Open access • Journal Article • DOI:10.1007/S10489-013-0499-4

Mining sequential patterns with periodic wildcard gaps — Source link

Youxi Wu, Lingling Wang, Jiadong Ren, Wei Ding ...+1 more authors

Institutions: Hebei University of Technology, Yanshan University, University of Massachusetts Boston, University of Vermont

Published on: 01 Jul 2014 - Applied Intelligence (Springer US)

Topics: Wildcard, Pattern matching, Data structure, Breadth-first search and Heuristic (computer science)

Mining Sequential Patterns with Periodic Wildcard Gaps

Youxi Wu, Lingling Wang, Jiadong Ren, Wei Ding, Xindong Wu

Abstract Mining frequent patterns with periodic wildcard gaps is a critical data mining problem to deal with complex real-world problems. This problem can be described as follows: given a subject sequence, a pre-specified threshold, and a variable gap-length with wildcards between each two consecutive letters. The task is to gain all frequent patterns with periodic wildcard gaps. State-of-the-art mining algorithms which use matrix or other linear data structures to solve the problem not only consume a large amount of memory but also run slowly. In this study, we use an Incomplete Nettree structure (the last layer of a Nettree which is an extension of a tree) of a sub-pattern *P* to efficiently create Incomplete Nettrees of all its super-patterns with prefix pattern *P* and compute the numbers of their supports in a one-way scan. We propose two new algorithms, MAPB (Mining sequentiAl Pattern using incomplete Nettree with Breadth first search) and MAPD (Mining sequentiAl Pattern using incomplete Nettree with Depth first search), to solve the problem effectively with low memory requirements. Furthermore, we design a heuristic algorithm MAPBOK (MAPB for tOp-K) based on MAPB to deal with the Top-*K* frequent patterns for each length. Experimental results on real-world biological data demonstrate the superiority of the proposed algorithms in running time and space consumption and also show that the pattern matching approach can be employed to mine special frequent patterns effectively.

Keywords Sequential pattern mining, Periodic wildcard gap, Pattern matching, Heuristic algorithm, Nettree

Y. Wu (✉), L. Wang, J. Ren, W. Ding, X. Wu

School of Computer Science and Engineering, Hebei University of Technology, Tianjin 300130, China
e-mail:wuc567@163.com

L. Wang
e-mail:wangling927927@126.com

J. Ren
School of Information Science and Engineering, Yanshan Univeristy, Qinghuangdao 066004 , China
e-mail:jdren@ysu.edu.cn

W. Ding
Department of Computer Science, University of Massachusetts Boston, Boston, 02125, USA
e-mail:wei.ding@umb.edu

X. Wu
Department of Computer Science, University of Vermont, Burlington, VT 05405, USA
e-mail:xwu@cs.uvm.edu

1. Introduction

Pattern mining plays an essential role in many critical data mining tasks, such as graph mining [1], spatiotemporal data mining [2], and univariate uncertain data streams [3]. Sequential pattern mining, first introduced by Agrawal and Srikant [4], is a very important data mining problem [5] with broad applications including analyzing moving object data [6], detecting intrusion [7], and discovering changes in customer behaviour [8] etc. In order to solve the specific applications, researchers have also proposed some special approaches. For example, Liao and Chen [9] proposed a Depth-First SPelling algorithm for mining sequential patterns in long biological sequences. To avoid producing a large number of uninteresting and meaningless patterns, Hu *et al.* [10] proposed an algorithm which considers compactness, repetition, recency and frequency jointly to select sequential patterns and it is efficient in the business-to-business environment. Shie *et al*. [11] proposed an efficient algorithm named IM-Span to mine user behaviour patterns in mobile commerce environments. Yin *et al.* [12] proposed an efficient algorithm USpan for mining high-utility sequential patterns. Zhu *et al.* [13] proposed an efficient algorithm Spider-Mine to mine the Top-*K* largest frequent patterns from a single massive network with any user-specified probability. Wu *et al.* [14] proposed a novel framework for mining the Top-*K* high-utility itemsets. Classical sequential pattern mining algorithms include GSP [4] and Prefix Span [15].

Researchers have recently focused on periodic detection [16] and mining frequent patterns with periodic wildcard gaps [17], since this kind of pattern can flexibly reflect sequential behaviours and is often exhibited in many real-world fields. For example, in business, retail companies may want to know what products customers will usually purchase at regular time intervals rather than in continuous time according to time gaps. So managers can adjust marketing methods or strategies and improve sales profit [18, 19]. In biology, periodic patterns are redeemed as having significant biological and medical values. Considering that genes and proteins both contain many repetitive fragments, many periodic patterns of different lengths and types can therefore be found. Some of the patterns with excessively high frequency have been determined as the suspected cause of some diseases [20, 21]. In data mining, researchers found that some frequent periodic patterns can be used in feature selection for the purpose of classification and clustering [22, 23]. So an extraordinary task is how to mine all these frequent patterns and the Top-*K* patterns for each length from a mass of data to improve its efficiency in each application field.

In this study, we focus on mining frequent patterns with periodic wildcard gaps. This problem can be described as follows: given a subject sequence, a pre-specified threshold, and a variable gap-length with wildcards between each two consecutive letters. Our goal is to gain all frequent patterns with periodic wildcard gaps and the Top-*K* frequent patterns for each length. The pattern with periodic wildcard gaps means that all these wildcard gap constraints are the same. For example, pattern $P1=p_0[min_0, max_0]p_1[min_1, max_1]p_2$ =A[0,

2]T[0, 2]G is a pattern with periodic wildcard gaps since min_0=min_1=0 and max_0=max_1 =2. We can easily know that both patterns $P2$=p_0[min_0, max_0]p_1[min_1, max_1]p_2=A[1,2]T[0,2]G and $P3$=p_0[min_0, max_0]p_1[min_1, max_1]p_2=A[0,3]T[0,2]G are not patterns with periodic wildcard gaps since the wildcard gaps between letters are not completely the same.

In existing work, a series of algorithms had been proposed on the pattern mining with flexible wildcards, as detailed in Sect. 2. But all these state-of-the-art mining algorithms use a matrix or other linear data structure to solve the problem and consume a large amount of memory and run slowly [20, 24, 25]. It is also very difficult to mine frequent patterns in long sequences. Therefore, we propose two more efficient mining algorithms by using an incomplete Nettree structure and can get all frequent patterns in a shorter time with less memory space. In addition, we also design a heuristic algorithm to gain the Top-K frequent patterns for each length effectively. The contributions of this paper are described specifically as follows:

- We propose an incomplete Nettree structure (see Part 4.2) which can be used to compute the numbers of supports of those super-patterns in a one-way scan.
- We design two algorithms named MAPB (Mining sequentiAl Pattern using incomplete Nettree with Breadth first search) and MAPD (Mining sequentiAl Pattern using incomplete Nettree with Depth first search) to solve the problem effectively with low memory requirements. Experiments on real-world bio-data show that MAPD is much faster than MAPB and can be employed to mine frequent patterns in long sequences.
- Furthermore, a heuristic algorithm named MAPBOK (MAPB for tOp-K) which is designed based on MAPB is presented to solve the Top-K mining problem, although MAPB is slower than MAPD. This algorithm can efficiently get the Top-K frequent patterns for each length without having to sort all frequent patterns or to check all possible candidate patterns.

The paper is organized as follows. Section 2 reviews related work. Section 3 presents the problem definition and preliminaries. Section 4 introduces Nettree and the incomplete Nettree structure and then proposes the MAPB and MAPD algorithms. The analysis of the time and space complexities of these two algorithms and an illustrative example are also shown in section 4. Section 5 proposes a heuristic algorithm to find the Top-K frequent patterns for each length and the longest frequent patterns. Section 6 reports comparative studies and Section 7 concludes the paper.

2. Related work

We discuss the existing work from the following features:

(1) Pattern P has many wildcard gap constraints. In some studies the gap constraints can be different [26]. But in this paper all these wildcard gap constraints are the same. The pattern is called a pattern with periodic wildcard gaps.

(2) In some studies, only one or two positions of the given sequence are considered [23], whereas we consider that a group of positions are used to denote a support of a pattern in the given sequence. This means that each position p_j should be considered. For example, for a given sequence S=$s_0s_1s_2s_3s_4s_5$= AATTGG and a pattern P=p_0[min_0, max_0]p_1[min_1, max_1]p_2=A[0,2]T[0,2]G, <0, 2, 4> is a support of P in S due to the fact that s_0=p_0=A, s_2=p_1=T, and s_4=p_2=G. So it is easy for us to know that there are 2*2*2=8 supports in the problem.

(3) Each position can be used many times in this study. In the literature [26-28] each position in the sequence can be used at most once, which is called the one-off condition first introduced by Chen *et al*. [29]. Under the one-off condition there are only two supports of P=A[0,1]T[0,1]G in S=$s_0s_1s_2s_3s_4s_5$=AATTGG which are <0, 2, 4> and <1, 3, 5>. Another kind of study is under the non-overlapping condition [30]. For instance, given $S1$=$s_0s_1s_2s_3s_4$ =ATATA and $P1$=A[0,1]T[0,1]A, it is easy to know that <0, 1, 2> and <2, 3, 4> are two supports of $P1$ in $S1$. We notice that position 2 is used twice in <0, 1, 2> and <2, 3, 4>, but position 2 does not appear at the same indication of the supports. Hence <0, 1, 2> and <2, 3, 4> are two supports of $P1$ in $S1$ under the non-overlapping condition rather than the one-off condition. But in this study, we do not have these kinds of constraints.

(4) In the literature [26, 27, 30] it is shown that the Apriori property can be used to prune candidate patterns if the sequential pattern mining has the one-off condition or the non-overlapping condition, since the number of supports of any super-pattern is less than that of its sub-patterns under these conditions. So if a pattern is not frequent, all its super-patterns are not frequent. Whereas the Apriori property does not hold in our mining problem, since the number of supports (even support ratio) of a super-pattern can be greater than that of its sub-patterns. To reduce redundant candidate patterns, the Apriori-like property was proposed [20]. The property can be described as that all super-patterns are not frequent if the support ratio of a pattern is less than a value which is less than the given threshold.

(5) Generally, pattern mining algorithms can find all frequent patterns whose number of supports is not less than the given threshold. However, it is not easy for users to use the frequent patterns when they face thousands of patterns or more. In some cases, parts of the duplicated frequent patterns need to be removed to prune the uninteresting patterns because these patterns and their super-patterns have the same number of supports. This is called closed frequent patterns. Moreover, some issues focus on the Top-K mining problem [13, 14, 31] since the Top-K frequent patterns are more useful than other frequent patterns.

