,d)})$ is a constraint in one of the calls to $Encode()$ in this phase. If this constraint is not satisfied in this phase, then, from the previous paragraph, it follows that $(e^*_{(d^2, \lceil \frac{j}{2} \rceil)}, S^*_{(d^2, \lceil \frac{j}{2} \rceil)})$ is a constraint in one of the calls to $Encode()$ in the next phase. Iterating this argument, we conclude that there exists a phase i and a number j such that the constraint $(e^*_{(j,d_i)}, (S^*_{(j,d_i)})$ (recall d_i denotes d^{2^i}) is satisfied in phase i by a particular call to $Encode()$. The original constraint (S, e) is now satisfied by the codes assigned to elements in Σ because these codes are concatenations of several smaller codes, one of which is output by the above call.

Finally, we reiterate that the above algorithm takes $O(sz\log^2\sigma(1 + \log z/\log\log\sigma)\frac{d}{\log d})$ time and produces codes of length $O(z(\log\sigma\log\log_d\sigma + \log(sd)))$, where d is any value between 2 and σ. For $d = O(1)$, the above bounds become $O(sz\log^2\sigma(1 + \log z/\log\log\sigma))$ and $O(z(\log\sigma\log\log\sigma + \log s))$, respectively.

7.3 Reducing the Code Length Further

In this section, we assume the condition that the number of sets S with size at least i is $O(\frac{s}{2^i})$, for all i (see Lemma 6.4). With this assumption, we show that the code length can be reduced to $O(\log^2 s)$ and the time to $O(s\log^2\sigma)$. Subsequently, we will show how a similar reduction in codes can be obtained for the sets obtained in the randomized algorithm for the Row Rotation problem (see Lemma 5.1); the expected code length can be reduced to $O(\frac{\log^2 s}{\log\log s})$ and the expected time will be $O(s\log^2\sigma)$, in this case. Constant probability bounds for these two quantities will follow from Markov's inequality.

The Deterministic Case. The trick is to apply distinct values of d for constraints involving sets of distinct sizes. The set sizes range from 1 to $O(\log s)$. We partition this range into doubling categories; for sets whose sizes are in category $[i, 2i)$, we choose $d = \Theta(2^{i/2})$. Then we apply the algorithm in the previous section to constraints involving sets of this category. This is repeated for each category. Finally, for each $e \in \Sigma$, the final code is obtained by concatenating the codes for e obtained for each of these categories.

The time taken for satisfying constraints involving sets in category $[i, 2i)$ is obtained by replacing s by $O(\frac{s*i}{2^i})$, z by i, and d by $\Theta(2^{i/2})$ in the running time derived at the end of the previous section. Also, recall that for each category for which its terms σ, s satisfy $s+1 < \sigma$, the term σ can be reduced to at most $s+1$ using time $O(s\log s)$. Then summing over all categories

(so i doubles at each successive term in this sum), the total running time becomes:

$$O\left(s\log^2\sigma\sum(\frac{i^2}{2^i}\frac{2^{i/2}}{i})(1+\frac{\log i}{\log\log\sigma})\right)=O(s\log^2\sigma)$$

Using the same substitutions, the total code length is:

$$O\left(\sum i(\log\sigma\log\log_{2^{i/2}}\frac{s\times i}{2^i}+\log(\frac{s\times i\times 2^{i/2}}{2^i}))\right),$$

which simplifies to

$$O\left(\sum i(\log\sigma\log\frac{\log s}{i}+\log s)\right)=O(\log^2 s).$$

The Randomized Case. Recall that Theorem 5.1 provides sets with the following property: the expected number of sets S with size at least i is $O(\frac{s}{i^{\Theta(i)}})$, for all i, $1\le i=O(\frac{\log s}{\log\log s})$. As before, we partition this range into doubling categories; for sets whose sizes are in category $[i,2i)$, we choose $d=\Theta(i^{\Theta(i)})$. The expected time taken for satisfying constraints involving sets in category $[i,2i)$ is obtained by replacing s by $O(\frac{s\times i}{i^{\Theta(i)}})$, z by i, and d by $\Theta(i^{\Theta(i)})$ (with an appropriate choice of constant in the $\Theta(i)$ term so the following inequalities hold). Summing over all categories (so i doubles at each successive term in this sum), we get the following bound on the total expected time:

$$O\left(s\log^2\sigma\sum(\frac{i^2}{i^{\Theta(i)}}\frac{i^{\Theta(i)}}{i\log i})(1+\frac{\log i}{\log\log\sigma})\right)=O\left(s\log^2\sigma\sum\frac{i}{i^{\Theta(i)}}\right)=O(s\log^2\sigma).$$

