Faster Pattern Matching with Character Classes using Prime Number Encoding

Chaim Linhart * , Ron Shamir

School of Computer Science, Tel Aviv University, Tel-Aviv 69978, ISRAEL

Abstract

In *pattern matching with character classes* the goal is to find all occurrences of a pattern of length m in a text of length n, where each pattern position consists of an allowed set of characters from a finite alphabet Σ. We present an FFT-based algorithm that uses a novel prime-numbers encoding scheme, which is $\log n / \log m$ times faster than the fastest extant approaches, which are based on boolean convolutions. In particular, if $m^{|\Sigma|} = n^{O(1)}$, our algorithm runs in time $O(n \log m)$, matching the complexity of the fastest techniques for wildcard matching, a special case of our problem. A major advantage of our algorithm is that it allows a tradeoff between the running time and the RAM word size. Our algorithm also speeds up solutions to approximate matching with character classes problems — namely, matching with k mismatches and Hamming distance, as well as to the *subset matching* problem.

Key words: Pattern matching with character classes, Subset matching, Hamming distance, FFT-based pattern matching

* Corresponding author.
Email addresses: chaiml@post.tau.ac.il (Chaim Linhart), rshamir@post.tau.ac.il (Ron Shamir).

1 Introduction

Generic pattern matching problems require finding all occurrences of a pattern p in a text t. Throughout this paper, we denote by m and n the length of the pattern and the text, respectively ($m < n$). In the classical string matching problem both p and t are strings over a finite alphabet $\Sigma = \{a_1, \ldots, a_\sigma\}$ of size σ. A myriad of efficient algorithms have been developed over the years, the fastest of which solve this problem in linear time, such as the Knuth-Morris-Pratt [1] and Boyer-Moore [2] algorithms.

1.1 Matching with Don't-Cares

A more general matching problem is obtained when we allow the pattern and the text to contain don't-care characters, or wildcards, denoted '$*$', which match all symbols in Σ. Formally:

Matching with don't-cares: Given a pattern p and a text t, which may contain don't-cares, find all occurrences of p in t. Here, p is said to occur at location i in t if: $\forall 1 \leq j \leq m$, $p[j] = t[i+j-1]$ or $p[j] =$'$*$' or $t[i+j-1] =$'$*$'.

If the number of don't-cares in the pattern is very small, the problem can be solved in linear time, for example by building a deterministic finite automaton (DFA) that detects all possible words that match the pattern. Another approach is to use the *match-count* algorithm, which finds the number of matching positions (or, equivalently, the Hamming distance) between the pattern and every length m substring of the text (see, e.g., [3, ch. 4.3]). The algorithm, first introduced by Fischer and Paterson [4], computes the contribution of each

alphabet symbol to the score independently, as follows. For the symbol $a \in \Sigma$, each occurrence of a in the text and in the pattern is replaced by the number 1, and all other symbols are encoded by 0. The number of matching a's between the pattern and every substring in the text is obtained by computing the convolution between the binary-encoded pattern and text. Using Fast Fourier Transform (FFT), the convolution can be computed in $O(n \log m)$ time under the RAM model of computation, which assumes that arithmetic operations on numbers with w bits take constant time, where $w = O(\log N)$ is the RAM word size and N is the maximal input size. Thus, the total running time of match-count is $O(\sigma n \log m)$, as it involves σ such convolutions. The algorithm can easily be extended to cope with wildcards in the pattern and in the text.

Fischer and Paterson further showed that a similar technique can be applied to solve matching with don't-cares in time $O(\log \sigma \cdot n \log m)$ [4]. Removing the dependence on σ remained an open problem until recently. Indyk introduced a randomized technique for computing boolean products, which yielded an $O(n \log m)$-time Monte Carlo algorithm for wildcard matching and other problems [5]. Kalai gave another elegant Monte Carlo algorithm with the same time complexity, based on integer codes [6]. Cole and Hariharan were the first to obtain an $O(n \log m)$-time deterministic algorithm, by encoding each symbol with a pair of rational numbers [7]. A simpler deterministic algorithm with the same time complexity was presented very recently by Clifford and Clifford [8]. All of the above algorithms compute convolutions using FFT; the main differences between them is in the way they encode the pattern and the text.

Matching with don't-cares can be generalized by allowing the pattern to contain any non-empty subset, or *class*, of characters at each position:

Matching with character classes: Given a pattern p with character classes ($p[j] \subseteq \Sigma$), and a text t, which may contain don't-cares ($t[i] \in \Sigma \cup$ '$*$'), find all occurrences of p in t. Here, p is said to occur at location i in t if:
$\forall 1 \leq j \leq m$, $t[i+j-1] \in p[j]$ or $t[i+j-1] =$'$*$'.

For example, the pattern `a[abcd]r[ab]` matches the text `abracadadrb` at locations 1 (the substring "`abra`") and 8 ("`adrb`"). W.l.o.g., we may assume that the text does not contain don't-cares — otherwise, we can add the don't-care symbol to all the character classes in the pattern, and treat it as a regular symbol in the alphabet. Matching with character classes, as well as with similar types of patterns, has been studied extensively (e.g., [9,10]). Most algorithms, however, have the same worst-case running time as the naïve algorithm — $O(nm)$. Bit-parallelism techniques improve this to $O(nm/w)$, where w is the RAM word size [11]. In general, the best worst-case performance is attained by the match-count algorithm — $O(\sigma n \log m)$.

