Verifying Candidate Matches in Sparse and Wildcard Matching [*]

[Extended Abstract] [†]

Richard Cole
Courant Institute, NYU
251 Mercer Street
NY, NY 10012
cole@cs.nyu.edu

Ramesh Hariharan
Indian Institute of Science
Bangalore 560012
India
ramesh@csa.iisc.ernet.in

ABSTRACT

This paper obtains the following results on pattern matching problems in which the text has length n and the pattern has length m.

- An $O(n \log m)$ time deterministic algorithm for the String Matching with Wildcards problems, even when the alphabet is large.
- An $O(k \log^2 m)$ time Las Vegas algorithm for the Sparse String Matching with Wildcards problem, where $k << n$ is the number of non-zeros in the text. We also give Las Vegas algorithms for the higher dimensional version of this problem.
- As an application of the above, an $O(n \log^2 m)$ time Las Vegas algorithm for the Subset Matching and Tree Pattern Matching problems, and a Las Vegas algorithm for the Geometric Pattern Matching problem.
- Finally, an $O(n \log^2 m)$ time deterministic algorithm for Subset Matching and Tree Pattern Matching.

The crucial new idea underlying the first three results above is that of confirming matches by convolving vectors obtained by coding characters in the alphabet with non-boolean (i.e., rational or even complex) entries; in contrast, almost all previous pattern matching algorithms consider only boolean codes for the alphabet. The crucial new idea underlying the fourth result is a simpler method of shifting characters which ensures that each character occurs as a singleton in some shift.

1. INTRODUCTION

This paper obtains deterministic and Las Vegas algorithms for a variety of pattern matching problems. Each of these problems requires finding all occurrences of a pattern p in a text t. All the Las Vegas algorithms we give have the following form: find a set of candidate matches and then verify them. Indeed, the deterministic algorithm for wildcard matching can also be viewed in this spirit. While this approach is not new (see e.g., [3] and [14]), the method in this paper appears to be considerably more general.

The criteria for defining pattern occurrences differ for each of the problems we consider. We describe previous history and our results for each of these problems in turn. In each problem, unless otherwise specified, we will use n to denote $|t|$ and m to denote $|p|$.

The Wildcard Matching Problem. Here, p is said to occur at location i in t if, for each non-wildcard symbol $p[j]$ in p, $0 \leq j < m$, the corresponding text symbol $t[i+j]$ is either identical to $p[j]$ or is itself a wildcard symbol. Let Σ denote the alphabet set from which the non-wildcard symbols in t, p are drawn.

A classic result of Fischer and Paterson [7] states that this problem can be solved in $O(n \log m \log |\Sigma|)$ time. Removing the dependence on $|\Sigma|$ in the above time complexity has been an important open problem for a long time. Recently, Indyk [11] removed the dependence on $|\Sigma|$ but by using a Monte Carlo randomized algorithm which took $O(n \log n)$ time. Kalai [13] gave another (simpler) Monte Carlo randomized algorithm with a running time of $O(n \log m)$.

We give a very simple deterministic algorithm for the above problem which runs in $O(n \log m)$ time. The main idea here is to assign a two character rational code to each non-wildcard character in t and p and then perform a convolution. This convolution will allow us to count the number of aligned matching non-wildcard characters.

Most previous approaches have used boolean codes for the alphabet in conjunction with convolution. One possible exception is Indyk's algorithm [11], which uses a boolean encoding for the alphabet but then performs a convolution modulo 2 in linear time; this modulo 2 convolution involves performing a regular convolution on vectors of size $\Theta(\frac{n}{\log n})$ with entries in a field defined by irreducible degree $\Theta(\log n)$ boolean polynomials. Kalai's algorithm [13] is another exception which effectively uses integer codes. Note that in

[*] This work was supported in part by NSF grants CCR9800085 and CCR0105678.

[†] Work partly done when both authors were visiting King's College, London, and when the second author was visiting NYU.

STOC 02 May 19-21, 2002, Montreal, Quebec, Canada
Copyright 2002 ACM 1-58113-495-9/02/0005 ...$5.00.

contrast to Indyk's and Kalai's algorithms, all matches reported by our algorithm are certain.

The Shift Matching Problem. In this problem, the characters in the text and pattern are either integers or wildcards. Pattern p is said to occur at location i in t if there exists an integer l_i such that one of the following conditions holds for all non-wildcard symbols $p[j]$ in p:

1. The text character $t[i+j]$ aligned with $p[j]$ is a wildcard.

2. $t[i+j] - p[j] = l_i$.

This problem has not been previously studied. Our motivation in defining this problem is its crucial use in solving the Sparse Wildcard Matching problem (which will be described shortly). We give a deterministic algorithm which takes $O(n \log \max\{N, m\})$ time, where N is a parameter such that the entries in t, p come from the range $0 \ldots N-1$. The main idea here is to assign a complex code to each non-wildcard character in t and p and then perform a convolution; again, this convolution will allow us to count the number of aligned pairs of characters satisfying condition 2 above.

The d-Dimensional Sparse Wildcard Matching Algorithm. In this problem, t, p are d-dimensional arrays of size n^d and m^d, respectively. Text t is sparse, i.e., it has only k non-zero characters, where $k << n^d$, and no wildcard characters. The pattern p comprises wildcard and non-zero non-wildcard characters. p is said to occur at location $i_1, \ldots, i_d$ in t if, for all non-wildcard characters $p[j_1, \ldots, j_d]$ in p, $t[i_1+j_1, \ldots, i_d+j_d] = p[j_1, \ldots, j_d]$; in other words, each non-wildcard character in p must be aligned with an identical character in t. Clearly, the number of non-wildcards in p must be at most k for it to occur anywhere in the text. We assume that both the text and the pattern are given by the implicit $O(kd)$ size description which specifies the list of non-zero entries in each.

This problem was defined (implicitly) by Cardoze and Schulman [2] with the aim of solving the Geometric Pattern Matching problem (which will be defined shortly). They gave a Monte Carlo randomized algorithm with running time $O(k \log k + kd)$ and failure probability inverse polynomial in k. The key idea in this algorithm was to hash the text and pattern down to strings of size $\Theta(k)$; this hashing preserves all matches and does not introduce new spurious matches, with high probability.

We will show how to solve this problem using a Las Vegas randomized algorithm whose running time is $O(dk \log k \log n)$, with failure probability inverse polynomial in k. For $md \leq n$, a variant of the standard trick of breaking the text into smaller pieces decreases this running time to

$$O(dk \log \min\{k, (dm)^d\} \log dm + dk \log k)$$

The failure probability is inverse polynomial in $\min\{k, (dm)^d\}$.

Our Las Vegas algorithm essentially adds a verification step to the Monte Carlo algorithm of Cardoze and Schulman [2]. Verification requires the detection of spurious matches introduced by the hashing mentioned above. To detect such matches, one needs to check whether each pair of aligned characters (in the text and pattern obtained after hashing) in a claimed match actually corresponds to a pair of aligned characters in the original text and pattern, respectively. This was precisely our motivation for defining the Shift Matching problem. Our algorithms for Wildcard Matching and Shift Matching above play a crucial role in detecting these spurious matches.