Using the same substitutions, the total code length is:

$$O\left(\sum i(\log\sigma\log\log_{i^{\Theta(i)}}\frac{s\times i}{i^{\Theta(i)}}+\log(\frac{s\times i\times i^{\Theta(i)}}{i^{\Theta(i)}}))\right),$$

which simplifies to

$$O\left(\sum i(\log\sigma\log\frac{\log s/i^{\Theta(i)}}{i\log i}+\log s)\right).$$

Using the fact that $i=O(\frac{\log s}{\log\log s})$, the above expression simplifies to $O(\frac{\log^2 s}{\log\log s})$.

7.4 Implementing $Encode(\mathcal{C},j)$

It suffices to show how to implement this procedure for $j=1$ in $O(|\mathcal{C}|z\log\sigma(1+\log z/\log\log\sigma))$ time with codes of length $O(z)$. If j is larger, then just repeating this procedure with the unsatisfied constraints (whose total size at least halves in each consecutive step) and concatenating the resulting codes together gives the desired result in the same time, using codes of length $O(zj)$. In the rest of this section, we assume $j=1$.

The main idea is to hash elements of the set $\{1\ldots\sigma\}$ into an $r=O(z)$ sized range using a hash function $h:\{1\ldots\sigma\}\to\{1\ldots r\}$, so that the following property is satisfied: for at least

half[4] the constraints $(S,e) \in \mathcal{C}$, $h(e) \neq h(f)$ for any $f \in S$. Suppose we can find such a hash function $h()$. Then it is easily seen that the following assignment of codes will satisfy half the constraints in $\mathcal{C}$: the code for element e has length r with a 1 at the $h(e)$th position and 0's elsewhere. It now remains to be described how a hash function $h()$ with the above property is determined in $O(|\mathcal{C}|z\log\sigma)$ time.

Hash Functions. We use hash functions of the form

$$h(e) = ae \pmod{p} \pmod{r},$$

where $p \in [r\log\sigma(1+\log z/\log\log\sigma), 2r\log\sigma(1+\log z/\log\log\sigma)]$ is a prime number, $1 \leq a \leq p$, and $r = \Theta(z)$. The aim is to choose a and p carefully so that the required property is satisfied.

We say that a prime p in the above range is *good* if for all but one quarter of the constraints $(S,e) \in \mathcal{C}$, $e \pmod{p} \neq f \pmod{p}$, for all $f \in S$. Having chosen a good p, those constraints which satisfy the above property are called *good* constraints. Given a good prime p, we say that a choice of a is *good* if for all but one quarter of the good constraints $(S,e) \in \mathcal{C}$, $ae \pmod{p} \pmod{r} \neq af \pmod{p} \pmod{r}$, for all $f \in S$.

We will need the following lemma.

Lemma 7.2 *There exists a good prime $p \in [r\log\sigma(1+\log z/\log\log\sigma), 2r\log\sigma(1+\log z/\log\log\sigma)]$. Further, given any good prime p, there exists a good a, $1 \leq a < p$.*

Proof. For the first part of the lemma, consider choosing a random prime p in the above range and consider some constraint $(S,e) \in \mathcal{C}$. The probability that $e \pmod{p} = f \pmod{p}$ for a fixed $f \in S$ is at most $\frac{\log\sigma/\log\log\sigma}{\Theta(r\log\sigma(1+\log z/\log\log\sigma)/\log(r\log\sigma(1+\log z/\log\log\sigma)))} = O(\frac{1}{r})$ (because $e-f$ can have at most $\log\sigma/\log\log\sigma$ prime factors while the number of primes in the range $[r\log\sigma(1+\log z/\log\log\sigma), 2r\log\sigma(1+\log z/\log\log\sigma)]$ is $\Theta(r\log\sigma(1+\log z/\log\log\sigma)/\log((r\log\sigma)(1+\log z/\log\log\sigma)))$. It follows that the probability that $e \pmod{p} = f \pmod{p}$ for any $f \in S$ is at most $O(\frac{z}{r})$, which can be made smaller than one quarter by an appropriate choice of constant for r. The first part of the theorem follows.

