DOI 10.1007/s10115-003-0111-z
Springer-Verlag London Ltd. © 2004
Knowledge and Information Systems (2004) 6: 243–268

Knowledge and
Information Systems

Discovering High-Order Periodic Patterns

Jiong Yang[1], Wei Wang[2] and Philips S. Yu[3]

[1] Computer Science Department, UIUC, Urbana, IL, USA
[2] Computer Science Department, UNC Chapel Hill, Chapel Hill, NC, USA
[3] IBM T. J. Watson Research Center, Hawthorne, NY, USA

Abstract. Discovery of periodic patterns in time series data has become an active research area with many applications. These patterns can be hierarchical in nature, where a higher-level pattern may consist of repetitions of lower-level patterns. Unfortunately, the presence of noise may prevent these higher-level patterns from being recognized in the sense that two portions (of a data sequence) that support the same (high-level) pattern may have different layouts of occurrences of basic symbols. There may not exist any common representation in terms of raw symbol combinations; and hence such (high-level) pattern may not be expressed by any previous model (defined on raw symbols or symbol combinations) and would not be properly recognized by any existing method. In this paper, we propose a novel model, namely *meta-pattern*, to capture these high-level patterns. As a more flexible model, the number of potential meta-patterns could be very large. A substantial difficulty lies in how to identify the proper pattern candidates. However, the well-known *Apriori* property is not able to provide sufficient pruning power. A new property, namely *component location* property, is identified and used to conduct the candidate generation so that an efficient *computation-based* mining algorithm can be developed. Last, but not least, we apply our algorithm to some real and synthetic sequences and some interesting patterns are discovered.

Keywords: Asynchronous pattern; Data mining; Frequent pattern; Meta-pattern; Periodic pattern

1. Introduction

Periodicity detection on time series data is a challenging problem of great importance in many real applications. The periodicity is usually represented as repeated occurrences of a list of symbols in a certain order at some frequency (Han et al, 1998, 1999; Yang et al, 2000). Due to the changes of system behavior, some pattern may only be notable within a portion of the entire data sequence; different patterns may present at different places and be of different durations. The evolution among patterns may also follow some

Received 25 Feb 2002
Revised 7 Sep 2002
Accepted 27 Sep 2002

regularity. Such regularity, if any, would be of great value in understanding the nature of the system and building prediction models. Consider the application of *inventory replenishment*. The history of inventory refill orders can be regarded as a symbol sequence. For brevity, let us only consider the replenishment of flu medicine. Figure 1(a) shows the history of refill orders of a pharmacy during 1999 and 2000 on a weekly basis. The symbol ‘r’ means a refill order of flu medicine was placed in the corresponding week, while, ‘-’ represents that no flu medicine replenishment was made in that week. It is easy to see that the replenishment follows a biweekly pattern during the first half of each year and a triweekly cycle during the second half of each year. This seasonal fluctuation also forms a high-level periodic pattern (the period length is one year). However, such high-level patterns may not be expressible by any previous model (defined in terms of raw symbols) due to the presence of noise, even when the noise is very limited. In the above example, a major outbreak of flu caused a provisional replenishment in the 4th week of 1999 (Fig. 1(a)). Afterwards, even though the replenishment frequency is back to once every other week, the occurrences of all subsequent replenishments become misaligned. Even though the biweekly replenishment cycle was notable in the first two quarters of both 1999 and 2000, the corresponding portions in the data sequence have a different layout of replenishment occurrences. This characteristic determines that the representation of the above two-level periodicity is beyond the expressive power of any traditional model of periodic patterns that only takes raw symbols as components. In any traditional model, each symbol specified in a pattern uniquely matches its counterpart in the data sequence, and all occurrences of a pattern have to share a unique common layout. This inherent limitation would prevent many interesting high-level patterns from being captured. Note that, even if the period length (i.e., 52 weeks) is given,[1] the only annual pattern that is able to be generated under the traditional model via pairwise comparison of symbols corresponding to each week shown in Fig. 1(b). The symbol ‘*’ denotes the don’t care position[2] and can match any symbol on the corresponding position. Clearly, little information is conveyed in this pattern as the important two-level periodicity is completely concealed.

To tackle the problem, we propose a so-called *meta-pattern* model to capture high-level periodicities. A meta-pattern may take occurrences of patterns/meta-patterns (of lower granularity) as components. In contrast, we refer to the patterns that contain only raw symbol(s) as the *basic patterns*, which may be viewed as special cases of meta-patterns. In general, the noise could occur anywhere, be of varied duration, and even occur multiple times within the portion where a pattern is notable as long as the noise is below some threshold. Even though the allowance of noise plays a positive role in characterizing system behavior in a noisy environment, it prevents such a meta-pattern from being represented in the form of an (equivalent) basic pattern. The model of meta-pattern provides a more powerful means of periodicity representation. The recursive nature of meta-pattern can not only tolerate a greater degree of noises/distortion, but also can capture the (hidden) hierarchies of pattern evolutions, which may not be expressible by previous models. In the previous example, the biweekly and the triweekly replenishment cycles can be easily represented by $P_1 = (r : [1, 1], * : [2, 2])$ and $P_2 = (r : [1, 1], * : [2, 3])$, respectively, where the numbers in brackets indicate the offset of the component within the pattern. The two-level periodicity can be easily represented as $(P_1 : [1, 24], * : [25, 25], P_2 : [26, 52])$, which can be interpreted as the pattern P_1 repeats at the first 24 weeks and the pattern P_2 repeats from week 26 to

[1] Note that in many applications, e.g. seismic periodicity analysis, the period length is usually unknown in advance and is part of the mining objective.