Subset Matching and Tree Pattern Matching. In the Subset Matching problem, 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. The pattern p is said to occur at text position i if the set $p[j]$ is a subset of the set $t[i+j]$, for all locations j in p.

This problem was defined by Cole and Hariharan [3], as an intermediate problem in solving the Tree Pattern Matching problem (to be defined shortly). The previous best algorithms known for this problem were an $O(s \frac{\log^3 s}{\log \log s})$ time Las Vegas algorithm due to Cole, Hariharan and Indyk [4], an $O(s \log s)$ time Monte Carlo algorithm due to Indyk [11], and an $O(s \log^3 s)$ time deterministic algorithm due to Cole, Hariharan and Indyk [4]. The above time complexities are for the case when $s \geq n, m$ (if not, then the running times become $O(n \frac{\log^2 s}{\log \log s} \log \max\{m, s\})$, $O(n \log s)$ and $O(n \log^2 s \log \max\{m, s\})$, respectively.

As a direct application of our 1-d Sparse Wildcard Matching algorithm, we give a Las Vegas algorithm with running time $O(s \log^2 s)$, with failure probability at most inverse polynomial in s (assuming $s \geq n, m$, if not, then the running time becomes $O(s \log s \log \max\{m, s\})$, with failure probability still inverse polynomial in s).

In addition, we give a deterministic algorithm with running time $O(s\log^2 s)$ (assuming $s \geq n, m$, if not, then the running time becomes $O(n\log s \log \max\{m, s\})$). This algorithm is based on the idea of choosing collections of shifts for characters in t and p such that each character occurs as a singleton in at least one of the collections. This idea was the basis for the algorithm in [3]; however, that paper chose shifts randomly, in contrast to our deterministic construction. A deterministic construction based on convolution was given in [4]. The present deterministic construction does not use convolutions and is faster by a $\log s$ factor. The resulting algorithm is substantially simpler than the $O(s \log^3 s)$ time deterministic algorithm in [4].

In the *Tree Pattern Matching* problem, t and p are ordered, node-labelled trees of size n and m respectively. 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. As shown in [3, 5], the Tree Pattern Matching problem can be reduced in linear time to the Subset Matching problem. The above results for the Subset Matching problem immediately lead to $O(n \log^2 m)$ time Las Vegas and deterministic algorithms for Tree Pattern Matching.

Geometric Pattern Matching. In this problem, t and p are collections of points in d-dimensional space. Let k denote the number of points in t. We assume that these points have integer coordinates and that the coordinates of points in t and p come from the ranges $[0 \ldots n-1]$ and $[0 \ldots m-1]$, respectively. The aim is to determine whether there exists a transformation from an allowed class of transformations which when applied to p ensures that each point in p is within a specified threshold distance Δ of some point in t. The two kinds of transformations we consider are translations and rigid motions, i.e., translations coupled with rotations.

The previous best algorithm for translations was a Monte Carlo algorithm due Cardoze and Schulman [2] and had a running time of $O(k(2\Delta+1)^d \log[k(2\Delta+1)^d])$. For rotations, Cardoze and Schulman [2] gave a Monte Carlo algorithm with a running time of $O(f(k,d,\Delta,\epsilon)\log f(k,d,\Delta,\epsilon))$ for an appropriate function $f(k,d,\Delta,\epsilon)$, where ϵ is a tolerance parameter in measuring distances.

For translations, we give a Las Vegas algorithm with running time $O(dk(2\Delta+1)^d \log[k(2\Delta+1)^d]\log n)$, with failure probability inverse polynomial in $k(2\Delta+1)^d$. For the case when $md \leq n$, this can be improved to $O(dk(2\Delta+1)^d \log\min\{k(2\Delta+1)^d, (dm)^d\}\log dm + dk(2\Delta+1)^d \log k(2\Delta+1)^d)$; the failure probability is at most inverse polynomial in $\min\{k(2\Delta+1)^d, (dm)^d\}$. For rigid motions, we give a Las Vegas algorithm with running time

$$O(f(k,d,\Delta,\epsilon)\log\min\{f(k,d,\Delta,\epsilon), \frac{m\sqrt{d}}{\epsilon}\}\log\frac{m\sqrt{d}}{\epsilon})$$

The failure probability is inverse polynomial in

$$\min\{f(k,d,\Delta,\epsilon), \frac{m\sqrt{d}}{\epsilon}\}$$

These algorithms are direct consequences of our Sparse Wildcard Matching algorithm.

Sparse Convolution. In the Sparse Convolution problem, the aim is to find the convolution vector w of two given vectors t and p, comprising only non-negative entries. We assume that t and p are given not as explicit vectors but rather as lists of location-value pairs comprising locations which have non-zero values. The aim is to compute w in an output sensitive way, i.e., in time proportional to the number of non-zero entries in w. This problem was posed in [15].

Let $||w||$ denote the number of non-zero entries in w. We show how to obtain these non-zero entries in $O(||w||\log^2 m)$ time, using a Las Vegas randomized algorithm, whose failure probability is inverse polynomial in m. To the best of our knowledge, this is the first algorithm for this problem. We remark that if we wanted to allow negative entries in t and p, we would need to define $||w||$ as the number of non-zero entries in $\tilde{w}$, where $\tilde{w}$ is as follows. Let $\tilde{t}$ and $\tilde{p}$ be t and p, respectively, with non-zero entries replaced by 1; $\tilde{w}$ is the product of $\tilde{t}$ and $\tilde{p}$.

Roadmap. Section 2 describes the definitions, notations and a basic tool used by our algorithms. Each subsequent section describes our algorithms for the problems listed above, in turn. Proofs of lemmas and the description of the Sparse Convolution algorithm are omitted for lack of space.

2. PRELIMINARIES

All algorithms in this paper will assume the RAM model of computation, which allows arithmetic on $\log N$ bit numbers in $O(1)$ time, where N is of the order of the maximum problem size.

In all our problems, we will use n to denote $|t|$ and m to denote $|p|$ (except in the d-Dimensional Sparse Wildcard Matching problem in which the corresponding terms are n^d and m^d, respectively). Using a standard reduction, we will assume that $n \leq 2m$ for all problems in which t and p are strings (using the standard trick of breaking the text into pieces of length $2m$, consecutive pieces overlapping by m). t and p are indeed strings in all problems, except for the d-Dimensional Sparse Wildcard Matching Problem, the Tree Pattern Matching problem, and the Geometric Pattern Matching problem.

The following definition will be central to the techniques used in this paper.

Convolution. The *convolution vector* of two vectors u, v is defined as the vector w such that $w[i] = \sum_{j=0}^{|u|-1} u[j]v[(i+j)(\text{mod}|v|)]$. We use the notation $u \oplus v$ to denote w. Note that this definition of convolution involves wrap-around (i.e., v is assumed to be a cyclic vector). In this paper, we will also use the non-wrap-around notion of convolution, i.e., $w[i] = \sum_{j=0}^{|u|-1} u[j]v[i+j]$, with out of range entries taken as 0. However, unless otherwise specified, all references to convolution will refer to the wrap-around definition.