For the second part of the lemma, we consider only good constraints remaining after the choice of an arbitrary good p in the above range. Now consider a random choice of a, $1 \leq a < p$, and consider a good constraint $(S,e) \in \mathcal{C}$. Since this is a good constraint, $(e-f) \not\equiv 0 \pmod{p}$, for each f in S. Consider one such f. The number of choices of a which lead to $a(e-f) \pmod{p} \equiv 0 \pmod{r}$ is at most $\frac{p}{r}$; so the probability that $ae \pmod{p} \pmod{r} = af \pmod{p} \pmod{r}$ for this f is $\frac{1}{r}$. The probability that $ae \pmod{p} \pmod{r} = af \pmod{p} \pmod{r}$ for some $f \in S$ is at most $\frac{z}{r}$, which can be made smaller than one quarter by an appropriate choice of r. The second part of the lemma follows. □

The Algorithm. It now suffices to show how to find a good prime p and then a good a, the existence of both being guaranteed by Theorem 7.2. We use exhaustive search in both cases.

For each prime p in the above range and for each constraint $(S,e) \in \mathcal{C}$, we check whether $e \pmod{p} \neq f \pmod{p}$, for all $f \in S$. This check takes $O(|\mathcal{C}|)$ time per choice of p, giving $O(|\mathcal{C}|z\log\sigma(1+\log z/\log\log\sigma))$ time overall.

Next, having fixed p, we go through each $a < p$. For each such a and for each good constraint $(S,e) \in \mathcal{C}$, we check whether $ae \pmod{p} \pmod{r} \neq af \pmod{p} \pmod{r}$, for all $f \in S$. The time taken is again $O(|\mathcal{C}|z\log\sigma(1+\log z/\log\log\sigma))$, as required.

[4] By size, and not by number. All further references to fractions of constraints will be by size and not number.

Remark on finding primes. We conclude with a technical issue. The above algorithm requires finding primes in the range $[r \log \sigma(1+\log z/\log\log \sigma), 2r \log \sigma(1+\log z/\log\log \sigma)]$. Let $y = r \log \sigma(1+\log z/\log\log \sigma)$. All primes in this range can be determined in $O(y \log y)$ and even $O(y \log\log y)$ time once and for all at the beginning of the algorithm. This doesn't contribute significantly to the complexity, since $r = O(z) = O(\log s)$.

7.5 Interval Codes

In this section, we consider the case when all the sets S are intervals of $[1 \ldots \sigma]$ of length z. In this case, we show how to construct codes of length $O(z + \log \sigma)$ in $O(\sigma)$ time.

We will actually solve the Data-Independent case for this problem, i.e., we will assume that all possible intervals of length z are present. Clearly, the resulting code applies to the Data-Dependent case as well.

First, we classify all (S, e) pairs, $e \notin S$, into two categories. Partition the interval $[1 \ldots \sigma]$ into disjoint intervals $[1 \ldots 2z], [2z+1 \ldots 4z]$, etc (we call these intervals *basic* intervals). Category 1 has all those (S, e) pairs for which S lies completely within one of these intervals. Category 2 contains all the remaining (S, e) pairs. Since $|S| \leq z$ and S is an interval, it follows that if S is in Category 2, then it must be in one of the intervals $[z+1 \ldots 3z], [3z+1 \ldots 5z]$, etc.

We will show how to obtain codes for each element in $[1 \ldots \sigma]$ so as to separate the pairs (S, e) in Category 1. Codes which separate the pairs (S, e) in Category 2 can be obtained similarly. The final code will be obtained by concatenating these two codes. This code will clearly separate all (S, e) pairs, $e \notin S$.

We process Category 1 as follows. (S, e) pairs in this category are further subdivided into two subcategories. Those pairs for which S and e are in the same basic interval are in Subcategory 1; the remaining pairs are in Subcategory 2. As before, we will obtain distinct codes which will separate pairs in each subcategory; the final code for Category 1 will be a concatenation of these two codes.

