Wu X, Qiang JP, Xie F. Pattern matching with flexible wildcards. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 29(5): 740–750 Sept. 2014. DOI 10.1007/s11390-014-1464-3

Pattern Matching with Flexible Wildcards

Xindong Wu[1,2] (吴信东), *Fellow*, *IEEE*, Ji-Peng Qiang[1] (强继朋), and Fei Xie[1,3] (谢　飞)

[1] *Department of Computer Science, Hefei University of Technology, Hefei 230009, China*

[2] *Department of Computer Science, University of Vermont, Burlington, VT 05405, U.S.A.*

[3] *Department of Computer Science and Technology, Hefei Normal University, Hefei 230601, China*

E-mail: xwu@hfut.edu.cn; qjp2100@163.com; xiefei9815057@sina.com

Received February 17, 2014; revised July 22, 2014.

Abstract Pattern matching with wildcards (PMW) has great theoretical and practical significance in bioinformatics, information retrieval, and pattern mining. Due to the uncertainty of wildcards, not only is the number of all matches exponential with respect to the maximal gap flexibility and the pattern length, but the matching positions in PMW are also hard to choose. The objective to count the maximal number of matches one by one is computationally infeasible. Therefore, rather than solving the generic PMW problem, many research efforts have further defined new problems within PMW according to different application backgrounds. To break through the limitations of either fixing the number or allowing an unbounded number of wildcards, pattern matching with flexible wildcards (PMFW) allows the users to control the ranges of wildcards. In this paper, we provide a survey on the state-of-the-art algorithms for PMFW, with detailed analyses and comparisons, and discuss challenges and opportunities in PMFW research and applications.

Keywords pattern matching, wildcards, bioinformatics, pattern mining

1 Introduction

In recent years, pattern matching with wildcards has been widely used in various application fields, such as bioinformatics[1], intrusion detection systems[2], and pattern mining[3-6]. In bioinformatics, for example, the DNA sequence *TATA* is a common promoter that often occurs after the sequence *CAATCT* within 30∼50 wildcards[7]. Pattern matching with wildcards (PMW) has become especially crucial in exploring valuable information from DNA sequences. In intrusion detection systems, various attacks can be detected by means of some predefined rules where each rule is represented by a regular expression that defines a set of strings. It would be convenient and efficient to merge regular expressions whose languages are similar to forming a compact pattern with variable-length gaps.

In text mining[8], it is very natural to consider sequential patterns with wildcards where patterns are composed of words that can capture semantic relations between words. Table 1 shows an example to demonstrate how textual patterns with wildcards outperform general patterns in text mining. The text is from the title and abstract of a paper published on the *Knowledge and Information System* in 2013[9]. We assume that the issue is to extract keywords from the text automatically[10]. The top frequent single words and textual patterns with wildcards are also listed in Table 1 where the number after each colon indicates the fre-

Table 1. A Motivation Example

Text	Frequent Words	Frequent Patterns with Wildcards
Topic-aware social influence *propagation models* ⋯ We introduce novel *topic-aware* influence-driven *propagation models* that ⋯ In particular, we first propose simple *topic-aware* extensions of the well-known Independent Cascade and Linear Threshold models. However, these *propagation models* have a very large number of parameters which could lead to ⋯ Keywords: social influence; *topic modeling*; *topic-aware propagation model*; viral marketing	Model: 11, topic: 7, propagation: 6, influence: 6, social: 4, ⋯	Topic model: 7, topic aware propagation model: 3, social influence: 3, influence driven propagation: 3, ⋯

Survey

This paper is supported in part by the National Natural Science Foundation of China under Grant Nos. 61229301 and 60828005, the Program for Changjiang Scholars and Innovative Research Team in University (PCSIRT) of the Ministry of Education, China, under Grant No. IRT13059, and the National Science Foundation (NSF) of USA under Grant No. 0514819.

quency of a word or a pattern. It can be seen that although the single words can capture the main topics of the paper, they are different from the keywords provided by the authors. For the keyword "topic model", it occurs only once continuously in the abstract. But the pattern "topic model" with wildcards occurs seven times. Furthermore, all occurrences of the pattern "topic aware propagation model" are not continuous.

Giving a pattern P and a text T, some classical matching algorithms return the matching positions of P in T. This matching can be exact, and can also be approximate. However, if wildcards are added into the process of matching, this PMW problem becomes much complicated. There are three types of research efforts on dealing with pattern matching with wildcards: 1) a fixed number of wildcards[11-15], 2) the number range of wildcards to be specified separately[16-21], namely flexible wildcards, and 3) an unbounded number of wildcards[5,22-24]. As the first two types have limited flexibilities for the user's queries, pattern matching with flexible wildcards (PMFW) has drawn more attention.

Due to the uncertainty of wildcards, not only is the number of all matches exponential with respect to the maximal gap flexibility and the pattern length, but the matching positions in PMW are also hard to choose. In Example 1 below, although the length of the text is 5 and the maximal gap flexibility is just 2, the number of matches has already reached 6 and the matching positions are even more complicated. Because of the flexible wildcards, there can be more than one, actually an exponential number of occurrences of a pattern at the same element position. Thus counting the matching positions of all matches one by one is computationally infeasible.

Example 1. Given $P = a[0,2]c[0,2]c$ and $T = acccc$, where [0, 2] means there are 0 to 2 wildcards at each of these places, the number of all matches is 6, and the matching positions are $\{(1,2,3)$, $(1,2,4)$, $(1,2,5)$, $(1,3,4)$, $(1,3,5)$, $(1,4,5)\}$ respectively.

To solve the above problems in PMW, this paper reviews research efforts in PMFW from the following three perspectives. First, the start position or the end position of each matching occurrence of the pattern in the text is reported[25-26] and this line of research is referred to as the EPP (Ending Positions of Pattern) problem. Second, only the number of all matches of the pattern in the text[27-28] is computed, and this is called the NAM (Number of All Matches) problem. Third, to avoid a text letter being shared by more than one matching occurrence, the one-off condition is incorporated into PMFW[29-30], called the OOC (One-off Condition) problem. The one-off condition means that every letter in T can be used at most once, namely, any two occurrences of P cannot share the same position in T.

The rest of the paper is organized as follows. We provide a brief review on pattern matching in Section 2. In view of the above-mentioned three lines of research within PMFW, we present recent studies in Sections 3∼5 respectively, along with an analysis and a comparison in each section. Meanwhile, the challenges and future work are discussed in Section 6.

2 Overview of Pattern Matching

2.1 Classical Pattern Matching

Given a text t and a pattern p, the classical pattern matching problem is to search for the appearances of p in t. The classical pattern matching is also called exact pattern matching. Lin *et al.*[31] categorized the exact pattern matching algorithms into four categories: automation-based[31-34], heuristic-based[35-36], hashing-based[37-38], and bit-parallelism based algorithms[39-40]. With the application problems extended, pattern matching has been generalized, such as approximate pattern matching, multiple pattern matching, and pattern matching with wildcards.