The Fast Convolution Theorem. The following theorem and its consequent corollary on the RAM model are standard (see for example, [17], page 1) and crucial to our algorithms. They hold for both definitions of convolution above.

THEOREM 1. *Consider two vectors u, v, each vector having length $O(m)$ and comprising l-bit entries. Let $M(l)$ be the time taken to multiply two l bit numbers. Then $u \oplus v$ with entries precise up to $l - \Theta(\log m)$ bits can be obtained in $O(m \log m * M(l))$ time.*

Since $M(l) = O(1)$ on the RAM model for $l = O(\log N)$, we get the following corollary.

COROLLARY 2. *If $l = O(\log N)$ then $u \oplus v$ with entries precise up to $l - \Theta(\log m)$ bits can be obtained in $O(m \log m)$ time.*

3. THE WILDCARD MATCHING ALGORITHM

As stated in Section 2, we assume that $n < 2m$ and show how Corollary 2 yields a simple $O(m \log m)$ time algorithm for Wildcard Matching. We assume, without loss of generality, that symbols in t and p are drawn from the integer alphabet $0 \ldots m-1$ (otherwise, sort and rename the symbols at an expense of $O(m \log m)$ time).

Our algorithm performs the following two non-wrap-around convolutions.

Step 1. The aim of this convolution is to compute, for each location i of t, the number of non-wildcards in p that are aligned with non-wildcards in t when $p[0]$ is aligned with $t[i]$. Call this count $nw[i]$. This is calculated as follows.

We obtain a new text t' from t by replacing each non-wildcard by 1 and each wildcard by 0. A new pattern p' is obtained from p in the same way. It is easily seen that $(t' \oplus p')[i] = nw[i]$. By Corollary 2, $t' \oplus p'$ can be computed in $O(m \log m)$ time.

Step 2. The aim of this convolution is to compute, for each location i of t, a quantity which indicates whether or not p matches at i. The following convolution will yield value $2 * nw[i]$ if and only if p matches at location i. Since $nw[i]$ is already known from Step 1, this information is sufficient to find all occurrences of p in t.

This convolution involves a new text t' obtained from t by replacing each non-wildcard character a by two adjacent numbers a and $1/a$ (i.e., if $t[j] = a$ then $t'[2j] = a$ and $t'[2j+1] = 1/a$), and each wildcard by two 0's. A new pattern p' is obtained from p in the same way, except that $1/a$ and a are switched (i.e., if $p[j] = a$ then $p'[2j] = 1/a$ and $p'[2j+1] = a$). It is easily seen that p occurs at location i if and only if $(t' \oplus p')[2i] = 2 * nw[i]$ (this uses the fact that $a/b + b/a \geq 2 + \frac{1}{m(m-1)}$ for $a \neq b$). It remains to determine the time taken for this convolution.

Time and Precision Analysis. Using the fact that $a/b + b/a \geq 2 + \frac{1}{m(m-1)}$ for $a \neq b$, it follows that $(t' \oplus p')[2i] - 2 * nw[i]$ either equals 0 or is at least $\frac{1}{m(m-1)}$. Therefore, $O(\log m)$ bits of precision are sufficient to detect which of these two cases occurs. By Corollary 2, $t' \oplus p'$ can be computed to this level of precision in $O(m \log m)$ time, provided the input vectors t' and p' themselves have entries which are correct up to $\Theta(\log m)$ bits of precision. Setting up t' and p' with this level of precision is easily done in $O(m \log m)$ time (the only issue is that of determining $\Theta(\log m)$ significant digits of $1/a$, which is easily done in $O(\log m)$ time). This concludes the algorithm.

4. THE SHIFT MATCHING ALGORITHM

As stated in Section 2, we assume that $n < 2m$. We show how Corollary 2 yields a simple $O(m \log(mN))$ time algorithm for Shift Matching. As in Wildcard Matching, there are two steps. The first step is identical to the first step in Wildcard Matching and obtains the count $nw[i]$ for all text locations i. The second step is also similar, but uses a different encoding, as detailed below. In what follows, let α denote $\frac{2\pi\sqrt{-1}}{N}$.

Step 2. The aim of this step is to compute, for each location i of t, a quantity which indicates whether or not p matches at i. The following convolution will yield a complex number with modulus $nw[i]$ if and only if p matches at location i. Since $nw[i]$ is already known from Step 1, this information is sufficient to find all occurrences of p in t.

This convolution involves a new text t' obtained from t by replacing each non-wildcard character a by $e^{\alpha a}$ and each wildcard character by 0. Similarly, a new pattern p' is obtained from p by replacing each non-wildcard character a by $e^{-\alpha a}$ and each wildcard character by 0. It is easily seen that p occurs at location i if and only if $(t' \oplus p')[i] = e^{\alpha l_i} * nw[i]$, for some integer l_i (which is also the difference between any aligned pair of non-wildcard text and pattern characters in this match). It remains to determine the time and precision required for this convolution.

Time and Precision Analysis. It is easily seen that $|(t' \oplus p')[i] - e^{\alpha l_i} * nw[i]|$ either equals 0 or is at least

$$|e^{\alpha l_i} - e^{\alpha(l_i - 1)}| = 2Sin\frac{\pi}{N} \geq \frac{1}{N}, \text{ for } N \geq 2.$$

Therefore, $O(\log N)$ bits of precision in the output are sufficient to detect which of these two cases occurs. By Corollary 2, $t' \oplus p'$ can be computed to this level of precision in $O(m \log m)$ time, provided the input vectors t' and p' themselves have entries which are correct up to $\Theta(\log m + \log N)$ bits of precision. Setting up t' and p' with this level of precision is easily done in $O(m \log(mN))$ time.

Remark. We claim that if required, then for each match $t[i]$ of p in t, l_i, if defined, can be computed as well in the above mentioned time (by computing l_i from $e^{\alpha l_i}$). Note that l_i is defined as long as the pattern and the text have at least one pair of aligned non-wildcard characters.

5. THE D-DIMENSIONAL SPARSE WILDCARD MATCHING ALGORITHM

Before we describe our algorithm for the Sparse Wildcard Matching problem, we need to describe two basic tools used in this algorithm: *dimension reduction* and *length reduction by hashing.*

Dimension Reduction. Consider two possibly sparse d-dimensional arrays t and p, with $|t| = n^d$, $|p| = m^d$. t comprises some zeros and k non-zeros and p comprises wildcards and non-zero non-wildcard characters. As in [2], we use random projections to obtain strings t' and p' from t and p, respectively. t' and p' have length polynomial in k and matches of p in t will be related to matches of p' in t' as described shortly.

Choose integers $b_1 \ldots b_d$ independently and uniformly at random from a range polynomial in k. Map each location $t[i_1, \ldots, i_d]$ to $t'[\sum_{r=1}^{d} b_r i_r]$ and likewise for p. It is easy to see that distinct non-zeros in t map to distinct locations in t', with failure probability inverse polynomial in k. Further, this property is easily verified in $O(kd + k \log k)$ time. A similar property holds for non-wildcards in p; these map to distinct locations in p'. A location in t' to which no non-zero in t maps is set to 0 and a location in p' to which no non-wildcard in p maps is set to the wildcard character. The following important lemma holds.

