Multi-pattern Matching with Wildcards

Meng Zhang*, Yi Zhang†, Jijun Tang‡ and Xiaolong Bai*
*College of Computer Science and Technology, Jilin University, Changchun, China
Email: zhangmeng@jlu.edu.cn, baixiaol@sina.com
†Department of Computer Science, Jilin Business and Technology College, Changchun, China
Email: whdzy2000@vip.sina.com
‡Department of Computer Science & Engineering, Univ. of South Carolina, USA
Email: jtang@cse.sc.edu

***Abstract*—Multi-pattern matching with wildcards is to find all the occurrences of a set of patterns with wildcards in a text. This problem arises in various fields, such as computational biology and network security. But the problem is not extensively studied as the single pattern case and there is no efficient algorithm for this problem. In this paper, we present efficient algorithms based on the fast Fourier transform. Let $P = \{p^1, \ldots, p^k\}$ be a set of patterns with wildcards where the total length of patterns is $|P|$, and a text t of length n over alphabet $a_1, \ldots, a_\sigma$. We present three algorithms for this problem where patterns are matched simultaneously. The first algorithm finds the matches of a small set of patterns in the text in $O(n \log |P| + occ \log k)$ time where occ is the total number of occurrences of P in t. The words used in the algorithm are of size $k\lceil 2 \lg \sigma \rceil + \sum_{i=1}^{k} \lceil \lg |p^i| \rceil$ bits. The second algorithm is based on a prime number encoding. It runs in time $O(n \log m + occ \log k)$ where m is the length of the longest pattern in P. The algorithm uses words with $k\lceil \lg(2m\sigma^2 + k^2) \rceil$ bits. The third one finds the occurrences of patterns in the text in time $O(n \log |P| \log \sigma + occ \log k)$ by computing the Hamming distance between patterns and the text. The algorithm uses words with $\sum_{i=1}^{k} \lceil \lg |p^i| \rceil$ bits. Moreover, we demonstrate an FFT implementation based on the modular arithmetic for machines with 64-bit word. Finally, we show that these algorithms can be easily parallelized, and the parallelized algorithms are given as well.**

***Keywords*-Algorithm; Multi-pattern matching; Wildcards; FFT.**

I. INTRODUCTION

The problem of multi-pattern matching with wildcards is to find a set of patterns $P = \{p^1, \ldots, p^k\}$ in a text t (both the text and the patterns allow to contain wildcards). Throughout the paper, k denotes the number of patterns, n denotes the length of t, Σ denotes the alphabet of σ symbols from which the symbols in P and t are chosen. The single pattern matching with wildcards problem has received much attention. Fischer and Paterson [12] presented the first solution based on the fast Fourier transforms (FFT). The running time is $O(n \log m \log \sigma)$ where m is the length of the pattern. Indyk [13] latter introduced a randomized $O(n \log n)$ time Monte Carlo algorithm. Kalai [14] gave a simpler and faster $O(n \log m)$ time algorithm. In 2002 the first deterministic $O(n \log m)$ time solution was presented by Cole and Hariharan [7]. It uses one convolution, and each symbol in the text and the pattern is encoded with a pair of rational numbers. Clifford and Clifford [5] recently gave a simpler deterministic algorithm with the same time complexity that uses three convolutions where the numbers used are as large as $4m(\sigma-1)^4/27$. By allowing to preprocess the text, Rahman and Iliopoulos [20] gave efficient solutions without using FFTs and developed an algorithm runing in time $O(n + m + occ)$ where occ is the total number of occurrences of P in t. Very recently, Linhart and Shamir [17] presented the prime number encoding. By this approach, if $m^\sigma = n$, the algorithm runs in $O(n \log m)$ time by computing a single convolution.

Much research focus on Multi-string matching problem. The first algorithm to solve this problem in $O(n \log \sigma)$ time is presented by Aho and Corasick [1] which generalizes the Knuth-Morris-Pratt algorithm [15]. The Commentz-Walter [8] algorithm is a direct extension of the Boyer-Moore algorithm [4] which also combined the idea of AC algorithm. Several parallel multi-string matching algorithms are presented that are either precise [10] or approximate [27], [25], [24], [26]. The factor recognition approach based algorithms [9], [19], [2] use either suffix automata or factor oracles for precise or weak factor recognition. For short patterns, bit parallelism leads to algorithms that are efficient in practice, see [18].

However, the problem of matching a set of patterns with wildcards is not extensively studied as the single pattern case. To date, there is no efficient algorithm for this problem. But multi-pattern matching problem arises in many applications, such as intrusion detection systems [29], anti-virus systems [28] and computational biology [17]. A close but different problem: matching a set of patterns with variable length don't cares was solved by Kucherov and Rusinowitch [16]. They proposed an algorithm that runs in time $O((n + L(P))\log L(P))$, where $L(P)$ is the total length of keywords in every pattern of the pattern set P, and $|t|$ is the length of the input text. A faster solution was given by [22] which runs in $O((n + \|P\|)\log \kappa / \log\log \kappa)$ time, where $\|P\|$ is the total number of keywords in all the patterns in P, and κ is the number of distinct keywords in all the patterns in P.

In this paper, we focus on the problem of matching a set of patterns with wildcards without preprocessing the text. We present three FFT based algorithms for

A preliminary version of this paper was presented at PAAP 2010 [23].

doi:10.4304/jsw.6.12.2391-2398

this problem. The first one extends the Clifford and Clifford algorithm [5] to handle multi-pattern and runs in $O(n \log |P| + occ \log k)$ time where $|P|$ denotes the total length of all the patterns in P. It can find the matches of a small set of patterns in the text by three convolutions. The words used in the algorithm are of size $k\lceil 2 \lg \sigma \rceil + \sum_{i=1}^{k} \lceil \lg |p^i| \rceil$ bits. The second one uses the prime number encoding to encode both the pattern and the text. It runs in time $O(n \log m + occ \log k)$ where m is the length of the longest pattern. The algorithm uses words with $k\lceil 2 \lg \sigma + \lg m + \lg \lg(m\sigma^2) \rceil$ bits. The drawback of the two methods is that when σ, $|P|$ and k are large, the word will be too long to fit into a machine word of modern processors (normally 32 or 64 bits). To shorten the word length, we present an algorithm that uses words with $\sum_{i=1}^{k} \lceil \lg |p^i| \rceil$ bits. The algorithm finds the occurrences of patterns in the text in time $O(n \log |P| \log \sigma + occ \log k)$ by computing the Hamming distances between the patterns and the text. The distances are computed by $2\lceil \lg \sigma \rceil$ convolutions. Moreover, we discuss the modular arithmetic based FFT and give all the necessary parameters for the FFT on the 64-bit architecture. The algorithms presented in this paper can be easily parallelized. On a q-processor PRAM model, the time complexity of the algorithms decreases by q times compared with that on a single processor.