We present an FFT-based algorithm, whose running time depends on the parameter $\kappa = \log_\sigma (\log n / \log m)$. If $\kappa < 1$, its time complexity is $O(\sigma^{1-\kappa} n \log m)$, which is $\log n / \log m$ times faster than match-count; if $\kappa = 1$, i.e., $m^\sigma = n$, our algorithm computes a single convolution, matching the $O(n \log m)$ running time of the fastest wildcard matching algorithms; and when $\kappa > 1$, our method runs in time $O(n \log (m/\kappa))$, asymptotically converging to the optimal linear-time performance of classical string matching methods as κ approaches

infinity. Notably, in the latter case we obtain an improvement for wildcard matching. Our algorithm uses a novel encoding scheme that is based on large prime numbers. The basic idea is to encode the text and the pattern in such a way that at match locations their convolution is congruent to 0 modulo some large number M. Unlike other methods, prime code exploits the entire RAM word, admitting improved performance for a longer word size. Table 1 summarizes the main results presented in this paper.

Table 1

Summary of the main results described in this paper.

Problem	Previous complexity	Prime-code compleixty	
Matching with character classes	$O(\sigma n \log m)$ [4]	$O(\sigma^{1-\kappa} n \log m)$	if $m^\sigma = n^{\omega(1)}$
		$O(n \log(\frac{m}{\kappa}))$	if $m^\sigma = n^{o(1)}$
Hamming distance with char. classes	$O(\sigma n \log m)$ [4]	$O(n\sigma(1 + \log^2 m / \log n))$	if $m^\sigma = n^{\omega(1)}$
		$O(n(\log m + \sigma))$	if $m^\sigma = n^{o(1)}$
Subset matching (Monte Carlo)	$O(\sigma n \log(\sigma n))$ [5]	$O(\sigma n \log m)$	if $m^\sigma = n^{\omega(1)}$
		$O(n \log n \log(\frac{m}{\kappa}) / \log m)$	if $m^\sigma = n^{o(1)}$

1.3 Approximate Matching with Character Classes

In many practical scenarios, one would like to find not only perfect matches, but also locations at which the pattern approximately matches the text, that is, matches it up to a specified small distance. A commonly used metric is the number of mismatched pattern positions, or Hamming distance. Returning to the previous example, the pattern `a[abcd]r[ab]` matches the text `abracadadrb` at location 4 (the substring "`acad`") with two mismatches (positions 3 and 4). Finding all pattern occurrences with at most k mismatches is often referred to as the *k-Mismatches problem*. The fastest algorithm for string matching with k mismatches runs in time $O(n\sqrt{k \log k})$ [12]. If the pat-

tern contains character classes, the most efficient algorithm is match-count, running in time $O(\sigma n \log m)$. In fact, match-count provides more information than just the k-mismatches — as explained earlier, it computes the number of mismatches at every text location. We call this the *Hamming distance problem*. In the restricted case of a pattern that contains only single symbols and don't-cares, but not other classes of characters, Abrahamson [10] showed that the Hamming distance can be computed in time $O(n\sqrt{m \log m})$, which is faster than match-count when $\sigma > \sqrt{m/\log m}$. The algorithm we developed for matching with character classes can also solve the k-mismatches and Hamming distance problems with small additional cost. This is an asymptotic improvement over the match-count algorithm.

1.4 Subset Matching

In the subset matching problem, both the pattern and the text are composed of character classes. According to the original definition by Cole and Hariharan [13], the pattern matches the text if every character class in the pattern is a subset of the corresponding character class in the text. For consistency with the definition of matching with character classes, we shall switch the roles of the pattern and the text, and obtain the following equivalent problem:

Subset Matching: Given a pattern p and a text t, both consisting of character classes ($p[j], t[i] \subseteq \Sigma$), find all occurrences of p in t. Here, p is said to occur at location i in t if: $\forall 1 \leq j \leq m$, $t[i+j-1] \subseteq p[j]$.

Obviously, matching with character classes is a special case of subset matching. Let s be the total number of characters in the pattern and text, i.e., $s = \sum_{j=1}^{m} |p[j]| + \sum_{i=1}^{n} |t[i]|$. The most efficient algorithms for subset matching

are an $O(s \log s)$ Monte Carlo algorithm due to Indyk [5], and an $O(s \log^2 s)$ deterministic algorithm by Cole and Hariharan [7]. Since s might be as large as $\sigma(n+m)$, the above methods have worst-case running times of $O(\sigma n \log(\sigma n))$ and $O(\sigma n \log^2(\sigma n))$, respectively. We develop a randomized variant of our prime-code technique that yields a Monte Carlo algorithm for subset matching. Assuming $\sigma = O(m)$, the algorithm runs in time $O(\sigma n \log m)$ if $\kappa < 1$, $O(n \log n)$ if $\kappa = 1$, and $O(n \log n \log(m/\kappa)/\log m)$ if $\kappa > 1$.

1.5 Motivation

Character classes are commonly used in regular expressions and in many applications of pattern matching in various fields. We briefly describe here two such applications in computational biology. The alphabet in both applications consists of the four DNA bases — $\Sigma = \{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}$.