Lemma 3. *If p matches starting at $t[i_1, \ldots, i_d]$ then p' matches starting at $t'[\sum_{r=1}^{d} b_r i_r]$. Further, if p does not match starting at $t[i_1, \ldots, i_d]$ then p' does not match at $t'[\sum_{r=1}^{d} b_r i_r]$, with failure probability inverse polynomial in k.*

Length Reduction by Hashing. Consider two possibly sparse strings t and p, with $|t| = n$, $|p| = m$. t comprises zeros and non-zeros and p comprises wildcards and non-zero non-wildcard characters (which we will sometimes refer to as just non-zeros). As in [2], we use hashing to obtain shorter strings (of an appropriately chosen length s) from t and p as follows.

Let $\mathcal{H}$ denote a family of hash functions given by $ax(\text{mod } q)(\text{mod } s)$, where p is a prime in $2n \ldots 4n$, $a \in 0 \ldots q-1$, and s is a number possibly much smaller than n. We choose a random hash function $h = ax(\text{mod } q)(\text{mod } s)$ from $\mathcal{H}$ (i.e., we choose a uniformly from $0 \ldots q-1$). Using h, we will map t and p to small strings t_h and p_h, respectively, as below.

Each location i in t_h and p_h will correspond to a set of non-zero locations in t and p, respectively, which map to i. t_h is obtained by mapping each location x in t to the following two locations in t_h:

- $ax(\text{mod } q)(\text{mod } s)$
- $[ax(\text{mod } q) + q](\text{mod } s)$

Thus each location in t has 2 images in t_h. p_h is obtained by mapping each location x in p to location $ax(\text{mod } q)(\text{mod } s)$ in p_h.

Definitions. A location is t_h is called *empty*, if no non-zero locations in t map to it, *singleton*, if exactly one non-zero location in t maps to it, and *multiple* otherwise. Analogous definitions hold for locations in p_h. A *wrap-around* placement of p_h starting at location i in t_h is a placement of p_h such that $p_h[j]$ is aligned with $t_h[(i+j)(\text{mod } s)]$.

The following properties of t_h and p_h will be crucial and are easy to show.

LEMMA 4. *Consider a wrap-around placement of p_h in t_h with $p_h[0]$ aligned with $t_h[h(i)]$. Then for each $j, 0 \leq j \leq m-1$, $p_h[h(j)]$ is aligned with $t_h[(h(i)+h(j))(\bmod\ s)]$, which is one of the images of $t[i+j]$.*

LEMMA 5. *Let k denote the number of non-zeros in t. Consider a wrap-around placement of p_h in t_h with $p_h[0]$ aligned with $t_h[h(i)]$ and consider any location $p[j]$. If q is indeed a prime, then with probability $O(\frac{k}{s})$, $p_h[h(j)]$ is aligned with a non-empty location in t_h if and only if $t[i+j] \neq 0$.*

5.1 The Monte-Carlo Algorithm

We assume that the text t has size n^d and the pattern p has size m^d. Let k denote the number of non-zeros in t. We describe the Monte Carlo algorithm of [2] first and then use the Wildcard Matching and Shift Matching algorithms above to get a Las Vegas algorithm. We describe this algorithm only for the case when all non-zeros in t and p equal 1. The Las Vegas algorithm to be described will handle the more general case as well.

First, we do the dimension reduction described above to obtain strings t' and p' from t and p, respectively. This takes $O(dk + k\log k)$ time. Note that t' and p' have length polynomial in k. Next, we set $s = O(k)$ (recall from above that s is the range of the hash functions to be chosen), with the constant chosen appropriately, so that the probability in Lemma 5 is a small enough constant. Then we choose $\Theta(\log k)$ hash functions h independently and uniformly at random from $\mathcal{H}$; for each chosen hash function h, we obtain t_h and p_h from t' and p' as described above. The time taken in this process is $O(polylog(k))$ for choosing q and $O(k\log k)$ for constructing t_h and p_h, over all h.

For each chosen hash function h, we find all wrap-around placements of p_h in t_h such that each non-empty location in p_h is aligned with a non-empty location in t_h. This is done easily by a simple reduction to the Boolean Wildcard Matching problem and takes $O(k\log^2 k)$ time over all chosen hash functions h. Location $t_h[i]$ is said to be a *match* for p_h if the wrap-around placement of p_h starting at $t_h[i]$ satisfies the above property.

Let M denote the set of *potential matches* of p in t (i.e., those placements of p in t for which the lexicographically least non-zero location in p is aligned with a non-zero in t); note that $|M| \leq k$. Then, for each $i_1, \ldots, i_d \in M$, we check in $O(d + \log k)$ time whether p_h matches t_h at $h(\sum_{r=1}^{d} b_r i_r)$ for all chosen h. Potential matches in M satisfying this condition are declared real matches. This takes $O(kd + k\log k)$ time overall.

It remains to show correctness. From Lemmas 4 and 3, it is easily seen that if p matches t at location $i_1, \ldots, i_d$, then p_h matches t_h at location $h(\sum_{r=1}^{d} b_r i_r)$, for all chosen h. Further, from Lemmas 5 and 3, if p does not match t at location $i_1, \ldots, i_d$, then p_h mismatches t_h at location $h(\sum_{r=1}^{d} b_r i_r)$, for at least one of the chosen h, with failure probability inverse polynomial in k. Thus, all matches of p in t will pass the above test with certainty, while the probability of any mismatch passing this test is inverse polynomial in k.

The total time taken is $O(polylog(k) + dk + k\log^2 k)$. Cardoze and Schulman [2] in fact obtain a slightly faster algorithm running in $O(polylog(k) + dk + k\log k)$ by using a linear time Monte Carlo algorithm for doing convolution modulo 2 [11].

5.2 The Las Vegas Algorithm

We now show how to convert the above Monte Carlo algorithm to a Las Vegas algorithm running in time $O(polylog(k)$ $+ dk\log k\log n)$, even for the case when the non-zeros in t and p come from a large alphabet. Note that this essentially adds a multiplicative factor of $O(d\log n)$ to the Monte Carlo algorithm.

The main idea is to focus on singleton locations in t_h and p_h, and compute various statistics on these singletons. The following fact is crucial and the proof is similar to the proof of Lemma 5. Recall that following the dimension reduction and hashing steps, each non-zero location in t has two images in t_h.

FACT 6. *The following holds with failure probability inverse polynomial in k: for each non-zero location in t, there exists a chosen hash function h such that both images of this non-zero are singletons in t_h.*

Step 1: Ensuring Singletons. We begin with the following preliminary test. Having chosen the $\Theta(\log k)$ hash functions h, we check whether, for each non-zero location in t, there exists a chosen hash function h such that both images of this non-zero location are singletons in t_h.

