Matching with don't-cares and a small number of mismatches

Chaim Linhart [*] , Ron Shamir

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

Abstract

In *matching with don't-cares and k mismatches* we are given a pattern of length m and a text of length n, both of which may contain don't-cares (a symbol that matches all symbols), and the goal is to find all locations in the text that match the pattern with at most k mismatches, where k is a parameter. We present new algorithms that solve this problem using a combination of convolutions and a dynamic programming procedure. We give randomized and deterministic solutions that run in time $O(nk^2 \log m)$ and $O(nk^3 \log m)$, respectively, and are faster than the most efficient extant methods for small values of k. Our deterministic algorithm is the first to obtain an $O(\text{poly}(k) \cdot n \log m)$ running time.

Key words: Analysis of algorithms, Pattern matching, Approximate wildcard matching

1 Introduction

The problem of *pattern matching with don't-cares* requires finding all occurrences of a pattern p of length m in a text t of length n, where the pattern and the text contain don't-cares (or wildcards), often marked as '*', that match all symbols. Fischer and Paterson developed an algorithm for solving this problem that utilizes boolean convolutions, computed using the Fast Fourier Transform (FFT) [1]. Assuming the RAM model, which is the computational model used by most studies on FFT-based pattern matching techniques, its running time is $O(\log|\Sigma| \cdot n \log m)$, where Σ is the alphabet. This time complexity has been improved over the past decade using various FFT-based methods.

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

Cole and Hariharan were the first to obtain an $O(n \log m)$ time deterministic algorithm [2], which was simplified by Clifford and Clifford [3].

In many practical scenarios, one may want to search for approximate matches, that is, locations in the text that match the pattern up to a small pre-specified distance. Perhaps the most widely used metric is the Hamming distance, which counts the number of mismatched pattern symbols. Applications of this variant of approximate matching are very common. For example, in bioinformatics they arise when comparing genes or proteins, and in the context of motif finding and primer design. The Hamming distance between the pattern and the text at every offset can be computed using the *match-count* algorithm, which computes $|\Sigma|$ boolean convolutions in time $O(|\Sigma| n \log m)$ [1]. Abrahamson combined the match-count algorithm with a divide-and-conquer technique to compute the Hamming distance in time $O(n\sqrt{m \log m})$ [4]. Randomized solutions for Hamming distance computation can also be obtained using sketching protocols (e.g., [5,6]).

In this paper we focus on the problem of *matching with k mismatches.* Given a pattern, a text and an integer k, we would like to report all locations in the text that match the pattern with at most k mismatches. This problem has been studied extensively for simple strings (i.e., without don't-cares). Currently, the most efficient method runs in time $O(n\sqrt{k \log k})$ [7]. As in the case of exact matching, searching for approximate matches becomes much more difficult when we allow don't-cares. This variant, which we call *matching with don't-cares and k mismatches*, has received attention only very recently (see details below). Here, we describe new efficient algorithms for matching with don't-cares and k mismatches, which are conceptually simpler, and in some cases faster, than extant techniques.

2 Problem definition and preliminaries

Let Σ be a finite alphabet, and denote by '$*$' the don't-care symbol. A text $t = t_1 \ldots t_n$ and a pattern $p = p_1 \ldots p_m$ are strings over $\Sigma \cup$'$*$'. Define $HD(i)$ to be the Hamming distance between p and $t_i \ldots t_{i+m-1}$:

$$HD(i) = |\ \{\ 1 \leq j \leq m \mid p_j \neq t_{i+j-1} \text{ and } p_j, t_{i+j-1} \neq \text{'}*\text{'}\ \}\ |$$

Matching with don't-cares and k mismatches: Given a pattern p and a text t with don't-cares, and an integer k, find all occurrences of p in t with at most k mismatches, i.e., report all locations i in t with $HD(i) \leq k$.

All the algorithms described in this paper assume the RAM model, wherein standard arithmetic on w-bit numbers are performed in constant time. Following common practice, we shall assume that the word size is $w = O(\log n)$.