The paper is organized as follows. Section II gives some basic notions. Section III presents the algorithms for multi-pattern matching with wildcards using Euclidean distance. In Section IV, we give the approach based on Hamming distance of bit vectors. Section V introduces the FFT based on modular arithmetic on 64-bit architectures. Some interesting issues are discussed in Section VI.

II. Preliminaries

Let Σ be a finite alphabet and $'*'$ the wildcard symbol. Denote by $|s|$ the length of a string s. A text $t = t[1] \dots t[n]$ and a pattern $p = p[1] \dots p[m]$ are strings over $\Sigma \cup \{'*'\}$. Given a pattern p and a text t, which both may contain wildcards, we say that p occurs at location j in t if:

$$p[i] = t[i+j-1] \text{ or } p[i] =' *' \text{ or } t[i+j-1] =' *', \quad \text{for } 1 \le i \le m. \quad (1)$$

We use $P = \{p^1, \dots, p^k\}$ to denote a set of patterns with wildcards. We use " $\cdot$ " to denote the concatenation of two patterns, for example $p^1 \cdot p^2$ is the concatenation of pattern p^1 and p^2. For an integer array $x = x[1], x[2], \dots, x[n]$, we use $x[i_1..i_2]$ where $1 \le i_1 \le i_2 \le n$ to denote the array $x[i_1], \dots, x[i_2]$ and use $2^e x$ to denote the integer array of length n, such that

$$(2^e x)[i] = x[i]2^e, \text{ for each } 1 \le i \le n. \quad (2)$$

The following definition is a basic technique used in this paper.

Convolution: The convolution, or cross-correlation, of two vectors a, b is the vector $a \oplus b$ such that $(a \oplus b)[i] = \sum_{j=1}^{|a|} a[j]b[(i+j-1) \bmod |b|], 1 \le i \le |b|$. Note that this definition of convolution involves wrap-around (i.e., b is assumed to be a cyclic vector).

Our algorithms are based on FFTs. An important property of FFT is that in the RAM model, $p \oplus t$ can be computed in $O(n \log n)$ time. By a standard trick [12], the running time can be further reduced to $O(n \log m)$. First, split the text into n/m pieces of length $2m$. The starting positions of the pieces are in the set $\{lm + 1 \mid 0 \le l < n/m\}$. The convolution between the pattern and each piece of the text is computed using FFT in time $O(m \log m)$ per piece. The overall time complexity is $O((n/m)m \log m) = O(n \log m)$.

III. Euclidean distance based multi-pattern matching with wildcards

In this section, we extend the wildcard matching algorithm of Clifford and Clifford [5] to multi-pattern. Generally speaking, the Clifford and Clifford algorithm first encodes each symbol by a unique positive number and replaces wildcards by 0's. Then, for each location $1 \le i \le n - m + 1$ in the text, the algorithm computes

$$\sum_{j=1}^{m} p[j]t[i+j-1](p[j] - t[i+j-1])^2 = \sum_{j=1}^{m} (p[j]^3 t[i+j-1] - 2p[j]^2 t[i+j-1]^2 + p[j]t[i+j-1]^3) \quad (3)$$

in $O(n \log m)$ time using FFTs. Wherever there is an exact match this sum will be exactly 0.

The numbers computed by the algorithm are as large as $4m(\sigma-1)^4/27$. In [6], the authors modified the algorithm as follows. First by replacing non-wildcards by 1's and wildcards by 0's in the text and the pattern, we get p' and t' respectively. Then, for each location $1 \le i \le n-m+1$ in the text, the algorithm computes

$$\sum_{j=1}^{m} p'[j]t'[i+j-1]\big(p[j] - t[i+j-1]\big)^2. \quad (4)$$

The result can be viewed as the squire of the Euclidean distance between the pattern and the substring starting from a location i in t. The maximal numbers used in the convolutions are reduced to $m\sigma^2$.

A. Algorithm 1

In our approach, for a pattern p and each location $1 \le i \le n$ in the text, we use the following wrap-around sum

$$d(p,t)[i] = \sum_{j=1}^{|p|} p'[j]t'[l(i,j)]\big(p[j] - t[l(i,j)]\big)^2 \quad (5)$$

where $l(i,j)$ denotes $(i+j-1) \bmod n$. We can compute (5) by the following formula which uses only three

convolutions.

$$\sum_{j=1}^{|p|} p'[j]t[l(i,j)]^2 - 2\sum_{j=1}^{|p|} p[j]t[l(i,j)] + \sum_{j=1}^{|p|} t'[l(i,j)]p[j]^2. \quad (6)$$

Wherever there is an exact match this sum will be exactly 0.

To match a set of patterns $p^1, \ldots, p^k$, we first construct a composed pattern of length $|P|$:

$$p = p^1 \cdot p^2 \cdot \ldots \cdot p^k.$$

Define $\alpha_1 = 0$, $\alpha_j = \sum_{l=1}^{j-1}(\lceil \lg |p^l| + 2\lg\sigma \rceil)$, for $1 \le j \le k$ and $o_1 = 0$, $o_j = \sum_{l=1}^{j-1} |p^l|$, for $1 \le j \le k$. We use $I[1..l]$ to denote the array of length n where all the entries are 1's. We construct I^P as follows

$$I^P = (2^{\alpha_1} I[1..|p^1|]) \cdot (2^{\alpha_2} I[1..|p^2|]) \cdot \ldots \cdot (2^{\alpha_k} I[1..|p^k|]).$$

Then we compute the following

$$R[i] = \sum_{j=1}^{|p|} I^P[j]p'[j]t'[l(i,j)]\big(p[j] - t[l(i,j)]\big)^2$$

$$= \sum_{j=1}^{|p|} I^P[j]p'[j]t[l(i,j)]^2 - 2\sum_{j=1}^{|p|} I^P[j]p[j]t[l(i,j)] + \sum_{j=1}^{|p|} I^P[j]p[j]^2 t'[l(i,j)]. \quad (7)$$

R can be computed using three convolutions. By checking whether the bit vector from $(\alpha_j + 1)$th significant bit to α_{j+1}th significant bit of the binary code of $R[i]$, denoted by $R[i]_{[\alpha_j+1..\alpha_{j+1}]}$, is all 0's, we will know whether p^j occurs at position $(i + o_j)$ mod n in t. That is to say, assume that t is a cyclic vector, then for each $|P|$-length factor of t starting from each position of t, the result of the matching of each pattern is stored in a disjoint bit interval in a word. We give the algorithm in Figure 1.