This property takes $O(k\log k)$ time to check. If this property does not hold then the verification fails, and the entire algorithm is repeated. By Fact 6, the probability of this happening is inverse polynomial in k. In the sequel, we assume that, for each non-zero location in t, there exists a chosen hash function h such that both images of this non-zero location are singletons in t_h.

Step 2: Checking that Singletons Match. For each chosen hash function h, we find all wrap-around placements of p_h in t_h such that the following conditions are satisfied:

1. Each non-empty location in p_h is aligned with a non-empty location in t_h.

2. For each pair of aligned singleton locations in t_h and p_h, if any, the following property holds: the non-zero characters in t and p, respectively, which map to these singleton locations are identical.

Both conditions are checked easily by a simple reduction to the Wildcard Matching problem. This step takes $O(k\log^2 k)$ time over all chosen hash functions h.

Definition. Location $t_h[i]$ is said to be a *match* for p_h if the wrap-around placement of p_h starting at $t_h[i]$ satisfies the above properties. We say that a potential match of p at $t[i_1, \ldots, i_d]$ is a *claimed match* if p_h matches at $t_h[h(\sum_{r=1}^{d} b_r i_r)]$, for all chosen hash functions h. Further, this claimed match of p at $t[i_1, \ldots, i_d]$ is said to *correspond* to a match of p_h at $t_h[h(\sum_{r=1}^{d} b_r i_r)]$.

Note that by Lemmas 4, 5, and 3, all matches of p in t must be claimed matches, and all claimed matches are true matches with failure probability inverse polynomial in k. Next, we need identify which, if any, of the claimed matches are false matches.

False Match Scenarios. We enumerate the scenarios in which a claimed match is a false match. Consider one such false match of p at location $t[i_1, \dots, i_d]$. Since this is not actually a match, there exists a non-zero location $p[j_1, \dots, j_d]$ whose value is different from $t[i_1 + j_1, \dots, i_d + j_d]$. For brevity, let $\hat{i}$ denote $h(\sum_{r=1}^{d} b_r i_r)$ and $\hat{j}$ denote $h(\sum_{r=1}^{d} b_r j_r)$. Since p_h matches t_h at $\hat{i}$, the location $t_h[\hat{i} + \hat{j}]$ aligned with $p_h[\hat{j}]$ is non-empty, for all chosen hash functions h.

There are only three possible false match scenarios.

1. $t_h[\hat{i} + \hat{j}]$ is singleton but $p_h[\hat{j}]$ is multiple, for some chosen h.

2. $t_h[\hat{i} + \hat{j}]$ and $p_h[\hat{j}]$ are both singletons, for some chosen h.

3. $t_h[\hat{i} + \hat{j}]$ is multiple, for all chosen h.

We briefly describe how to detect false matches in each scenario. Whatever remains will be a true match and all true matches will indeed be detected.

Detecting Scenario 1. Note that a true match can never lead to Scenario 1. In other words, if p indeed matches t at location $i_1, \dots, i_d$ then Scenario 1 does not hold. This can be seen as follows. If multiple non-zeros in p map to $p_h[\hat{j}]$ then the non-zeros in t aligned with these non-zeros in p must all map to $t_h[\hat{i} + \hat{j}]$, by Lemmas 4 and 3. But this would mean that $t_h[\hat{i} + \hat{j}]$ is multiple and not singleton. Therefore, for any claimed match of p at $t[i_1, \dots i_d]$, if Scenario 1 holds for the wrap-around placement of p_h at $t_h[\hat{i}]$ for even one of the chosen hash functions h, then this claimed match is not an actual match.

Detecting Scenario 1 is easily done in $O(k \log^2 k)$ time using a simple reduction to the Boolean Wildcard Matching problem. All claimed matches satisfying Scenario 1 are eliminated by this process.

Detecting Scenario 2. Consider a particular claimed but false match of p at $t[i_1, \dots, i_d]$ in which $p[j_1, \dots, j_d]$ is non-zero and the aligned character $t[i_1 + j_1, \dots, i_d + j_d]$ is either 0, or non-zero and different from $p[j_1, \dots, j_d]$. Consider that hash function h for which Scenario 2 holds in the corresponding match of p_h in t_h. So both $t_h[\hat{i} + \hat{j}]$ and $p_h[\hat{j}]$ are singletons.

They key property which enables us to identify this mismatch via Shift Matching is the following. Clearly, the unique non-zero location in p which maps to $p_h[\hat{j}]$ is the location $p[j_1, \dots, j_d]$. Let $t[i'_1, \dots, i'_d]$ denote that unique non-zero location in t which maps to $t_h[\hat{i} + \hat{j}]$; note that $i'_r \neq i_r + j_r$ for some r, otherwise the claimed match being considered would not have passed Step 2. Then the vector $(i'_1 - j_1, \dots, i'_d - j_d)$ is not equal to the vector $(i_1, \dots, i_d)$. This suggests that Scenario 2 can be detected by performing Shift Matching on each dimension separately as follows.

Consider each dimension r and each chosen hash function h in turn. Obtain new strings t'_h and p'_h from t_h and p_h, respectively, by replacing each singleton by the rth dimension of the unique non-zero location in t and p, respectively, which maps to this singleton. All other locations in t'_h and p'_h get wild-cards. For each wrap-around placement $t'_h[i]$ of p'_h, we determine whether this placement of p'_h matches t'_h under the definition of Shift Matching, and if so, we compute the quantity l_i (see the remark at the end of Section 4). The time taken in this process is $O(dk \log k \log n)$ over all h and all d dimensions.

Finally, a particular claimed match of p at $t[i_1, \dots, i_d]$ is eliminated unless the following holds for all chosen h: p'_h matches at $t'_h[\hat{i}]$ and $l_{\hat{i}}$ for dimension d' is either undefined (i.e., one at least of each pair of aligned characters in p'_h and t'_h is a wildcard, so Scenario 2 does not hold trivially) or equals $i_{d'}$, for all dimensions d'.

Detecting Scenario 3. Assuming that Scenarios 1 and 2 never apply, we detect false matches corresponding to Scenario 3 using the following statistic. Recall that by Step 1, for each non-zero location in t, there exists a chosen hash function h such that both images of this non-zero location are singletons in t_h. We *associate* each non-zero location in t with exactly one of the various h's for which the above property holds. Singletons in t_h corresponding to non-zero locations in t associated with h, are called *special singletons.*

For each claimed match of p in t, and for each chosen hash function h, we consider the corresponding match of p_h in t_h. In this match, we count the number of singletons associated with t_h which are aligned with singletons in p_h. This is done in $O(k \log k)$ time per chosen hash function h by convolving the string obtained by replacing special singletons in t_h by 1 and others by 0, with the string obtained by replacing singletons in p_h by 1 and others by 0. The total time taken by this step is $O(k \log^2 k)$.

For each claimed match of p in t, we sum the above count over all hash functions h. By the following lemma, this claimed match is eliminated if and only if the above sum does not equal $\#p$, where $\#p$ is the number of non-zeros in p.

LEMMA 7. *The above sum equals $\#p$ for a claimed match if and only if it is a true match.*

Total Time Taken. The total time taken by the algorithm is $O(polylog(k) + dk \log k \log n) = O(dk \log k \log n)$, with failure probability inverse polynomial in k.