Convolution: The convolution of two vectors a, b is the vector $a \oplus b$ such that:

$$(a \oplus b)[i] \stackrel{\text{def}}{=} \Sigma_{j=1}^{|a|} a_j b_{i+j-1} \ , \quad \text{for } 1 \leq i \leq |b| - |a| + 1$$

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 $\lceil n/m \rceil$ 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 [1]).

3 Related work and previous results

Both match-count and Abrahamson's technique for Hamming distance matching can easily handle don't-cares. Thus, matching with don't-cares and k mismatches can be solved in time $O(|\Sigma| n \log m)$ or $O(n\sqrt{m \log m})$ [1,4]. Intuitively, finding the locations at which the pattern matches the text with at most k mismatches should be easier than computing the exact Hamming distance at all locations. Indeed, Clifford et al. [8] recently developed several faster algorithms for this problem. Their algorithms, as well as the new ones we introduce in this work, extend the elegant technique for wildcard matching reported by Clifford and Clifford [3], which we now describe in brief (note that this technique also appears in [9] in the context of string matching with L_2 distance, and that similar methods based on manipulation of polynomials for solving various pattern matching problems were suggested earlier, e.g., [10,11]).

3.1 Simple matching with don't-cares

The simple algorithm for matching with don't-cares first encodes the pattern and the text, as follows. Each symbol is replaced by a unique positive number, and don't-cares are replaced by 0's. Then, for each location i in the text, the algorithm computes the sum $A_0[i]$:

$$A_0[i] = \sum_{j=1}^{m} x_{i,j} \tag{1}$$

where:

$$x_{i,j} = p_j \ t_{i+j-1} \ (p_j \ - \ t_{i+j-1})^2 \tag{2}$$

It is easy to see that $A_0[i] = 0$ if and only if there is an exact match at offset i. The key observation is that this sum can be computed efficiently for all offsets using three FFTs, since:

$$A_0[i] = \sum_{j=1}^{m} p_j^3 \, t_{i+j-1} \; - 2 \sum_{j=1}^{m} p_j^2 \, t_{i+j-1}^2 \; + \sum_{j=1}^{m} p_j \, t_{i+j-1}^3 \tag{3}$$

For example, the first sum in (3) is a convolution between $p_1^3, \ldots, p_m^3$ and the text $t_1, \ldots, t_n$. Thus, the total running time is $O(n \log m)$ [3].

3.2 Matching with don't-cares and k mismatches

Clifford et al. [8] further developed the above idea and devised an algorithm for solving the 1-mismatch problem — given a pattern and a text that contain don't-cares, the algorithm reports all text locations that match the pattern with at most one mismatch. In short, their algorithm computes (again, with FFTs) an additional array $A_1[i] = \sum_{j=1}^{m} (i + j - 1)\, x_{i,j}$. If there is a single mismatch at offset i, then the value $B[i] = A_1[i]/A_0[i]$ is the position of the mismatch. Thus, there is one mismatch iff $A_0[i] = x_{i,B[i]-i+1}$, which could easily be verified in constant time per text offset. Clifford et al. used this procedure as a building block for solving the k mismatches with don't-cares problem. They present a randomized algorithm that runs in $O(n(k + \log n \log\log n)\log m)$ time and gives the correct answer with high probability. Their deterministic algorithms, based on tools developed for group testing and for k-selectors, run in time $O(nk^2 \log^3 m)$ and $O(nk \operatorname{polylog} m)$, respectively (the latter with $O(\operatorname{poly} m)$ time preprocessing).