Mining frequent patterns with periodic wildcard gaps essentially relies on a counting mechanism to calculate the number of supports (or occurrences) of a pattern and then determine whether the pattern is frequent or not. However, on one hand, the number of supports generally grows exponentially with the length of the checked patterns. On the other hand, both the number of supports and support ratio do not satisfy monotonicity and the Apriori property cannot be used to reduce the size of the set of candidate patterns. Thus it is infeasible for traditional algorithms to solve this problem. Researchers have explored several major approaches to untangle this hard problem: (1) The set of length-m candidate patterns is used to generate a set of length-(m+1) candidate patterns. Then select frequent patterns from the set of candidate patterns and iterate this process till it is done [20, 22, 23]. (2) The Apriori-like property is employed to reduce the size of the set of candidate patterns [20]. (3) Min *et al*. [24] redefined the problem of [20] and the support ratio is monotonous in the new definitions. So the Apriori property can be used. (4) Pattern matching techniques are used to calculate the numbers of supports of patterns in the given sequence to find frequent patterns [24, 25].

Table 1 shows a comparison of related work.

Table 1. Comparison of related work

Algorithms	Periodic wildcard gaps	Apriori property	Occurrences	Output patterns
He *et al.* [26]	No	Apriori	One-off	All frequent patterns
Ding *et al.* [30]	No	Apriori	Non-overlap	All/closed frequent patterns
Xie *et al.* [27]	Yes	Apriori	One-off	All frequent patterns
Li *et al.* [23]	Yes	Apriori	First-last	Closed/long frequent patterns
Ji *et al.* [21]	Yes	Apriori-like	All	Minimal distinguishing subsequence patterns
Min *et al.* [24]	Yes	Apriori	All	All frequent patterns
Zhang *et al.* [20]	Yes	Apriori-like	All	All frequent patterns
Zhu and Wu [25]	Yes	Apriori-like	All	All frequent patterns
This paper	Yes	Apriori-like	All	All/Top-*K* frequent patterns for each length

In Table 1, Zhang *et al.* [20] studied the problem of mining frequent patterns with periodic wildcard gaps in a genome sequence and proposed the MPP algorithm based on the Apriori-like property which employed an effective pruning mechanism to reduce the size of the set of candidate patterns. Min *et al.* [24] redefined the issue of [20] and the Apriori property can become available. The mining algorithm uses a pattern matching approach to calculate the number of supports (or occurrences) of a pattern in the given sequence. Zhu and Wu [25] explored the MCPaS algorithm to discover frequent patterns with periodic wildcard gaps from multi-sequences. The proposed Gap Constrained Search (GCS) algorithm used a sparse array to calculate the number of supports of a pattern. Ji *et al.* [21] proposed an algorithm to mine all minimal distinguishing subsequences satisfying the minimum and maximum gap constraints and a maximum length constraint. Li *et al.* [23] proposed Gap-BIDE to discover the complete set of closed sequential patterns with gap constraints, and Gap-Connect to mine the approximate set of long patterns. They further explored several feature selection methods from the set of gap-constrained patterns on the issue of classification and clustering. A triple (*bP*, *eP*, *count*) was used to represent a set of supports of a pattern which share the same beginning position and ending position, where *bP*, *eP*, and *count* are the beginning position of the supports in the sequence, the ending position of the supports in the sequence, and the total number of supports, respectively. Xie *et al.* [27] and He *et al.* [26] both studied the problem of mining frequent patterns under the one-off condition which can greatly improve the time performance. In [27], the MAIL algorithm employing the left-most and the right-most pruning methods is designed to find frequent patterns with periodic wildcard gaps. In [26], two heuristic algorithms, namely one-way scan and two-way scan, are proposed to mine all frequent patterns. Ding *et al.* [30] studied the repetitive gapped subsequence mining problem and focused on mining closed frequent patterns with non-overlapping supports.

According to Table 1, we can see that studies [20, 24, 25] are the most closely related with our study. The support ratio is used to determine whether a pattern is frequent or not in this kind of issue and a pattern and its super-patterns generally have different support ratios. So we pay more attention to mining all frequent patterns rather than closed frequent patterns. Whereas the algorithm in [20] is less effective to mine all frequent patterns, especially in long sequences. Moreover, traditional Top-*K* mining studies such as Zhu *et al.* [13] and Wu *et al.* [14] focus on mining the Top-*K* frequent patterns in all frequent patterns. But those strategies are invalid when users are interested in the Top-*K* frequent patterns with length *t* (1<=*t* <=*m* and *m* is the length of the longest frequent patterns). For example, suppose the length of the longest frequent patterns is 10 and the numbers of frequent patterns for various lengths are 4, 16, 64, 256, 1024, 4096, 13438, 5767, 604, and 1, respectively. Users may be interested in the Top-10 frequent patterns with length 8 and the Top-10 frequent patterns with length 9. But the traditional Top-*K* mining algorithm can only find the Top-10 frequent patterns in all frequent patterns. Therefore, more effective mining algorithms especially in long sequences and mining the Top-*K* frequent patterns for each length are worth exploring.

3. Problem definition and preliminaries

We first give some definitions related with the proposed methods.

Definition 1. $S=s_0 \ldots s_i \ldots s_{n-1}$ is called a subject **sequence**, where $s_i \in \sum$ is a symbol and n is the length of S. $\sum$ can be a different symbol set in different applications and the size of $\sum$ is denoted by $|\sum|$. In a DNA sequence, $\sum$={A,T,C,G } and $|\sum|$=4.

Definition 2. $P=p_0[min_0,max_0]p_1 \ldots [min_{j-1},max_{j-1}]\ p_j \ldots [min_{m-2},max_{m-2}]\ p_{m-1}$ is a **pattern**, where $p_j \in \sum$, m is the length of P, min_{j-1} and max_{j-1} are integer numbers and $0 \le min_{j-1} \le max_{j-1}$. min_{j-1} and max_{j-1} are gap constraints, presenting the minimum and maximum number of wildcards between two consecutive letters p_{j-1} and p_j, respectively. If $min_0 = min_1 = \ldots = min_{m-2} = M$ and $max_0 = max_1 = \ldots = max_{m-2} = N$, pattern P is called a **pattern with periodic wildcard gaps**. For example, A[1,3]C[1,3]C is a pattern with periodic wildcard gaps [1,3] .

Definition 3. If a sequence of indices $D=<d_0, d_1, \ldots, d_{m-1}>$ is subject to $M \le d_j - d_{j-1} - 1 \le N$ ($1 \le j \le m-1$), D is an **offset sequence** of P with periodic wildcard gaps [M, N] in S. The number of offset sequences of P in S is denoted by $ofs(P, S)$.

Definition 4. If an offset sequence $I=<i_0,\ldots ,i_j ,\ldots ,i_{m-1}>$ is subject to $s_{i_j} = p_j$ ($0 \le j \le m-1$ and $0 \le i_j \le n-1$), I is a **support** (or occurrence) of P in S. The number of supports of P in S is denoted by $sup(P, S)$.

Definition 5. $r(P, S) = sup(P, S)/ofs(P,S)$ is called the **support ratio**. If $r(P, S)$ is not less than a pre-specified threshold ρ, pattern P is a **frequent pattern,** otherwise P is an **infrequent pattern**.

Apparently, $sup(P, S)$ is not greater than $ofs(P, S)$ since each offset sequence can be a support. So $0 \le r(P, S) \le 1$. Zhang *et al.* [20] gave a method to calculate $ofs(P,S)$ with gap constraints [M, N]. Suppose m and n are the length of pattern P and the given sequence, respectively. Let $W=N-M+1$, $l_1 = \lfloor (n+M)/(M+1) \rfloor$, and $l_2 = \lfloor (n+N)/(N+1) \rfloor$. $ofs(P,S)$ is calculated as follows.

For $m > l_1$, $ofs(P,S)=0$. (1)

For $m \le l_2$, $ofs(P,S)=(n-(m-1)((M+N)/2+1))W\text{^}(m-1)$ (2)

For $l_2 < m \le l_1$, $ofs(P,S)$ can be calculated by a recursive formula. (3)

Example 1. Suppose patterns $P1$= A[1,3]C and $P2$=G[1, 3]C, a sequence $S=s_0s_1s_2s_3s_4s_5$=AGCCCT and ρ =0.25.

We can easily enumerate all 9 offset sequences of $P1$ and $P2$ in S which are <0,2>,<0,3>,<0,4>, <1,3>, <1,4>, <1,5>, <2,4>, <2,5>, and <3,5>. Hence $ofs(P1, S)=ofs(P2, S)=9$. The

supports of $P1$ in S include <0,2>,<0,3>, and <0,4> and $sup(P1, S)$=3. $P1$ is a frequent pattern since the support ratio $r(P1, S)$=3/9=0.333. We can see that $sup(P2, S)$=2 and the supports are <1, 3> and <1, 4>. So $r(P2, S)$=2/9=0.222 < ρ and $P2$ is an infrequent pattern.

Definition 6. Given patterns P and Q, if Q is a sub-string of P, P and Q are called the **super-pattern** and **sub-pattern,** respectively. If sub-pattern Q contains the first $|P|$-1 characters of P, Q is a **prefix pattern** of P denoted by *Prefix* (P). Similarly, if sub-pattern Q contains the last $|P|$-1 characters of P, Q is a **suffix pattern** of P denoted by *Suffix* (P).

Example 2. Suppose patterns $P3$= A[1, 3]C[1, 3]T, $Q1$=A, $Q2$=C, and $Q3$=A[1, 3]C.

$Q1$, $Q2$, and $Q3$ are sub-patterns of $P3$ and $P3$ is a super-pattern of $Q1$, $Q2$, and $Q3$. The prefix pattern and suffix pattern of $P3$ are $Q3$ and C[1,3]T, respectively.

One of the most important tasks of the mining problem is how to calculate the number of supports of a pattern in order to determine whether the pattern is frequent or not. Zhu and Wu [25] proposed the GCS algorithm to calculate the number of supports of pattern P in sequence S. An illustrative example shows the principle of GCS.

Example 3. Suppose a pattern P=T[0,3]C[0,3]G and a sequence $S=s_0s_1s_2s_3s_4s_5s_6s_7s_8s_9$ = TTCCTCCGCG.

Figure 1 shows the principle of the GCS algorithm which creates a two-dimensional array A. If $p_i <> s_j$ ($0 \le i \le m$-1), then $A(i, j)$=0. If i=0 and $p_i=s_j$, then $A(i, j)$=1, otherwise $A(i, j)=\sum_{k=M}^{k=N} A(i\text{-}1, j\text{-}k\text{-}1)$ ($0<i \le m$-1). The sum of the elements in the last row is the number of supports of the pattern in the sequence. Hence we know that the number of supports of P in S is 0+0+0+0+0+0+5+0+4=9.