Transcription factors (TFs) are specialized proteins that bind to regulatory regions in the DNA and control gene expression. A TF usually binds to many different DNA segments that share a common pattern, or *motif*, characteristic of the TF. Such binding-site motifs are often modeled using patterns with character classes. For example, *p53*, the most frequently mutated tumor suppressor in human cancers, binds to two repeats of `[AG][AG][AG]C[AT][AT]G[CT][CT][CT]` [14]. In a typical setting, the TF binding pattern is short ($10 \leq m \leq 20$), and the goal is to efficiently locate all its occurrences in many long regulatory regions ($n \approx 10^8$).

The second application is in the design of degenerate primers for Polymerase Chain Reaction (PCR) experiments. PCR is a technique for amplifying a specific region of DNA, so that enough copies of it are available for testing or

sequencing. The first step in PCR is to synthesize two DNA segments, or *primers*, lying on opposite sides of the target region. A PCR primer is called degenerate if some of its positions have several possible bases. Thus, a degenerate primer can be described as a pattern with character classes. Degenerate primers can be used to amplify several related genomic sequences in a single PCR experiment. We studied the computational problem of designing highly degenerate primers [15], and applied our algorithms in experiments for studying the human and canine olfactory receptor genes [16,17]. A common problem in the design of degenerate primers is to verify that the primers do not bind to DNA regions others than those they are meant to amplify. Thus, one needs to search for all occurrences of a candidate primer, typically of length $20 \leq m \leq 30$, in the entire genome ($n \approx 6 \cdot 10^9$ in human). One may also want to allow a small number of mismatches (e.g., $k = 3$), as the PCR technique usually tolerates a few mismatches.

Subset matching is applicable in many pattern matching scenarios, such as geometric pattern matching and general pattern matching. Most notably, Cole and Hariharan showed that tree pattern matching, an important problem which has been studied extensively, can be reduced to subset matching in linear time [13]. In computational biology, subset matching can be applied to search for conserved TF binding sites in aligned sequences of multiple species. The straightforward solution is to search for the occurrences of the TF's motif in each species separately, as described earlier, and then check which locations match the motif in all species. An alternative approach is to combine the sequences into a single consensus sequence, in which each position contains the set of bases that appear in that position in one or more species, and apply subset matching to search for the motif in the consensus sequence.

2 Preliminaries

All the algorithms described in this paper assume the RAM model, wherein standard arithmetic on w bit numbers is performed in constant time. Following standard practice, we shall assume that the word size is $w = O(\log n)$ (see, e.g., [5,6]).

Convolution: The convolution, or cross-correlation, of two vectors a, b is the vector $a \oplus b$ such that $(a \oplus b)[i] = \Sigma_{j=1}^{|a|} a[j]b[i+j-1]$ for $1 \leq i \leq |b|-|a|+1$.

Given a pattern p of length m and a text t of length n ($m < n$), both encoded using numbers with w bits, the convolution $p \oplus t$ can be computed in $O(n \log m)$ time, as follows. First, the text is split into n/m pieces of length $2m$, with overlap m between consecutive pieces. The convolution between the pattern and each piece of the text is then computed using FFT in time $O(m \log m)$ per piece (as in [4]).

3 Matching with Character Classes

In this section we describe our encoding scheme and how it can be applied to solve pattern matching with character classes. Since our algorithm is based on computing convolutions on segments of length $2m$, we may assume w.l.o.g. that $\sigma \leq 2m$, as each $2m$-long piece of text contains at most $2m$ distinct symbols.

3.1 Prime Code

A prime code assigns to each symbol $a_i \in \Sigma$ a distinct prime number p_i, where $p_1 < p_2 < \ldots < p_\sigma$ (notice that we use p_i to denote the i-th prime number, whereas $p[i]$ is the character class at position i in the pattern). Denote $M = p_1 \cdot \ldots \cdot p_\sigma$. We further require that all primes are larger than m (i.e., $p_1 > m$). We first describe how such prime numbers can be found, and then explain how to encode the pattern and the text.

Finding Prime Numbers $p_1, \ldots, p_\sigma > m$: Following are well known bounds on the number $\pi(x)$ of primes less than or equal to x [18]:

$$\forall x \geq 17 \quad \frac{x}{\ln x} < \pi(x) < 1.26\frac{x}{\ln x}$$

Using these bounds, for $m \geq 17$ we get:

$$\pi(5m\ln m) - \pi(m) > \frac{5m\ln m}{\ln(5m\ln m)} - 1.26\frac{m}{\ln m} > \frac{5m\ln m - 2.6m}{2\ln m} > 2m \geq \sigma$$

Thus, if we search for prime numbers between $m+1$ and $5m\ln m$, we are guaranteed to find at least σ prime numbers, as required. Since testing for primality takes polynomial time (i.e., testing whether x is prime takes polylog(x) time) [19], the prime numbers we seek can be found in $O(m \cdot \text{polylog}(m))$ time. Alternatively, we could apply Eratosthenes' sieve to obtain all the primes up to $5m\ln m$ in time $O(m\log m\log\log m)$. This can be improved to $o(m\log m)$ time using modern sieves, such as the sieve of Atkin [20]. Notice that each of the primes we obtain is a number with at most $\log_2(5m\ln m) < 2\log_2 m$ bits. The prime numbers depend only on m — if we are given a list of equal-length patterns, this step of the algorithm needs to be performed only once.

Text Code: The symbol a_i in the text is encoded by the integer M/p_i.