4 Main ideas and results

Our approach is based on the fact that at a fixed location i in the text, the number of mismatches between the pattern and the text is the number of non-zero's in the array $x_{i,1}, \ldots, x_{i,m}$. Denote:

$$C[i] \;=\; \sum_{1 \le j_1 < j_2 < \ldots < j_{k+1} \le m} x_{i,j_1} \cdot x_{i,j_2} \cdot \ldots \cdot x_{i,j_{k+1}} \tag{4}$$

where the sum is over all possible $(k{+}1)$-tuples of ordered indices from $\{1, \ldots, m\}$. We claim that there is a k-mismatch if and only if $C[i] = 0$. This is because if there are k or less mismatches at location i, then every set of $k+1$ indices must contain at least one position j' where the pattern matches the text, i.e., $x_{i,j'} = 0$, which implies that $C[i] = 0$. Conversely, if there are more than k

mismatches at text location i, and let $j_1, \ldots, j_{k+1}, \ldots$ denote their positions in the pattern, then $x_{i,j_1} \cdot x_{i,j_2} \cdot \ldots \cdot x_{i,j_{k+1}} > 0$. Since all $x_{i,j}$'s are non-negative, we get $C[i] \geq x_{i,j_1} \cdot x_{i,j_2} \cdot \ldots \cdot x_{i,j_{k+1}} > 0$, as required.

Alas, the value $C[i]$ is a sum of $\binom{m}{k+1}$ products of $k{+}1$ $x_{i,j}$'s — how can we compute it efficiently? Our main observation is that $C[i]$ can be expressed using a recursion, whose base is made up of $k{+}1$ arrays of the type $D_s[i] = \Sigma_{j=1}^{m} x_{i,j}^{s}$. Each of these arrays can be broken up into $O(k)$ convolutions and computed using FFTs. A dynamic programming procedure is then applied to compute $C[i]$ and report the results. An additional obstacle we need to overcome is that the numbers computed by the algorithm are too large to fit inside a single RAM word. We use simple tools from number theory to solve this problem. The total time complexity of our randomized algorithm, which reports the correct locations with high probability, is $O(nk^2 \log m)$. The running time of our deterministic solution is $O(nk^3 \log m)$. It is the first deterministic algorithm that solves matching with don't-cares and k mismatches in $O(\mathrm{poly}(k) \cdot n \log m)$ time. In particular, for constant k, it matches the $O(n \log m)$ time complexity for exact matching with don't-cares [2,3].

5 The algorithm

Our algorithm for matching with don't-cares and k mismatches, called k-MISMATCH, consists of four main steps, outlined in Figure 1.

```
Algorithm k-MISMATCH (pattern p, text t, integer k):
1. Encode $p$ and $t$ using positive integers, '*' using 0
2. Compute the arrays $D_1[i] = \Sigma_{j=1}^{m} x_{i,j}$ , ... , $D_{k+1}[i] = \Sigma_{j=1}^{m} x_{i,j}^{k+1}$,
   where $x_{i,j} = p_j t_{i+j-1}(p_j - t_{i+j-1})^2$
3. Compute the array $C[i] = \sum x_{i,j_1} \cdot x_{i,j_2} \cdot \ldots \cdot x_{i,j_{k+1}}$
4. Report a match at location $i$ iff $C[i] = 0$
```

Fig. 1. Algorithm for matching with don't-cares and k mismatches. See also Figure 2.

In step 1, the pattern and the text are encoded as in [3] — each alphabet symbol is replaced by a unique positive integer, and don't-cares are replaced by 0's. Step 2 computes the arrays $D_s[i]$ for $s = 1, \ldots, k{+}1$:

$$\begin{aligned} D_s[i] &= \sum_{j=1}^{m} x_{i,j}^s = \sum_{j=1}^{m} p_j^s\ t_{i+j-1}^s\ (p_j - t_{i+j-1})^{2s} \\ &= \sum_{j=1}^{m} p_j^{3s}\ t_{i+j-1}^s - 2s \sum_{j=1}^{m} p_j^{3s-1}\ t_{i+j-1}^{s+1} + \cdots + \sum_{j=1}^{m} p_j^s\ t_{i+j-1}^{3s} \end{aligned}$$