According to (7), we can see that for the resulting array R computed by Algorithm 1,

$$R[i] = \sum_{j=1}^{k} d(p^j, t)[i + o_j]2^{\alpha_j}, \text{ for } 1 \le i \le n. \quad (8)$$

For any $p^j \in P$, according to (5), we have $d(p^j, t)[i + o_j] \le |p^j|\sigma^2$. So $\alpha_{j+1} - \alpha_j$ (that equals $\lceil \lg |p^j| + 2\lg\sigma \rceil$) bits are enough to represent $d(p^j, t)[i + o_j]$. As a result, the binary code of $d(p^j, t)[i + o_j]$ equals $R[i]_{[\alpha_j+1..\alpha_{j+1}]}$. Thus we can get $d(p^j, t)[i + o_j]$ by computing $(R[i] \bmod 2^{\alpha_{j+1}})/2^{\alpha_j}$.

To verify the correctness of Algorithm 1, suppose that a pattern $p^j \in P$ occurs in the text t starting from position x. We have $d(p^j, t)[x] = 0$. If p^j does not occur in t starting from x, we have $d(p^j, t)[x] \neq 0$. Let $i = x - o_j$. Thus $R[i]_{[\alpha_j+1..\alpha_{j+1}]} = 0$ indicates $d(p^j, t)[i + o_j] = 0$, that is, p^j occurs at position $(i + o_j)$ mod n of t.

The algorithm takes $O(n \log |P|)$ time to compute R and uses $O(nk)$ time to check R to find whether there is any occurrence of patterns. Each entry of R uses $k\lceil 2\lg\sigma \rceil + \sum_{i=1}^{k} \lceil \lg |p^i| \rceil$ bits. The size of the words used in the FFTs are of the same size. If σ, $|P|$ and k are small enough, each word can fit into a single machine word of modern processors that is typically 32 bits or 64 bits. For example, for DNA sequences where $\sigma = 4$, Algorithm 1 can process four patterns each with 16 symbols or three patterns each with 64 symbols. But when σ, $|P|$ and k are not small the words used in FFTs have to be very long. For example, let $\sigma = 256$, even for two patterns, Algorithm 1 uses words exceeding 32 bits. The algorithm in Section IV tries to shorten the word size to cope with texts on larger alphabets and pattern sets that have larger number of patterns and longer length.

Algorithm 1

Input: Text t and pattern set $P = \{p^1, p^2, \ldots, p^k\}$.

1: $R \leftarrow \{0, 0, \ldots, 0\}$
2: $\alpha_1 \leftarrow 0, o_1 \leftarrow 0$
3: **for** $j \leftarrow 2$ to k **do**
4: $\quad \alpha_j \leftarrow \alpha_{j-1} + \lceil \lg |p^{j-1}| + 2\lg\sigma \rceil$
5: $\quad o_j \leftarrow o_{j-1} + |p^{j-1}|$
6: **end for**
7: $L \leftarrow o_k + |p^k|$
8: $p \leftarrow p^1 \cdot p^2 \cdot \ldots \cdot p^k$,
9: $I^P[i] = (2^{\alpha_1} I[1..|p^1|]) \cdot (2^{\alpha_2} I[1..|p^2|]) \cdot \ldots \cdot (2^{\alpha_k} I[1..|p^k|])$
10: Compute $(p^i)'$ where $1 \le i \le k$ and t' by replacing non-wildcards by 1's and wildcards by 0's in the t and p^i.
11: For $1 \le i \le n$, compute $R[i] = \sum_{j=1}^{L} I^P[j]p'[j]t[(i + j - 1) \bmod n]^2 - 2\sum_{j=1}^{L} I^P[j]p[j]t[(i + j - 1) \bmod n] + \sum_{j=1}^{L} I^P[j]t'[(i + j - 1) \bmod n]p[j]^2$ using FFT.
12: **for** $pos \leftarrow 1$ to n **do**
13: $\quad$ **for** $j \leftarrow 1$ to k **do**
14: $\quad\quad$ **Output** "p^j occurs at $(pos + o_j)$ mod n in t" if $R[pos]_{[\alpha_j+1..\alpha_{j+1}]} = 0$.
15: $\quad$ **end for**
16: **end for**

Fig. 1. Algorithm 1.

The time complexity of checking the matches of P in t can be further reduced. Let the length of the longest pattern in P be m. Checking the matches of P in t can be done in time $O(n \log(m\sigma) + occ \log k)$ where occ is the times of occurrences of patterns in t. We first transform R to array ϱ such that $\varrho[i]_{[l]} = 0$ if $l \notin \{\alpha_1 + 1, \alpha_2 + 1, \ldots, \alpha_k + 1\}$ and for $1 \le j \le k$,

$$\varrho[i]_{[\alpha_j+1]} = \begin{cases} 1 & \text{if } R[i]_{[\alpha_j+1..\alpha_{j+1}]} = 0; \\ 0 & \text{if } R[i]_{[\alpha_j+1..\alpha_{j+1}]} \neq 0. \end{cases}$$

For p^j position $\alpha_j + 1$ is called the indication position (id position) of p^j. The transformation is described as

follows. Let there be d different lengths of patterns. We order the lengths in an increasing order, use m_j where $1 \leq j \leq d$ to denote the jth minimal length. First, set $\varrho[i] = 0$ for $1 \leq i \leq n$. Then for each pattern p^j we compute the bit $\bigwedge_{e=\alpha_j+1}^{\alpha_{j+1}} \overline{R[i]}_{[e]}$ and set this bit to $\varrho[i]_{[\alpha_j+1]}$. It follows that if $R[i]_{[\alpha_j+1..\alpha_{j+1}]} = 0$ then $\bigwedge_{e=\alpha_j+1}^{\alpha_{j+1}} \overline{R[i]}_{[e]} = \varrho[i]_{[\alpha_j+1]} = 1$. The computation is by bitwise `shiftright` and bitwise `and` operations. In the transformation, We use d bit masks, say $Vmask[1], \ldots, Vmask[d]$. The bit mask $Vmask[j]$ is a word where the bits on the indication locations for patterns whose length is m_j are set to bit 1 and other bits are set to 0s. The transformation is given in Figure 2. The time complexity of the algorithm is $O(n \log(m\sigma^2))$

Transform R
Input: An array R of length n.