[2] It is introduced to represent the position(s) in a pattern where no strong periodicity exhibits.

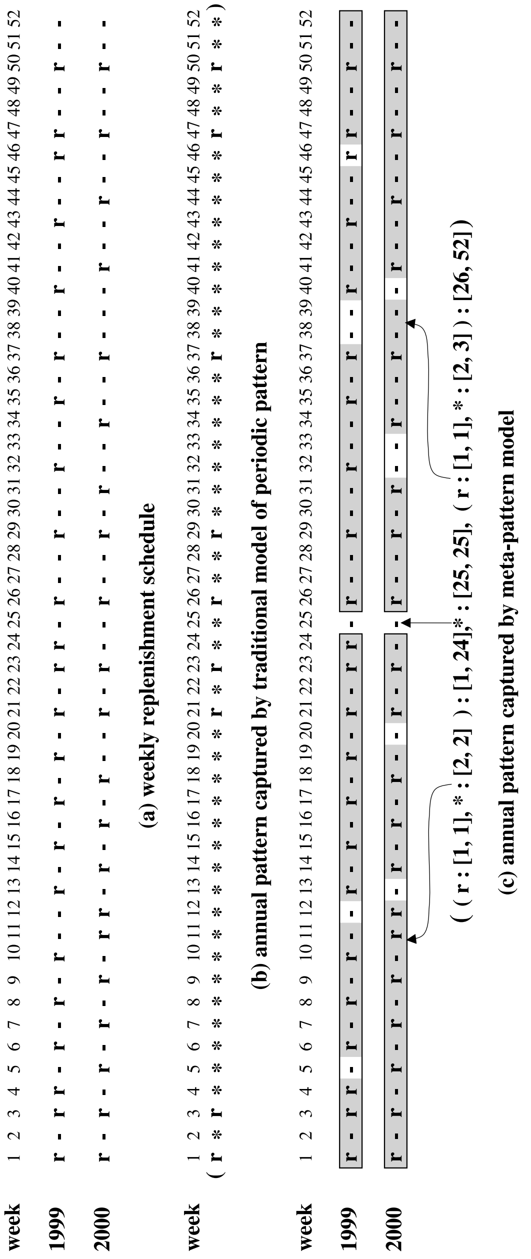

week
1999
2000
(a) weekly replenishment schedule
(r * r * * * * * * * * * * * * * * * * * r * r * * r * r * * * * * * * r * * * * * * * * * r * * r * * *)
(b) annual pattern captured by traditional model of periodic pattern
((r : [1, 1], * : [2, 2]) : [1, 24], * : [25, 25], (r : [1, 1], * : [2, 3]) : [26, 52])
(c) annual pattern captured by meta-pattern model

Fig. 1. Meta-pattern.

week 52. As shown in Fig. 1 (c), each rectangular box denotes the portion where the corresponding low-level pattern (i.e., either $(r : [1, 1], * : [2, 2])$ or $(r : [1, 1], * : [2, 3])$) is notable.

Unfortunately, the flexibility of meta-pattern poses serious challenges in the discovery process, which may not be encountered in mining basic patterns:

- While a basic pattern has two degrees of freedom, the period (i.e., the number of positions in the pattern) and the choice of symbol for each single position, a meta-pattern has an additional degree of freedom: the length of each component in the pattern. It is incurred by the fact that a component may occupy multiple positions. This extra degree of freedom increases the number of potential meta-pattern candidates dramatically.
- Many patterns/meta-patterns may collocate or overlap for any given portion of a sequence. As a result, during the meta-pattern mining process, there could be a large number of candidates for each component of a (higher-level) meta-pattern. This also aggravates the mining complexities.

Therefore, how to identify the ‘proper’ candidate meta-patterns is crucial to the overall efficiency of the mining process, and will be the focus of the algorithmic part of the paper. To tackle this problem, we employ a so-called *component location property*, in addition to the traditionally used Apriori property, to prune the search space. This is inspired by the observation that a pattern may participate in a meta-pattern only if its notable portions exhibit a certain cyclic behavior. A *computation-based* algorithm is devised to identify the potential period of a meta-pattern and, for each candidate period, the potential components and their lengths within the meta-pattern. The set of all meta-patterns can be categorized according to their structures and are evaluated in a designed order so that the pruning power provided by both properties can be fully utilized.

In summary, we claim the following contributions in this paper:

- A model of meta-pattern is identified to capture the cyclic relationship among discovered periodic patterns and to enable a recursive construction of exhibited cyclic regularities.
- The *component location property* is proposed to provide further pruning power, in addition to the traditional Apriori property.
- A *computation-based* algorithm is designed to identify and to verify potential meta-pattern candidates.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of recent related research. The general model is presented in Section 3. Section 4 outlines the major steps of our algorithm. The algorithms of component generation and the pattern verification are elaborated in Sections 5 and 6, respectively. Section 7 presents experimental results. The conclusion is drawn in Section 8.