	0	1	2	3	4	5	6	7	8	9
	T	T	C	C	T	C	C	G	C	G
T	1	1	0	0	1	0	0	0	0	0
C	0	0	2	2	0	2	1	0	1	0
G	0	0	0	0	0	0	0	5	0	4

Figure 1. Gap constrained pattern search

We know that the array in Figure 1 is a sparse array. So GCS must deal with much useless zero data that makes it ineffective to compute the number of supports of a pattern. We employ an incomplete Nettree data structure to overcome its shortcomings and improve the effectiveness. It is necessary to design a good data structure to speed up the calculation of the number of supports of a pattern, in terms of the mining efficiency.

What is more, a preferable pruning strategy also plays an important role. As we know, the Apriori property does not hold in our mining problem. Example 4 shows that not only the number of supports but also the support ratio of a pattern does not satisfy monotonicity.

Example 4. Suppose a sequence S=TCGGG, a pattern Q=T[0,3]C and its super-pattern P=T[0,3]C[0,3]G.

We know that $sup(Q,S)$=1 and $sup(P,S)$=3. So the number of supports of a pattern can exceed that of its sub-pattern. The offset sequences of Q in S are <0,1>, <0,2>, <0,3>, <0,4>, <1,2>, <1,3>, <1,4>, <2,3>, <2,4>, and <3,4>. The offset sequences of P in S are <0,1,2>, <0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>, <0,3,4>, <1,2,3>, <1,2,4>, <1,3,4>, and <2,3,4>. So both $ofs(Q,S)$ and $ofs(P,S)$ are 10. Hence $r(Q, S)$ is less than $r(P, S)$. Therefore, this example shows that not only is $sup(Q,S)$ less than $sup(P,S)$ but also $ofs(Q,S)$ is less than $ofs(P,S)$.

But luckily, Zhang *et al*. [20] proposed the Apriori-like property to prune candidate patterns effectively and can get all frequent patterns. This property means that if the support ratio of a pattern is less than some value, all its super-patterns are not frequent.

Lemma 1. For any length-m pattern Q and its length-(m+1) super-pattern P, we know that $sup(P)<=sup(Q)*W$, where $W=N-M+1$ and M and N are the minimum and maximum gap, respectively.

Proof: Let $I=<i_0,i_1\ldots, i_j, \ldots, i_{m-1}>$ be a support of the pattern Q. P has at most W supports with prefix-support I in sequence S which are $I_1=<i_0, i_1, \ldots, i_j,\ldots, i_{m-1}, i_{m-1}+M+1>$, $I_2=<i_0,i_1, \ldots, i_j, \ldots, i_{m-1}, i_{m-1}+M+2>,\ldots I_W=<i_0,i_1, \ldots, i_j, \ldots, i_{m-1}, i_{m-1}+N+1>$. So $sup(P)<= sup(Q)* W$. Likewise, when Q is the suffix pattern of P, we can get the same formula as above.

Theorem 1. If the support ratio of a length-m pattern Q is less than $\frac{n-(d-1)*(w+1)}{n-(m-1)*(w+1)}*\rho$, all its super-patterns which contain it can be thought infrequent, where d ($d>m$) is the length of the longest frequent patterns.

Proof: We know that $\frac{\sup(Q,S)}{ofs(Q,S)} < \frac{n-(d-1)*(w+1)}{n-(m-1)*(w+1)}*\rho$. Suppose P is a super-pattern of pattern Q with length k ($m<k<=d$), then we know that $ofs(Q,S)=(n-(m-1)*(w+1))*W^{m-1}$ and $ofs(P)=(n-(k-1) *(w+1))*W^{k-1}$ according to the offset equation. So $ofs(P,S)= \frac{n-(k-1)*(w+1)}{n-(m-1)*(w+1)}*W^{k-m}* ofs(Q,S)$. We know that $sup(P,S)<= sup(Q,S)*W^{k-m}$ according to Lemma 1. Hence $\sup(P,S)/ofs(P,S)<= \frac{\sup(Q,S)}{ofs(Q,S)}*\frac{n-(m-1)*(w+1)}{n-(k-1)*(w+1)}*\rho$ <= $\frac{\sup(Q,S)}{ofs(Q,S)}*\frac{n-(m-1)*(w+1)}{n-(d-1)*(w+1)}*\rho < \rho$. Therefore **Theorem 1** is proved.

Definition 7. All frequent patterns with the Apriori or Apriori-like property can be described by a tree which is called a **frequent pattern tree**. The root of the tree is null. A path from the root to a node in the tree is a frequent pattern.

Example 5. Suppose all frequent patterns of a DNA sequence are {A, T, C, G, AA, AC, AG, TA, TT, TC, CA, CC, CG, GA, GT , GC, GG, AGA, AGT, AGC} which can be expressed as a frequent pattern tree as shown in Figure 2.

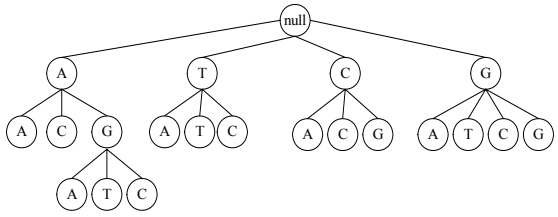


Figure 2. A frequent pattern tree

4. Nettree and the algorithms

4.1. Nettree

Definition 8. A **Nettree** data structure is a collection of nodes. The collection can be empty, otherwise a Nettree consists of some distinguished nodes $r_1,\ldots,r_m$ (root), and 0 or more non-empty subNettrees $T_1,T_2, \ldots, T_n$, each of whose roots can be connected by at least an edge from root r_i, where $1 \le m$, $1 \le n$, and $1 \le i \le n$. A generic Nettree is shown in Figure 3.

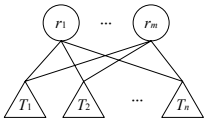


Figure 3. A generic Nettree

A Nettree is an extension of a tree because it has similar concepts of a tree, such as the root, leaf, level, parent, child, and so on. In order to show the differences between tree data structure and Nettree data structure, Property 1 is given as follows.

Property 1. A Nettree [32,33] has the following four properties.

(1) A Nettree may have more than one root.

(2) Some nodes except roots in a Nettree may have more than one parent.

(3) There may be more than one path from a node to a root.

(4) Some nodes may occur more than once.

Definition 9. Given a Nettree, if the same node can occur once or more, we use the node name to denote each of them if each node occurs only once. Otherwise n_j^i denotes node i on the j^{th} level if a node occurs more than once on different levels in the Nettree [32, 33].

Example 6. Figure 4 shows a Nettree. Node 3 occurs 3 times in the Nettree. n_1^3, n_2^3 and n_3^3 denote node 3 on the 1st, 2nd and 3rd levels, respectively. Node n_2^4 has two parents, n_1^1 and n_1^3.

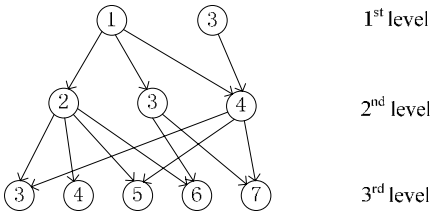


Figure 4. A Nettree

Definition 10. A path from a root to node n_j^i is called a **root path**. $R(n_j^i)$ denotes the Number of Root Paths (NRP) of node n_j^i. The NRPs of a root and a j^{th} level node n_j^i (j>1) are 1 and the sum of the NRPs of its parents, respectively.

A pattern matching problem with gaps can be used to construct a Nettree according to the pattern and the sequence [32, 33]. On one hand, a Nettree can clearly express all supports of the pattern in the sequence [32]. On the other hand, it is easy to calculate the number of supports and find the optimal supports under the one-off conditions [33]. An illustrative example is shown as follows:

Example 7. Suppose pattern P = T[0,3]C and sequence S=$s_0s_1s_2s_3s_4s_5s_6s_7s_8s_9$=TTCCTCCGCG.

Figure 5 shows a Nettree for the pattern matching problem. The numbers in the white and grey circles are the name of a node and its NRP, respectively. A path from a root to a 2nd level node in the Nettree is a support of pattern P in sequence S. For example, path (0,3) which is from node 0 in the first level to node 3 in the second level is a support <0,3> of P in S. So a Nettree can show all supports of P in S effectively. Hence, the number of supports is 2+2+2+1+1=8. This is the solution of literature [32]. However, if each position can be used no more than once, called the one-off condition [29], supports <0,3> and <4,5> are not the optimal supports under the one-off condition. This instance has many optimal solutions and one of the solutions is <0,3>, <1,5> and <4,8>. Literature [33] designed an algorithm to find the optimal supports using Nettree.

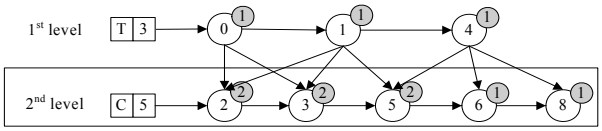


Figure 5. A Nettree for P in S

Therefore, literatures [32] and [33] which deal with two pattern matching problems use Nettree to calculate the number of supports and find the optimal supports under the one-off condition when pattern P and sequence S are given, respectively. Literature [32] can be seen as calculating the searching space of literature [33]. Whereas this study focuses on a pattern mining problem that is to find all frequent patterns when sequence S is given.

4.2. Algorithms

It is very easy to get a path from a node in the first level to a node in the last level in a Nettree if a Nettree has parent-child and child-parent relationships. A path is a support (or occurrence) of pattern P in sequence S for the pattern matching problem. So a Nettree has parent-child and child-parent relationships in [32, 33]. However, we only want to know the number of supports in the pattern mining problem. Hence it is not necessary to store the parent-child and child-parent relationships of the Nettree in this study. The Nettree is usually stored in a node array and each element in the node array stores the node name and its NRP.

Lemma 2. The sum of NRPs of the j^{th} ($1 \le j \le |P|$) level nodes is the number of supports of sub-pattern Q which is the first j characters of pattern P.

Proof: Let Q be the first j characters of pattern P. $R(n_j^i)$ is the number of supports of Q at position i. So the sum of NRPs of the j^{th} ($1 \le j \le |P|$) level nodes is the number of supports of Q in S.

By Lemma 2, we know that the sum of NRPs of the first level nodes, 1+1+1=3, is the number of supports of *Prefix* (P) in S in Example 7. The sum of NRPs of the last level nodes (2+2+2+1+1=8) is the number of supports of pattern P. It is easily seen that the top m-1 level nodes of the Nettree of the length-m pattern P are redundant in calculating the number of supports of its super-pattern. So we only store the last level nodes of the Nettree of a pattern which is called **an incomplete Nettree**.

The reasons for using an incomplete Nettree to solve the problem are as follows. (1) The incomplete Nettree only stores useful information and avoids calculating useless data, which makes the algorithm more effective and consumes less memory. But in [24, 25], a two-dimensional array is used to calculate the number of supports. So the two algorithms in [24, 25] have to deal with useless information which influences the efficiency of mining since the array is sparse. (2) We can use the incomplete Nettree of pattern P to construct a new incomplete Nettree and calculate the number of supports of its super-pattern. This can take full advantage of the previous calculating results.