1: **for** $i \leftarrow 1$ to n **do**
2: $R[i] \leftarrow \overline{R[i]}$
3: **end for**
4: **for** $j \leftarrow 1$ to d **do**
5: $Vmask[j] = 0$
6: **for** each pattern p^e such that $|p^e| = m_j$ **do**
7: $Vmask[j]_{[\alpha_e+1]} \leftarrow 1$
8: **end for**
9: **end for**
10: **for** $i \leftarrow 1$ to n **do**
11: $j \leftarrow 1$, $x \leftarrow Rr \leftarrow R[i]$, $RE \leftarrow 0$
12: **for** $s \leftarrow 1$ to $\lceil \lg(m\sigma^2) \rceil - 1$ **do**
13: $Rr \leftarrow Rr >> 1$
14: $x \leftarrow x \ \wedge \ Rr$
15: **if** $s = \lceil \lg(m_j\sigma^2) \rceil - 1$ **then**
16: $RE \leftarrow x \ \wedge \ Vmask[j]$
17: $j \leftarrow j + 1$
18: **end if**
19: **end for**
20: $\varrho[i] \leftarrow RE$
21: **end for**

Fig. 2. Transform R to ϱ.

When ϱ is available, we next check each entry of ϱ to find matches. For an entry $x = \varrho[i]$, we use an implicit binary tree T to find the id positions on which the bit values 1. Each tree node corresponds to a subset of the indication positions. A node $u = [n_1..n_2]$ consists of the indication positions of p_j where $j \in [n_1..n_2]$. The root is the set of all id positions, denoted by $[1..k]$. Each leaf contains one id position. For node u, the two children are $[n_1..(n_1+n_2)/2]$ and $((n_1+n_2)/2..n_2]$. We compute a bit mask $Mask(u)$ from u as follows: the bits of $Mask(u)$ on positions in $[n_1..n_2]$ are set to 1s, other bits are set to 0s.

We start at the root of T and check $x \wedge Mask([1..k])$. If it is 0, then there is no match for patterns $p^1, p^2, \ldots, p^k$. If it is not 0, at least one pattern matches, we continue to search in the left subtree of the root by checking $x \wedge Mask([1..k/2])$. If it is 0 then there is no match for patterns $p^1, p^2, \ldots, p^{k/2}$, and we prune the left branch; otherwise, at least one pattern matches, so we continue to search the left subtree of the root. After searching in the left subtree, we search in the right subtree. In this manner we traverse T depth-first, checking $x \wedge Mask(u)$ for each visited node u and pruning the branch if it is 0 along the way. At last, all the occurrences will be found by visiting the leaves corresponding to the matched patterns.

The time complexity is $O(n + occ \log k)$. So the total time complexity for finding the matches in R is $O(n \log(m\sigma^2) + occ \log k)$. For $|P| > m$ and $|P| \geq \sigma$, the total time complexity of Algorithm 1 is $O(n \log |P| + occ \log k)$.

B. Prime number encoding based algorithm

In this section, we introduce another strategy for matching a set of patterns with wildcards. Other than concatenating the patterns to a long one, this method aligns the patterns and generates a composed pattern with the length of the longest pattern. The algorithm is based on a prime number encoding of patterns. Let m be the length of the longest pattern in P. We extend each pattern to a similar length m by padding $'*'$s to the end of a pattern. Denote the resulting pattern set by P'. By padding m 0's to the end of the input, any matching of P in t is exactly a matching of P' in t for the same pattern on the same position.

We first pick up k distinct prime numbers $\rho_1, \rho_2, \ldots, \rho_k$. Denote $M = \rho_1 \cdot \rho_2 \cdots \rho_k$. We further require that ρ_i are larger than $|p^i|\sigma^2$.

Now we consider the encoding method. For k non-negative integers $x_1, x_2, \ldots, x_k$, where each x_i is not greater than σ, define an integer X that is less than M, such that for $1 \leq i \leq k$

$$X \equiv x_i \pmod{\rho_i}. \tag{9}$$

According to the Chinese Remainder Theorem (CRT, in short) [11], define $c_i = M/\rho_i\big((M/\rho_i)^{-1} \bmod \rho_i\big)$. For c_i, we have $c_i \equiv 1 (\bmod \rho_i)$ and $c_i \equiv 0 (\bmod \rho_{i'}), i' \neq i$. By the CRT, $X = \sum_{i=1}^{k} c_i \cdot x_i$.

We construct a composed pattern γ of length m from P, where

$$\gamma[i] = \sum_{j=1}^{k} c_j \cdot p^j[i], \text{ for } 1 \leq i \leq m, \tag{10}$$

and another composed pattern γ' of length m, where

$$\gamma'[i] = \sum_{j=1}^{k} c_j \cdot (p^j)'[i], \text{ for } 1 \leq i \leq m. \tag{11}$$

The text is encoded as follows:

$$\tau[i] = \sum_{j=1}^{k} c_j \cdot t[i], \text{ for } 1 \leq i \leq n. \tag{12}$$

τ' is encoded as follows:

$$\tau'[i] = \sum_{j=1}^{k} c_j \cdot t'[i], \text{ for } 1 \le i \le n. \quad (13)$$

Since $\rho_j > |p^j|\sigma^2$ and $d(p^j, t)[i] \le |p^j|\sigma^2$, for $1 \le j \le k$, we have

$$\gamma \bmod \rho_j = p^j, \tau \bmod \rho_j = t. \quad (14)$$

Then we can use Clifford and Clifford algorithm to match P' in t as follows. We compute Equation (5) for γ and τ and get array $d(\gamma, \tau)$. For an arbitrary array A, define $A \bmod p$ as the array of the same length of A where $(A \bmod p)[i] = A[i] \bmod p$. According to the CRT, for $1 \le j \le k$, we have

$$\begin{aligned} d(\gamma, \tau)[i] \bmod \rho_j &= d(\gamma \bmod \rho_j, \tau \bmod \rho_j)[i] \\ &= d(p^j, t)[i], \text{ for } 1 \le i \le n. \end{aligned} \quad (15)$$

Suppose that a pattern $p^j \in P$ occurs in the text t starting from position i. We have $d(p^j, t)[i] = 0$. So, $d(\gamma, \tau)[i] \bmod \rho_j = 0$. If p^j does not occur in t starting from x, we have $d(\gamma, \tau)[i] \bmod \rho_j \neq 0$. To find the matches of patterns, we have to check every entry of $d(\gamma, \tau)$, say $d(\gamma, \tau)[i]$, for all ρ_js such that $d(\gamma, \tau)[i] \bmod \rho_j = 0$. This straight forward method can find the occurrences in $O(nk)$ time. However, by using the method of Linhart and Shamir [17], the time complexity can be reduced to $O(n + occ \log k)$.