Pattern Code: A character class $[a_{i_1}, \ldots, a_{i_c}]$ in the pattern is encoded by an integer $n_{\mathcal{S}}$, where $\mathcal{S} = \{i_1, \ldots, i_c\}$, s.t.:

$$n_{\mathcal{S}} \equiv \begin{cases} 0 \pmod{p_i} & \forall i \in \mathcal{S} \\ 1 \pmod{p_j} & \forall j \notin \mathcal{S} \end{cases} \tag{1}$$

The Chinese Remainder Theorem (CRT, in short) guarantees that such integers exist (see, e.g., [21, ch. 31.5]). In fact, for each subset $\mathcal{S} \subseteq \{1, \ldots, \sigma\}$ there exists a single integer $0 \leq n_{\mathcal{S}} < M$, for which Equation 1 holds. Moreover, this integer can be found using the CRT, as follows. For each j $(1 \leq j \leq \sigma)$, we obtain a pair of integers r_j, q_j s.t.: $r_j p_j + q_j(M/p_j) = 1$. These integers can be computed using Euclid's gcd algorithm in time $O(\log p_j)$. Denoting $c_j = q_j(M/p_j)$, it follows from the CRT that:

$$n_{\mathcal{S}} = \Sigma_{j \notin \mathcal{S}}\ c_j = 1 - \Sigma_{j \in \mathcal{S}}\ c_j \bmod M$$

Hence, given the coefficients $c_1, \ldots, c_\sigma$, a character class with c symbols is encoded in linear time. Let s_p denote the total number of characters in the pattern, i.e., $s_p = \Sigma_{j=1}^{m} |p[j]|$. Encoding the entire pattern takes $O(s_p)$ time, plus $O(\sigma \log p_\sigma) = O(m \log m)$ (since $\sigma \leq 2m$ and $p_\sigma \leq 5m \ln m$) for computing the c_j's, which can be done in pre-processing, as they do not depend on the content of the pattern. The attentive reader may have noticed that we ignored a crucial problem — the numbers we are dealing with might be too large to fit into a single RAM word. We will address this issue in the next section.

Our basic algorithm for pattern matching with character classes, called PMCC, is outlined in Figure 1.

INPUT: Pattern p with character classes, text t over alphabet Σ, $\sigma = |\Sigma|$

OUTPUT: All occurrences of p in t

Algorithm PMCC:

1. Pre-processing:

   1a. Find σ prime numbers — $p_1, \ldots, p_\sigma > m$

   1b. Set $M \leftarrow p_1 \cdot \ldots \cdot p_\sigma$

   1c. Compute coefficients $c_1, \ldots, c_\sigma$ using the CRT:

   $$c_i \equiv 1 \pmod{p_i} \quad \text{and} \quad c_i \equiv 0 \pmod{p_j} \text{ for } j \neq i$$

2. Encode the pattern and the text using a prime code:

   2a. Text: Replace the symbol a_i by M/p_i

   2b. Pattern: Replace the character class $[a_{i_1}, \ldots, a_{i_c}]$ by $n_{\{i_1,\ldots,i_c\}}$:

   $$n_{\{i_1,\ldots,i_c\}} = 1 - (c_{i_1} + \ldots + c_{i_c}) \bmod M$$

3. Compute the convolution $p \oplus t$ using FFT

4. Report a match at location i iff $(p \oplus t)[i] \equiv 0 \pmod{M}$

Fig. 1. Algorithm PMCC for pattern matching with character classes.

Theorem 1 *If $m^\sigma = n^{O(1)}$, algorithm PMCC solves pattern matching with character classes using a single convolution in time $O(n \log m)$.*

Proof: We first prove that PMCC produces the correct output. Let us compare the pattern to the substring at location i in the text. The value of the convolution at this location is: $(p \oplus t)[i] = \Sigma_{j=1}^{m} p[j]t[i+j-1]$. Let $[a_{i_1}, \ldots, a_{i_c}]$

be the character class at position j in the pattern, and let a_k be the symbol at position $i+j-1$ in the text. It is easily seen that:

$$p[j]t[i+j-1] = n_{\{i_1,\ldots,i_c\}} \cdot M/p_k \equiv \begin{cases} 0 & \text{, if } a_k \in \{a_{i_1}, \ldots, a_{i_c}\} \\ M/p_k & \text{, otherwise} \end{cases}$$

(all congruences are modulo M). Denote by e_k the number of times the symbol a_k in the text does not match the corresponding character class in the pattern, when the pattern is aligned against text location i. Thus, $(p \oplus t)[i] \equiv R$, where $R = \Sigma_{k=1}^{\sigma}\ e_k \cdot (M/p_k)$. Since $p_k > m$ for all k, we get $R < M/m \cdot \Sigma_{k=1}^{\sigma} e_k$. Obviously, $\Sigma_{k=1}^{\sigma} e_k \leq m$, so $R < M$. Of course, $R \geq 0$, and this inequality strictly holds iff $\exists k, e_k > 0$. The correctness of the algorithm immediately follows.

We now analyze the running time of PMCC. As explained in Section 3.1, step 1 can be performed in time $O(m \log m)$, and encoding the text and the pattern in step 2 takes $O(n + s_p)$ time. Step 4 takes $O(n)$ time. Henceforth we shall ignore these pre-processing and linear-time phases of the algorithm, and focus on step 3, which determines the overall time complexity. Since we showed that $\log_2 p_i < 2 \log_2 m$ (Section 3.1), it follows that $\log_2 M = \Sigma_{k=1}^{\sigma} \log_2 p_k < 2\sigma \log_2 m$. Thus, for $m^\sigma = n^{O(1)}$, we get $\log_2 M = O(\log n)$, i.e., M fits into a single machine word, so the convolution in step 3 can be computed in time $O(n \log m)$, as required.