2. Related Work

Most previous work on mining sequence data fell into two categories: discovering sequential patterns (Agrawal and Srikant, 1995; Berndt and Clifford, 1996; Padmanabhan and Tuzhilin, 1996; Srikand and Agrawal, 1996; Mannila et al, 1997; Bettini et al, 1998; Chakrabarti et al, 1998; Das et al, 1998; Guralnik et al, 1998; Padmanabhan and Tuzhilin, 1998; Garofalakis et al, 1999; Spiliopoulou, 1999; Han et al, 2000; Zaki, 2000, 2001) and mining periodic patterns(Han et al, 1998; Ozden et al, 1998; Han et al, 1999; Yang et al, 2000). The primary difference between them is that the models of sequential pattern purely take into account the number of occurrences of the pattern, while the frameworks for periodic patterns focus on characterizing cyclic behaviors.

2.1. Sequential Patterns

Discovering frequent sequential patterns was first introduced in Agrawal and Srikant (1995). The input data is a set of sequences, called data sequences. Each data sequence is a list of transactions and each transaction consists of a set of items. A sequential pattern also consists of a (fully ordered) list of transactions. The problem is to find all frequent sequential patterns with a user-specified minimum support, where the support of a sequential pattern is the percentage of data sequences that contain the pattern. Apriori-based algorithms, such as AprioriALL (Agrawal and Srikant, 1995) and GSP (Srikand and Agrawal, 1996), were proposed to mine patterns with some minimum support in a level-wise manner. To further improve the performance, a projection-based algorithm called FreeSpan (Han et al, 2000) was introduced to reduce the candidate patterns generated and hence reduce the number of scans through the data. Additional useful constraints (such as time constraint and regular expression constraint) and/or taxonomies were also studied extensively in Garofalakis et al (1999), Srikand and Agrawal (1996), and Zaki (2000) to enable more powerful models of sequential patterns.

As a more generative model, the problem of discovering frequent episodes from a sequence of events was presented in Mannila et al (1997). An episode is defined to be a collection of events that occur relatively close to each other in a given partial order. A time window is moved across the input sequence and all episodes that occur in some user-specified percentage of windows are reported. This model was further generalized by Padmanabhan and Tuzhilin (1996) to suit temporal logic patterns.

2.2. Periodic Patterns

Full cyclic pattern was first studied in Ozden et al (1998). The input data to Ozden et al (1998) is a set of transactions, each of which consists of a set of items. In addition, each transaction is tagged with an execution time. The goal is to find association rules that repeat themselves throughout the input data. Han et al (1998, 1999) presented algorithms for efficiently mining partial periodic patterns. In practice, not every portion in the time series may contribute to the periodicity. For example, a company's stock may often gain a couple of points at the beginning of each trading session but it may not have much regularity at later times. This type of looser periodicity is often referred to as *partial periodicity*. We will see later that our model also allows partial periodicity.

The inclusion of a user-defined calendar is studied in Ramaswamy et al (1998). A user explicitly defines a calendar and interesting patterns are discovered based on the calendar. For example, if a user defines temporal subsequence to start on the days when the US government announces the unemployment rate as the calendar and this calendar is applied to the stock prices in the New York Stock Exchange, then some interesting patterns can be discovered relating the reaction of stock prices to these announcements.

To accommodate the phenomenon that the system behavior may change over time, a flexible model of asynchronous periodic pattern was proposed in Yang et al (2000). In this model, a qualified pattern may be present only within a subsequence and whose occurrences may be shifted due to disturbance. Two parameters, min_rep and max_dis, are employed to specify the minimum number of repetitions required within each segment of non-disrupted pattern occurrences and the maximum allowed disturbance between any two successive valid segments. The intuition behind this is that a pattern needs to repeat itself at least a certain number of times to demonstrate its significance and periodicity. On the other hand, the disturbance between two valid segments has to be within some reasonable bound. Otherwise, it would be more appropriate to treat such disturbance as a signal of 'change of system behavior' instead of random noise injected into some persistent behavior. The parameter max_dis acts as the boundary to separate these two phenomena. For example, Fig. 1(c) shows six valid segments of pattern $(r : [1, 1], * : [2, 2])$ (for both 1999 and 2000) if we set $min\_rep = 2$. (Each valid segment is indicated by a shaded region.) Moreover, if we set $max\_dis = 3$, the three valid segments in 1999 (separated by a disturbance of 1 symbol) form a valid symbol subsequence of $(r : [1, 1], * : [2, 2])$, while the three valid segments in 2000 from another valid symbol subsequence. Obviously, the appropriate values of these two parameters are application dependent and need to be specified by the user. Note that, due to the presence of disturbance, some subsequent valid segment may not be well synchronized with the previous ones. (Some position shifting occurs.) Upon satisfying these two requirements, the longest valid subsequence of a pattern is returned. A two-phase algorithm is devised first to generate potential periods by distance-based pruning followed by an iterative procedure to derive and validate candidate patterns and locate the longest valid subsequence. Still, this paper only concerns patterns constructed from raw symbols and does not address the problem of meta-patterns. As we pointed out earlier, due to the extra degree of freedom possessed by meta-patterns and the massive amount of potential candidate components, no direct generalization of existing algorithm (designed for mining patterns consisting of only raw symbols) can be applied to mine meta-patterns efficiently. In fact, serious challenges exist in how to quickly identify the 'proper' candidate meta-patterns so that unnecessary computation can be avoided. Thus, we will focus on candidate meta-pattern generation (rather than the candidate meta-pattern validation) in this paper. In addition, Sheng and Hellerstein (2001) presented a method to discover patterns with unknown periodicity. This work differs from the meta-pattern because it does not address the problem of finding nested (hierarchical) patterns.