However, if we can simultaneously calculate the numbers of supports of super-patterns with the same prefix pattern in a one-way scan, the pattern mining algorithm will further

enhance its efficiency. For example, suppose that a pattern P=T[0,3]C is frequent in a DNA sequence and we need to determine whether super-patterns P_1= T[0,3]C[0,3]A, P_2=T[0,3]C[0,3]T, P_3=T[0,3]C[0,3]C, and P_4=T[0,3]C[0,3]G are frequent or not. It will be less effective to calculate the numbers of supports of these four super-patterns in a four-way scan than in a one-way scan. The principle of calculating the numbers of supports of all super-patterns $P[M, N]a$ ($a \in \Sigma$) in a one-way scan is as follows.

According to gap constraints [M, N], we create all child nodes of node q which is a node in an incomplete Nettree of P. If $s_j = \Sigma_k$ ($q+M+1 \le j \le q+N+1$, $0 \le k < |\Sigma|$), we check whether node j is a node in the incomplete Nettree of super-pattern P_k or not. If it is not in an incomplete Nettree of P_k, we create node j and store it in the incomplete Nettree and the NRP of node j is the NRP of node q, otherwise we only update the NRP of node j, making it plus the NRP of node q. So the number of supports of super-pattern P_k is the sum of the NRPs of all nodes of the incomplete Nettree of P_k. Hence, we can calculate the numbers of supports of these super-patterns in a one-way scan. The algorithm is named INSupport.

Here we employ a structure named IINettree (Information of the Incomplete Nettree) to describe the mined pattern string, the number of its supports and its incomplete Nettree in INSupport. For the sake of simplicity, we use the terms *pattern, sup* and *INtree* for short. So IINettree can be expressed as {*pattern*, *sup*, *INtree*}. Due to that the representation of the incomplete Nettree is relatively straightforward, an incomplete Nettree is composed of a size and an array which contains names of all nodes and their corresponding NRPs. Hence, IINettree can again be written as {*pattern*, *sup*, {*size*, ($name_0$, NRP_0), …, ($name_{size-1}$, NRP_{size-1})}}. The inputs of INSupport are P, S and INtree which is an incomplete Nettree of P. The output of INSupport is *superps* which is an array of IINettree. The size of *superps* is $|\Sigma|$. For example, given a sequence S= TTCCTCCGCG and a prefix pattern P=T[0,3]C (shown in Example 8). $superps_2$ can be expressed as {T[0,3]C[0,3]C, 15, {4, (3,2),(5,4),(6,6),(8,3)}} since "C" is the third letter in $\sum$ in a DNA sequence. Algorithm INSupport is shown as follows.

Algorithm 1: INSupport (*P*,*S*, *INtree*);

Input: *P*, *S*, *INtree*

Output: *superps*

```
1: superps.sup=0;
2: superps.pattern= P[M, N]Σ;
3:for (i=0; i<|INtree|; i++)
4:      oldnode= INtree i;
5:       for  (j=oldNode.name+M+1;  j ≤ oldNode.name  +N+1;
j++)
6:            if (sj==Σk) then
7:                superps k.sup +=oldnode.NRP;
8:                position=search(superps k.INtree, j);
9:           //The result will be -1 if j is not in superps k.INtree.
10:               if (position==-1) then
11:                   newnode.name= j;
12:                   newnode.NRP=oldNode.NRP;
13:                   superpsk.INtree size++ =newNode;
14:               else
15:                       superpsk.INtree position.NRP+= oldNode.
NRP;
16:               end if
17:           end if
18:      end for
19:end for
20: return superps;
```

The principle of MAPB is given as follows. Firstly, each character in Σ is seen as a length-1 pattern P. We create $|\Sigma|$ incomplete Nettrees for each pattern and calculate the numbers of their supports. Then we need to determine whether the support ratio of each pattern is not less than $\beta = \rho *(n-(d-1)*(w+1))/(n-(m-1)*(w+1))$ or not, where $w=(M+N)/2$, d and m are the length of the longest frequent patterns and the length of pattern P, respectively. If it is not less than the value, pattern P and its incomplete Nettree will be stored in a queue. After this, prefix pattern P and its incomplete Nettree is dequeued and we need to check whether pattern P is a frequent pattern or not. Algorithm 1 is used to calculate the number of supports of all super-patterns with pattern P and create $|\Sigma|$ incomplete Nettrees of the super-patterns. Finally, we check whether each super-pattern is not less than β or not. If yes, we store it and its incomplete Nettree in the queue and then iterate this process till the queue is empty. Apparently, this method is a kind of Apriori-like property to prune the number of candidate patterns and constructs the frequent pattern tree based on BFS. The algorithm named MAPB is given as follows.

Algorithm 2. MAPB

Input: $S=s_0s_1\ldots s_{n-1}$, M, N, ρ, and d, where d is the length of the longest frequent patterns

Output: All patterns with frequency not less than ρ

```
1:patterns.pattern=Σ;
2:for (i=0; i<|S|; i++)
3:      if (si==Σj) then
4:           node.name=i;
5:           node.NRP=1;
6:           patternsj.INtree size++ =node;
7:      end if
8:end for
9:for (j=0; j<|Σ|; j++)
10:         patternsj.sup= patternsj.INtree.size;
11:        if  (patternsj.sup/|S|>= ρ *(n-(d-1)*(w+1))/n))  then
meta. enqueue(patternsj);
12:end for
13:while (!meta.empty())
14:      subP =meta.dequeue ();
15:      P=subP.pattern;
16:      INtree= subP.INtree;
17:      length=|P|;
18:      calculate r(P,S);
19:      if (r(P,S)>= ρ) then C length= C length U P;
20:      superps = INSupport(P,S, INtree);
21:      for (j=0; j<|Σ|; j++)
22:           Q= superps j. pattern;
23:           calculate r (Q,S);
24:           if(length+1<=d) then
25:                    if  (r(Q,S)  >= ρ *(n-(d-1)*(w+1))/
(n-length* (w+1))) then meta.enqueue(superps j);
26:           else
27:                              if  (r(Q,S)  >= ρ )  then
meta.enqueue(superps j);
28:           end if
29:      end for
30:end while
31:return C=U Ci
```

Although MAPB employs a pattern matching strategy (INSupport) to calculate the numbers of supports of candidate patterns, INSupport and the algorithm in [32] are significantly different. The reasons are given as follows. Firstly, the algorithm in [32] is used to calculate the number of supports for

only one pattern, while INSupport can simultaneously calculate the numbers of supports for many patterns with a common prefix pattern. Secondly, the algorithm in [32] does not need any previous result to solve the problem, while INSupport calculates the numbers of supports depending on previous results.

MAPB stores the frequent patterns in a queue. A new algorithm named MAPD stores the frequent patterns in a stack and the frequent pattern tree is constructed based on DFS. We can know that lines 11, 14, 25, and 27 of MAPD are then modified as:

11: if ($patterns_j.sup/|S| >= \rho * (n-(d-1)*(w+1))/n$) then $meta.Push(patterns_j)$;

14: $subPattern=meta.Pop()$;

25: if ($r(Q,S) >= \rho * (n-(d-1)*(w+1))/(n-length*(w+1))$) then $meta.Push(superps_j)$;

27: if ($r(Q,S) >= \rho$) then $meta.Push(superps_j)$;

4.3. A running example

An illustrative example is used to show how Algorithm 1 works.

Example 8. Suppose sequence $S=s_0s_1s_2s_3s_4s_5s_6s_7s_8s_9$= TTCCTCCGCG, patterns P_0= T[0,3]C[0,3]A, P_1= T[0,3]C[0,3]T, P_2=T[0,3]C[0,3]C, P_3=T[0, 3]C[0,3]G and the incomplete Nettree of P=T[0,3]C which is shown in the black frame in Figure 5.

R(node) is used to express the NRP of a node according to Definition 10. We create child nodes of n_2^2 from s_3 to s_6 since gap constraints are [0, 3]. Because s_3=C, P_2=T[0,3]C[0,3]C and node n_3^3 is not in the incomplete Nettree of P_2, we create node n_3^3 in the incomplete Nettree of P_2 and $R(n_3^3)=R(n_2^2)=2$. We create node n_3^4 in the incomplete Nettree of P_1 and $R(n_3^4)=R(n_2^2)=2$. Similarly, we create nodes n_3^5 and n_3^6 in the incomplete Nettree of P_2, $R(n_3^5)= R(n_3^6)= R(n_2^2)=2$. Then we create child nodes of n_2^3 from s_4 to s_7. Because s_4=T and node n_3^4 is in the incomplete Nettree of P_1, we update the value of $R(n_3^4)= R(n_3^4)+ R(n_2^3)=4$. Similarly, we can know that $R(n_3^5)= R(n_3^5)+ R(n_2^3)=4$, $R(n_3^6)= R(n_3^6)+ R(n_2^3)=4$, and $R(n_3^7) = R(n_2^3)=2$. Finally, we create all child nodes of node n_2^5, n_2^6, and n_2^8. Figure 6 shows incomplete Nettrees of P, P_0, P_1, P_2, and P_3. It is straightforward to know that the numbers of supports of P_0, P_1, P_2, and P_3 in S are 0, 4, 2+4+6+3=15, and 5+4=9, respectively.

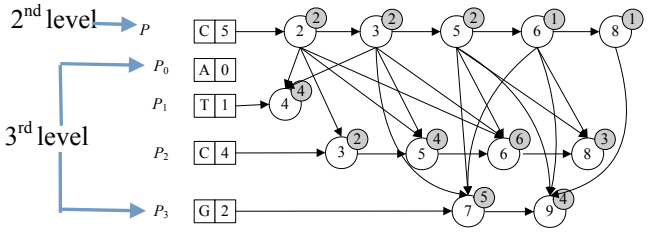


Figure 6. Incomplete Nettrees of P, P_0, P_1, P_2, and P_3

4.4. Correctness and completeness

The main difference between MAPB and MAPD is using different strategies to create the frequent pattern tree. So we only prove the correctness and completeness of MAPB. The same proofs apply to the correctness and completeness of MAPD.

Theorem 2. (Correctness of MAPB) The output of MAPB is the frequent patterns.

Proof: We know that Algorithm 1 is correct according to Lemma 2. So MAPB can calculate the numbers of supports of super-patterns with prefix P correctly. Zhang *et al*. [20] gave the method to calculate $ofs(P,S)$ and proved the correctness of the method. Therefore Theorem 2 is proved.

Theorem 3. (Completeness of MAPB) MAPB can find all frequent patterns whose lengths are less than l_2.