The approximate pattern matching focuses on the problem of pattern matching that allows errors[41]. Given a text T, a pattern P and a pre-specified positive integer k, the approximate pattern matching problem is defined to find all substrings S's in T such that the distance between S and P is not larger than k. One of the best studied distance functions is the edit distance, which allows deleting, inserting and substituting characters in both strings.

Given a set of patterns $P = \{P_1, P_2, \ldots, P_l\}$ and a text T, the multiple pattern matching is to find all occurrences of these patterns in T. One can solve this problem by scanning T for each pattern separately, but that requires scanning T as many times as the number of patterns. Therefore, some efficient algorithms have been designed by scanning T only once for all patterns[36,42]. The Wu-Manber algorithm[36] is based on the Boyer-Moor algorithm[35] that uses the "bad character" heuristic to skip over characters in the input text. The authors of [42] proposed an efficient multi-pattern matching algorithm based on a bit representation of patterns and text using a compact encoding scheme.

2.2 Pattern Matching with Wildcards

Fischer and Paterson first generalized pattern matching to consider "don't care" letters, also called wildcards or gaps in the literature[11]. A wildcard can match any letter in a given alphabet. The objective was to find all locations of the occurrences of a pattern P in a text T, where locations are matching positions of an occurrence in the text, and an occurrence means a matching of P in T. The user can specify a fixed number of wildcards between every two consecutive letters in P. The running time in [11] is $O(n \log^2 m \log\log m \log |\Sigma|)$ where n is the text length, m is the pattern length, and $|\Sigma|$ is the alphabet size. Muthukrishnan and Palem[12] proved the problem of pattern matching with wildcards to be at least as hard as the Boolean convolution problem. Indyk[13] and Cole and Hariharan[15] improved the time efficiency of the problem. The time complexities in [13] and [15] are $O(n \log n)$ and $O(n \log m)$ respectively where n is the text length and m is the pattern length. However, the number of wildcards in [11-15] is a constant but not a range, which limits flexibilities for the user's queries.

The problem of pattern matching with flexible wildcards has been studied in [25] where the pattern contains variable length gaps with a certain range. Rahman *et al.*[25] presented an $O(n+m+\alpha \log g)$ algorithm to report whether a pattern occurs in a text where n is the text length, m is the pattern length, α is the total number of occurrences of the substring in the text, and g is the maximal gap length between any two consecutive substrings. Bille *et al.*[26] gave an algorithm that takes $O(n \log k+m+\alpha)$ time to find all ending positions of the substring in the text where n is the text length, k is the number of substrings in the pattern, m is the pattern length, and α is the total number of occurrences of the substring in the text. Min *et al.*[27] and Zhu and Wu[28] also studied the pattern matching problem with flexible wildcards to count the number of occurrences of a pattern in the text. The detailed algorithm analysis and comparison will be given in Subsection 4.2.

Kucherov and Rusinowitch[22] and Zhang *et al.*[23] studied the multi-pattern matching problem with unbounded wildcards, that is, the gap length between any two consecutive substrings is arbitrarily large within the text. A VLDC pattern is the sequence of keywords $p_1@p_2 \cdots @p_m$ where p_i $(1 \leqslant i \leqslant m)$ is a string over an alphabet Σ called keyword and @ represents the variable length wildcards. Given a set of VLDC patterns P and a text T, the multi-VLDC matching problem is to determine whether one of the patterns of P matches T. In [22], an algorithm is proposed to solve the problem in $O((|t| + |P|) \log |P|)$ where $|t|$ is the text length, $|P|$ is the total length of keywords in every pattern of P. Zhang *et al.*[23] presented a faster and simpler algorithm that takes $O((|t| + ||P||) \log K/ \log\log K)$ where $|t|$ is the text length, $\|P\|$ is the total number of keywords in every pattern of P, and K is the number of distinct keywords in P.

Another dimension of the pattern matching problem with wildcards is studied in [29-30] where the occurrences of a pattern do not share the characters in the text, called the one-off condition. The one-off condition has two advantages. On the one hand, it makes the sequential pattern mining problem satisfy the Apriori property[43] that can greatly improve the time efficiency of the pattern mining. On the other hand, it also makes great sense in many practical applications. For example, in text mining, the user is interested in how frequently the words co-occur in the document that captures the semantic relations between words. Obviously, it is more reasonable that each occurrence of a word is counted only once in the occurrences of a pattern.

Ding *et al.*[5] also studied the pattern matching problem with wildcards applied in sequential pattern mining where the occurrences of a pattern satisfy the non-overlapping condition that is very similar, but different from the one-off condition. Let $occ_1 = (l_1, \ldots, l_m)$ and $occ_2 = (l'_1, \ldots, l'_m)$ be two occurrences of a length-m pattern. If $l_i \neq l'_i$ for every $1 \leqslant i \leqslant m$, occ_1 and occ_2 are non-overlapping. However, occ_1 and occ_2 satisfy the one-off condition only if $l_i \neq l'_j$ for every $1 \leqslant i, j \leqslant m$.

In Table 2, some important features of different types of pattern matching with wildcards are listed.

In the sections to follow, we focus on the problem of pattern matching with flexible wildcards (PMFW), based on our own research efforts since 2004. This problem has also been extended to approximate matc-

Table 2. Different Types of Pattern Matching Problems with Wildcards

Algorithm	Input	Constraints	Output
Literature [11-14]	One text and one pattern	Fixed wildcards	Occurrences
Literature [16-20]	One text and one pattern	Flexible wildcards	Ending positions of occurrences
Literature [22]	One text and multi-patterns	Unbounded wildcards	Whether one of the patterns occurs in the text
Literature [27-28]	One text and one pattern	Flexible wildcards	Number of occurrences
Literature [29-30]	One text and one pattern	Flexible wildcards	Occurrences with the one-off condition
Literature [23]	One text and one pattern	Unbounded wildcards	Occurrences with the non-overlapping condition
Literature [24]	One text and one pattern	Unbounded wildcards	Occurrences with the one-off condition
Literature [21]	One text and one pattern	Unbounded wildcards, flexible wildcards	Ending positions of occurrences

hing[44-45] and pattern mining[4,46], which are outside the scope of this paper.

3 EPP: Ending Positions of Pattern

3.1 Problem Statement

Let $P = p_1[N_1, M_1]p_2[N_2, M_2] \cdots [N_{k-1}, M_{k-1}]p_k$ be a pattern with wildcards of k subpatterns and $T = t_1t_2 \ldots t_n$ be a text of length n, where subpattern p_i ($1 \leqslant i \leqslant k$) is a letter or a string without any wildcards, $[N_{i-1}, M_{i-1}]$ is a string that represents the local length constraint over a finite alphabet Σ, and N_{i-1} and M_{i-1} are the number range of wildcards between p_{i-1} and p_i, s.t. $0 \leqslant N_{i-1} \leqslant M_{i-1}$.