For $x = d(\gamma, \tau)[i]$, we use an implicit binary tree T to find ρ_j such that $x \bmod \rho_j = 0$. Each tree node corresponds to a subset of the pattern set. The root is the set of all patterns, denoted by $[1..k]$. Each leaf contains one pattern. For a node $u = [n_1..n_2]$, the two children are $[n_1..(n_1+n_2)/2]$ and $((n_1+n_2)/2..n_2]$. For node u, we compute an integer $Modul(u)$ as follows: $Modul(u) = \rho_{n_1} \cdot \rho_{n_1+1} \cdots \rho_{n_2}$.

We start at the root of T and check $x \bmod M$. If it is not 0, then there is no match for patterns. If it is 0, at least one pattern matches, we continue to search in the left subtree of the root by checking $x \bmod \rho_1 \cdot \rho_2 \cdots \rho_{k/2}$. If it is not 0 then there is no match for patterns $p^1, p^2, \ldots, p^{k/2}$, and we prune the left branch; otherwise, at least one of these patterns matches, so we continue to search the left subtree of the root. We traverse T in a depth-first manner, pruning some of the branches along the way. In the end, all the occurrences will be found by visiting the leaves corresponding to the matched patterns.

The algorithm takes $O(n \log m)$ time to compute R and uses $O(n + occ \log k)$ time to check R to find whether there is any occurrence of patterns. The total running time is $O(n \log m + occ \log k)$.

Now we consider the size of words used in the algorithm. For the number $\pi(x)$ of primes not exceeding x, the following bound is well known:

$$\forall x > 17 \quad \frac{x}{\ln x} < \pi(x) < 1.26\frac{x}{\ln x}. \quad (16)$$

According to the bound, we can verify that for $m\sigma^2 > 17$,

$$\pi(2m\sigma^2 + k^2) - \pi(m\sigma^2) > \frac{2m\sigma^2 + k^2}{\ln(2m\sigma^2 + k^2)} - 1.26\frac{m\sigma^2}{\ln(m\sigma^2)} > k. \quad (17)$$

Thus, if we search for prime numbers between $m\sigma^2+1$ and $2m\sigma^2 + k^2$, we will find at least k prime numbers. Using the sieve algorithm of Atkin et al. [3] it takes $o(2m\sigma^2 + k^2)$ time to find these prime numbers. Each prime number is at most $\lg(2m\sigma^2 + k^2)$ bits long. Therefore, $\lg M \le k \lg(2m\sigma^2 + k^2)$. Thus, each entry of R uses $k\lceil\lg(2m\sigma^2 + k^2)\rceil$ bits.

IV. Hamming distance based approach

A. Hamming distance of bit vectors with wildcards

Let $B = b_1 b_2 \ldots b_n$ be a binary pattern with wildcards. Denote $\overline{b_1}\overline{b_2}\ldots\overline{b_n}$ by $\overline{B}$ where if $b_i = *$ then $\overline{b}_i = *$. Given a binary pattern p and a bit string t (t is assumed to be a cyclic vector), such that both p and t have wildcards. The matchings between the pattern and a factor of t of length m have seven cases: 11, $1*$, $*1$, 00, $0*$, $*0$ and $**$ alignments. The Hamming distance between the two strings is the sum of the number of nonmatchings, that is $\#10 + \#01$.

The Hamming distance between p and the factor starting from each position of t of length m can be computed by the following method: For a bit vector v with wildcards, denote by $h^x(v)$ the bit vector where each wildcard of v is replaced by the bit x. Then $(h^0(\overline{p}) \oplus h^0(t))[i]$ equals $\#01$ between p and the factor of t starting from i, and $(h^0(\overline{t}) \oplus h^0(p))[i]$ equals $\#10$ between the same pair of strings. The Hamming distance between p and the factor of t starting from i can be computed by the following:

$$H(p, t)[i] = (h^0(\overline{p}) \oplus h^0(t))[i] + (h^0(\overline{t}) \oplus h^0(p))[i]. \quad (18)$$

This distance can be computed by two convolutions.

B. Multi-pattern matching by Hamming distance of bit vectors

We present an algorithm for multi-pattern matching with wildcards based on Hamming distance. In this approach, we derive $\lceil\lg\sigma\rceil$ bit vectors from the input text and $\lceil\lg\sigma\rceil$ patterns from the pattern and then perform $\lceil\lg\sigma\rceil$ times of pattern matchings for $\lceil\lg\sigma\rceil$ pairs of pattern and bit text. For $1 \le s \le \lceil\lg\sigma\rceil$, denote the sth bit of the integer encoding of a character $c \in \Sigma$ by $c_{[s]}$, for $c = *$, $c_{[s]} = *$. For a string t over Σ, denote the bit vector $t[1]_{[s]}t[2]_{[s]}\ldots t[n]_{[s]}$ by $t_{[s]}$.

To match a set of patterns, we first construct a series of patterns $B_{\langle 1\rangle}$, $B_{\langle 2\rangle}$, …, $B_{\langle\lceil\lg\sigma\rceil\rangle}$ from P, such that for $1 \le s \le \lceil\lg\sigma\rceil$,

$$B_{\langle s\rangle} = 2^{a_1}p^1_{[s]} \cdot 2^{a_2}p^2_{[s]} \cdot \ldots \cdot 2^{a_k}p^k_{[s]},$$

where $a_1 = 0$, $a_j = \sum_{l=1}^{j-1}\lceil\lg|p^l|\rceil$.

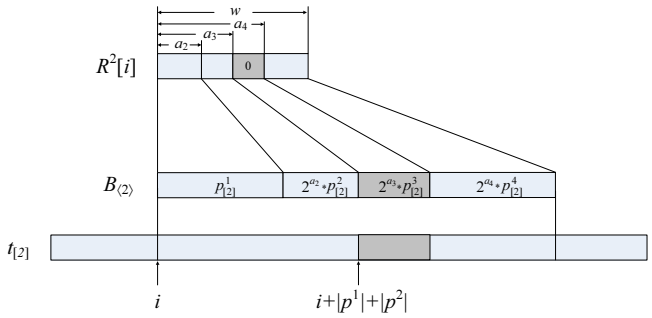

Fig. 3. An example of the run of Algorithm 3. Pattern p_2^3 is matched in t^2. The matching position is $i+|p^1|+|p^2|$ and $R^2[i][a_3+1..a_4]=0$.