For $md \leq n$, the above time can be improved to

$$O(dk \log \min\{k, (dm)^d\} \log dm + dk \log k)$$

using a variant of the standard trick of breaking t into subarrays of smaller size; the failure probability is at most inverse polynomial in $\min\{k, (dm)^d\}$. Breaking the text involves dividing it into smaller texts of size dm and overlap m along each dimension, chosen so as to ensure that the number of non-zero elements in the smaller texts sum (over all smaller texts) to $O(k(1 + \frac{1}{d})^d) = O(k)$.

Remark on improving success probability. The failure probability can in fact be reduced to $\frac{1}{k^{\Theta(\log k)}}$ based on the observation that failure in the verification process results solely from failure in Step 1. The probability of failure in Step 1 can be reduced to $\frac{1}{k^{\Theta(\log k)}}$ by repeating the dimension reduction and hashing steps $\Theta(\log k)$ times. The extra time taken in this process is $O(dk \log k + k \log^2 k)$.

6. APPLICATIONS OF SPARSE WILDCARD MATCHING

6.1 Subset Matching and Tree Pattern Matching

First, consider Subset Matching. For each distinct character $c \in \Sigma$ which occurs in p, we obtain a 1-d instance of the Sparse Wildcard matching problem as follows. We obtain a new text t_c from t by replacing each set $t[i]$ which does not have c by 0 and each set $t[i]$ which has c by 1. We obtain a new pattern p_c from p by replacing each set $p[i]$ which does not have c by a wildcard and each set $p[i]$ which has c by 1. It is easily seen that all occurrences of p in t can be found by solving the Sparse Wildcard matching problem on p_c, t_c for all $c \in \Sigma$ and taking the intersection of the sets of matches obtained. This gives a Las Vegas algorithm with running time $O(s \log^2 s)$, with failure probability at most inverse polynomial in s. Further, as mentioned in the introduction, this leads to improved algorithms for Tree Pattern Matching.

6.2 Geometric Pattern Matching

It is easily seen that the case of translations reduces to the d-dimensional Sparse Wildcard Matching problem in which the text has size n^d, the pattern has size m^d, the number of 1's in the text (all non-zeros are 1's) is $O((2\Delta+1)^d k)$ and the number of 1's in the pattern is the number of points in p. Using the above mentioned algorithm for the d-dimensional Sparse Wildcard Matching problem, we get an algorithm with running time $O(dk(2\Delta+1)^d \log[k(2\Delta+1)^d] \log n)$, with failure probability inverse polynomial in $k(2\Delta+1)^d$. For the case when $md \leq n$, this can be improved to $O(dk(2\Delta+1)^d \log \min\{k(2\Delta+1)^d, (dm)^d\} \log dm + dk(2\Delta+1)^d \log k(2\Delta+1)^d)$; the failure probability is at most inverse polynomial in $\min\{k(2\Delta+1)^d, (dm)^d\}$.

Next, consider the case of rigid motions. In [2, 12], it is shown how this problem reduces to several instances of the 1-d Sparse Wildcard Matching problem; the total number of non-zeros over all problems is $f(k,d,\Delta,\epsilon)$, where $f()$ is as in the introduction, and the pattern size in each such problem is $O(\frac{m\sqrt{d}}{\epsilon})$. Our algorithm for the 1-d Sparse Wildcard Matching problem leads to a Las Vegas algorithm having $O(f(k,d,\Delta,\epsilon) \log \min\{f(k,d,\Delta,\epsilon), \frac{m\sqrt{d}}{\epsilon}\} \log \frac{m\sqrt{d}}{\epsilon})$ running time and failure probability inverse polynomial in $\min\{f(k,d,\Delta,\epsilon), \frac{m\sqrt{d}}{\epsilon}\}$.

7. FAST DETERMINISTIC SUBSET MATCHING AND TREE PATTERN MATCHING

We give a deterministic $O(s\log^2 s)$ algorithm for the Subset Matching Problem based on the following crucial fact from [3, 5]. This immediately leads to an $O(n\log^2 m)$ time deterministic algorithm for Tree Pattern Matching.

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' as follows. For each character e in the above alphabet, a shift $shift(e)$ is chosen, where $shift(e)$ is an integer in an appropriate range (to be described later). 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'.

We will perform character shifting $O(\log s)$ times; the ith such operation will result in new strings t'_i, p'_i. We will ensure that the various shiftings satisfy the following properties:

- Each character in each set in t is in a singleton set in at least one of the t'_i's.
- The t'_i's and p'_i's have length $O(s)$ each.

We will show how to perform character shiftings satisfying these two properties in Section 7.1. The overall time taken in this process will be $O(s\log^2 s)$.

Next, we show how to find all matches of p in t using the t'_i's and p'_i's. Since the techniques here are similar in spirit to those used in the Sparse Wildcard Matching problem and the Sparse Matching problem, we will describe them in less detail.

The Algorithm.

Step 1. We find a candidate set of matches in this step. Location i is termed a candidate match if for all t'_j, p'_j's, the placement of p'_j starting at $t'_j[i]$ satisfies the following property: each non-empty set in p'_j is aligned with a non-empty location in t'_j; further each singleton in t'_j is aligned with either a singleton or an empty location in p'_j. This is easily done in $O(s\log^2 s)$ time overall using the Wildcard Matching algorithm.

It is easily seen that all true matches will survive this stage. In particular, note that a singleton set S in t'_j cannot be aligned with a non-empty non-singleton set $S' = \{a, b, \dots\}$ in a true match because the characters in t matching a, b in this true match must then appear in S and S would no longer be singleton.

Step 2. For each candidate match i and for all t'_j, p'_j, we check whether the placement of p'_j starting at $t'_j[i]$ satisfies the following property: if a singleton in p'_j is aligned with a singleton in t'_j, then the two characters corresponding to these singleton sets are identical. This is easily done in $O(s\log^2 s)$ time overall using a simple reduction to the Wildcard Matching problem and using the algorithm in Section 3. Candidate matches which violate this property are discarded. All true matches will survive this stage as well.

Step 3. Recall that each character in each set in t is in a singleton set in at least one of the t'_j's. We *associate* each character in t with exactly one of the t'_j's in which it appears as a singleton.

For each remaining candidate match i and for each t'_j, p'_j, we compute the following quantity: the number $n(i,j)$ of singletons associated with t'_j which are aligned with singletons in p'_j in the placement of p'_j starting at $t'_j[i]$. This is easily done in $O(s\log^2 s)$ time overall using Corollary 2.

Finally, we claim that, analogous to Lemma 7, a claimed match i is a true match if and only if $\sum_j n(i,j)$ equals the total sum of the sizes of the pattern sets. The total time taken is $O(s\log^2 s)$.

7.1 Computing Good Character Shifts

Our aim is to obtain $O(\log s)$ collections of character shifts in $O(s \log^2 s)$ time so that the two properties mentioned earlier are indeed satisfied. Each successive collection of shifts we obtain has the property that a constant fraction of the characters in t which have not yet appeared as singletons in previous collections of shifts appear as singletons in this collection.