Definition 1 (Occurrence). *An occurrence* $C = (c_1, \ldots, c_i, \ldots, c_k)$ *is a list of position indices of* P *in* T*. If there exists a sequence of position indices* $C = (c_1, \ldots, c_i, \ldots, c_k)$ *where* $1 \leqslant c_i \leqslant n$, $1 \leqslant i \leqslant k-1$ *such that*

1) $T_{c_i} = p_i$,

2) $N_i + |p_i| \leqslant c_{i+1} - c_i \leqslant M_i + |p_i|$,

where C *is a match of* P *in* T *for all* $1 \leqslant i \leqslant k$.

Definition 2 (EPP Problem). *The objective of the EPP problem is to return the end position indices of all occurrences of* P *in* T*, namely all* c_k.

In Example 1, the result of the EPP problem is the set of positions {3, 4, 5}.

3.2 Algorithms

3.2.1 *Algorithm of Rahman et al.*

Algorithm 1 outlines the steps of the algorithm of Rahman *et al.*[25] This algorithm uses the famous Aho-Corasick (AC) automaton[32] to report the occurrences of the subpatterns $\{p_1, \ldots, p_k\}$. For each subpattern p_i ($2 \leqslant i \leqslant k$), the algorithm initializes a sort list L_i. Initially, when the j-th occurrence x of p_1 is reported, the range $[(endpos(x) + N_1 + 1), (endpos(x) + M_1 + 1)]$ is stored to L_2, where $endpos(x)$ is the end position of x. Once an occurrence y of p_i ($i > 1$) is reported, L_i is searched and if $startpos(y)$ is not within L_i, then it is discarded, where $startpos(y)$ is the start position of y. Otherwise, the new range is calculated and stored in L_{i+1} when i is less than k, or the end position of y is returned if i equals to k.

Algorithm 1.

```
1. Build the AC-automaton for the subpatterns $p_1, p_2, \ldots, p_k$
2. Scan text $T$ from $t_1$ to $t_n$, each time when an occurrence
   $x$ of $p_i$ is reported do
3.     if $i = 1$ or $startpos(x)$ is contained in the range in
           $L_i$ then
4.         if $i < k$ then
               Append the range $R(x) = [endpos(x) + N_i + 1,$
               $endpos(x) + M_i + 1]$ to the end of $L_{i+1}$.
5.         if $i = k$ then Report $endpos(x)$;
```

Example 2. Given $T = acacgactgcagctat$, $P = ac[0,6]g[2,5]ct$.

As shown Fig.1, at position 2, AC automation reports the occurrence of p_1, and Algorithm 1 stores the range (3, 9) to L_2. When reporting the occurrence of p_2 at position 5, the range (8, 11) is put into L_3, since $L_2 = \{(3,9), (5,11)\}$ contains 5. At position 8, although AC automation reports the occurrence of p_3, the algorithm does not return an occurrence of P, because 7 does not belong to $L_3 = \{(8,11)\}$. When the occurrence x of p_3 is at position 14, the algorithm returns 14 since $startpos(x) = 13$ belongs to $L_3 = \{(8,11), (12,15), (15,18)\}$.

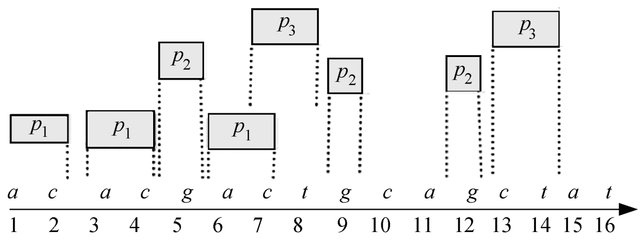

Fig.1. Example for Algorithm 1.

The time complexity is $O(n \log k + m + \alpha \log w)$, and the space complexity is $O(m + \alpha)$, where $m = \sum_{i=1}^{k} |p_i|$ is the sum of the lengths of the strings, α is the total number of occurrences of all subpatterns in the text, and $w = \max_{1 \leqslant i \leqslant k}(M_i - N_i)$ is the maximal gap length in P.

3.2.2 *Algorithm of Bille et al.*

Bille *et al.*[26] studied on how to increase the effectiveness.

In Example 2, let us look at the algorithm of Rahman *et al.* when the occurrence x of p_2 at position 9 is reported. At this time, $L_2 = \{(3,9), (5,11), (8,14)\}$, and $L_3 = \{(8,11)\}$. Since L_2 contains position 9, the range (12, 15) is put into L_3.

Bille *et al.* found that L_i might contain useless and overlapping ranges. Therefore in their algorithm, they remove useless ranges, and merge some ranges if they have overlaps. Using this algorithm, at position 9 of Example 2, $L_2 = \{(3,14)\}$ and $L_3 = \{(8,15)\}$, through reducing the ranges in L_i, the algorithm not only needs less memory to store the ranges, but also enhances the searching efficiency.

The algorithm of Bille *et al.* has a time complexity of $O(n \log k + m + \alpha)$ and a space complexity of

$O(m+A)$, where $A = \sum_{i=1}^{k} |N_i|$ is the sum of the lower local length constraints.

3.3 Comparison

Both algorithms perform on-line searching, and they scan the given text only once. The algorithm of Rahman *et al.* may require a large space when it is used for processing a large biological text, since α typically increases with the length of the text, hence the space complexity is likely to become a bottleneck. Conversely, the algorithm of Bill *et al.* is not affected much by the length of the text.

Both algorithms would perform poorly when most subpatterns in the given pattern are single letters, because all these subpatterns would be used to build the AC automaton. The AC automaton is a Trie structure and its effectiveness is reduced with single letters. A comparison is given in Table 3.

Table 3. Comparison of EPP Algorithms

	Rahman *et al.*	Bille *et al.*
Data structure	AC automaton	AC automaton
Space complexity	$O(n \log k + m + \alpha)$	$O(n \log k + m + \alpha)$
Time complexity	$O(m + \alpha)$	$O(m + A)$

4 NAM: Number of All Matches

4.1 Problem Statement

Let $P = p_1[N_1, M_1]p_2[N_2, M_2] \cdots [N_{m-1}, M_{m-1}]p_m$ where every p_i $(1 \leqslant i \leqslant m)$ is a single letter. If there is no wildcard between two adjacent characters in p, [0, 0] will be used for all $[N_i, M_i]$ $(i = 1, \ldots, m-1)$.