Define

$$B_{\langle s,0\rangle} = 2^{a_1}h^0(p^1_{[s]}) \cdot 2^{a_2}h^0(p^2_{[s]}) \cdot \ldots \cdot 2^{a_k}h^0(p^k_{[s]}),$$
$$\widetilde{B}_{\langle s,0\rangle} = 2^{a_1}h^0(\overline{p^1_{[s]}}) \cdot 2^{a_2}h^0(\overline{p^2_{[s]}}) \cdot \ldots \cdot 2^{a_k}h^0(\overline{p^k_{[s]}}). \quad (19)$$

According to the computing of Hamming distance, for each $B_{\langle s\rangle}$ and P, we compute

$$R^s[i] = (\widetilde{B}_{\langle s,0\rangle} \oplus h^0(t_{[s]}))[i] + (h^0(\overline{t_{[s]}}) \oplus B_{\langle s,0\rangle})[i], \quad \text{for } 1 \le i \le n. \quad (20)$$

At last we compute the resulting array R such that

$$R[i] = R^1[i] \vee R^2[i] \vee \ldots \vee R^{\lceil \lg \sigma \rceil}[i], \text{ for } 1 \le i \le n. \quad (21)$$

By checking whether $R[i]_{[a_j+1..a_{j+1}]} = 0$, we will know whether p^j occurs at position $(i+o_j) \bmod n$ in t.

According to the definition of $B_{\langle s\rangle}$, we can see that for a resulting array R^s computed by the algorithm,

$$R^s[i] = \sum_{j=1}^{k} H(p^j_{[s]}, t_{[s]})[i+o_j]2^{a_j}, \text{ for } 1 \le i \le n. \quad (22)$$

For $p^j \in P$, we have $H(p^j_{[s]}, t_{[s]})[i+o_j] \le |p^j|$. So, $a_{j+1} - a_j = \lceil \lg |p^j| \rceil$ bits are enough to represent $H(p^j_{[s]}, t_{[s]})[i+o_j]$. Then we have

$$H(p^j_{[s]}, t_{[s]})[i+o_j] = R^s[i]_{[a_j+1..a_{j+1}]}. \quad (23)$$

Thus we can get $H(p^j_{[s]}, t_{[s]})[i+o_j]$ by computing $(R^s[i] \bmod 2^{a_{j+1}})/2^{a_j}$.

To see why this method works, suppose that a pattern $p^j \in P$ occurs in t starting from position x. We have $H(p^j_{[s]}, t_{[s]})[x] = 0$ for $1 \le s \le \lceil \lg \sigma \rceil$. Otherwise $H(p^j_{[s]}, t_{[s]})[x] \ne 0$ for at least one s. Let $x = i + o_j$. Since $R^s[i]_{[a_j+1..a_{j+1}]} = H(p^j_{[s]}, t_{[s]})[i+o_j]$, we have that $\bigvee_{s=1}^{\lceil \lg \sigma \rceil} R^s[i]_{[a_j+1..a_{j+1}]} = 0$ indicates $H(p^j_{[s]}, t_{[s]})[i+o_j] = 0$, for $1 \le s \le \lceil \lg \sigma \rceil$. Thus $R[i]_{[a_j+1..a_{j+1}]} = 0$ indicates that p^j occurs at position $(i+o_j) \bmod n$ of t. An example of the running of this algorithm is shown in Figure 3 where we only illustrate one bit vector.

The algorithm is given in Figure 4.

Algorithm 3 takes $O(n \log |P| \log \sigma)$ time to compute R and uses $O(nk)$ time to check whether there is any occurrence of patterns. By the method in Section III-A, the time for checking matches in R can be reduced to $O(n \log m + occ \log k)$. The total time complexity is therefore $O(n \log |P| \log \sigma + occ \log k)$. The words used in the algorithm are of size $w = \sum_{i=1}^{k} \lceil \lg |p^i| \rceil$ bits. If we use a $O(n \log m)$ time single pattern matching algorithm to match each pattern one by one, the total time is $O(n \sum_{i=1}^{k} \log |p^i| + occ \log k) = O(nw + occ \log k)$. Thus if $w > \lg |P| \lg \sigma$, Algorithm 3 takes less time for matching P against t than the time taken by single pattern matching algorithms. If $w \le \lg |P| \lg \sigma$, we can partition the pattern set into a set of subsets of P guaranteeing that $w > \lg |LP| \lg \sigma$ for each subset LP. We then match all these subsets of P using our algorithm one by one. The overall running time is certainly less than the time for running a single pattern algorithm for all the patterns.

If the occurrences of patterns are rare, we can revise Algorithm 3 to have a better performance. Recall that to use FFT efficiently, the input text is split into $n/|P|$ pieces of length $2|P|$ and each piece is matched against P independently. Then in the running of Algorithm 3, the pattern set is matched against every piece. Denote the piece starting at position $(l-1)|P|+1$ in t where

Algorithm 3
Input: text t and pattern set $P = \{p^1, p^2, \ldots, p^k\}$.

1: $R \leftarrow \{0, 0, \ldots, 0\}$
2: $a_1 \leftarrow 0, o_1 \leftarrow 0$
3: **for** $j \leftarrow 2$ to k **do**
4: $\quad a_j \leftarrow a_{j-1} + \lceil \lg |p^{j-1}| \rceil$
5: $\quad o_j \leftarrow o_{j-1} + |p^{j-1}|$
6: **end for**
7: $L \leftarrow o_k + |p^k|$
8: **for** $s \leftarrow 1$ to $\lceil \lg \sigma \rceil$ **do**
9: $\quad B_{\langle s,0\rangle} = 2^{a_1}h^0(p^1_{[s]}) \cdot 2^{a_2}h^0(p^2_{[s]}) \cdot \ldots \cdot 2^{a_k}h^0(p^k_{[s]})$
10: $\quad \widetilde{B}_{\langle s,0\rangle} = 2^{a_1}h^0(\overline{p^1_{[s]}}) \cdot 2^{a_2}h^0(\overline{p^2_{[s]}}) \cdot \ldots \cdot 2^{a_k}h^0(\overline{p^k_{[s]}})$
11: **end for**
12: **for** $s \leftarrow 1$ to $\lceil \lg \sigma \rceil$ **do**
13: $\quad$ For $1 \le i \le n$, compute $R^s[i] \leftarrow \sum_{r=1}^{L} \widetilde{B}_{\langle s,0\rangle}[r]h^0(t_{[s]})[(i+r-1) \bmod n] + \sum_{r=1}^{L} B_{\langle s,0\rangle}[r]h^0(\overline{t_{[s]}})[(i+r-1) \bmod n]$ using FFT.
14: $\quad$ **for** $i \leftarrow 1$ to n **do**
15: $\quad\quad R[i] \leftarrow R[i] \vee R^s[i]$
16: $\quad$ **end for**
17: **end for**
18: **for** $pos \leftarrow 1$ to n **do**
19: $\quad$ **for** $j \leftarrow 1$ to k **do**
20: $\quad\quad$ **Output** "p^j occurs at $(pos+o_j) \bmod n$ in t" if $R[pos]_{[a_j+1..a_{j+1}]} = 0$.
21: $\quad$ **end for**
22: **end for**

Fig. 4. Algorithm 3.