Definitions. We associate an integer *weight* $w(a)$ with each instance a of each character in Σ which appears in some set in t. Note that different instances of the same character could have different weights. The term *weight-sum* denotes the sum $\sum 2^{w(a)}$, where the sum is over all instances a of all characters in t. A character in t is *live* if it has not appeared as a singleton in any of the previous collections of shifts. Let t' be obtained from t by a character shifting step. The *weight* of set $t'[i]$ is defined as $(\prod_{live\ a\in t'[i]} 2^{w(a)})\times$ $(\sum_{non-live\ a\in t'[i]} 2^{w(a)})$, with the first term missing if there are no live characters in $t'[i]$ and the second term missing if there are no non-live characters in $t'[i]$. The weight $wt(t')$ of t' is defined to be the sum of the weights of non-empty sets in t'. Let s' denote the sum of the sizes of the sets in t.

The intuition for defining the weight of t' as above is to penalize live characters occurring with other live or non-live characters, while not putting any constraint on non-live characters. This ensures that a small weight for t' forces a good fraction of the live characters to occur as singleton but allows non-live characters to occur with other non-live characters.

Successive runs of the following key procedure shall give us the successive collections of shifts.

The Key Procedure. Given a weight assignment to each instance of each character in t, with weight-sum $W = O(s)$, this procedure uses a collection of character shifts to obtain t' and p' with the following properties:

- The weight of t' is at most $\left(1+\frac{1}{k}\right)W$, for some constant $k > 2$.
- $|t'|, |p'| = O(s)$.

This procedure takes $O(s\log s + L\log^2 s)$ time, where L is the number of live characters in the current run of this procedure. As we will show, L decreases geometrically with each run. Next, we describe how this procedure indeed gives us the requisite t'_i, p'_i's satisfying the two properties mentioned earlier.

Using the Above Procedure. We begin with character weights equal to 2 and weight-sum $4s' = O(s)$. Next, we perform $O(\log s)$ iterations, modifying the character weights in each iteration. In general, suppose $i-1$ iterations have been performed and $t'_1, \dots, t'_{i-1}$ have been determined. We identify those instances of characters in t which appear in singleton sets in at least one of $t'_1, \dots, t'_{i-1}$; these characters now get weight 1. We then give new weights to the remaining characters by weighting then equally and fixing the weight-sum restricted to these characters to be between $4s'$ and $8s'$ (this range is needed if we want character weights to be integral); the weight-sum W obtained by including weight 1 characters would then be at most $10s' = O(s)$. Note that $W/s' \geq 4$. Next, we call the above procedure with these weights to obtain t'_i. Finally, we stop when all instances of all characters in t satisfy the singleton appearance property. The following lemma shows that L decreases geometrically and therefore $O(\log s)$ iterations will suffice. The total time taken by the above procedure is thus $O(s\log^2 s)$.

Lemma 8. *The text t'_i obtained in the ith iteration of the above procedure has the property that all but a $\frac{2}{k}$ fraction of the live characters (i.e., those which have not appeared in singleton sets in any of $t'_1, \dots, t'_{i-1}$) appear in singleton sets in t'_i.*

Corollary 9. *After $O(\log s)$ iterations, each instance of each character would have appeared in a singleton set in some t'_i.*

It remains to describe how the above key procedure works.

7.2 Computing Small Weight t': The Row Rotation Problem

We formulate the problem of determining shifts to the characters in Σ to obtain a t' with small weight as follows. We will restrict our shifts to size $Y = O(s)$. Consider a $\Sigma\times Y$ size matrix A. This matrix has two kinds of entries: empty and non-empty. $A[e, j]$ is empty if $e \in \Sigma$ does not occur in text set $t[j]$, and $A[e, j]$ equals the weight of the instance of character e in the set $t[j]$, otherwise. Clearly, the total number of non-empty entries in A and their weight-sum W are both $O(s)$. The aim is to find collections of circular shifts of the rows (these circular shifts can then be linearized) so that the properties mentioned earlier are satisfied. We achieve this using a deterministic $O(s\log s + L\log^2 s)$ time algorithm with 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.

7.2.1 Shifting Megarows

First, we describe how a good relative shift between two megarows v, w is determined. Our algorithm needs the following definitions.

Megarows as Vectors. We define a vector v of weights for each megarow as follows. Consider the non-empty entries $a_1, a_2 \dots$, if any, in the ith position in the shifted rows forming the megarow; then the weight stored in $v[i]$ is:

1. 0, if all entries in this position are empty.
2. $(\prod_{live\ a_j} 2^{w(a_j)})\times(\sum_{non-live\ a_j} 2^{w(a_j)})$, with the first term (second term, respectively) missing if there are no live (non-live, respectively) entries.

The weight, $wt(v)$, of the megarow corresponding to v is $\sum_i v[i]$. Vector v is called the *megarow weight vector* or *megarow-vector* for short. We will say that $v[i]$ is *live* if at least one of $a_1, a_2, \dots$ is live. One technical issue before we proceed is that of representing a megarow-vector. We represent such a vector as a list of non-zero entries.

Let v, w, respectively, be the megarow-vectors for the two megarows being combined. 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 with megarow-vector u. Our aim is to choose the shift w' of w so as to keep the weight of u as small as possible. In fact, we will be able to keep the weight of u below $wt(v)+wt(w)+\frac{wt(v)wt(w)}{Y}$. Note that the new megarow-vector u obtained by combining the megarows corresponding to v and w will have the following properties.

1. $u[i] \leq v[i] \times w'[i]$, for i such that both $v[i]$ and $w'[i]$ are non-zero and at least one of $v[i], w'[i]$ is live.

2. $u[i] = v[i] + w'[i]$, otherwise.

Contributions to $wt(u) - wt(v) - wt(w)$ come from just the first term above and this is the *excess* which we seek to minimize in our algorithm. In the algorithm, this excess will be upper bounded at each step by a *potential*, which we shall define shortly.

Our Algorithm. Let sh denote the amount by which w needs shifting to obtain w'. We will determine sh by determining the $O(\log s)$ bits in the binary representation of sh one by one in increasing order of significance. In other words, we will first determine whether sh is even or odd. If sh is committed to the even option in this step then we will determine whether sh is 0 (mod 4) or 2 (mod 4) in the next step. And if sh is committed to the odd option in the first step then we will determine whether sh is 1 (mod 4) or 3 (mod 4) in the next step. This is repeated until sh is completely determined. Note that there are two options available at each step and we will pick the one which leads to the smaller increase in weight. We explain this procedure in further detail below.