■

We now show how to adjust the PMCC algorithm so that it could solve instances with $m^\sigma = n^{\omega(1)}$, and how to improve its performance when $m^\sigma = n^{o(1)}$.

Theorem 2 *Pattern matching with character classes can be solved in time:*

$$\begin{cases} O(\sigma^{1-\kappa} n \log m) \text{ , if } \ 0 \leq \kappa \leq 1 \\ O(n \log m) \qquad \text{ , if } \ \kappa = 1 \\ O(n \log(m/\kappa)) \ \text{ , if } \ \kappa \geq 1 \end{cases}$$

where $\kappa = \log_\sigma (\log n / \log m)$*, or, more generally,* $\kappa = \log_\sigma (w / \log m)$*, where* w *is the RAM word size.*

Proof: The Case $m^\sigma = n^{\omega(1)}$*:* In order to ensure that all the numbers the algorithm computes do not exceed $O(\log n)$ bits, we apply a standard trick — we partition Σ into smaller alphabets, $\Sigma = \bigcup \Sigma_j$, each of size at most $\log n / \log m$. For each of these $\lceil (\log m / \log n)\sigma \rceil$ alphabets, we solve the problem using PMCC, ignoring all symbols that are not in the active alphabet — such symbols in the text are replaced by the symbol '$*$', which is also added to all the character classes in the pattern, as well as to the alphabet. Finally, we report a match at each location for which a match was found over all the alphabets. Denoting $\kappa = \log_\sigma (\log n / \log m)$, the total running time is $O(\sigma \log m / \log n \cdot n \log m) = O(\sigma^{1-\kappa} n \log m)$.

The Case $m^\sigma = n^{o(1)}$*:* In this case, PMCC utilizes only a small part of the RAM word. We can improve its running time by avoiding this waste, as follows. The main idea is to work on κ-tuples. We first rename the pattern using the alphabet Σ^κ, padding the pattern with character classes that consist of the entire alphabet, if required. The new pattern is a pattern with character classes over the new alphabet (notice that this trick does not work for a pattern with don't-cares — renaming it with Σ^κ results in a pattern with character classes, not only single symbols and don't-cares). Next, we rename the text using Σ^κ, each time starting at a different offset $0 \leq i < \kappa$. For each offset, we run

PMCC and report the matches. Since the text and the pattern are of length n/κ and m/κ, respectively, and all the numbers involved fit into a RAM word ($\log_2 M < 2\sigma^\kappa \log_2 (m/\kappa) = O(\log n)$), each offset is handled in time $O(n/\kappa \cdot \log(m/\kappa))$. The total time complexity is therefore $O(n \log(m/\kappa))$.

■

3.3 Approximate Matching

In this section we describe simple post-processing procedures that can be applied to our PMCC algorithm in order to solve two approximate matching with character classes problems — Hamming distance and matching with k mismatches.

3.3.1 Hamming Distance

Interestingly, the prime-code convolution vector $p \oplus t$ contains more information than merely the locations of the matches. As we shall now show, the number of mismatches at every location in the text can easily be derived from it, thus computing the Hamming distance for patterns with character classes more efficiently than match-count.

Theorem 3 *The Hamming distance for patterns with character classes can be computed in time:*

$$\begin{cases} O(n(\sigma^{1-\kappa} \log m + \sigma)) = O(n\sigma(1 + \log^2 m / \log n)) & \text{, if } \ 0 \leq \kappa < 1 \\ O(n(\log m + \sigma)) & \text{, if } \ \kappa \geq 1 \end{cases}$$

Proof: Recall that for a fixed location i in the text: $(p \oplus t)[i] \equiv R \pmod{M}$, where $R = \Sigma_{k=1}^{\sigma} e_k(M/p_k)$, and e_k is the number of mismatches for the symbol a_k in the text. Since $R \equiv e_k(M/p_k) \pmod{p_k}$, we get: $e_k = R \cdot (M/p_k)^{-1} \bmod p_k$. (Notice that the modular inverse $(M/p_k)^{-1}$ is the integer q_k we computed earlier for encoding the pattern). As described in Section 3.2, if $m^\sigma = n^{O(1)}$ all the numbers computed by our matching algorithm fit into a machine word, so we can compute e_k from $(p \oplus t)[i]$ in constant time. If $m^\sigma = n^{\omega(1)}$, the algorithm performs a separate convolution for each partial alphabet Σ_j; we thus calculate e_k from the convolution vector we computed for the partial alphabet that contains a_k. Therefore, in both cases the Hamming distance $\Sigma_k e_k$ at every text location can be calculated in total time $O(\sigma n)$, given the convolution vector(s). In fact, we can compute a weighted Hamming distance — $\Sigma_k w_k e_k$, where each mismatched text symbol is assigned a pre-defined weight w_k.

■

3.3.2 k-Mismatches

We would now like to find all text locations at which there are at most k mismatches ($1 \leq k < m$). Obviously, we could compute the Hamming distance and solve the problem in $O(\sigma n)$ time. However, when k is small, as is often the case in practice, there is a more efficient alternative.