3. Definition of Meta-Patterns

Let $\Im = \{a, b, c, \ldots\}$ be a set of literals. A traditional periodic pattern (Han et al, 1999; Yang et al, 2000) consists of a tuple of k components, each of which is either a literal or '*'. k is usually referred to as the *period* of the pattern. '*' can be substituted for any literal and is used to enable the representation of partial periodicity. For example, $\ldots, a, b, c, a, b, d, a, b, b, \ldots$ is a sequence of literals and $(a, b, *)$[3] represents that the incident 'b following a' occurs for every 3 time instances in the data sequence. The period of this pattern is 3 by definition. Note that the third component in the pattern is filled by a '*' since there is no strong periodicity present in the data sequence with respect to this component. Because a pattern may start anywhere in the data sequence, only patterns whose first component is a literal in $\Im$ need be considered. In this paper, we refer to this type of pattern as a *basic* pattern as each component in the pattern is restricted to be either a literal or a '*'. In contrast, *a meta-pattern* may have pattern(s)/meta-pattern(s) as its component(s). This enables us to represent complicated basic patterns in a

[3] Since each component corresponds to exactly one symbol, we do not have to explicitly record the offset of a component within the pattern as this information can be easily derived.

more concise way and possibly to reveal some hidden patterns among discovered ones. Formally, a *meta-pattern* is a tuple consisting of k components $(x_1, x_2, \ldots, x_k)$ where each x_i $(1 \leq i \leq k)$ can be one of the following choices augmented by the offsets of the starting and ending positions of the component with respect to the beginning of the meta-pattern.

- a symbol in $\Im$;
- 'don't care' *;
- a pattern/meta-pattern.

We also require that at least one position of a meta-pattern has to correspond to a non '*' component to ensure a non-trivial pattern. For example, $((r : [1, 1], * : [2, 2]) : [1, 24], * : [25, 25], (r : [1, 1], * : [2, 3]) : [26, 52])$ is a meta-pattern with three components: $(r : [1, 1], * : [2, 2])$, *, and $(r : [1, 1], * : [2, 3])$. The *length* of a component is the number of positions that the component occupies in the meta-pattern. In the previous example, the component length of $(r : [1, 1], * : [2, 2])$ is 24. We also say that 52 is the *span* of this meta-pattern, which is equal to the sum of the length of all components in the meta-pattern. This pattern can be interpreted as '*the pattern* $(r : [1, 1], * : [2, 2])$ *is true for 24 positions (or weeks in previous example) followed by the pattern* $(r : [1, 1], * : [2, 3])$ *for 27 positions with a gap of one position in between, and such a behavior repeats for every 52 positions*'. For brevity, we sometimes omit the augmenting offset of a component if the length of the component is only one position. For example, $(r, *)$ is the abbreviation of $(r : [1, 1], * : [2, 2])$ and $((r, *) : [1, 24], *, (r, * : [2, 3]) : [26, 52])$ is equivalent to $((r : [1, 1], * : [2, 2]) : [1, 24], * : [25, 25], (r : [1, 1], * : [2, 3]) : [26, 52])$. It is obvious that the meta-pattern is a more flexible model than the basic pattern and the basic pattern can be viewed as a special (and simpler) case of the meta-pattern. Because of the hierarchical nature of the meta-pattern, the concept of *level* is introduced to represent the 'depth' of a meta-pattern. By setting the level of basic pattern to be 1, the *level* of a meta-pattern is defined as the maximum level of its components plus 1. According to this definition, the level of $(r, * : [2, 3])$ is 1 and the level of $P_1 = ((r, *) : [1, 24], *, (r, * : [2, 3]) : [26, 52])$ is 2. Note that the components of a meta-pattern do not have to be of the same level. For instance, $(P_1 : [1, 260], * : [261, 300])$ is a meta-pattern (of level 3) which has a level-2 component and a level-1 component.

All terminologies associated with the basic patterns (Yang et al, 2000) (i.e., level-1 patterns) can be generalized to the case of meta-patterns (i.e., higher-level patterns). We now give a brief overview of terms defined in Yang et al (2000) for basic patterns. Given a symbol sequence $D' = d_1, d_2, \ldots, d_s$ and a basic pattern $P = (p_1, p_2, \ldots, p_s)$, we say D' *supports* P iff, for each $i (1 \leq i \leq s)$, either $p_i = *$ or $p_i = d_i$. D' is also called a *match* of P. Given a pattern P and a symbol sequence D, a list of j disjoint matches of P in D is called a *segment* with respect to P iff they form a contiguous portion of D. j is referred to as the *number of repetitions* of this segment. Such a segment is said to be a *valid segment* iff j is greater than or equal to the required minimum repetition threshold min_rep. A *valid subsequence* in D (with respect to P) is a set of disjoint valid segments where the distance between any two consecutive valid segments does not exceed the required maximum disturbance threshold max_dis. P is said to be a *valid pattern* in D if there exists a valid subsequence in D with respect to P. The parameters min_rep and max_dis, in essence, define the significance of the periodicity and the boundary to separate noise and change of system behavior. The appropriate values of min_rep and max_dis are application dependent and are specified by the user.