Thus, each array $D_s[i]$ is a linear combination of $2s{+}1$ convolutions of the type $p^a \oplus t^b$, so a total of $O(k^2)$ convolutions plus $O(k^2)$ linear-time operations on arrays of length n are required in step 2. In order to perform step 3, we need to define another family of arrays. Let s and t be positive integers, $s{+}t \le k{+}2$. Define the following array:

$$F_{t,s}[i] = \sum_{\substack{1\le j_1\le m \\ 1\le j_2<\ldots<j_t\le m \\ \forall l>1\ \ j_l\ne j_1}} x_{i,j_1}^s \cdot x_{i,j_2} \cdot \ldots \cdot x_{i,j_t}$$

If $s = 1$, we also require $j_1 < j_2$, so that each term occurs only once in the above sum. Informally, $F_{t,s}[i]$ is the sum of all terms of the type $x_{i,j_1}^s \cdot x_{i,j_2} \cdot \ldots \cdot x_{i,j_t}$, where the indices $j_1, \ldots, j_t$ are chosen in such a way that each term is taken exactly once. Notice that $F_{1,s}[i] = D_s[i]$, and $F_{k+1,1}[i] = C[i]$.

Lemma 1 *The following recursion holds:*

$$F_{t+1,s}[i] = \tfrac{1}{c}\ (F_{t,1}[i] \cdot F_{1,s}[i] - F_{t,s+1}[i]) \quad \text{, where } c = \begin{cases} t{+}1 & \text{, if } s = 1 \\ 1 & \text{, if } s > 1 \end{cases}$$

Proof: By definition:

$$F_{t,1}[i] \cdot F_{1,s}[i] = \Big(\sum_{1\le j_1<\ldots<j_t\le m} x_{i,j_1} \cdot \ldots \cdot x_{i,j_t} \Big) \cdot \Big(\sum_{1\le j\le m} x_{i,j}^s \Big)$$

Opening the above parentheses, we get two types of terms — one with $t{+}1$ distinct x's (when the index j in $x_{i,j}^s$ is not one of $j_1, \ldots, j_t$), and one with t distinct x's (when $j \in \{j_1, \ldots, j_t\}$). Collecting each type to a separate sum, we get using simple algebra:

$$\begin{aligned} & F_{t,1}[i] \cdot F_{1,s}[i] = \\ &= c \cdot \sum_{\substack{1\le j_1\le m \\ 1\le j_2<\ldots<j_{t+1}\le m \\ \forall l>1\ \ j_l\ne j_1}} x_{i,j_1}^s \cdot x_{i,j_2} \cdot \ldots \cdot x_{i,j_{t+1}} + \sum_{\substack{1\le j_1\le m \\ 1\le j_2<\ldots<j_t\le m \\ \forall l>1\ \ j_l\ne j_1}} x_{i,j_1}^{s+1} \cdot x_{i,j_2} \cdot \ldots \cdot x_{i,j_t} \\ &= c \cdot F_{t+1,s}[i] + F_{t,s+1}[i] \end{aligned}$$

■

By Lemma 1, step 3 of the algorithm can be computed using dynamic programming. We first set $F_{1,s}[i] = D_s[i]$ for $s = 1, \ldots, k{+}1$. We then compute

$F_{2,s}[i]$ for $s = 1, \ldots, k$ using the recursion. We continue in this way, as illustrated in Figure 2, until we obtain $F_{k+1,1}[i]$, which is the array $C[i]$ we wished to compute. Note that by examining the arrays $F_{1,1}[i], \ldots, F_{k+1,1}[i]$, we can infer the exact number of mismatches at each k-mismatch location:

$$HD(i) = \min\{\ t \mid F_{t+1,1}[i] = 0\ \}$$

The number of arrays the algorithm computes in step 3 is $k(k+1)/2$. Since each array is calculated in linear time, the running time of this step is $O(nk^2)$.