We run a number of iterations (this number will be specified shortly). Iteration $j \geq 1$ computes the jth least significant bit of sh and is performed in the following setting. Consider non-empty locations in v and partition these into residue classes modulo 2^{j-1}. This results in 2^{j-1} sets of locations (e.g., for $j = 1$, there is only one set comprising all non-empty locations, and for $j = 2$, there are two sets, one comprising the even locations and another comprising the odd locations); call these $V_1 \ldots V_{2^{j-1}}$. Let $W_1 \ldots W_{2^{j-1}}$ denote the analogous sets for w. Going into the jth iteration, we would have computed a matching between the V's and the W's (initially, i.e., for $j = 1$, there is only one set V_1 for v and this is matched to the only set W_1 for w). For simplicity, we assume that V_l is matched to W_l, $1 \leq l \leq 2^{j-1}$, using appropriate permutations to rename sets if necessary. Further, we would have a *potential* for this matching defined as the following quantity summed over all l such that either V_l or W_l contains a live location:

$$\begin{aligned}&(\text{sum of values in non-zero locations in } V_l)\\ \times&(\text{ sum of values in non-zero locations in } W_l)\end{aligned}$$

We now describe how to perform the jth iteration so that a new matching is computed for sets corresponding to residue classes modulo 2^j. The potential of this resulting matching will be at most half the potential of the original matching and the time taken in this process will be proportional to the number of live locations in v and w put together (this will be explained shortly).

Note that each residue modulo 2^{j-1} corresponds to exactly one of two possible residues modulo 2^j (i.e., $l(\text{mod } 2^{j-1})$ equals either $l(\text{mod } 2^j)$ or $l + 2^{j-1}(\text{mod } 2^j)$). Based on this observation, we split V_l and W_l into two sets each; call these sets V_l^1, V_l^2 and W_l^1, W_l^2, respectively. Setting the jth bit of sh to 0 corresponds to matching V_l^1 with W_l^1 and V_l^2 with W_l^2, for each l. And setting the jth bit of sh to 1 corresponds to matching V_l^1 with W_l^2 and V_l^2 with W_l^1, for each l. It is easily seen that one of these two choices will lead to a potential which is at most half the potential of the previous matching. We perform $\log Y = \Theta(\log s)$ iterations. The final potential will then be $\frac{1}{Y}$ times the initial potential, which is $wt(v)wt(w)$. It is easily seen that the weight $wt(u)$ of the new megarow obtained by shifting w by sh and combining it with v is at most

$$wt(v) + wt(w) + \frac{wt(v)wt(w)}{Y}$$

.

Implementation Details: Tries. The main issue in implementing the above algorithm is the maintenance of the sets and their associated potentials so that the jth iteration can be performed in time proportional to the number of live locations in v and w put together. The key observation which makes this possible is the fact that computing the potential requires computing sums only over l such that either V_l or W_l contains a live location. Thus, in each iteration it suffices to maintain only sets V_l, W_l such that at least one of these has a live location. Clearly, the number of such sets is bounded by the number of live locations in v and w put together. Maintenance of these sets is easily handled with the following preprocessing.

Initially, we build a trie of all non-empty locations in v. For each non-empty location in v, the reverse of its binary representation is used to build the trie. A similar trie is built for w. Note that each non-empty set V_l will be identical to the set of leaves in some appropriate subtree of the trie for v, and similarly for w. Thus, instead of maintaining the V_l's, we will maintain nodes in the trie. Each node in the trie will maintain two quantities, the sum of the values of the leaves in its subtree, and a binary string identifying the path from the root to that node. Using this information, it is easily seen that given a particular V_l (i.e., a node in the trie), the sets V_l^1, V_l^2 along with the associated potentials can be obtained in $O(1)$ time (using table look-up if necessary to identify the nodes for the new sets).

Time Complexity. The tries are built in time linear in the number of non-empty items in v and w using constant time LCA computation on consecutive items [18]. The time taken in trie building is thus $O(\max\{wt(v), wt(w)\})$. Subsequent to this, the jth iteration runs in time bounded by the number of live locations in v and w put together. The number of iterations performed is $O(\log s)$. The total time over all iterations is thus $O(\max\{wt(v), wt(w)\} + \#\mathit{live\ locations} \times \log s)$.

7.2.2 Pairing Megarows and Analysis

The order is which megarows are paired depends on the weights of the associated vectors. Recall $W = O(s)$ denotes the sum of the megarow-vector weights at the very beginning (because, each megarow is just a single row at the very beginning). We need to show that the final megarow-vector weight is $(1 + \frac{1}{k})W$. We also need to show that the total time taken is $O(s \log^2 s)$.

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. It is easily seen that the new megarow which results from combining two paired megarows in the same category will be in a strictly higher category. As we will show, megarow-vector weights will always be $O(s)$. It follows that the number of phases will be $\Theta(\log s)$.

Bounding Megarow-Vector Weights. The megarow-vector weights are bounded by the following lemma.

Lemma 10. *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* $\frac{N}{O} \leq (1 + \frac{2^{j+1}}{Y})$.

By choosing $Y = \Theta(s)$ appropriately, we get:

Corollary 11. *The final megarow-vector weight is* $W(1+\frac{1}{k})$*, for an appropriately chosen constant* $k > 2$.

Time Complexity. It remains to determine the time taken by the above algorithm. Recall that the process of merging two megarows v, w takes time $O(\max\{wt(v), wt(w)\}\} + \#live\ locations \times \log s)$, where *#live locations* is the number of live locations in v and w put together. Since, by Corollary 11, the sum of megarow-vector weights is always bounded by $O(s)$, each phase takes time $O(s+L\log s)$. Since the number of phases is $O(\log s)$, the total time taken is $O(s \log s + L \log^2 s)$, as required.

8. REFERENCES

[1] L. Adleman, M. Huang. Recognizing Primes in Random Polynomial Time. Proceedings of the *19th ACM Symposium on Theory of Computing*, 1987, pp. 462–469.

[2] D. Cardoze, L. Schulman. Pattern Matching for Spatial Point Sets. Proceedings of the *39th IEEE Symposium on Foundations of Computer Science*, 1998, pp. 156–165.

[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] 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.

[7] 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.

[8] S. Goldwasser and J. Kilian, Almost all primes can be quickly certified. Proceedings of *18th Annual IEEE Symposium on Foundations of Computer Science*, 1986, pp. 316–329.

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

[10] 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.

[11] 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.

[12] 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.

[13] A. Kalai, Efficient Pattern Matching with Don't Cares. Proceedings of the *13th ACM-SIAM Symposium on Discrete Algorithms*, 2002, pp. 655-656.

[14] S. Muthukrishnan, Detecting false matches in string matching algorithms. Proceedings of the *4th Conference on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, 684, Springer-Verlag, 1993, pp. 164–178.

[15] S. Muthukrishnan, New results and open problems related to non-standard stringology. Proceedings of the *6th Conference on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, 937, Springer-Verlag, 1995, pp. 298–317.

[16] M. O. Rabin, A probabilistic algorithm for testing primality. *Journal of Number Theory*, 12, 1980, pp. 128–138.

[17] A. Shokrollahi, J. Buhler, V. Stemann, Fast and precise computations of discrete fourier transforms using cyclotomic integers. Proceedings of the *29th ACM Symposium on Theory of Computing*, 1997, pp. 40-47.

[18] B. Schieber, U. Vishkin, On finding lowest common ancestors in trees: simplification and parallelization. *SIAM Journal on Computing*, 17, 1988, pp. 1253–1262.