Definition 3 (Global Length Constraint). *The global length constraints* $[G_N, G_M]$ *specify the length range of each matching substring of* T *in which* P *occurs. Each matching substring has to satisfy* $\sum_{i=1}^{k} |p_i| + \sum_{i=1}^{k} |N_i| \leqslant G_N \leqslant G_M \leqslant \sum_{i=1}^{k} |p_i| + \sum_{i=1}^{k} |M_i|$.

Definition 4 (NAM Problem). *The objective of the NAM problem is to compute the number of all matches. In Example* 1*, the result of NAM is* 6*. NAM does not care about the specific positions of all matches.*

4.2 Algorithms

We present PAIG[27] and GCS[47] using the following example.

Example 3. $T = agaagaggaagaa$ and $P = a[0,2]g[1,2]a[0,3]a$.

4.2.1 GCS

GCS[47] employs a searching table to calculate the matching numbers of a pattern with wildcards. The matching table $\boldsymbol{H}$ is an $m \times n$ matrix of integers, where $H[i][j]$ is the number of matches of p_i ending at t_j. For example, $H[3][10] = 2$ indicates there are two matches of P ending at position 10. Given a pattern $P = p_1p_2 \ldots p_m$, and a text $T = t_1t_2 \ldots t_n$, the searching process of GCS consists of three steps. First, GCS scans the text from the left to the right. For any position j $(1 \leqslant j \leqslant n)$, if $t_j = p_1, H[1][j]$ is set to 1, otherwise $H[1][j]$ is set to 0. Second, for any position j $(1 \leqslant j \leqslant n)$, if $t_j = p_i$ $(1 < i \leqslant m)$, then $H[i][j]$ is updated to $\sum_{k=1}^{\max(1, j-g_{i-1})} H[i-1][k]$ where g_{i-1} is the gap length between p_{i-1} and p_i. Finally, the number of times that P occurs in T is $\sum_{j=1}^{n} H[m][j]$.

Table 4 is the matching table of GCS for Example 3. Firstly, the first row $\boldsymbol{H}[0]$ is initialized. Secondly, all the cells (except the first row) are updated according to the values of the previous row. For example, $H[2][5] = H[1][2] + H[1][3] + H[1][4] = 2$. Finally, GCS sums up the numbers of the last row and returns the sum as the total of matches. Therefore, the number of times is $H[4][6] + H[4][10] + H[4][12] + H[4][13] = 13$.

Table 4. Matching Table of GCS

T		1	2	3	4	5	6	7	8	9	10	11	12	13
		a	g	a	a	g	a	g	g	a	a	g	a	a
1	a	1	0	1	1	0	1	0	0	1	1	0	1	1
2	g	0	1	0	0	2	0	2	1	0	0	2	0	0
3	a	0	0	0	1	0	0	0	0	2	3	0	0	2
4	a	0	0	0	0	0	1	0	0	0	2	0	5	5

The GCS algorithm can be easily extended to calculate the support of a pattern according to the frequency information of its prefix pattern for sequential pattern mining. For this purpose, two maps, the rear map (RM) and the head map (HM), are constructed. The rear map records the number of occurrences and the ending positions of a pattern. $RM[x]$ indicates the number of times that a pattern ends at x. The head map is built to record locations and times of the length-2 pattern's starting position information. $HM[x]$ indicates the number of times that a pattern starts at x. Given a pattern $P = p_1p_2 \ldots p_l$, and its rear map $RM_{p_1p_2 \ldots p_l}$, the length-2 pattern $Q = p_lp_{l+1}$ and its head map $HM_{p_{l+1}}$, then the support of pattern $L = p_1p_2 \ldots p_lp_{l+1}$ is calculated by the following equation:

$$Sup(p_0p_1 \ldots p_lp_{l+1}) = \sum_{x=1}^{n} RM[x]HM[x].$$

For Example 3, the rear map of $P = a[0,2]g[1,2]a[0,3]a$, the head map of length-2 pattern $Q = a[0,2]g$, and the rear map of $L = a[0,2]g[1,2]a[0,3]a[0,2]g$ are given in Table 5. Then, $Sup(L)$ can be available through them, $Sup(L) =$

$RM_P[6] \times HM_Q[6] + RM_P[10] \times HM_Q[10] = 4$. The support of L can also be computed by $Sup(L) = RM_L[7] + RM_L[8] + RM_L[11] = 4$.

Table 5. Rear Map and Head Map for Example 3

T	1	2	3	4	5	6	7	8	9	10	11	12	13
	a	g	a	a	g	a	g	g	a	a	g	a	a
RM_P	0	0	0	0	0	1	0	0	0	2	0	5	5
HM_Q	1	0	1	2	0	2	0	0	1	1	0	0	0
RM_L	0	0	0	0	0	0	1	1	0	0	2	0	0

When calculating the matching numbers that p_i ends at the positions in the text, GCS needs to scan the whole text, and for each position, backtrack w steps where w is the gap length. Therefore, the time complexity of GCS is $O(mnw)$ where m is the pattern length, n is the text length, and w is the maximal gap length between any two consecutive characters. Furthermore, GCS needs $O(n)$ space to construct the matching table.

4.2.2 PAIG

PAIG[27] also uses an $n \times (m-1)$ matching table as illustrated in Table 4. The meaning of each cell in the table is explained as follows. The row information indicates the starting position of a match in the text; the column information indicates the length of the pattern; the numbers in the cells indicate the ending indices of matches; and the numbers in brackets indicate counters of respective matches. For example, 8(1), 9(2) on row 6 indicate that starting from t_6, there are $1 + 2 = 3$ matches of $P_2 = a[0,2]g[1,2]a$, 1 ending at t_9, and the other two ending at position 10. The final matching result is obtained through summing up all counters of the last column. It is 13 for the example in Table 6.

Table 6. Matching Table of PAIG

Position	T	$P_1 = a[0,2]g$	$P_2 = a[0,2]$ $g[1,2]a$	$P_3 = a[0,2]$ $g[1,2]a[0,3]a$
0	a	1(1)	3(1)	5(1)
1	g	-	-	-
2	a	4(1)	-	-
3	a	4(1), 6(1)	8(1), 9(1)	9(1), 11(2), 12(2)
4	g	-	-	-
5	a	6(1), 7(1)	8(1), 9(2)	9(1), 11(3), 12(3)
6	g	-	-	-
7	g	-	-	-
8	a	10(1)	12(1)	-
9	a	10(1)	12(1)	-
10	g	-	-	-
11	a	-	-	-
12	a	-	-	-

Computational time is saved through skipping some empty cells. Once an empty cell is identified, all remaining cells on the same row should also be empty. Therefore PAIG can simply skip them and go to the beginning of the next row. In applications, only a small fraction of the table needs to be filled.