$1 \leq l \leq n/|P|$ by $piece(t, l)$. The resulting arrays computed by Algorithm 3 taking $piece(t, l)$ and P as input are denoted by $R^s(l)$, for each $1 \leq s \leq \lceil \lg \sigma \rceil$. We have $R^s(l)[i] = R^s[i + (l-1)|P|]$ where $1 \leq i \leq |P|$. Algorithm 3 is revised as follows. In computing R^s, for piece $piece(t_{[s]}, l)$, if $R^s(l)[i]_{[a_j+1..a_{j+1}]} \neq 0$ for all $1 \leq i \leq |P|$, then we have that no pattern occurs in piece $piece(t, l)$. In the followed computing, all the pieces $piece(t_{[s']}, l)$ where $s' > s$ are neglected. That is to say, $piece(t_{[s']}, l)$ will not be matched against $B_{\langle s' \rangle}$. The revised algorithm is correct for the situation that no pattern occurs in the neglected pieces. Since the occurrences of patterns are rare, the number of inspected pieces of bit vectors of the input by the revised algorithm is small. The cost is that in processing a bit vector of the input, the revised algorithm has to check the resulting array to determine which piece of the input can be neglected. In the original algorithm, the check is done only once when all the convolutions are finished.

V. FFT BASED ON INTEGER ARITHMETIC

In this section, we give an FFT implementation based on modular arithmetic other than complex numbers. In [21], Yap and Li present a fast integer multiplication algorithm based on an efficient FFT implementation built on the integer arithmetic. The approach aims at reducing the overhead in the implementation of FFT on machines in which the words are 32-bit or 64-bit. This FFT implementation fits the context of our multi-pattern matching algorithm well. The basic idea of the approach is to perform FFT in the ring $\mathbb{Z}_M = \{0, 1, \ldots, M-1\}$ of numbers modulo M. Here M is a specially chosen prime number. In a w-bit machine, M is at most w-bits so that the component-wise product can be done in $O(1)$ machine operations. For an n-length vector, in the field $\mathbb{Z}_M$, we need primitive n-th roots of unity. According to [21], the modular M is a prime number that can be expressed as

$$M = nd + 1. \tag{24}$$

We next pick up a primitive element of $\mathbb{Z}_M$, say e, and set

$$\omega = e^d \bmod M. \tag{25}$$

Then we have that each of $\omega^i \bmod M$ is distinct and $\neq 1$, for $i = 1, 2, 3, \ldots, n-1$. We have $w^n \equiv e^{nd} \equiv e^{M-1} \equiv 1(\bmod M)$ by Fermat's little theorem. That is ω is an n-th primitive root of unity.

In practice, to proceed the recursion of FFT, we need n to be a power of 2. For the case that the machine word is 32 bits, Yap and Li [21] choose $M_{32} = 2,013,265,921$ as the modulo, a prime number with only 31 bits that can be expressed as $M_{32} = 2^{27}d+1$ where $d = 15$ and $n = 2^{27}$. The number 31 can be proved to be a primitive element of $\mathbb{Z}_{M_{32}}$. The primitive 2^{27}-th roots of unity can be chosen as $\omega = 31^{15} \bmod M_{32} = 440,564,289$. Using ω, we can implement the FFT algorithm on a 32-bit machine, where we compute everything mod M_{32} and each component of the vectors is of the length 27 bits.

Based on Yap and Li's approach, we present the parameters for the 64-bit architecture. We choose $M_{64} = 6,269,010,681,299,730,433$ as the modulo, a prime number with only 63 bits. M_{64} has similar properties with M_{32}, which can be expressed as $M_{64} = 2^x d + 1$ where $d = 87$ and $x = 56$. Among the prime numbers that can be expressed as $2^x d + 1$, M_{64} has the maximal x. The number 5 turns out to be the smallest primitive element of $\mathbb{Z}_{M_{64}}$. The primitive 2^{56}-th roots of unity can be chosen as

$$\omega = 5^{87} \bmod M_{64} = 4,467,632,415,761,384,939.$$

Using ω, we can implement the FFT algorithm on a 64-bit machine, where we compute everything mod M_{64} and each component of the vectors is of the length 56 bits.

VI. PARALLELIZED ALGORITHMS

The algorithms in this paper can be easily parallelized. For Algorithm 1, we design a parallel multi-pattern matching algorithm with no communication. According to the trick in Section II, we first split the text into $n/|P|$ pieces each of length $2|P|$. The starting positions of the pieces are in the set $\{l|P| + 1 \mid 0 \leq l < n/|P|\}$. The convolution between the composed pattern and each piece of the text is computed using FFT in time $O(|P| \log |P|)$ per piece. We do not use the parallelized FFT, but use the sequential version of FFT conducted by processors. As each piece can be matched independently, no communication is needed. Since each piece is of length $2|P|$ and completely overlaps with the adjacent pieces, any occurrence of a pattern will keep integrate in at least one piece. Therefore, the parallelized Algorithm 1 is correct. On a q-processor PRAM model, the overall time complexity of parallelized Algorithm 1 is $O((n \log |P| + occ \log k)/q)$.

Algorithm 2 and Algorithm 3 can be parallelized in the same way as Algorithm 1. Readily, the time complexity of the parallelize Algorithm 2 and Algorithm 3 on a q-processor PRAM model are $O((n \log m + occ \log k)/q)$ and $O((n \log |P| \log \sigma + occ \log k)/q)$ accordingly.

VII. CONCLUSION AND FURTHER RESEARCH

We have presented three algorithms for multi-pattern matching with wildcards. The first one can find the matches of a small set of patterns in a text in $O(n \log |P| + occ \log k)$ time. The words used in the algorithm are of size $k\lceil 2 \lg \sigma \rceil + \sum_{i=1}^{k} \lceil \lg |p^i| \rceil$ bits. The second algorithm finds the occurrences of patterns in time $n \log m + occ \log k$ based on a prime number encoding of the pattern set and the text, using the words of $k\lceil \lg(2m\sigma^2 + k^2) \rceil$ bits. The last algorithm is based on Hamming distance between bit vectors. It runs in $O(n \log |P| \log \sigma + occ \log k)$ time and uses the words with $\sum_{i=1}^{k} \lceil \lg |p^i| \rceil$ bits. If the number of wildcards in the patterns is very small, the problem can be solved by building a deterministic finite automaton (DFA) that detects all possible words that match the pattern. One advantage of our algorithm over finite automata based algorithms is that the patterns in our algorithm need not

be preprocessed but taken as the input. In automata based approaches, the pattern set is used to build automata that will be further optimized for a low memory usage or a better performance. Whenever built, it is difficult to add or remove patterns from the existing data structures for the automata. In our approach, as patterns are taken as the input, adding or deleting a pattern has very low costs, thus our approach has more flexibilities.