Theorem 4 *Matching with k mismatches for patterns with character classes can be solved in time:*

$$\left\{ \begin{array}{ll} O(\sigma^{1-\kappa} n \log m) & \text{, if } \ 0 \leq \kappa < 1 \\ O(n \log m) & \text{, if } \ \kappa \geq 1 \end{array} \right\} \ + \ O(n \cdot \min\{\sigma, k(1 + \log \frac{\sigma}{k})\})$$

Proof: The idea is to identify which symbols have mismatches when the pattern is compared to a substring in the text. To this end, for every text location we perform a binary search using boolean queries on the value of R modulo subsets of the prime numbers, as follows.

Suppose $k < \sigma$, and let $\mathcal{T}$ be a balanced binary tree, whose leaves, ordered from left to right, are the alphabet symbols $a_1, \ldots, a_\sigma$. Each node in $\mathcal{T}$ corresponds to a subset of Σ, comprised of the symbols at the leaves of its subtree. An example is illustrated in Figure 2. We further assume that $(p \oplus t)[i]$ fits into a single RAM word, so $m^\sigma = n^{O(1)}$. We start at the root of $\mathcal{T}$, and check $R \bmod p_1 p_2 \ldots p_{\lceil \sigma/2 \rceil}$ — If it is 0, then there are no mismatches for the symbols $a_1, \ldots, a_{\lceil \sigma/2 \rceil}$, and we prune the left branch; otherwise, at least one of these symbols has mismatches, so we continue the search in the left subtree of the root. Similarly, if $R \bmod p_{\lceil \sigma/2 \rceil + 1} \ldots p_\sigma \neq 0$, we continue to the right subtree. In this manner we traverse $\mathcal{T}$ breadth-first, pruning some of the branches along the way. Each non-pruned branch contains one or more mismatched symbols (that is, at least one of the leaves in its subtree has mismatches). Thus, if the number of non-pruned branches exceeds k, there are more than k mismatched symbols, which clearly implies that there are more than k mismatches at the current text location, so we stop the search. Otherwise, we end up with at most k leaves that correspond to the mismatched symbols. We calculate the exact number of mismatches for each of these symbols, as we have done for the Hamming distance computation, and report a match if their sum does not exceed k. In the example illustrated in Figure 2 there are three mismatched symbols — a_1, a_2 and a_4, and a total of four mismatches. The total time required for the above search is proportional to the number of branches we traverse, which is $O(k + k \log \frac{\sigma}{k})$.

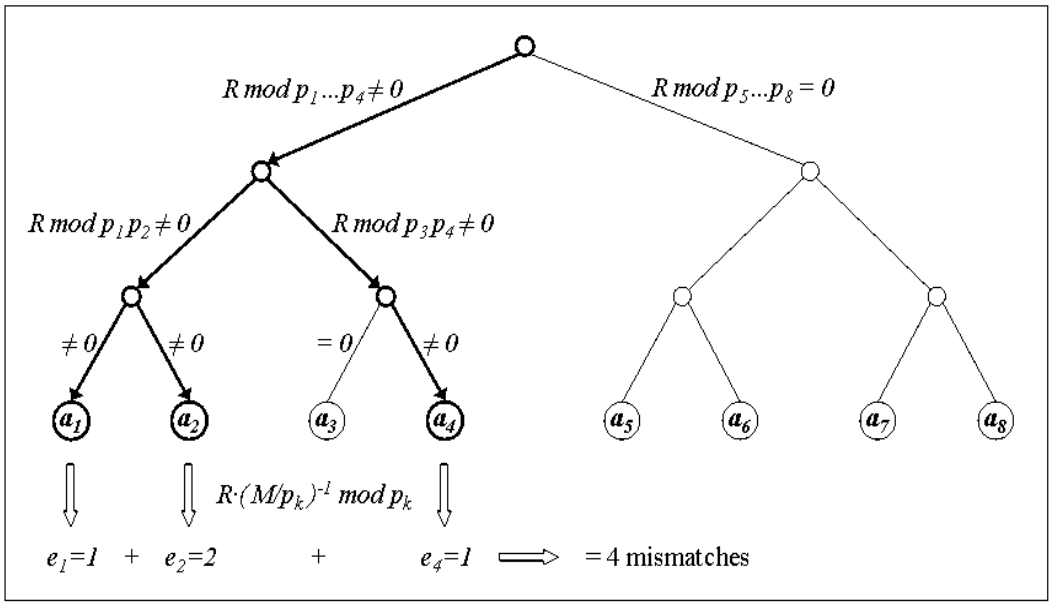

Fig. 2. Illustration of the k-Mismatches algorithm for $\sigma = 8$. For every text location, the algorithm traverses the alphabet tree $\mathcal{T}$ breadth-first, continuing the search only in edges with mismatches (thick arrows).

If $m^\sigma = n^{\omega(1)}$, we cannot perform the above search, since we do not have the value $(p \oplus t)[i]$. Instead, the alphabet is partitioned into partial alphabets — $\Sigma_1, \ldots$, of size at most $\log n / \log m$, and the matching algorithm computes $\lceil (\log m / \log n)\sigma \rceil$ convolutions, one for each partial alphabet. Thus, we can implement the same breadth-first binary search as for the case $m^\sigma = n^{O(1)}$, except that rather than starting at the root of $\mathcal{T}$, we begin the search from the level in the tree that contains the nodes that correspond to the partial alphabets $\Sigma_1, \ldots$ (or the next level if $\lceil (\log m / \log n)\sigma \rceil$ is not an integer power of 2). For example, if $(\log m / \log n)\sigma = 2$, the matching algorithm computes two convolutions (one for $\Sigma_1 = \{a_1, \ldots, a_{\sigma/2}\}$, and one for $\Sigma_2 = \{a_{\sigma/2+1}, \ldots, a_\sigma\}$), so we start the breadth-first search at the second level of $\mathcal{T}$ (whose nodes correspond to Σ_1 and Σ_2); in order to check whether $R \bmod p_1 p_2 \ldots p_{\lceil \sigma/2 \rceil}$ is 0, we use the values $(p \oplus t)[i]$ computed in the first convolution, and for $R \bmod p_{\lceil \sigma/2 \rceil + 1} \ldots p_\sigma$ we use the output of the second