First, consider Subcategory 1. We will assign a code of length $2z$ to each element in $[1 \ldots 2z]$; this code will separate all (S, e) pairs which are in this basic interval. It is now easily seen that assigning element e, $2z+1 \leq e \leq \sigma$, the same code as $e \bmod 2z$ will separate all pairs in this subcategory. The codes assigned to elements in $[1 \ldots 2z]$ are as follows. For $e \in 1 \ldots z$, the code has $z+1-e$ leading ones, followed by $2z-(z+1-e) = z+e-1$ trailing 0's. For $e \in z+1 \ldots 2z$, the code has $3z - e$ leading 0's, followed by $e - z$ trailing ones. A little inspection shows that this code separates all pairs (S, e) in this category.

Second, consider Subcategory 2. All (S, e) pairs in this subcategory have the property that S is completely within some basic interval and S and e are in distinct basic intervals. The code we give e is the binary representation of the first element in the basic subinterval containing e, followed by the bitwise complement of this binary representation. This code separates all pairs in this subcategory due to the following property: if f, g are the first elements in distinct basic intervals, then some bit in the code for f obtained above has a 1, and the corresponding bit in the code for g is 0. The length of this code is $2 \log \lceil \frac{\sigma}{2z} \rceil$.

The total code length is thus $O(z + \log \sigma)$.

References

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

[2] E. Cambouropoulos, M. Crochemore, C.S. Iliopoulos, L. Mouchard, Y.J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop on Combinatorial Algorithms*, 1999, pp. 129–144.

[3] R. Cole, R. Hariharan. Tree pattern matching and subset matching in randomized $O(n \log^3 m)$ time. Proceedings of the *29th ACM Symposium on Theory of Computing*, 1997, pp. 66–75.

[4] R. Cole, R. Hariharan, P. Indyk. Tree pattern matching and subset matching in deterministic $O(n \log^3 m)$ time. Proceedings of the *10th ACM-SIAM Symposium on Discrete Algorithms*, 1999, pp. 245–254.

[5] R. Cole, R. Hariharan. Tree pattern matching to subset matching in linear time. Submitted to *SIAM Journal on Computing*, 2000.

[6] H. Chernoff. *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Annals of Mathematical Statistics, 23, 1952, pp. 493–509.

[7] T. Crawford, C.S. Iliopoulos, R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, Vol. 11, 1998, pp. 73–100.

[8] M. Dubiner, Z. Galil, E. Magen. Faster tree pattern matching. Proceedings of the *31st IEEE Symposium on Foundations of Computer Science*, 1990, pp. 145–150.

[9] A.G. Dyachkov, V.V. Rykov. A survey of superimposed code theory. *Problems of Control and Information Theory*, 12, 4, 1983 (English translation).

[10] 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.

[11] C.M. Hoffman, M.J. O'Donell. Pattern matching in trees. *Journal of the ACM*, 1982, pp. 68–95.

[12] P. Indyk. Deterministic superimposed coding with applications to pattern matching. Proceedings of the *38th IEEE Symposium on Foundations of Computer Science*, 1997, pp. 127–136.

[13] P. Indyk. Faster algorithms for string matching problems: matching the convolution bound. Proceedings of the *39th IEEE Symposium on Foundations of Computer Science*, 1998, pp. 166-173.

[14] P. Indyk, R. Motwani, S. Venkatasubramanian, Geometric matching under noise: combinatorial bounds and algorithms. Proceedings of the *10th ACM-SIAM Symposium on Discrete Algorithms*, 1999, pp. 457–465.

[15] W.H. Kautz, R.C. Singleton, Nonrandom binary superimposed codes, *IEEE Transactions on Information Theory*, 10, 1964, pp. 363–377.

[16] S.R. Kosaraju. Efficient tree pattern matching. Proceedings of the *30th IEEE Symposium on Foundations of Computer Science*, 1989, pp. 178–183.

[17] R. Motwani, P. Raghavan. Randomized algorithms. Cambridge University Press, 1995, pp. 396.

[18] E. Porat. Private communication.