For example, starting from position 2, there is no match of $P_2 = a[0,2]g[1,2]a$, thus there should be no match of $P_3 = a[0,2]g[1,2]a[0,3]a$, either. For the same reason, on row indices of 2, 5, 7, 8, 11, 12 and 13, PAIG only fills the first column.

There are n rows and $(m-1)$ columns in the matching table. The length of a variable length cell is at most $2\sum_{i=0}^{m-1} w_i = O(mw)$, where w is the maximal gap flexibility. Therefore, the time complexity of PAIG is $O(n \times m \times mw \times w \times \log(mw)) = O(nm^2w^2 \log(mw))$. The space complexity of PAIG is $O(n)$.

4.3 Comparison

Both PAIG and GCS employ an incremental approach to fill matching tables. In other words, they are dynamic programming oriented approaches. This is why they are efficient.

In PAIG, more information is recorded. Therefore the starting-ending pairs of matches are available. In GCS, only the ending points of matches are available. The data structure of GCS is simpler, and it is a matrix of integers. For this reason, PAIG can be easily revised to suit global length constraints, while GCS cannot. For Example 3, if we require that each match have a length between 7 and 9, then according to the last column in Table 4, the following starting-ending pairs satisfy this requirement: (3, 9), (3, 11), (5, 11) and (5, 12). There are $1 + 2 + 3 + 3$ matches.

PAIG and GCS have the same space complexity. While employing certain memory sharing techniques, the space complexity of PAIG is $O(mw)$[44]. However, it is required that the sequence be segmented and read a number of times. If we read the whole sequence into the memory, the space complexity is $O(n + mw) = O(n)$, since very long gaps or sequences are unreasonable. To reduce it further, GCS can fill a matching table row by row instead of column by column. The mod operation is also needed to keep only w rows. In that case, the space complexity is $O(mw)$. However, this approach is more complex, and also requires the segmentation of the sequence. In most applications, reading the whole sequence into the memory is a better choice.

The time complexity of GCS is more efficient than that of PAIG for the worst case, as given above. But PAIG is more efficient for the best case. According to the analysis in Subsection 4.2.2, when we fill the matching table of PAIG, if one element is empty, all remaining cells in the same row are simply skipped. Therefore the time complexity of PAIG is $O(n)$. However, each element in the matching table of GCS should be filled, and the time complexity is $O(nm)$. A comparison between these two algorithms is given in Table 7.

Table 7. Comparison of NAM Algorithms

		PAIG	GCS
Global length constraints		Yes	No
Data structure		Table	Table
Space complexity		$O(n)$	$O(n)$
Time complexity	Worst	$O(nm^2w^2\log(mw))$	$O(nmw)$
	Best	$O(n)$	$O(nm)$

5 OOC: One-off Condition

5.1 Problem Statement

Definition 5 (One-off Condition). *Suppose* $C_1 = (e_1, \ldots, e_m)$ *and* $C_2 = (h_1, \ldots, h_m)$ *are two matching position indices of* P *in* T. *If* $\forall 1 \leqslant i,\ j \leqslant m$, $e_i \neq h_j$ *then* C_1 *and* C_2 *satisfy the one-off condition.*

This one-off condition has some similarity but is different from the non-overlapping condition in [5] (mentioned in Subsection 2.2). [5] studies the pattern matching problem with unbounded wildcards, that is, the gap length between any two consecutive substrings can be arbitrarily large within the text.

Example 4. Given $T = tatttcc$, $P = t[1,2]t[1,2]c$. All occurrences of P in T are {(1, 3, 6), (1, 4, 6), (1, 4, 7), (3, 5, 7)}. (1, 4, 6) and (3, 5, 7) satisfy the one-off condition, while (1, 3, 6) and (3, 5, 7) do not, because position 3 occurs in both (1, 3, 6) and (3, 5, 7).

Definition 6 (OOC Problem). *The objective of the OOC problem is to find the maximum number of occurrences of* P *in* T*, on the condition that any two occurrences of* P *in* T *must satisfy the one-off condition. OOC outputs the matching positions.*

In Example 4, the objective of the OOC problem is to get the optimal solution {(1, 4, 6), (3, 5, 7)}.

5.2 Algorithms

5.2.1 SAIL

SAIL[29] scans the input text T. For every position i if T_i equals p_m, it checks whether there is an occurrence of P in T through a sliding window. SAIL first calculates the range of possible positions of p_1 by considering global constraints when scanning a possible position of p_m, and then deals with local constraints. Basically, SAIL conducts two phases, a *forward phase* and a *backward phase*. The matching results are provided by the following procedures of SAIL.

1) List $p_1, \ldots, p_j, \ldots, p_m$ and start from T_1 in a *forward* manner to locate every pattern letter p_j's possible positions in T.

2) SAIL sets "1" to the corresponding cell in Table 8 to indicate a seed position (see Definition 7 below). This mechanism guarantees that as long as a sequence $c_1, c_2, \ldots, c_j$ exists in T, which satisfies the corresponding local constraints, the cell for a seed position of p_j is marked with "1" in the search table.

3) SAIL keeps searching seed positions of every pattern letter p_j until one seed position for the last pattern letter p_m is found. Once one seed position of p_m is found, the *backward phase* is triggered to locate an optimal occurrence of P in T. Actually one seed position of p_m indicates that there is a sequence $c_1, c_2, \ldots, c_m$ which satisfies the local constraints, consequently, in the *backward phase*, an optimal occurrence is guaranteed to be returned.

4) In the *backward phase*, using c_m, a seed position of p_m, which trivially becomes an optimal position $\overleftrightarrow{C_m}$ of p_m, SAIL finds $\overleftrightarrow{C_{m-1}}$. Using $\overleftrightarrow{C_{m-1}}$, SAIL finds $\overleftrightarrow{C_{m-2}}$. This procedure is repeated until all optimal positions of one optimal occurrence are located. The search table is re-initialized for the next round search. Letters used for matching this occurrence are marked to avoid being used again (for the OCC condition). In Table 8, the trace of one optimal occurrence has been highlighted.

Table 8. Search Table for $P = a[0,3]e[2,3]f[0,3]b[0,3]g$ and $T = aadfeefefhbbg$

	a	a	d	f	e	e	f	e	f	h	b	b	g
a	$\uparrow$ **1** $\leftarrow$	1 $\leftarrow$	0 $\leftarrow$	0	0	0	0	0	0	0	0	0	0
e	0	0	0	0	$\uparrow$ **1** $\leftarrow$	1 $\leftarrow$	0	0	0	0	0	0	0
f	0	0	0	0	0	0	0	0	$\uparrow$ **1**	0 $\leftarrow$	0	0	0
b	0	0	0	0	0	0	0	0	0	0	$\uparrow$ **1**	1 $\leftarrow$	0
g	0	0	0	0	0	0	0	0	0	0	0	0	$\uparrow$ **1**

Definition 7 (Seed Position). *When* $j = 1$, *position* i *is a seed position of* p_j *if* $t_i = p_j$; *and when* $2 \leqslant j \leqslant m$, *if* $t_i = p_j$ *and* $\exists$ *a seed position* x *of* p_{j-1} *such that* $N_{j-1} \leqslant y - x - 1 \leqslant M_{j-1}$, *position* y *is a seed position of* p_j.