The overall running time of the algorithm is dominated by the time taken to perform the $O(k^2)$ FFTs in step 2, which is $O(nk^2 \log m)$. However, as mentioned earlier, there is still one flaw we must address — the algorithm computes numbers as large as $\binom{m}{k+1}|\Sigma|^{4(k+1)} < m^{5(k+1)}$ (see (4)), i.e., numbers with $O(k \log m)$ bits, whereas the RAM model commonly used in the pattern-matching literature permits unit-cost operations only on $O(\log n)$-bit words. To solve this problem, we perform all computations modulo some large prime number q that fits into a single RAM word, as described in the next sections.

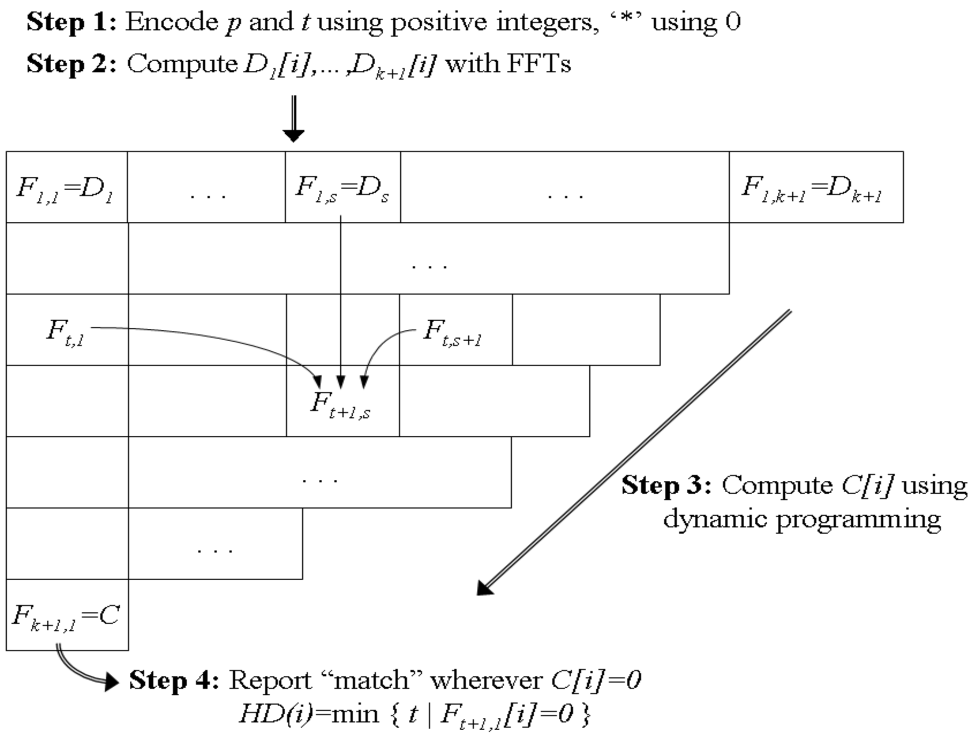
Step 1: Encode *p* and *t* using positive integers, '*' using 0
Step 2: Compute $D_1[i], \ldots, D_{k+1}[i]$ with FFTs
$F_{1,1}=D_1$
. . .
$F_{1,s}=D_s$
. . .
$F_{1,k+1}=D_{k+1}$
$F_{t,1}$
$F_{t,s+1}$
$F_{t+1,s}$
Step 3: Compute $C[i]$ using dynamic programming
$F_{k+1,1}=C$
Step 4: Report "match" wherever $C[i]=0$
$HD(i)=\min\{ t \mid F_{t+1,1}[i]=0 \}$

Fig. 2. Our algorithm for matching with don't-cares and k mismatches.

5.1 Randomized algorithm

Our randomized algorithm, called k-MISMATCH-RAN, is outlined in Figure 3. The algorithm randomly chooses two large prime numbers — q_1 and q_2, each with $O(\log n)$ bits, and computes the array $C_q[i] = C[i] \bmod q$, where $q = q_1 q_2$, using the procedure described above (integers in $\mathbb{Z}_q$ fit into a single RAM word, as required). Finally, it reports a match at location i if $C_q[i]=0$. $C[i]$ is an integer between 0 and some large number N, where $N < m^K$ and $K = 5(k+1)$. Thus, it has at most K prime factors larger than m. We therefore choose q_1 and q_2 randomly and uniformly from the primes within a sufficiently large interval, to guarantee that the probability of reporting a false match is small. The following lemma specifies the required interval.