Proof: We know that d is less than or equal to l_2 and if the support ratio of pattern P is less than $\frac{n-(d-1)*(w+1)}{n-(m-1)*(w+1)}*\rho$, all its super-patterns which contain it can be thought infrequent according to Theorem 1, where w= $(M+N)/2$ and l_2 = $\lfloor (n+N)/(N+1) \rfloor$ and n, m, d, M, and N are the length of sequence, the length of pattern, the length of the longest frequent patterns, the minimum gap, and the maximum gap, respectively. So MAPB enqueues pattern P if its support ratio is not less than $\frac{n-(d-1)*(w+1)}{n-(m-1)*(w+1)}*\rho$ and checks whether its super-patterns are frequent or not. Therefore all frequent patterns can be found by MAPB. Hence Theorem 3 is proved. Generally d is far less than l_2 in our mining problem, so the algorithms can usually find all frequent patterns in a sequence.

4.5. Complexities analysis

Both MAPB and MAPD can create the same frequent pattern tree. So the time complexities of these two algorithms are the same. Algorithm 1 creates the child nodes of the incomplete Nettree of a sub-pattern. It is easy to see that the average size of the incomplete Nettrees is $n/|\Sigma|$, where n is the length of the sequence. Each node has $W=N-M+1$ children. So the time complexity of Algorithm 1 is $O(W*n/|\Sigma|)$. Algorithm 1 runs $\sum_{j=1}^{d} len_j$ times, where len_j and d are the number of length-j frequent patterns and the length of the longest frequent pattern, respectively. Hence the time complexities of MAPB and MAPD are both $O(\sum_{j=1}^{d} len_j * W*n/|\Sigma|)$. However, the mining algorithm using GCS (MGCS) creates j rows for pattern P and $|\Sigma|$ rows for all super-patterns with prefix P to calculate the numbers of supports of these patterns. Each row has n elements and each element is calculated W times. So the time complexity of GCS is $O((j+|\Sigma|)*n*W)$. Thus, the time complexity of MGCS is $O(\sum_{j=1}^{d} len_j * (j+|\Sigma|)*n*W)$.

The average size of an incomplete Nettree is $n/|\Sigma|$ and each node stores its name and NRP. So the space complexity of an incomplete Nettree is $O(n/|\Sigma|)$. MAPB uses a queue and the max size of the queue is $O(|\Sigma|\hat{}x)$, where x is the position of max len_j. So the space complexity of MAPB is $O(|\Sigma|\hat{}(x-1)*n)$. MAPD uses a stack and the max size of the stack is $O(d*|\Sigma|)$. So the space complexity of MAPD is $O(d*n)$. We can see that the space complexity of MGCS is $O((d+|\Sigma|)*n)$. Table 2 gives a comparison of the time and space complexities among MGCS, MAPB, and MAPD.

Table 2. Comparison of the time and space complexities

Algorithm	Time complexity	Space complexity
MGCS	$O(\sum_{j=1}^{d} len_j * (j+\|\Sigma\|)*n*W)$	$O((d+\|\Sigma\|)*n)$
MAPB	$O(\sum_{j=1}^{d} len_j * W*n/\|\Sigma\|)$	$O(\|\Sigma\|\hat{}(x-1)*n)$
MAPD	$O(\sum_{j=1}^{d} len_j * W*n/\|\Sigma\|)$	$O(d*n)$

5. Top-*K* mining

Both MAPB and MAPD are mining algorithms based on the pattern matching approach to discover all possible frequent patterns. This kind of pattern mining approach can also be employed to find special frequent patterns effectively. For example, when we discover thousands of frequent patterns or more, it is difficult to use all these patterns. The most valuable patterns are Top-*K* frequent patterns with various lengths and the longest frequent patterns, where *K* is a specified parameter, and this is called the Top-*K* mining problem.

Like Apriori, algorithm MPP-best [20] acts iteratively, generating length-(m+1) candidate patterns using length-m frequent patterns and verifying their supports, till there is no candidate pattern. So it is difficult to employ this kind of approach to construct an algorithm to solve this problem.

An easy method is to propose an algorithm which finds all frequent patterns and sorts these patterns and then outputs the Top-*K* frequent patterns to satisfy users' needs. Another similar method is to introduce an algorithm that checks all possible candidate patterns and selects the Top-*K* frequent patterns. But these methods take a long time to solve the Top-*K* mining problem in long sequences using MAPD because many useless frequent patterns are found or useless possible candidate patterns are checked. To discover these patterns effectively, a heuristic algorithm named MAPBOK is proposed. The principle of MAPBOK is as follows.

If sub-pattern P is a Top-*K* length-m frequent pattern, its super-pattern Q is a Top-*K* length-(m+1) frequent pattern in high probability. Firstly, we mine the Top-(e**K*) length-m frequent patterns and output the Top-*K* frequent patterns, where e is not less than 1. Then the Top-(e**K*) frequent patterns are used to mine the length-(m+1) frequent super-patterns. We select the Top-(e**K*) frequent super-patterns and iterate this process till no new frequent patterns can be found. Apparently, this method constructs the frequent pattern tree based on BFS. The algorithm of MAPBOK is not given, since the change is straightforward from MAPB.

It is easy to see that the time complexity of MAPBOK is $O(e*K*d*W*n/|\Sigma|)$ since MAPBOK runs Algorithm 1 $O(e*K*d)$ times. The space complexity of MAPBOK is $O(e*K* n/|\Sigma|)$ since the max size of the queue is $O(e*K)$.

6. Performance evaluation

From Table 1 in section 2, we know that this study focuses on the same pattern mining problem as literatures [20] and [25]. Besides, the most related issue is literature [24] in which the problem is redefined and the Apriori property is used. Here we call the algorithm in [24] AMIN. Therefore we present experimental results by comparing MPP-best, MGCS, AMIN, MAPB, and MAPD. In order to show that our algorithms are superior to the state-of-the-art algorithms, we evaluate their performance based on the running time and memory requirements. For the Top-*K* mining problem for each length, we pay more attention to the mining accuracy of longer frequent patterns. Weighted accuracy can be calculated according to the following equation.

$$\text{Accuracy} = (\sum_{i=3}^{d} i * a_i)/(\sum_{i=3}^{d} i * b_i) \quad (4)$$

where a_i, b_i, and d are the numbers of correct Top-*K* frequent patterns, Top-*K* length-i frequent patterns, and the length of the longest frequent patterns, respectively. Generally, b_i is K. But when c is less than K, b_i becomes c, where c is the number of length-i frequent patterns.

6.1. Experimental environment and data

The data used in this paper are DNA sequences provided by the National Center for Biotechnology Information website. Homo Sapiens AX829174, AL158070 and AB038490 are chosen as our test data and can be downloaded from http://www.ncbi.nlm.nih.gov/nuccore/AX829174, http://www.ncbi.nlm.nih.gov/nuccore/AL158070.11 and http://www.ncbi.nlm.nih.gov/nuccore/ AB038490, respectively. The source codes of MPP-best, MGCS, MAPB, MAPD, and MAPBOK can be obtained from http://wuc.scse.hebut.edu.cn/msppwg/index.html. All experiments were run on a laptop with Pentium(R) Dual-Core T4500@ 2.30GHz CPU and 2.0 GB of RAM, Windows 7. Java Development Kit (JDK) 1.6.0 is used to develop all algorithms. In this study, the greatest length of frequent patterns is considered to be 13, the minimum and maximum gap constraints are 9 and 12, respectively and the threshold ρ is 3*10^-5 because all these parameters were used in [20, 24, 25]. What is more, considering that the default stack memory of JVM is too small, we assign 1.5GB memory space for every algorithm which is the maximal memory space of the Java virtual machine on the laptop. Table 3 shows all the sequences used in this paper.

Table 3. Bio-data sequences

Sequence	From	Length
*S*1	Homo Sapiens AX829174	1000
*S*2	Homo Sapiens AX829174	2000
*S*3	Homo Sapiens AX829174	4000
*S*4	Homo Sapiens AX829174	8000
*S*5	Homo Sapiens AX829174	10011
S6	Homo Sapiens AL158070	20000
*S*7	Homo Sapiens AL158070	40000
*S*8	Homo Sapiens AL158070	80000
*S*9	Homo Sapiens AL158070	167005
*S*10	Homo Sapiens AB038490	15000
*S*11	Homo Sapiens AB038490	30000
*S*12	Homo Sapiens AB038490	60000
*S*13	Homo Sapiens AB038490	131892

With the above presented test environment and data, Table 4 and Table 5 show the mining results and the comparison of max size of MAPB and MAPD, respectively.

Table 4. Mining results

Sequence	The length of the longest frequent patterns, the number of frequent patterns for various lengths and total frequent patterns
*S*1	13, {4,16,64,256,1024,4096,13374,5678,1514,623, 242, 55, 12}, 26958
*S*2	12, {4,16,64,256,1024,4096,15205,3436,350,85,8,3}, 24547
*S*3	10, {4,16,64,256,1024,4096,15965,1937,59,3}, 23424
*S*4	10, {4,16,64,256,1024,4096,14970,4283,241,1}, 24955
*S*5	10, {4,16,64,256,1024,4096,14422,4811,299,1}, 24993
*S*6	10, {4,16,64,256,1024,4096,12619,7068,614,8}, 25769
*S*7	10, {4,16,64,256,1024,4096,12388,6960,749,11}, 25568
*S*8	10, {4,16,64,256,1024,4096,12947,6303,666,11},25387
*S*9	10, {4,16,64,256,1024,4096,13438,5767,604,1}, 25270
*S*10	10, {4,16,64,256,1024,4096,12507,7197,563,2}, 25729
*S*11	11, {4,16,64,256,1024,4096,11126,7634,1164,54,1}, 25439
*S*12	10, {4,16,64,256,1024,4096,12799,6404,699,11}, 25373
*S*13	10, {4,16,64,256,1024,4096,12913,6558,672,11},25614

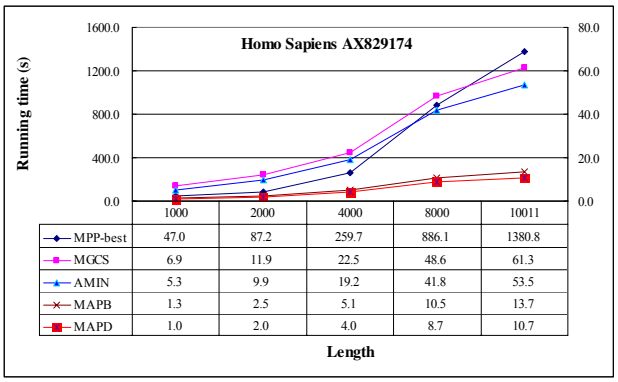
Table 5. Comparison of max size

Sequence	Max size of MAPB	Max size of MAPD
*S*1	14092	24
*S*2	15402	25
*S*3	16017	25
*S*4	15119	26
*S*5	14582	26
*S*6	12878	27
*S*7	12754	26
*S*8	/	27
*S*9	/	26
*S*10	12768	28
*S*11	11666	26
*S*12	12949	28
*S*13	/	28

Note: "Empty" means overflow error.