Similarly, given a symbol sequence $D' = d_1, d_2, \ldots, d_s$, for any meta-pattern $X = (x_1 : [1, t_1], x_2 : [t_1 + 1, t_2], \ldots, x_l : [t_{l-1} + 1, s])$, D' *supports* X iff, for each component x_i, either (1) x_i is '*' or (2) x_i is a symbol and $d_{t_{i-1}+1} = \ldots = d_{t_i} = x_i$ or (3) x_i is a (meta-)pattern and $d_{t_{i-1}+1}, \ldots, d_{t_i}$ is a valid subsequence with respect to x_i. D' is in turn called a *match* of P. We can define *segment*, *subsequence*, and *validation* in a similar manner to that of a basic pattern. Given a meta-pattern X and a symbol sequence D, a list of j disjoint matches of X in D is called a *segment* with respect to X iff they form a contiguous portion of D. j is referred to as the *number of repetitions* of this segment. Such segment is said to be a *valid segment* iff j is greater than or equal to the required minimum repetitions min_rep. A *valid subsequence* in D (with respect to X) is a set of disjoint valid segments where the distance between any two consecutive valid segments does not exceed the required maximum disturbance max_dis. P is said to be a *valid pattern* in D if there exists a valid subsequence in D with respect to X. Look back to the medicine replenishment example: the max_dis parameter solves the shift problem. Our model can tolerate that the pattern shifts at most max_dis symbols. In other words, our model can capture the patterns that have at least min_rep perfect repetitions continuously and at most max_dis interruption between two portions of perfect repetitions.

In this paper, given a symbol sequence and two parameters min_rep and max_dis, we aim at mining valid meta-patterns together with their longest valid subsequences (i.e., the valid subsequence which has the most overall repetitions of the corresponding meta-pattern). Since a meta-pattern can start anywhere in a sequence, we only need to consider those starting with a non '*' component.

4. Algorithm Outline

The great flexibility of the model poses considerable difficulties to the generation of candidate meta-patterns. Therefore, we will focus on the efficient candidate generation of meta-patterns in the remainder of this paper. The well-known Apriori property holds on the set of meta-patterns of the same span, which can be stated as follows: *for any valid meta-pattern* $P = (P_1 : [1, t_1], P_2 : [t_1 + 1, t_2], \ldots, P_s : [t_{s-1} + 1, t_s])$, *the meta-pattern constructed by replacing any component* P_i *with '*' in* P *is also valid.* For example, let $X_1 = ((a, b, *) : [1, 19], * : [20, 21])$ and $X_2 = ((a, b, *) : [1, 19], * : [20, 21], (b, c) : [22, 27], * : [28, 30], X_1 : [31, 150])$. If X_2 is a valid meta-pattern, then the pattern $X_3 = ((a, b, *) : [1, 19], * : [20, 21], (b, c) : [22, 27], * : [28, 150])$ (generated by replacing X_1 with '*') must be valid as well. Note that X_2 is a level-3 meta-pattern which has three non '*' components: $(a, b, *)$, (b, c), and X_1; whereas X_3 is a level-2 meta-pattern that has two non '*' components: $(a, b, *)$ and (b, c). Intuitively, X_3 should be examined before X_2 so that the result can be used to prune the search space.

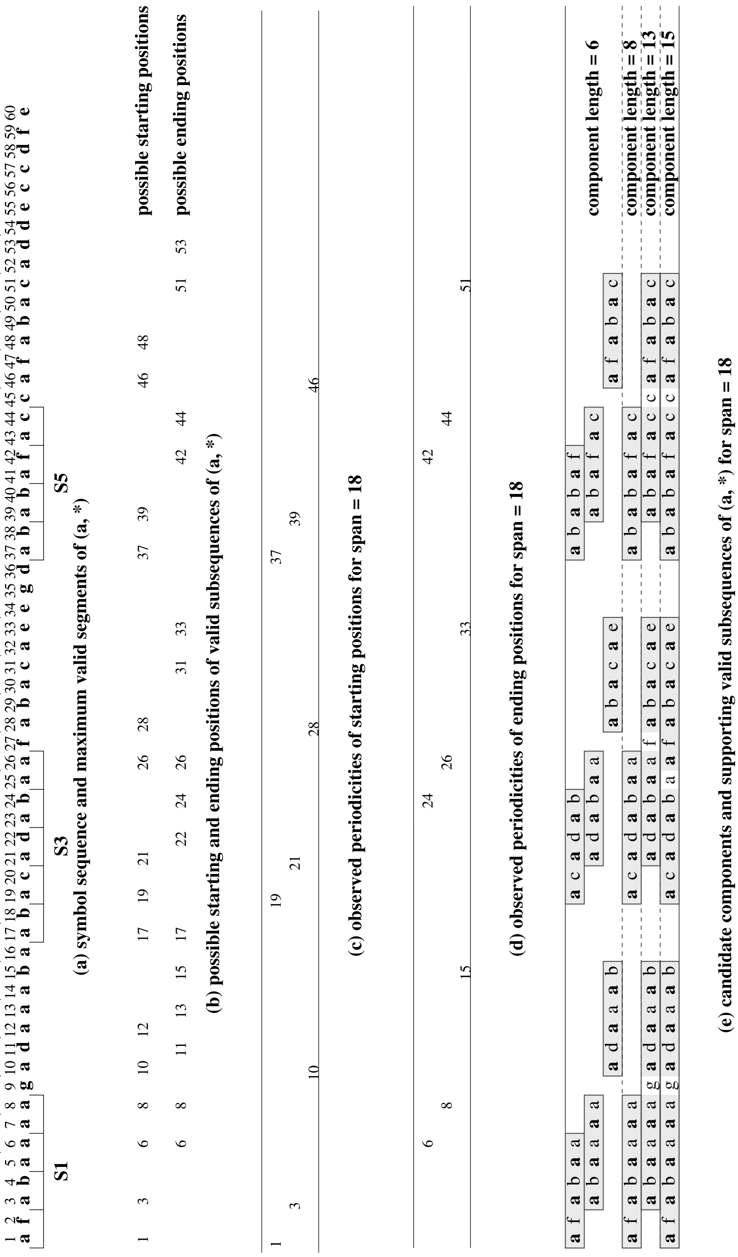
Nevertheless, because of the hierarchical characteristic of the meta-pattern, the Apriori property does not render sufficient pruning power as we proceed to high-level patterns from discovered low-level patterns. After identifying valid meta-patterns of level l, the brute force method (powered by the Apriori property) to mine patterns of level $l + 1$ is first to generate all possible candidates of level $l + 1$ by taking valid lower-level patterns as component(s); and then, verify them against the symbol sequence. While the verification of a base pattern can be performed efficiently (e.g., in linear time with respect to the length of the symbol sequence (Yang et al, 2000)), the verification for a candidate meta-pattern may be a cumbersome process because of the typically complicated structure of the candidate meta-pattern. In fact, considerable difficulty lies in determining whether a certain portion of the raw symbol sequence corresponds to a valid subsequence of a component of the candidate pattern, especially when the component itself is also meta-pattern. One strategy to speed up the process is to store all valid subsequences of each valid low-level pattern when the pattern is verified. Then the procedure of determining whether a portion of the sequence is a valid subsequence of a given component can be accomplished via table look-up operations. Even though this strategy requires additional storage space, it can usually lead to at least an order of magnitude of performance improvement. We will refer to this method as the *match-based approach* in the remainder of this paper: However, this match-based approach is still very cumbersome, and more specifically, suffers from two major drawbacks:

- The number of candidate patterns of a certain level (say level l) is typically an exponential function of the number of discovered lower-level meta-patterns. While a basic pattern has two degrees of freedom – the period and the choice of symbol at each position/component – a meta-pattern has an additional degree of freedom: the length of each component. This additional degree of freedom dramatically increases the number of candidate patterns generated. If there are v valid lower-level patterns, the number of candidate patterns of span s and with exactly k components for level l is in the order of $\Theta(v^k \times (2k)^s)$.
- There are typically a huge number of valid subsequences associated with each valid pattern even though only a few of them may eventually be relevant. Generating and storing all of them would consume a significant amount of computing resources and storage space, which in turn leads to unnecessary inefficiency.

To overcome these drawbacks, we made the following observation.

Property 4.1. (Component Location Property) A valid low-level meta-pattern may serve as a component of a higher-level meta-pattern only if its presence in the symbol sequence exhibits some cyclic behavior and such cyclic behavior has to follow the same periodicity as the higher-level meta-pattern by sufficient number of times (i.e., at least min_rep times).

In the above example, the meta-pattern X_1 can serve as a component of a higher-level meta-pattern (e.g., X_2) only if the locations of valid subsequences of X_1 exhibit a cyclic behavior with a period equal to the span of X_2 (i.e., 150). Otherwise, X_1 could not serve as a component of X_2. This property suggests that we can avoid the generation of a huge number of unnecessary candidate meta-patterns by deriving candidates from qualified span-component combinations according to the component location property. To identify qualified span-component combinations, we need to detect the periodicities exhibited by the locations of valid subsequences of each low level meta-pattern. This can be achieved without generating all valid subsequences for a meta-pattern. In fact, only the set of *maximum valid segments* is sufficient. For a given pattern, a valid segment is a maximum valid segment if it is not a portion of another valid segment: for example, if $min\_rep = 3$ and $max\_dis = 6$, $\{S_1, S_2, S_3, S_4, S_5S_6\}$ is the set of maximum valid segments of basic pattern $(a, *)$ for the symbol sequence in Fig. 2(a). Usually, the number of maximum valid segments is much smaller than the number of valid subsequences. The total number of distinct valid subsequences of $(a, *)$ in the symbol sequence given in Fig. 2(a) would be in the order of hundreds. It is in essence an exponential function of the number of maximum valid segments. Furthermore, for each maximum valid segment, we only need to store a pair of location indexes indicating its starting and ending positions. In the above example, the segment S_1 occupies 8 positions (positions 1 to 8) in Fig. 2(a) and its location indexes are the pair $(1, 8)$. The location indexes of maximum valid segments indeed provide a compact representation of all necessary knowledge of a valid low-level meta-pattern and is motivated by the following observations:



S1
S2
S3
S4
S5
S6
(a) symbol sequence and maximum valid segments of (a, *)
possible starting positions
possible ending positions
(b) possible starting and ending positions of valid subsequences of (a, *)
(c) observed periodicities of starting positions for span = 18
(d) observed periodicities of ending positions for span = 18
component length = 6
component length = 8
component length = 13
component length = 15
(e) candidate components and supporting valid subsequences of (a, *) for span = 18

Fig. 2. Computation-based approach.

- Given the set of location indexes of maximum valid segments of a pattern, it is easy to compute all possible starting positions and ending positions of valid subsequences. Any starting position of a valid segment is also a starting position of a valid subsequence because a valid subsequence is essentially a list of valid segments. Given a maximum valid segment S containing r repetitions of the pattern, there are $r - min\_rep + 1$ distinct starting positions that can be derived from S. More specifically, they are the positions of the first $r - min\_rep + 1$ occurrences of the pattern in S, respectively. For instance, positions 1 and 3 are the two starting positions derived from S_1. Similarly, all possible ending positions can be computed as well.
- The starting positions of the valid subsequences that exhibit cyclic behavior also present the same periodicity and so do their ending positions. Figure 2(b) shows the set of possible starting positions and ending positions of valid subsequences of $(a, *)$. When $min\_rep = 3$, by careful examination, the potential periodicities of $(a, *)$ (i.e., the possible spans of meta-patterns that $(a, *)$ may participate in as a component) include 7, 9, 11, 18, and 20. The periodic behavior discovered on starting positions and ending positions for span = 18 is shown in Fig. 2(c) and (d), respectively.