Lemma 2 *For $n \geq 17$ and $K=5(k+1) \leq 5n$, there are more than nK primes in the interval $[n+1, 6n(K+1)\ln n]$. All these primes are $O(\log n)$-bit numbers.*

Proof: Following are well known bounds on the number $\pi(x)$ of primes less than or equal to x [12]:

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

Since $\ln(6n(K+1)\ln n) < 4\ln n$ for $n \geq 17$, it follows from the above bounds that:

$$\pi(6n(K+1)\ln n) - \pi(n) > \frac{6n(K+1)\ln n}{4\ln n} - \frac{1.26n}{\ln n} >$$
$$\frac{6n(K+1)\ln n - 6n}{4\ln n} > nK$$

■

Algorithm k-MISMATCH-RAN (p, t, k):
```
1. Randomly choose two prime numbers — $q_1, q_2 \in [n+1, 6n(K+1)\ln n]$,
   where $K = 5(k+1)$
2. $C_q[i]$ = $k$-MISMATCH($p$,$t$,$k$) mod$q_1 q_2$
3. Report a match at location $i$ iff $C_q[i]=0$
```

Fig. 3. Randomized algorithm for matching with don't-cares and k mismatches.

In order to obtain the primes q_1 and q_2, one can randomly draw numbers from the above interval and check each number for primality. This takes $O(\text{polylog} n)$ expected time [13], and can be done in preprocessing, as it depends only on the length of the text. Since there are more than nK primes in the interval, out of which at most K are factors of $C[i]$, it follows that if $C[i]>0$ the probability that $C_q[i]=0$ is:

$$P(C_q[i]=0 \mid C[i]>0) < \binom{K}{2} / \binom{nK}{2} = \frac{K(K-1)}{nK(nK-1)} < 1/n^2$$

In other words, the algorithm reports a false match at a given location i with probability less than $1/n^2$ (a true match is always reported correctly, since $C[i]=0$ implies $C[i] \equiv 0 \pmod{q}$). Thus, the probability that the algorithm reports any false match in the entire text is less than $1/n$.

Theorem 1 *Algorithm k-MISMATCH-RAN solves matching with don't-cares and k mismatches in $O(nk^2 \log m)$ time and gives the correct output (i.e., does not report false matches anywhere in the text) with probability at least $1-\frac{1}{n}$.*

For $k = O(\sqrt{\log n \log\log n})$ our algorithm improves upon the randomized technique of Clifford et al. [8], which runs in $O(n(k + \log n \log\log n)\log m)$ time.

5.2 Deterministic algorithm

The deterministic algorithm, called k-MISMATCH-DET and outlined in Figure 4, chooses K prime numbers — $q_1, \ldots, q_K > m$, and computes the array $C[i]$ modulo each of these primes separately. Lemma 3 specifies the interval that contains these primes.

```
Algorithm k-MISMATCH-DET (p, t, k):
1. Find prime numbers — $q_1, \ldots, q_K > m$, where $K = 5(k+1)$
2. For each prime $q_r$ do:
        Let $C_r[i]$ = k-MISMATCH(p,t,k) mod $q_r$
3. Report a match at location i iff $\forall r\ C_r[i] \equiv 0 \pmod{q_r}$
```

Fig. 4. Deterministic algorithm for matching with don't-cares and k mismatches.