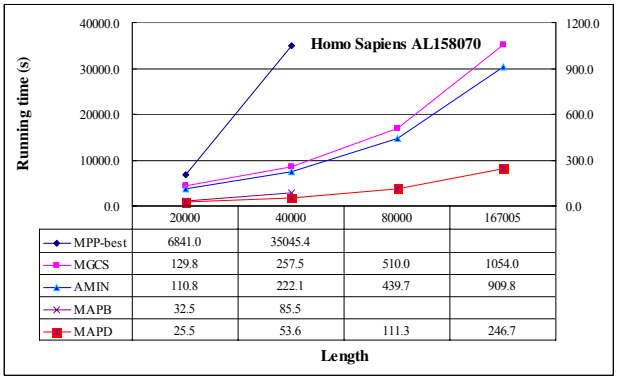
6.2. Running time evaluation

Here we compare the running time on several real DNA fragments of different lengths (shown in Figures 7~9). It is worth noting that the running time of MPP-best refers to the time data on the left and the other algorithms refer to the right because the MPP-best needs to take much time to finish the mining task. We can analyze the running results from the following aspects:



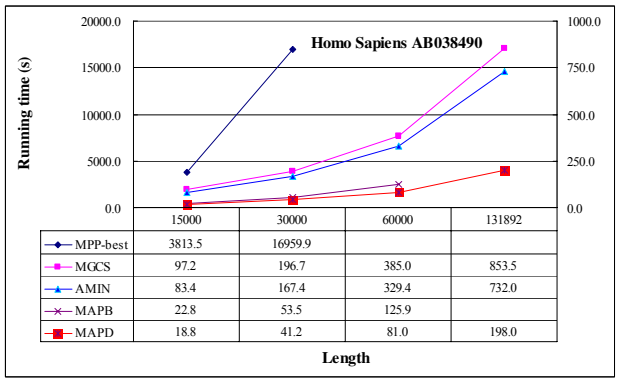
	1000	2000	4000	8000	10011
MPP-best	47.0	87.2	259.7	886.1	1380.8
MGCS	6.9	11.9	22.5	48.6	61.3
AMIN	5.3	9.9	19.2	41.8	53.5
MAPB	1.3	2.5	5.1	10.5	13.7
MAPD	1.0	2.0	4.0	8.7	10.7

Figure 7. Comparison of the running time on Homo Sapiens AX829174



	20000	40000	80000	167005
MPP-best	6841.0	35045.4		
MGCS	129.8	257.5	510.0	1054.0
AMIN	110.8	222.1	439.7	909.8
MAPB	32.5	85.5		
MAPD	25.5	53.6	111.3	246.7

Figure 8. Comparison of the running time on Sapiens AL158070



	15000	30000	60000	131892
MPP-best	3813.5	16959.9		
MGCS	97.2	196.7	385.0	853.5
AMIN	83.4	167.4	329.4	732.0
MAPB	22.8	53.5	125.9	
MAPD	18.8	41.2	81.0	198.0

Figure 9. Comparison of the running time on Homo Sapiens AB038490

Note: "Empty" means overflow error.

(1) From Figures 7~9, we can see that MPP-best not only is the slowest, but also cannot be used to find frequent patterns in long sequences. Only MPP-best uses the left axis to show its running time and other algorithms use the right axis to show theirs running time. The value of the left axis is far greater than that of the right axis. So we can easily notice that MPP-best is the slowest. For example, MPP-best costs nearly 10 hours to find the frequent patterns in sequence with length 40000 (shown in Figure 8), while MAPD costs no more than 1 minute. Furthermore, we observe that the running time of MPP-best grows faster than the length of a sequence. For example, the data table in Figure 7 shows that the running time of MPP-best grows about 29 times when the length of a sequence grows about 10 times from *S*1 to *S*5. Moreover, according to the running time in the tables of Figures 7~9, we can see that the running time of MGCS, AMIN MAPB, and MAPD is in linear growth with the length of a sequence. For example, when the length of a sequence grows about 10 times from *S*1 to *S*5, theirs running time grows about 10 times. However, if we mine frequent patterns in long sequences using MPP-best, we will also face the risk of out of memory when the length of a sequence is longer than 60000. The reason is that MPP-best employs a PIL (Partial Index List) structure which consumes lots of memory to calculate the number of supports.

(2) It is clear that MGCS and AMIN run faster than MPP-best according to Figures 7~9. In [25], MCPaS is about 40 times faster than MPP-best in *S*1 but MGCS is about 6.84 times faster than MPP-best in our experiment. The reason is that MCPaS employs not only GCS but also an effective pruning strategy. If both MPP-best and MCPaS employ the same pruning strategy, MCPaS cannot be about 40 times faster than MPP-best. AMIN is a little faster than MGCS according to the figures. The reason is that AMIN redefines the issue and can use Apriori property which is a more effective pruning strategy than Apriori-like property. AMIN is a kind of approximate algorithm and [24] detailed the difference between the approximate solutions and the accurate solutions. Moreover, both MGCS and AMIN can be used to mine in long sequences. The reason is that MGCS and AMIN use pattern matching approaches to calculate the number of supports for each candidate pattern. And this kind of approach consumes less memory than MPP-best.

(3) Both MAPB and MAPD run faster than MPP-best and MGCS although they employ the same pruning strategy. MAPD runs about 45 and 128 times faster than MPP-best in *S*1 and *S*5, respectively. And MAPD runs about 5 times faster than MGCS in all sequences. So does MAPB. Hence these two

algorithms are better than their peers because they avoid an amount of duplicated calculations.

(4) The running time of MAPB and MAPD should be consistent but actually MAPD runs faster than MAPB because MAPB consumes too much memory, which can influence the mining efficiency. The main reason is that the size of the maximum width of the frequent pattern tree is far greater than its maximum depth according to Table 5. So the max size of the queue is far greater than that of the stack. Therefore MAPB consumes much more memory than MAPD to solve the problem. Hence, MAPD is faster than MAPB and more suitable for the long-sequence mining problem.

(5) We can observe that MAPD can be used to mine frequent pattern in long sequences, while MAPB can not. For example, when the length of a sequence reaches 80000, MAPD can smoothly finish the mining task while MAPB leads to memory-overflow error. Hence it can be concluded that MAPB consumes more memory space than MAPD and it is more desirable to employ DFS to construct the frequent pattern tree.

(6) From Table 4, we can calculate $\sum_{j=1}^{d} len_j *(j+|\Sigma|)$ and $\sum_{j=1}^{d} len_j$, where d and Len_j are the maximum length of frequent patterns and the number of the length-j frequent patterns, respectively. For example, $\sum_{j=1}^{d} len_j * (j+|\Sigma|)$ and $\sum_{j=1}^{d} len_j$ in $S1$ are 274198 and 26958, respectively. The time complexities of MGCS and MAPD are $O(\sum_{j=1}^{d} len_j *(j+|\Sigma|)* W*n)$ and $O(\sum_{j=1}^{d} len_j * W*n/|\Sigma|)$, respectively. The running times of MGCS and MAPD in $S1$ are 6864ms and 1031ms, respectively, combined with the above analysis, MAPD should be about 274198/26958 *4≈40 times faster than MGCS. But MAPD is actually about 6864/1031≈6.65 times faster than MGCS. This is due to the fact that MGCS is easier to realize. Thus it also verifies the correctness of the time complexity analysis of MGCS and MAPD.

(7) From Tables 4 and 5, we can observe that the max size of the queue is slightly greater than the max number of frequent patterns for various lengths, because MAPB employs BFS to construct the frequent pattern tree. The max size of the stack is less than $d*|\Sigma|$, because MAPD employs DFS to construct the frequent pattern tree, where d means the max length of frequent patterns. For example, the number of length-7 frequent patterns is 15205 and the max size of queue is 15402 in $S2$ and the max size of stack is 24 which is less than 13*4=52 in $S1$.

Both MGCS and AMIN which consume less memory can be used to find frequent patterns in long sequences because the two algorithms calculate the number of supports for each pattern without using the previous results. MAPB can not be used to discover frequent patterns in long sequences, but it can avoid an amount of duplicated calculations using the previous results. MAPD not only can avoid an amount of duplicated calculations, but also consumes less memory. So MAPD is the fastest algorithm and can be used in long sequences. Therefore MAPD is superior to the state-of-the-art algorithms. Based on MAPB we propose a heuristic algorithm named MAPBOK. We will show the performance of MAPBOK in the next subsection.

6.3. Top-K mining evaluation

In order to illustrate how e, K and the length of a sequence affect the running time of MAPBOK, the results of various sequences mined with different e, K and the lengths of the sequences are shown in Figures 10~18.