Two important issues were taken into consideration to design SAIL.

1) Online searching: SAIL returns each matching substring of P in T as soon as it appears in the input T.

2) Optimization: under the one-off condition, SAIL determines which occurrence is an optimal one if multiple occurrences exist for a p_k's position by applying the left-most strategy.

In Example 4, SAIL's output is (1, 3, 6), which is not the optimal solution {(1, 4, 6), (3, 5, 7)}. Hence, SAIL is not a complete algorithm, because it loses occurrences when the text is available in an offline manner[48]. Meanwhile, the completeness of SAIL has been proven under a particular restriction[48], i.e., if the pattern does not have any recurring letters.

The time complexity of SAIL is $O(n + fG_M mw)$, where f is the frequency of P's last letter occurring in T. The space complexity of SAIL is $O(mL)$, where $L = m + \sum_{i=1}^{k} |b_i|$ is the maximum length of a pattern occurrence with local length constraints.

5.2.2 *BPBM*

BPBM[30] is an online algorithm based on bit-parallel technology to simulate the matching process which adopts two nondeterministic finite state automatons (NFAs). The first NFA is used to identify pattern matching. When it moves to a terminal state, it indicates a pattern match in the current search window. The second NFA is used to identify whether there exists a prefix of the pattern in the current search window to accelerate the scanning process by dropping useless sequences in the text.

In BPBM, given a pattern P, the first automaton is defined by $A = (\sum_P, I, F, D)$, where $|\sum_P|$ is the number of letters in the pattern alphabet that contains all distinct letters in pattern P without wildcards. The bit masks I, F, D are used to represent the gap initial state, the gap final state, and the transition state of P, respectively. I has 1 in the gap initial bits, and F has 1 in the gap final bits. D has 1 in the active bits corresponding to each NFA active state of letter $\delta \in \sum_P \cup \{\phi\}$ in P, where ϕ denotes a wildcard. The BPBM algorithm builds a table B, which also stores bit masks $b_1 \dots b_L$ for wildcards and each letter in the pattern. For letter $\delta \in \sum_P\{\phi\}$, we set the j-th bit to 1 if $p_j = \delta$ or $p_j = \phi$.

BPBM obtains D by the following formulae:

$$D = (D \ll 1)\&B[\delta],$$
$$D = D|((F - (D\&I))^\wedge F).$$

The rationale is as follows. First, $D = D\&B[\delta]$ checks whether the current letter δ is active. $D\&I$ isolates the active gap initial bits. Subtracting this from F generates ε-transition outcomes from each gap final bit to the corresponding gap initial bit b_i. If b_i is active, the outcome will have 1 only in bit $(b_i + N_i + M_i - 1)$ from bit b_i. Else, the outcome will have 1 at the gap final bit. Then, it fills the upper limit of a gap final bit with "^F" operation, successfully propagating the active bits to the desired target bits. Once the propagation has been done, we OR the results with the already active bits in D, as shown in Table 9.

The existing algorithms, Gaps-Shift-And and Gaps-BNDM[17], based on the bit-parallelism technology cannot handle a pattern that has only one letter between successive wildcards and when the minimum local length constraints are zero. BPBM can not only handle these conditions through improving the formula of ε-transition in the matching process, but also return a concrete matching position sequence, which makes it more applicable.

Table 9. BPBM with $P = a[0,2]g[0,3]g$ and $T = aagg$

Preprocessing	$I = 00010001$, $F = 01001000$, $B[a] = 11101110$, $B[g] = 01111111$, $B[\phi] = 011001110$, $D = 00000001$	
T	D	Description
g	00001111	D right transition. The 5th~8th states are active.
g	01111111	D right transition. The 2nd~4th states are active.
a	11101110	D right transition. Have an occurrence.
a	11011100	D right transition. Have an occurrence.

The time complexity of BPBM is $O(n+fG_M\lceil L/w\rceil)$, where w is the number of bits in a machine word. BPBM is superior to SAIL, when L has a low w. The space complexity is $O((m + |\sum_P|)\lceil L/w\rceil)$.

5.2.3 *SBO*

SBO[49] is an off-line algorithm based on a new non-linear structure called Nettree. A Nettree is a kind of directed acyclic graph (DAG) with two kinds of edge labels, "parent-child" and "child-parent", where each node has zero or more children nodes and zero or more parent nodes. The creation steps of the Nettree are given below.

SBO keeps searching seed positions of every pattern letter p_j ($1 \leqslant j \leqslant m$). Once one seed position of p_j is found at T_i, node n_j^i will be created in the j-th layer, and the node will be added at the tail of the j-th layer. If n_j^i is a seed position of seed position n_{j-1}^q, a parent-child relation between node n_j^i and n_{j-1}^q will be created. Meanwhile, from the m-th level to the first layer, a child-parent relation will be created, too. For each node n_j^i, SBO defines n_j^i's indegree $R_{\text{in}}(n_j^i)$ as the number of paths from the layer level to n_j^i, n_j^i's outdegree $R_{\text{out}}(n_j^i)$ as the number of paths from n_j^i to the m-th layer, and $R(j) = \sum_{j=1}^{m}(R_{\text{in}}(n_j^i) \times R_{\text{in}}(n_j^i))$ as the degree of the position T_j. For Example 4, the Nettree is shown in Fig.2, $R_{\text{in}}(n_2^3) = 1$, $R_{\text{out}}(n_2^3) = 1$, $R(3) = R(n_1^3) + R(n_2^3) = 2$.

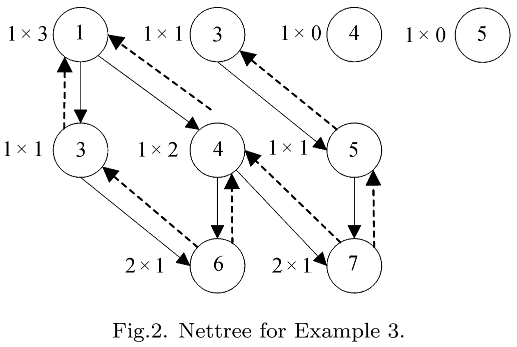

Fig.2. Nettree for Example 3.

The SBO algorithm has two strategies: strategy of greedy-search parent (SGSP) and strategy of right-most parent (SRMP), to find two occurrences of the same m-th layer, and then select the better occurrence from the results of SGSP and SRMP. The main idea of SGSP is to find an approximately optimal parent according to the degree of the position. If there are many candidate positions, SGSP selects the position whose degree R is the smallest. SRMP is to find the right-most parent of the current node at each step in the process of searching for an occurrence.