Thus, our strategy is to first compute the set of possible starting positions and identify, if any, the ones that exhibit some periodicity. The same procedure is also performed on the ending positions. If the same periodicity exists for both starting positions and ending positions, we then examine, for each pair of starting and ending positions, whether a valid subsequence exists and what is the possible format of the higher-level meta-pattern (i.e., possible span of the meta-pattern and possible length of the component). Figure 2(e) shows some candidate components generated from $(a, *)$ and the valid subsequences that support them. It is important to notice that the maintenance of the location indexes of maximum valid segments leads to a double-win situation. Besides its positive role in candidate generation, it also enables the verification process to be accomplished efficiently without the expensive generation and maintenance of all valid subsequences or the necessity to resort to the raw symbol sequence. As a result, we devise an efficient *computation-based* algorithm (as opposite to the traditional match-based approach) in the sense that the discovery of valid meta-patterns (other than base patterns) can be accomplished through pure computation (performed on the location indexes of maximum valid segments) without ever resorting back to the raw symbol sequence. It has been demonstrated that this advantage offers at least two orders of magnitude speed-up compared to the match-based approach. The component location property can provide a substantial inter-level pruning effect during the generation of high-level candidates from valid low-level meta-patterns, whereas the traditional Apriori property can render some pruning power to conduct the mining process of meta-patterns of the same level. While all meta-patterns can be categorized according to their levels and the number of non '*' components in the pattern as shown in Fig. 3, the pruning effects provided by the component location property and the Apriori property are indicated by dashed arrows and solid arrows, respectively. Consequently, the algorithm consists of two level of iterations. The outer iteration exploits the component location property, while the inner iteration utilizes the Apriori property. More specifically, each outer iteration discovers all meta-patterns of a certain level (say, level l) and consists of the following two phases:

1. *Candidate component generation.* For each newly discovered valid pattern/meta-pattern of level l, generate candidate components for meta-patterns of level $l + 1$. The component location property is employed in this phase.
2. *Candidate pattern generation and verification.* This phase generates candidate meta-patterns of level $l + 1$ based on the candidate components discovered in the previous

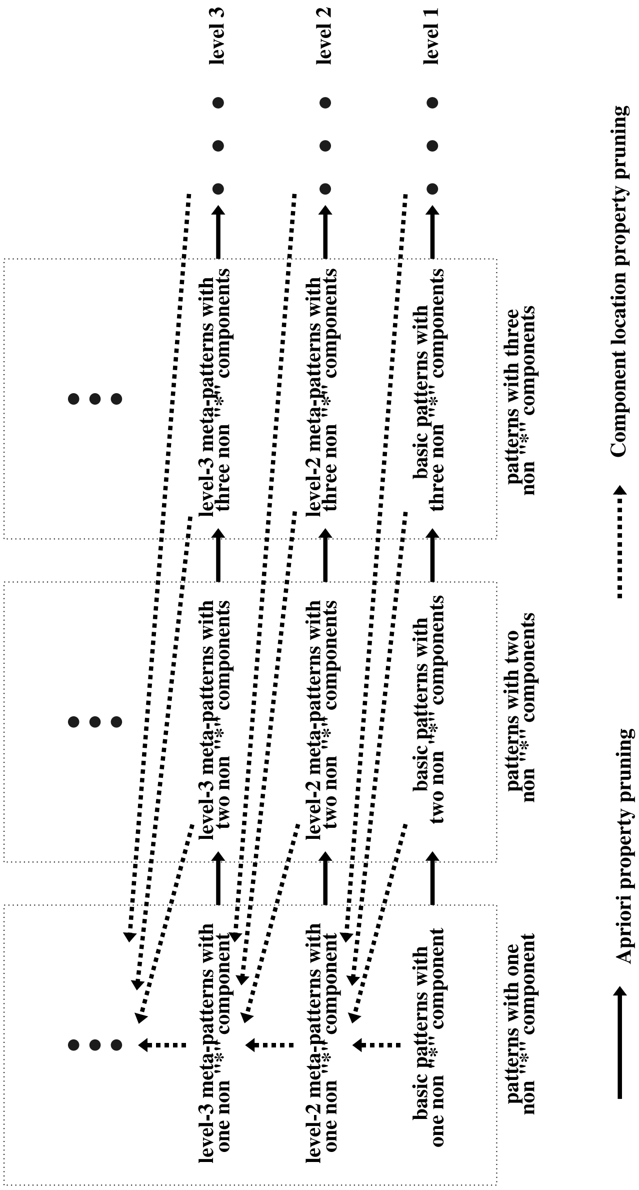

level-3 meta-patterns with one non "*" component
level-3 meta-patterns with two non "*" components
level-3 meta-patterns with three non "*" components
level 3
level-2 meta-patterns with one non "*" component
level-2 meta-patterns with two non "*" components
level-2 meta-patterns with three non "*" components
level 2
basic patterns with one non "*" component
basic patterns with two non "*" components
basic patterns with three non "*" components
level 1
patterns with one non "*" component
patterns with two non "*" components
patterns with three non "*" components
Apriori property pruning
Component location property pruning

Fig. 3. Pruning directions.

step and validates them. This phase utilizes the Apriori property and contains an iteration loop. During each iteration, meta-patterns with a certain number (say k) of non '*' components are examined, which includes the following two steps.