convolution. Searching for the mismatched symbols at a given location takes in this case $O(k + k \log \frac{\sigma}{k} + (\log m / \log n)\sigma) = O(k(1 + \log \frac{\sigma}{k}) + \sigma^{1-\kappa})$ time.

■

4 Subset Matching

In the subset matching problem, both the pattern and the text consist of subsets of Σ (see Section 1.4 for the definition of the problem). Unlike in the previous problem, here a $2m$-long piece of text may contain more than $2m$ different symbols. However, we shall still assume that $\sigma = O(m)$; we shall not analyze the performance of the algorithm for larger alphabets. We now describe a randomized version of the prime code technique for solving subset matching. The difference lies in the way we encode the text; the pattern is encoded as in Section 3.1.

Randomized Text Code: A non-empty character class $[b_{j_1}, \ldots, b_{j_d}]$ in the text is encoded by an integer $r \cdot M/p_{\mathcal{S}_t}$, where $p_{\mathcal{S}_t} = \Pi_{k=1}^{d} p_{j_k}$, and r is a random totative [1] of $p_{\mathcal{S}_t}$; in other words: (i) $1 \leq r < p_{\mathcal{S}_t}$, (ii) r is relatively prime to $p_{\mathcal{S}_t}$, (iii) r is chosen uniformly among the numbers that fulfill (i) and (ii). An empty character class in the text is encoded by 0.

Given the primes $p_{j_1}, \ldots, p_{j_d}$, each totative of $p_{\mathcal{S}_t}$ is uniquely characterized by its set of residues $r_{j_1}, \ldots, r_{j_d}$ modulo $p_{j_1}, \ldots, p_{j_d}$, respectively. Therefore, in order to uniformly select a random totative r, we choose random residues, and then compute the corresponding r using the CRT. Similarly to the analysis of

[1] A totative of x is a positive integer smaller than and relatively prime to x.

the pattern code in Section 3.1, encoding the text takes $O(s_t)$ time, where s_t is the total number of characters in the text, plus $O(m \log m)$ for pre-processing.

4.1 The SSM Algorithm

We now give a randomized algorithm, called SSM, for solving subset matching. The algorithm, outlined in Figure 3, is a variant of PMCC that uses the randomized text code described above.

INPUT: Pattern p and text t with character classes, alphabet Σ, $\sigma = |\Sigma|$

OUTPUT: All occurrences of p in t

Algorithm SSM:

1. Pre-processing: Same as in algorithm PMCC (Figure 1)
2. For $k = 1, \ldots, 2 \log_2 n / \log_2 m$:
   3. Encode the pattern and the text using a randomized prime code:
      3a. Text: Replace the character class $[b_{j_1}, \ldots, b_{j_d}]$ by $r \cdot M / p_{\mathcal{S}_t}$:
      $p_{\mathcal{S}_t} = p_{j_1} \cdot \ldots \cdot p_{j_d}$
      r is a random totative of $p_{\mathcal{S}_t}$ (i.e., $1 \leq r < p_{\mathcal{S}_t}$, $gcd(r, p_{\mathcal{S}_t}) = 1$)
      3b. Pattern: Replace the character class $[a_{i_1}, \ldots, a_{i_c}]$ by $n_{\{i_1,\ldots,i_c\}}$:
      $n_{\{i_1,\ldots,i_c\}} = 1 - (c_{i_1} + \ldots + c_{i_c}) \bmod M$
   4. Compute the convolution $C_k = (p \oplus t) \bmod M$ using FFT
5. Report a match at location i iff $\forall k \; C_k[i] = 0$

Fig. 3. Algorithm SSM for subset matching.

Theorem 5 *Algorithm SSM is a Monte Carlo algorithm for solving subset matching. If* $\sigma = O(m)$ *then with probability at least* $1 - \frac{1}{n}$ *the algorithm reports no false matches in time:*

$$\begin{cases} O(\sigma n \log m) & \text{, if } \ 0 \leq \kappa \leq 1 \\ O(n \log n) & \text{, if } \ \kappa = 1 \\ O(n \log n \log(m/\kappa)/\log m) & \text{, if } \ \kappa \geq 1 \end{cases}$$

where $\kappa = \log_\sigma (w/\log m)$ *and* w *is the RAM word size.*

Proof: We first analyze a single iteration k of steps 3 and 4. The convolution C_k at location i in the text is: $(p \oplus t)[i] = \Sigma_{j=1}^{m} p[j]t[i+j-1]$. Let $\mathcal{S}_p = [a_{i_1}, \ldots, a_{i_c}]$ be the character class at position j in the pattern, and let $\mathcal{S}_t = [b_{j_1}, \ldots, b_{j_d}]$ be the character class at position $i+j-1$ in the text. If $\mathcal{S}_t \subseteq \mathcal{S}_p$, then:

$$p[j]t[i+j-1] = n_{\{i_1,\ldots,i_c\}} \cdot rM/(p_{j_1} \cdot \ldots \cdot p_{j_d}) \equiv 0 \pmod{M}$$