The time complexity of SBO is $O(wn(n+m^2))$. The space complexity of SBO is $O(wmn)$.

5.3 Comparison

SBO consumes more space than SAIL and BPBM, because the Nettree is a tree and contains the whole candidate seed positions. In general, BPBM has a lower time and space complexity than SAIL. All three algorithms adopt heuristic strategies, and cannot guarantee the completeness. When the text is available in an offline manner, SBO can obtain more occurrences of the pattern than SAIL and BPBM in most cases as it relies on using two heuristic strategies repeatedly, and meanwhile, it also consumes more time than SAIL and BPBM. A comparison is given in Table 10.

Table 10. Comparison of OOC Algorithms

	SAIL	BPBM	SBO
Matching strategy	Left-most	Left-most	SGSP, SRMP
Data structure	Sliding window	Bit-parallel	Nettree
Time complexity	All polynomial time and SBO > SAIL > BPBM		
Space complexity	SBO > SAIL > BPBM		
Completeness	All incompleteness, in general SBO > SAIL = BPBM		

6 Conclusions

In this paper, we have presented state-of-the-art algorithms for dealing with pattern matching with flexible wildcards (PMFW), along three lines of efforts, EPP, NAM and OOC.

For the EPP problem, the algorithm of Bille *et al.* has a high efficiency in time and space, when most of the subpatterns are not just a single letter. In the NAM problem, PAIG is more efficient when the gaps in the pattern are not flexible, while GCS is more efficient for the other case. For the OCC problem, existing algorithms can get complete solutions when the pattern does not have recurring letters. With the online condition, BPBM has a better performance than the others. In the offline condition, SBO can get more occurrences than the others in most cases.

PMFW is still an open problem. Much work still needs to be along the following directions.

Further Exploration of the NP-Hard Complexity of the OOC Problem. For the OOC problem, there does not yet exist any polynomial algorithm that can achieve a complete solution in a general case, and therefore, it is assumed to be NP-hard[24,29].

Multiple Pattern Matching with Flexible Wildcards. Multiple pattern matching with flexible wildcards is also an important research task. How to expand existing pattern matching algorithms with multiple patterns at the same time? The objective is to achieve optimized solutions in both efficiency and solution coverage. The following three key issues need further investigations: 1) sharing the search space of common subpatterns among multiple patterns, 2) resource competition constraints among multiple patterns, and 3) association analysis among multiple patterns.

Approximate Pattern Matching with Flexible Wildcards. The PMFW problem can be extended with approximate matching. A new problem allowing the positions and the number of wildcards in the pattern to have possible uncertainty should be studied. Since the matching positions of the pattern in a text is uncertain, the spatial properties of the pattern become more complex, which could result in more challenges on how to design and analyze matching algorithms. This problem would play an important role in sequence alignment, protein structure prediction and so on.

Applications in Bioinformatics. Studies have shown that many human diseases are related to some repeating parts of genes, such as bacteria, viruses, and nervous system diseases. Sequence pattern mining with wildcards can help to find some interesting repetitive patterns. From the biological point of view, evaluating sequence patterns by exploring sequential pattern mining in bioinformatics can provide further insights.

References

[1] Cole J R, Chai B, Farris R J *et al.* The Ribosomal Database Project (RDP-II): Sequences and tools for high-throughput rRNA analysis. *Nucleic Acids Research*, 2005, 33(Database Issue): 294-296.

[2] Mendivelso J, Pinzón Y, Lee I. Finding overlaps within regular expressions with variable-length gaps. In *Proc. the 2013 Research in Adaptive and Convergent Systems*, Oct. 2013, pp.16-21.

[3] Patnaik D, Laxman S, Chandramouli B, Ramakrishnan N. A general streaming algorithm for pattern discovery. *Knowledge and Information Systems*, 2013, 37(3): 585-610.

[4] Xie F, Wu X, Hu X *et al.* Sequential pattern mining with wildcards. In *Proc. the 22nd IEEE International Conference on Tools with Artificial Intelligence*, Oct. 2010, pp.241-247.

[5] Ding B, Lo D, Han J, Khoo S. Efficient mining of closed repet-

itive gapped subsequences from a sequence database. In *Proc. the 25th IEEE International Conference on Data Engineering*, Mar. 29-April 2, 2009, pp.1024-1035.

[6] El-Ramly M, Stroulia E, Sorenson P. From run-time behavior to usage scenarios: An interaction-pattern mining approach. In *Proc. the 8th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, July 2002, pp.315-324.

[7] Manber U, Baeza-Yates R. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 1991, 37(3): 133-136.

[8] de Pablo-Sánchez C, Segura-Bedmar I, Martínez P, Iglesias-Maqueda A. Lightly supervised acquisition of named entities and linguistic patterns for multilingual text mining. *Knowledge and Information Systems*, 2013, 35(1): 87-109.

[9] Barbieri N, Bonchi F, Manco G. Topic-aware social influence propagation models. *Knowledge and Information Systems*, 2013, 37(3): 555-584.

[10] Wei Y, Dominique F, Jean-Paul B. An automatic keyphrase extraction system for scientific documents. *Knowledge and Information Systems*, 2013, 34(3): 691-724.

[11] Fischer M J, Paterson M S. String matching and other products. Technical Report, Massachusetts Institute of Technology, 1974.

[12] Muthukrishnan S, Palem K. Non-standard stringology: Algorithms and complexity. In *Proc. the 26th Annual ACM Symposium on Theory of Computing*, May 1994, pp.770-779.

[13] Indyk P. Faster algorithms for string matching problems: Matching the convolution bound. In *Proc. the 39th Symp. Foundations of Computer Science*, Nov. 1998, pp.166-173.

[14] Clifford P, Clifford R. Simple deterministic wildcard matching. *Information Processing Letters*, 2007, 101(2): 53-54.

[15] Cole R, Hariharan R. Verifying candidate matches in sparse and wildcard matching. In *Proc. the 34th Annual ACM Symposium on Theory of Computing*, May 2002, pp.592-601.

[16] Guo D, Hu X, Xie F, Wu X. Pattern matching with wildcards and gap-length constraints based on a centrality-degree graph. *Applied Intelligence*, 2013, 39(1): 57-74.

[17] Navarro G, Raffinot M. Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In *Proc. the 5th Annual International Conference on Computational Biology*, April 2001, pp.231-240.

[18] Morgante M, Policriti A, Vitacolonna N, Zuccolo A. Structured motifs search. *Journal of Computational Biology*, 2005, 12(8): 1065-1082.

[19] Cole R, Gottlieb L, Lewenstein M. Dictionary matching and indexing with errors and don't cares. In *Proc. the 36th Annual ACM Symposium on the Theory of Computing*, June 2004, pp.91-100.