Lemma 3 *For $m \geq 17$, the interval $[m+1, 19m \ln m]$ contains more than $5m$ primes.*

Proof: Since $\ln(19m\ln m) < 3\ln m$ for $m \geq 17$, then using the bounds on $\pi(x)$ (see proof of Lemma 2) we get:

$$\pi(19m\ln m) - \pi(m) > \frac{19m\ln m}{3\ln m} - \frac{1.26m}{\ln m} > \frac{19m\ln m - 4m}{3\ln m} > 5m$$

■

It follows from the above lemma that the primes $q_1, \ldots, q_K$ ($K \leq 5m$) can be found in time $o(m \ln m)$ using modern sieve techniques [14], and that they are $O(\log n)$-bit numbers, as required. The algorithm completes by reporting all locations for which $C[i]$ is 0 modulo all K primes. This always yields the correct answer, since:

$$0 \leq C[i] < m^K < \prod_{j=1}^{K} q_j$$

The running time of the deterministic algorithm is $o(m \ln m)$ for finding the prime numbers, plus $O(Knk^2 \log m)$ for computing $C[i]$ modulo each of the K primes. Thus, its total running time is $O(nk^3 \log m)$.

Theorem 2 *Algorithm k-MISMATCH-DET solves matching with don't-cares and k mismatches in $O(nk^3 \log m)$ time.*

Our deterministic algorithm is faster than the $O(nk^2 \log^3 m)$ time deterministic method of [8] for $k = O(\log^2 m)$.

6 Summary

We presented efficient randomized and deterministic algorithms for matching with don't-cares and k mismatches with running times $O(nk^2 \log m)$ and $O(nk^3 \log m)$, respectively. For small values of k, our algorithms are faster than the recently published methods of Clifford et al. [8]. For small alphabets ($|\Sigma| = O(k^2 \min\{k, \log^2 m\})$), the match-count algorithm is currently the fastest. Our solution is the first $O(\text{poly}(k) \cdot n \log m)$ time deterministic algorithm. For fixed values of k, this matches the $O(n \log m)$ time complexity of exact matching with don't-cares [2,3]. An interesting open question is whether an $O(f(k) n \log m)$ algorithm can be found with $f(k) = o(k^3)$, or even $f(k) = O(k)$.

Acknowledgments

This study was supported in part by the Israel Science Foundation (grant 802/08). We thank the anonymous referees for their constructive comments.

References

[1] M. Fischer, M. Paterson, String matching and other products, in: R. M. Karp (Ed.), Proc. 7th SIAM-AMS Complexity of Computation, 1974, pp. 113–125.

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

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

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

[5] J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. Strauss, R. Wright, Secure multiparty computation of approximations, ACM Transactions on Algorithms 2 (3) (2006) 435–472.

[6] E. Porat, O. Lipsky, Improved sketching of hamming distance with error correcting, in: A. Apostolico, M. Crochemore, K. Park (Eds.), Proc. 18th Annual Symposium on Combinatorial Pattern Matching, 2007, pp. 173–182.

[7] A. Amir, M. Lewenstein, E. Porat, Faster algorithms for string matching with k mismatches, Journal of Algorithms 50 (2) (2004) 257–275.

[8] R. Clifford, K. Efremenko, E. Porat, A. Rothschild, k-mismatch with don't cares, in: L. Arge, M. Hoffmann, E. Welzl (Eds.), Proc. 15th Annual European Symposium on Algorithms, 2007, pp. 151–162.

[9] E. Porat, O. Lipsky, Approximated pattern matching with the L_1, L_2 and L_∞ metrics, in: Proc. 15th Symposium on String Processing and Information, 2008, to appear (original manuscript from 2002).

[10] T. Eilam-Tzoreff, U. Vishkin, Matching patterns in strings subject to multi-linear transformations, in: R. Capocelli (Ed.), Sequences: combinatorics, compression, security, and transmission, Springer-Verlag, 1990, pp. 45–58.

[11] A. Amir, Y. Aumann, G. Benson, A. Levy, O. Lipsky, E. Porat, S. Skiena, U. Vishne, Pattern matching with address errors: rearrangement distances, in: Proc. 17th annual ACM-SIAM symposium on Discrete algorithms, 2006, pp. 1221–1229.

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

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

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