It remains to determine whether there exists an $O(n \log |P|)$ algorithm using words of a proper size. That is in the resulting array, each bit of an entry indicates whether a pattern occurs at the corresponding location of the text. The modifications on FFT itself would be necessary to reach this goal.

ACKNOWLEDGMENTS

This work was in part supported by Fundamental Research Funds for the Central Universities No.200903186, Education Department of Jilin Province No.599, NSF of Science & Technology Department of Jilin Province No.20101522, NSF grant OCI 0904179, Chinese NSF No.60703024 and Program for New Century Excellent Talents in University NCET-09-0428.

REFERENCES

[1] A. V. Aho, M. J. Corasick: Efficient String Matching: An Aid to Bibliographic Search, *Comm. ACM* 18, 333–340, 1975.
[2] C. Allauzen, M. Raffinot. Factor oracle of a set of words. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.
[3] A. Atkin, D. Bernstein. Prime sieves using binary quadratic forms, *Math. Comp.* 73, 1023–1030, 2004.
[4] R. S. Boyer, J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
[5] P. Clifford and R. Clifford, Simple deterministic wildcard matching. *Inf. Process. Lett.* 101(2): 53–54, 2007.
[6] R. Clifford, K. Efremenko, E. Porat and A. Rothschild, From coding theory to efficient pattern matching. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithm*, 778–784, 2009.
[7] R. Cole and R. Hariharan, Verifying candidate matches in sparse and wildcard matching. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, 592–601, 2002.
[8] B. Commentz-Walter. A string matching algorithm fast on the average. In Proc. of the *6th International Colloquium on Automata, Languages and Programming*, LNCS 71, 118–132, 1979.
[9] M. Crochemore, A. Czumaj, L. Gąsieniec, T. Lecroq, W. Plandowski, and W. Rytter. Fast practical multi-pattern matching. *Information Processing Letters*, 71(3/4):107–113, 1999.
[10] Maxime Crochemore, Zvi Galil, Leszek Gasieniec, Kunsoo Park, Wojciech Rytter. Constant-Time Randomized Parallel String Matching. *SIAM J. Comput*. 26(4): 950–960, 1997.
[11] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, MIT Press and McGraw-Hill, 2001.
[12] M. Fischer and M. Paterson, String matching and other products. In *Proceedings of the 7th SIAM-AMS Complexity of Computation*, 113–125, 1974.
[13] P. Indyk, Faster algorithms for string matching problems. Matching the convolution bound. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, 166–173, 1998.
[14] A. Kalai, Efficient pattern-matching with don't cares. In Proceedings of the 13th *Annual ACM-SIAM Symposium on Discrete Algorithms*, 655–656, Philadelphia, PA, USA, 2002.
[15] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.
[16] G. Kucherov and M. Rusinowitch, Matching a Set of Strings with Variable Length don't Cares. *Theor. Comput. Sci.* 178(1-2): 129–154, 1997.
[17] C. Linhart and R. Shamir, Faster pattern matching with character classes using prime number encoding. *J. Comput. Syst. Sci.*75(3): 155–162, 2009.
[18] G. Navarro, M. Raffinot. Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences. Cambridge University Press, 2002.
[19] M. Raffinot. On the multi backward dawg matching algorithm (MultiBDM). In R. Baeza-Yates, editor, Proceedings of the *4th South American Workshop on String Processing*, 149–165, 1997.
[20] M. Rahman and C. Iliopoulos, Pattern Matching Algorithms with Don't Cares. *SOFSEM* (2), 116–126 2007.
[21] C. Yap and C. Li. QuickMul: Practical FFT-based Integer Multiplication. Technical report, Department of Computer Science, Courant Institute, New York University, October 2000.
[22] M. Zhang, Y. Zhang and L. Hu, A faster algorithm for matching a set of patterns with variable length don't cares. *Inf. Process. Lett.* 110(6): 216–220, 2010.
[23] M. Zhang, Y. Zhang and J. Tang, Matching a set of patterns with wildcards. *Third International Symposium on Parallel Architectures, Algorithms and Programming* (PAAP'10), 169–174, IEEE Computer Society, 2010.
[24] C. Zhong, Z. Fan and D. Su, Parallel Approximate Multi-Pattern Matching on Heterogenerous Cluster Systems, Proceedings of the *9th International Conference on Parallel and Distributed Computing, Appplications and Technologies*, 74–79, IEEE Computer Society Press, 2008.
[25] C. Zhong, D. Fan, Parallel Algorithms for Approximate String Matching with Multi-Round Distribution Strategy on Heterogeneous Cluster Computing Systems(in Chinese). *Journal of Computer Research and Development*,45(S1):105–112,2008.
[26] Z. Fan, C. Zhong, X. Cui, L. Xu, Parallel Algorithm for Approximate Multiple Object Strings Matching on Heterogeneous Cluster Computing Systems with Limited Memory(in Chinese), *Journal of Chinese Computer Systems*, 30(2):225–229,2009.
[27] C. Zhong, G. Chen, Parallel Algorithms for Approximate String Matching on PRAM and LARPBS(in Chinese), *Journal of Software*, 15(2):159–169, 2004.
[28] Clam AntiVirus. URL: `http://www.clamav.net`.
[29] Snort Intrusion Detection System. URL: `http://www.snort.org`.

BIOGRAPHIES

Meng Zhang received his Ph.D. in Computer Science from Jilin University, in 2003. He is currently an Associate Professor in College of Computer Science and Technology, Jilin University. His main research interests include stringology, network security and computational biology.

Yi Zhang received her Ph.D. in Computer Science from Jilin University, in 2009. She is currently an Associate Professor in Department of Computer Science, Jilin Business and Technology College. She is also a Postdoc researcher in Jilin university. Her main research interests include artificial intelligent and computational biology.

Jijun Tang Prof. Jijun Tang received his Ph.D. in Computer Science from University of New Mexico, in 2004. He is currently an Associate Professor in Department of Computer Science and Engineering, University of South Carolina. His main research interests include high performance algorithm development, computational biology and engineering simulation. During the past five years, His research has been supported by ONR, NSF and NIH.

Xiaolong Bai is currently a third year undergraduate student in College of Computer Science and Technology of Jilin University. His research interests include algorithm design and network security.