[20] Kalai A. Efficient pattern-matching with don't cares. In *Proc. the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2002, pp.655-656.

[21] Haapasalo T, Silvasti P, Sippu S *et al.* Online dictionary matching with variable-length gaps. In *Proc. the 10th Int. Conf. Experimental Algorithms*, May 2011, pp.76-87.

[22] Kucherov G, Rusinowitch M. Matching a set of strings with variable length don't cares. *Theoretical Computer Science*, 1997, 178(1/2): 129-154.

[23] Zhang M, Zhang Y, Hu L. A faster algorithm for matching a set of patterns with variable length don't cares. *Information Processing Letter*, 2010, 110(6): 216-220.

[24] Wu X, Zhu X, He Y, Arslan A N. PMBC: Pattern mining from biological sequences with wildcard constraints. *Computers in Biology and Medicine*, 2013, 43(5): 481-492.

[25] Rahman M S, Iliopoulos C S, Lee I *et al.* Finding patterns with variable length gaps or don't cares. In *Proc. the 12th Annual International Computing and Combinatorics Conference*, August 2006, pp.146-155.

[26] Bille P, Gørtz I L, Vildhøj H W, Wind D K. String matching with variable length gaps. *Theoretical Computer Science*, 2012, 443(20): 25-34.

[27] Min F, Wu X, Lu Z. Pattern matching with independent wildcard gaps. In *Proc. the 8th IEEE Int. Conf. Dependable, Autonomic and Secure Computing*, December 2009, pp.194-199.

[28] Zhu X, Wu X. Mining complex patterns across sequences with gap requirements. In *Proc. the 20th Int. Joint Conf. Artificial Intelligence*, January 2007, pp.2934-2940.

[29] Chen G, Wu X, Zhu X, Arslan A, He Y. Efficient string matching with wildcards and length constraints. *Knowledge and Information Systems*, 2006, 10(4): 399-419.

[30] Guo D, Hong X, Hu X *et al.* A bit-parallel algorithm for sequential pattern matching with wildcards. *Cybernetics and Systems*, 2011, 42(6): 382-401.

[31] Lin P C, Li Z X, Lin Y D *et al.* Profiling and accelerating string matching algorithms in three network content security applications. *IEEE Communications Surveys and Tutorials*, 2006, 8(2): 24-37.

[32] Aho A V, Corasick M J. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 1975, 18(6): 333-340.

[33] Tuck N, Sherwood T, Calder B, Varghese G. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proc. the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, March 2004, pp.2628-2639.

[34] Norton M. Optimizing pattern matching for intrusion detection. http://pdf.aminer.org/000/309/890/optimizing_pattern_match.pdf, July 2014.

[35] Boyer R S, Moore J S. A fast string searching algorithm. *Communications of the ACM*, 1977, 20(10): 762-772.

[36] Wu S, Manber U. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona, 1994.

[37] Muth R, Manber U. Approximate multiple string search. In *Proc. the 7th Annual Symposium on Combinatorial Pattern Matching* (*CPM*), June 1996, pp.75-86.

[38] Karp R M, Rabin M O. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 1987, 31(2): 249-260.

[39] Baeza-Yates R, Gonnet G H. A new approach to text searching. *Communications of the ACM*, 1992, 35(10): 74-82.

[40] Navarro G, Raffinot M. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proc. the 9th Annual Symp. Combinatorial Pattern Matching*, July 1998, pp.14-33.

[41] Navarro G. A guided tour to approximate string matching. *ACM Computing Surveys*, 2001, 33(1): 31-88.

[42] Kim S, Kim Y. A fast multiple string pattern matching algorithm. In *Proc. the 17th AoM/IAoM Conference on Computer Science*, August 1999.

[43] Agrawal R, Srikant R. Mining sequential patterns. In *Proc. the 11th Int. Conf. Data Engineering*, March 1995, pp.3-14.

[44] Akutsu T. Approximate string matching with variable length don't care characters. *Information Processing Letters*, 1995, 55(5): 235-239.

[45] Lee I, Apostolico A, Iliopoulos C S, Park K. Finding approximate occurrences of a pattern that contains gaps. In *Proc. the 14th Australasian Workshop on Combinatorial Algorithms*, July 2003, pp.89-100.

[46] Zhang M, Kao B, Cheung D W, Yip K. Mining periodic patterns with gap requirement from sequences. In *Proc. the ACM SIGMOD International Conference on Management of Data*, June 2005, pp.623-633.

[47] Min F, Wu X. A comparative study of pattern matching algorithms on sequences. In *Proc. the 12th International Confer-*

ence on Rough Sets, Fuzzy Sets, Data Mining and Granular Computing, Dec. 2009, pp.510-517.

[48] Wang H, Xie F, Hu X, Li P, Wu X. Pattern matching with flexible wildcards and recurring characters. In *Proc. the 2010 IEEE International Conference on Granular Computing*, Aug. 2010, pp.782-786.

[49] Wu Y, Wu X, Jiang H, Min F. A heuristic algorithm for MPMGOOC. *Chinese Journal of Computers*, 2011, 34(8): 1452-1462. (in Chinese)

Xindong Wu is a Yangtze River Scholar in the School of Computer Science and Information Engineering at the Hefei University of Technology, China, a professor of computer science at the University of Vermont, USA, and a fellow of IEEE and AAAS. He received his B.S. and M.S. degrees in computer science from the Hefei University of Technology, China, and his Ph.D. degree in artificial intelligence from the University of Edinburgh, Britain. His research interests include data mining, big data analytics, knowledge-based systems, and Web information exploration. He is currently the steering committee chair of the IEEE International Conference on Data Mining (ICDM), the editor-in-chief of Knowledge and Information Systems (KAIS, by Springer), and a series editor-in-chief of the Springer Book Series on Advanced Information and Knowledge Processing (AI & KP). He was the editor-in-chief of the IEEE Transactions on Knowledge and Data Engineering (TKDE, by the IEEE Computer Society) between 2005 and 2008. He served as program committee chair/co-chair for the 2003 IEEE International Conference on Data Mining, the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, and the 19th ACM Conference on Information and Knowledge Management.

Ji-Peng Qiang received his B.S. degree in computer science from Hefei University in 2010, M.S. degree in computer science from Hefei University of Technology in 2013. Currently he is pursuing his Ph.D. degree in computer science from Hefei University of Technology. His research interests include pattern matching and multiple document summarization.

Fei Xie is an associate professor in the Department of Computer Science and Technology, Hefei Normal University. He received his M.S. and Ph.D. degrees both in computer science from Hefei University of Technology. Currently, he is a postdoctoral fellow in computer science of Hefei University of Technology. His research interests include pattern mining and text mining.