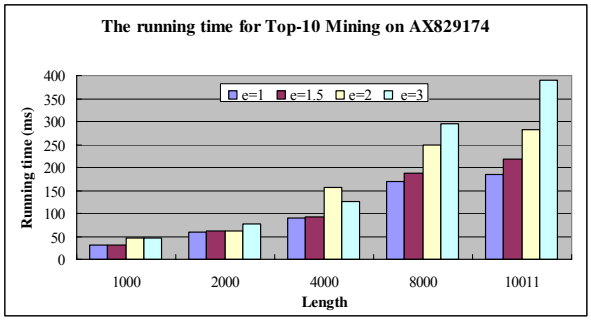


Figure 10. The running time for Top-10 Mining on AX829174

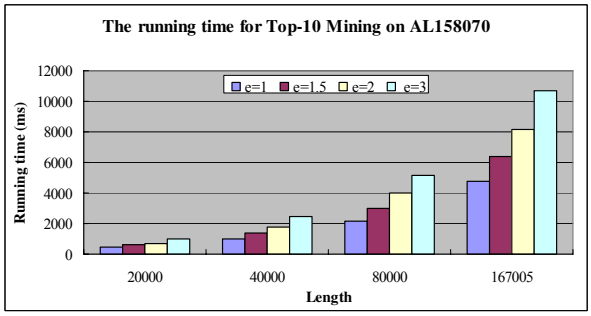


Figure 11. The running time for Top-10 Mining on AL158070

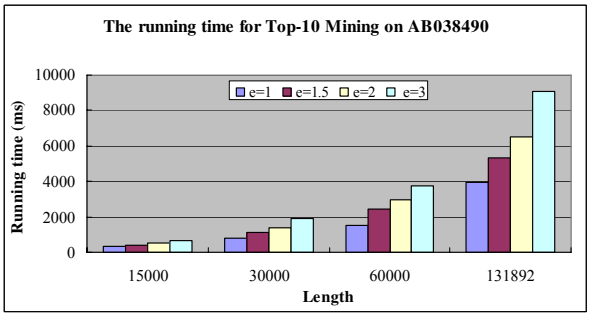


Figure 12. The running time for Top-10 Mining on AB038490

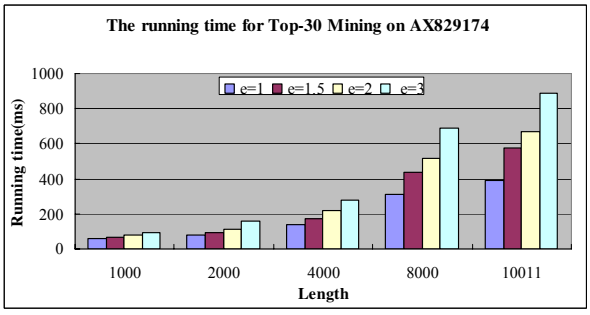


Figure 13. The running time for Top-30 Mining on AX829174

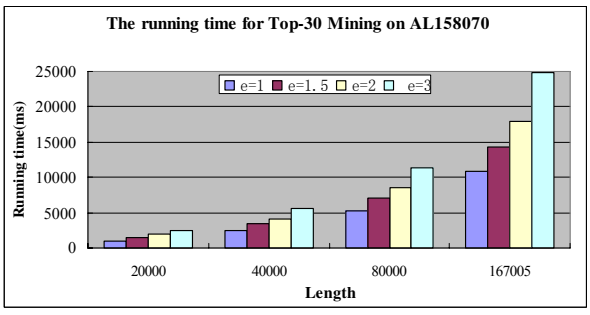


Figure 14. The running time for Top-30 Mining on AL158070

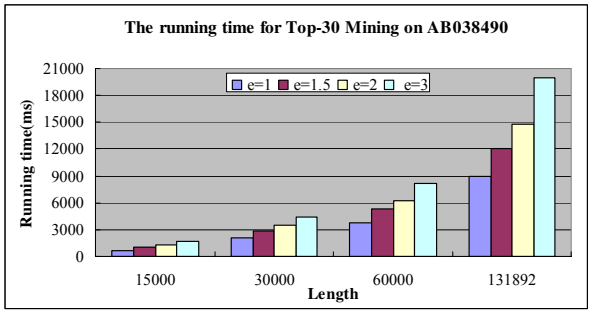


Figure 15. The running time for Top-30 Mining on AB038490

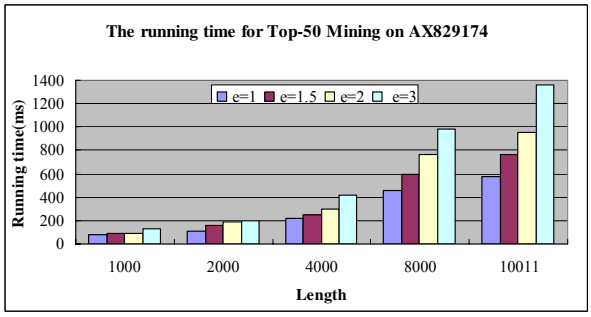


Figure 16. The running time for Top-50 Mining on AX829174

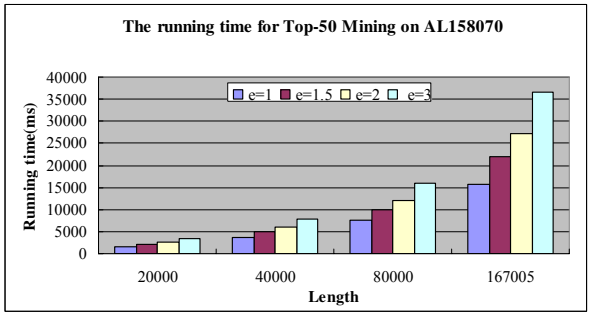


Figure 17. The running time for Top-50 Mining on AL158070

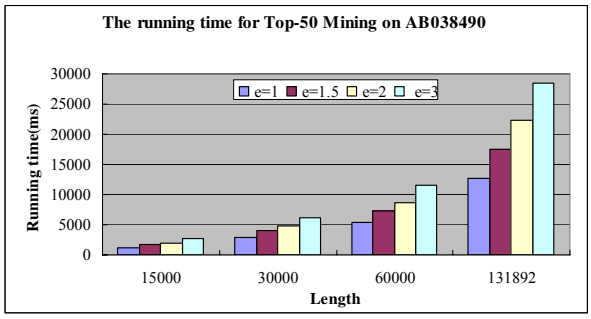


Figure 18. The running time for Top-50 Mining on AB038490

From the above figures, we can clearly observe that the longer the sequence is, the greater e and K are, and the longer the running time is.

(1) It is found that the running time approximately grows in a linear way with the length of a sequence. For example, the running time grows about 10 times while the length of a sequence grows from 1000 to 10011 no matter what e is in Figure 10.

(2) Obviously, the greater e is, the running time is longer for every sequence. Taking the length-131892 sequence on AB038490 in Figure 15 for example, the running times are about 9000, 12000, 15000, and 20000 ms when e are 1, 1.5, 2, and 3, respectively. However, we find that when e increases from 1 to 3, the running time is less than three times. This is no surprise because the number of candidate patterns is invariable when the length of frequent patterns is shorter or longer. For example, the number of length-2 candidate patterns is 16 no matter what e and K are. So the running time grows less than three times when e changes from 1 to 3.

(3) Of course K also has an important effect on the running time. The running time will be longer as K is greater because the number of candidate patterns is also increasing. We can clearly see that with regard to the same sequence and e in Figures 10, 13 and 16, the running time for the Top-10 mining in Figure10 is the shortest, the Top-30 mining in Figure13 takes second place and the Top-50 mining in Figure 16 needs the longest time. The other figures also show the same phenomenon. Likewise, we can observe that the running time is less than 5 times when K increases from 10 to 50. The reason is extremely similar to e in (2) and we will not elaborate it again here.

Therefore, these phenomena are consistent with the time complexity of MAPBOK $O(e*K*d*W*n/|\Sigma|)$ mentioned above and validate the correctness of the analysis of time complexity of MAPBOK.

Figures 19~27 show how e, K and the length of a sequence affect the accuracy of MAPBOK.