Thus, if the pattern matches the text at location i, the above holds for all $1 \leq j \leq m$, and we get $C_k[i] = 0$. Conversely, suppose there is a mismatch at position j in the pattern, and let $a_k \in \mathcal{S}_t - \mathcal{S}_p$. In this case, $n_{\{i_1,\ldots,i_c\}} \equiv 1 \pmod{p_k}$, and r modulo p_k is a random totative of p_k (since it is a random totative of $p_{\mathcal{S}_t}$), so:

$$p[j]t[i+j-1] \bmod p_k \sim U(1, \ldots, p_k - 1)$$

Fixing the remaining variables, the congruence $(p \oplus t)[i] \equiv 0 \pmod{M}$ has a unique solution for $p[j]t[i+j-1] \bmod p_k$. Therefore, the probability that the congruence holds is at most $1/(p_k - 1) \leq 1/m$. In other words, if the pattern does not match the text at location i, then $C_k[i] = 0$ with probability

at most $1/m$.

If we perform $(\log_2 n + c)/\log_2 m$ iterations of steps 2 and 3, each time encoding the text using new random residues, the probability of having any false matches at any position is at most $n/m^{(\log_2 n+c)/\log_2 m} = 1/2^c$. Specifically, for $c = \log_2 n$ the probability for failure is at most $1/n$, and the entire algorithm is $\Theta(\log n/\log m)$ times slower than the deterministic algorithm for pattern matching with character classes (Theorem 2).

■

Acknowledgments

We would like to thank Michal Ziv-Ukelson for many helpful discussions, and Ze'ev Rudnick for his help in Number Theory. We also thank an anonymous referee for constructive comments. This study was supported by the Israel Science Foundation (grants 309/02 and 802/08 and Converging Technologies Program Grant 1767.07).

References

[1] D. Knuth, J. Morris, V. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.

[2] R. S. Boyer, J. S. Moore, A fast string searching algorithm, Commun. ACM 20 (10) (1977) 762–772.

[3] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University

Press, 1997.

[4] M. Fischer, M. Paterson, String matching and other products, in: R. M. Karp (Ed.), Complexity of Computation, SIAM-AMS Proceedings, 1974, pp. 113–125.

[5] P. Indyk, Faster algorithms for string matching problems: matching the convolution bound, in: Proc. 39th IEEE Annual Symposium on Foundations of Computer Science, 1998, pp. 166–173.

[6] A. Kalai, Efficient pattern-matching with don't cares, in: Proc. 13th annual ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 655–656.

[7] R. Cole, R. Hariharan, Verifying candidate matches in sparse and wildcard matching, in: Proc. 34th Symposium on Theory of Computing, 2002, pp. 592–601.

[8] P. Clifford, R. Clifford, Simple deterministic wildcard matching, Inform. Process. Lett. 101 (2) (2007) 53–54.

[9] R. Pinter, Efficient string matching with don't-care patterns, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, Vol. F12 of NATO ASI Series, Springer-Verlag, 1985, pp. 11–29.

[10] K. Abrahamson, Generalized string matching, SIAM J. Comput. 16 (1987) 1039–1051.

[11] R. A. Baeza-Yates, G. H. Gonnet, A new approach to text searching, Commun. ACM 35 (10) (1992) 74–82.

[12] A. Amir, M. Lewenstein, E. Porat, Faster algorithms for string matching with k mismatches, in: Proc. 11th annual ACM-SIAM Symposium on Discrete Algorithms, 2000, pp. 794–803.

[13] R. Cole, R. Hariharan, Tree pattern matching and subset matching in randomized $o(nlog^3m)$ time, in: Proc. 29th ACM Symposium on Theory of Computing, 1997, pp. 66–75.

[14] J. Hoh, S. Jin, T. Parrado, J. Edington, A. Levine, J. Ott, The p53MH algorithm and its application in detecting p53-responsive genes, Proc. Natl. Acad. Sci. USA 99 (13) (2002) 8467–8472.

[15] C. Linhart, R. Shamir, The degenerate primer design problem: Theory and applications, J. Comput. Biol. 12 (4) (2005) 431–456.

[16] T. Fuchs, B. Malecova, C. Linhart, R. Sharan, M. Khen, R. Herwig, D. Shmulevich, R. Elkon, M. Steinfath, J. O'Brien, U. Radelof, H. Lehrach, D. Lancet, R. Shamir, DEFOG: A practical scheme for deciphering families of genes, Genomics 80 (3) (2002) 295–302.

[17] T. Olender, T. Fuchs, C. Linhart, R. Shamir, M. Adams, F. Kalush, M. Khen, D. Lancet, The canine olfactory subgenome, Genomics 83 (3) (2004) 361–372.

[18] J. Rosser, L. Schoenfield, Approximate formulas for some functions of prime numbers, Illinois J. Math. 6 (1962) 64–94.

[19] M. Agrawal, N. Kayal, N. Saxena, PRIMES is in P, Ann. of Math. 160 (2) (2004) 781–793.

[20] A. Atkin, D. Bernstein, Prime sieves using binary quadratic forms, Math. Comp. 73 (2004) 1023–1030.

[21] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, MIT Press and McGraw-Hill, 2001.