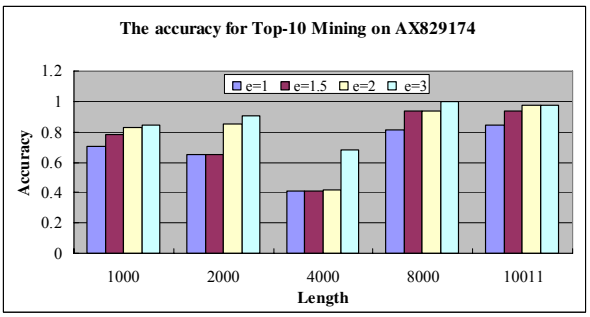


Figure 19. The accuracy for Top-10 Mining on AX829174

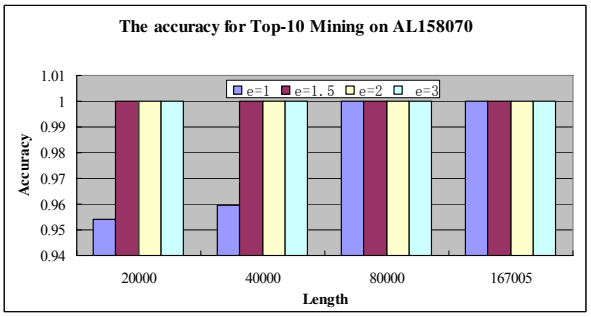


Figure 20. The accuracy for Top-10 Mining on AL158070

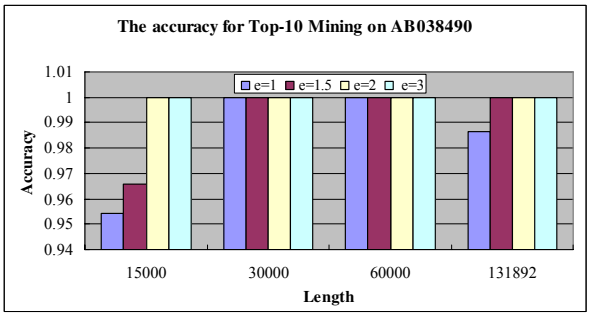


Figure 21. The accuracy for Top-10 Mining on AB038490

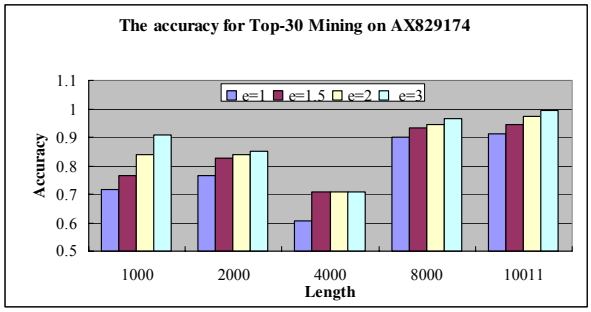


Figure 22. The accuracy for Top-30 Mining on AX829174

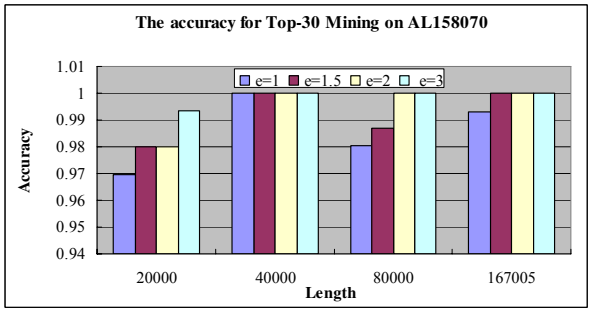


Figure 23. The accuracy for Top-30 Mining on AL158070

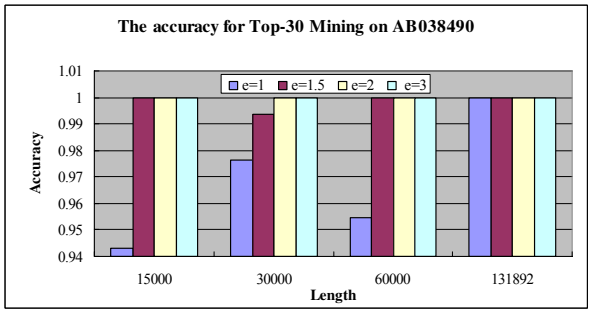


Figure 24. The accuracy for Top-30 Mining on AB038490

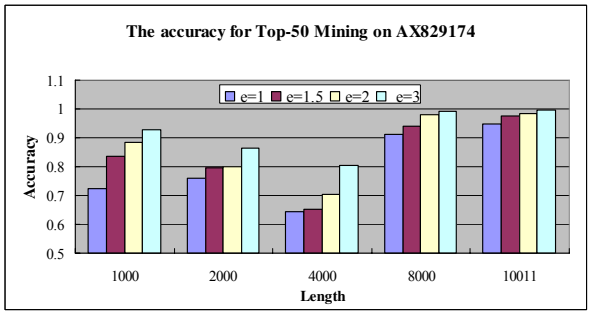


Figure 25. The accuracy for Top-50 Mining on AX829174

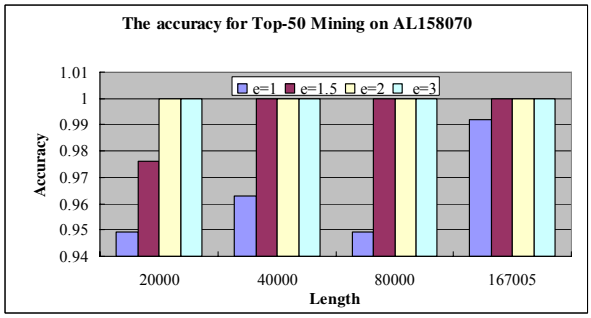


Figure 26. The accuracy for Top-50 Mining on AL158070

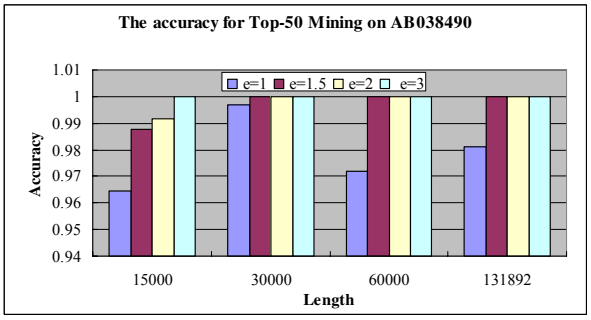


Figure 27. The accuracy for Top-50 Mining on AB038490

(4) According to Figures 19, 22 and 25, we find that the accuracy of MAPBOK is low in short sequences no matter what e and K are. The accuracy can sometimes be less than 0.5 when both e and K are very small. But meanwhile MAPD can get the Top-K patterns precisely in a short time. For example, it takes about 10 seconds to solve the Top-K problem in $S5$ with length 10011. For longer sequences, we can observe that the lowest accuracy for MAPBOK is more than 0.94 and, compared with MAPD, the running time is much less. When e and K are larger, the performance is more prominent. For example, the average accuracy from $S6$ to $S13$ of MAPBOK is 0.995 and the average running time is about 23 times faster than that of MAPD in the case of K=30 and e=1.5. So in terms of the Top-K mining problem it can be concluded that MAPBOK is more suitable for long sequences and MAPD suitable for shorter sequences. The main reason for this situation is that frequent patterns may change easily when sequences are short and it is not easy to change frequent patterns when sequences are long.

(5) From the accuracy results of every sequence, it can be clearly observed that e is an important factor to improve accuracy. Accuracy will be higher and can even reach 100% when e is bigger. We use average accuracy to reflect the effect. Taking the Top-30 mining from $S6$ to $S13$, for example, when e=1 the average accuracy of MAPBOK is 0.977 and when e=2 the average accuracy reaches 0.997. This can self-evidently be attributed to more generated candidate patterns. Although with e being greater the running time is also growing (here the average running time grows about 0.67 times) because more patterns need to be checked, it is still much less than that of MAPD. Hence, it is preferable to suitably increase e in order to improve accuracy. According to Figures 20, 21, 23, 24, 26, and 27, when the length of a sequence is equal to or greater than 15000, e=1.5 or e=2 can achieve a satisfying accuracy.

(6) As we know, the larger e and K are, the number of candidate patterns is bigger. So K has the same effect on accuracy as e. Accuracy will be higher when K is bigger according to Figures 20, 21, 23, 24, 26, and 27.

7. Conclusions

In this paper, we propose two new effective pattern mining algorithms, MAPB and MAPD, to find frequent patterns with periodic wildcard gaps. A pattern matching approach is used to calculate the number of supports of a pattern in the given sequence and determine whether the pattern is frequent or not. MAPB and MAPD store frequent patterns and their incomplete Nettrees in a queue and a stack, respectively. Experimental results validate that both MAPB and MAPD are superior to the state-of-the-art algorithms and MAPD achieves better performance than MAPB. So MAPD can be used to discover all frequent patterns in long sequences. However, it takes a

long time to solve the Top-*K* frequent patterns for each length in long sequences using MAPD. We furthermore propose a heuristic algorithm, named MAPBOK based on MAPB, which can accelerate the mining speed and achieve high accuracy.

Acknowledgments

This research is supported by the National Natural Foundation of China under grants No. 61229301, 61170190, and 61370144, the Natural Science Foundation of Hebei Province of China under grant No. F2013202138, and the Key Project of the Educational Commission of Hebei Province under grant No. ZH2012038.

References

[1] Kang U, Tsourakakis CE, Appel AP, Faloutsos C, Leskovec J (2011). Hadi: Mining radii of large graphs. ACM Transactions on Knowledge Discovery from Data (TKDD), 5(2): Article NO. 8

[2] Zheng YT, Zha ZJ, Chua TS (2012) Mining travel patterns from geotagged photos. ACM Transactions on Intelligent Systems and Technology (TIST), 3(3): Article No. 56

[3] Liu YH (2013) Stream mining on univariate uncertain data. Applied Intelligence, 39(2): 315-344

[4] Agrawal R, Srikant R (1995) Mining sequential patterns. In: Proceedings of International Conference on Data Engineering, San Jose, CA, pp 3-14

[5] Mooney CH, Roddick JF (2013) Sequential pattern mining - approaches and algorithms. ACM Computing Surveys, 45(2): Article No. 19

[6] Li Z, Han J, Ji M, Tang LA, Yu Y, Ding B, Lee JG, Kays R (2011) MoveMine: Mining moving object data for discovery of animal movement patterns. ACM Transactions on Intelligent Systems and Technology (TIST), 2(4): Article No. 37

[7] Wu SY, Yen E (2009) Data mining-based intrusion detectors. Expert Systems with Applications, 36(3-1): 5605-5612.

[8] Huang TCK (2012) Mining the change of customer behavior in fuzzy time-interval sequential patterns. Applied Soft Computing, 12(3): 1068-1086

[9] Liao VCC, Chen MS (2013) DFSP: a Depth-First SPelling algorithm for sequential pattern mining of biological sequences. Knowledge and Information Systems. Published online: 26 January.

[10] Hu YH, Chen YL, Tang K. Mining sequential patterns in the B2B environment. Journal of Information Science. 2009, 35(6): 677-694

[11] Shie BE, Yu PS, Tseng VS (2013) Mining interesting user behavior patterns in mobile commerce environments. Applied Intelligence, 38 (3): 418-435

[12] Yin J, Zheng Z, Gao L (2012) USpan: An Efficient Algorithm for Mining High Utility Sequential Patterns. In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Beijing, China, pp 660-668

[13] Zhu F, Qu Q, Lo D, Yan X, Han J, Yu PS (2011) Mining Top-K Large Structural Patterns in a Massive Network. Proceedings of the VLDB Endowment, 4(11): 807-818

[14] Wu C, Shie BE, Yu PS, Tseng VS (2012) Mining Top-K high utility itemsets. In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Beijing, China, pp 78-86

[15] Pei J, Han J, Mortazavi-Asl B, Pinto H, Chen Q, Dayal U, Hsu M (2001) PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: Proceedings of International Conference on Data Engineering, Heidelberg, Germany, pp 215-224

[16] Rasheed F, Alhajj R (2010) STNR: A suffix tree based noise resilient algorithm for periodicity detection in time series databases. Applied Intelligence, 32(3): 267-278

[17] Wang YT, Cheng JT (2011) Mining periodic movement patterns of mobile phone users based on an efficient sampling approach. Applied Intelligence, 35(1): 32-40

[18] Yen SJ, Lee YS (2012) Mining time-gap sequential patterns. In: 25th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, Dalian, China, 7345: 637-646

[19] Yen SJ, Lee YS (2013) Mining non-redundant time-gap sequential patterns. Applied Intelligence, 39(4): 727-738

[20] Zhang M, Kao B, Cheung DW, Yip KY (2007) Mining periodic patterns with gap requirement from sequences. ACM Transactions on Knowledge Discovery from Data, 1(2): Article No. 7

[21] Ji X, Bailey J, Dong G (2007) Mining minimal distinguishing subsequence patterns with gap constraints. Knowledge and Information Systems, 11(3): 259-286.

[22] Li C, Wang J (2008) Efficiently mining closed subsequences with gap constraints. In: SIAM International Conference on Data Mining, Georgia, USA, pp 313-322

[23] Li C, Yang Q, Wang J, Li M (2012) Efficient mining of gap-constrained subsequences and its various applications. ACM Transactions on Knowledge Discovery from Data, 6(1): Article No. 2

[24] Min F, Wu Y, Wu X (2012) The Apriori property of sequence pattern mining with wildcard gaps. International Journal Functional Informatics and Personalised Medicine. 4(1): 15–31

[25] Zhu X, X. Wu (2007) Mining complex patterns across sequences with gap requirements. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, pp 2934-2940

[26] He Y, Wu X, Zhu X, Arslan AN (2007) Mining frequent patterns with wildcards from biological sequences. In: IEEE International Conference on Information Reuse and Integration, Las Vegas, USA, pp 329-334

[27] Xie F, Wu X, Hu X, Gao J, Guo D, Fei Y, Hua E (2010) Sequential pattern mining with wildcards. Proceedings of the 22nd International Conference on Tools with Artificial Intelligence, Arras, France, pp 241-247

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

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

[30] Ding B, Lo D, Han J, Khoo SC (2009) Efficient mining of closed repetitive gapped subsequences from a sequence database. In: Proceedings of Conference on Data Engineering, Shanghai, China, pp 1024-1035

[31] Ahmed CF, Tanbeer SK, Jeong BS, Lee YK (2011) HUC-Prune: an efficient candidate pruning technique to mine high utility patterns. Applied Intelligence, 34(2): 181-198

[32] Wu Y, Wu X, Min F, Li Y (2011) A Nettree for pattern matching with flexible wildcard constraints. In: Proceedings of the 2010 IEEE International Conference on Information Reuse and Integration, Las Vegas, USA, pp 109-114

[33] Wu Y, Wu X, Jiang H, Min F (2011) A Nettree for approximate maximal pattern matching with gaps and one-off constraint. In: Proceedings of the 22nd

International Conference on Tools with Artificial Intelligence, Arras, France, pp 38-41