(a) If $k = 1$, the candidate singular meta-patterns[4] of level $l + 1$ are generated from candidate components derived in the previous phase. Otherwise, the candidate meta-patterns of level $l + 1$ with k non '*' components are generated based on the discovered level-$(l + 1)$ meta-patterns with $(k - 1)$ non '*' components.

(b) The newly generated candidate patterns are validated.

This inner iteration continues until no new candidate patterns of level $(l + 1)$ can be generated.

The entire procedure terminates when no new candidate components can be generated. In the following sections, we will elaborate on each phase in detail.

5. Candidate Component Generation

For a given pattern X of level l, a three-step algorithm is devised to generate candidate components (based on X) of patterns of level $l + 1$. The formal description of the algorithm is in Algorithm 5.1.

Algorithm 5.1. Component candidate generation

```
/* To find component candidates for patterns of level $l + 1$ */

Component_Generation($min\_rep$, $Seg_X$)
/* $Seg_X$ is the set of maximum valid segments for an $l$th-level pattern $X$ */
{
    $starting \leftarrow \emptyset$
    /* It holds possible starting positions of valid subsequences of $X$. */
    $ending \leftarrow \emptyset$
    /* It holds possible ending positions of valid subsequences of $X$. */
    $start \leftarrow \emptyset$
    /* It holds starting positions in $starting$ that exhibit some periodicity. */
    $end \leftarrow \emptyset$
    /* It holds ending positions in $ending$ that exhibit some periodicity. */
    $candidates \leftarrow \emptyset$
    /* It holds the set of valid subsequences (of $X$) that support candidate
    components derived from $X$. */
    Find_Start_End($Seg_X$, $starting$, $ending$)
    Periodicity($starting$, $start$)
    /* Finding positions in $starting$ that exhibits some span $s$
    and storing them into $start$ structures. */
    Periodicity($ending$, $end$)
    /* Finding positions in $ending$ that exhibits some span $s$
    and storing them into $end$ structures. */
    **for each** retained span $s$ **do**
```

[4] A singular meta-pattern is a meta-pattern that has only one non '*' component. Otherwise, it is called a complex meta-pattern.

```
        Component ($s$, $start$, $end$, $candidates$)
        /* For a given span $s$, find the possible length of the component
        that involves $X$ */
    return ($candidates$)
}
```

The proposed algorithm assumes that for a given lth-level pattern X, the set of the maximum valid segments of X are stored in Seg_X. Note that only the location indexes are physically stored and all other information associated with each maximum segment can be derived easily. For example, S_1, S_2, S_3, S_4, S_5, and S_6 are the maximum valid segments of $(a, *)$ in Fig. 2(a). Looking ahead, in the meta-pattern validation procedure (see Section 6.2), the algorithm will output all maximum valid segments of a valid meta-pattern as a by-product of the longest valid subsequence discovery without incurring any extra overhead. Note that we only keep track of the set of maximum valid segments instead of all valid subsequences because the number of the valid subsequences is much larger than the number of the maximum valid segments. The benefits of only tracking maximum valid segments are shown in Section 7.2.2.

In the subroutine $Find\_Start\_End$ in Algorithm 5.2, each maximum valid segment (in Seg_X) is examined successively to compute the set of possible starting and ending positions of valid subsequences. A possible starting position is the one that has at least min_rep repetitions immediately following it. Similarly, a possible ending position is the one with at least min_rep repetitions immediately preceding it. Figure 2(b) shows the possible starting/ending positions derived from the maximum valid segments in Fig. 2(a).

Even though two maximum segments in Seg_X may overlap, their $start\_rep$ sets are disjoint from each other (that is, no position (in the input sequence) serves in the set $start\_rep$ for multiple maximum segments). For instance, S_1 and S_2 overlap with each other, and {1, 3, 5, 7} and {6, 8, 10, 12, 14, 16} are the sets of starting positions of repetitions in S_1 and S_2 respectively. They are obviously disjoint. Therefore, each position in the input sequence serves as a starting position at most once. The same proposition also holds for the ending position. Thus, the computational complexity of each invocation of $Find\_Start\_End$ is $O(N)$, where N is the length of the input sequence.

Algorithm 5.2. Find valid starting and ending positions

```
Find_Start_End($Seg_X$, $starting$, $ending$)
{
    for each segment $seg \in Seg_X$ do
        for each $start\_rep \in seg$ do
        /* $start\_rep$ records the starting position of each repetition
        in a maximum segment. */
            if (Valid_Start($start\_rep$, $seg$)) do
            /* If there are at least $min\_rep$ repetitions following $start\_rep$,
            then Valid_Start returns true, otherwise false. */
                $starting \leftarrow starting \cup start\_rep$
        for each $end\_rep \in seg$ do
        /* $end\_rep$ records the ending position of each repetition
        in a maximum segment. */
            if (Valid_End($end\_rep$, $seg$)) do
            /* If there are at least $min\_rep$ repetitions preceding $end\_rep$,
            then Valid_End returns true, otherwise false. */
                $ending \leftarrow ending \cup end\_rep$
    return
}
```

The *Periodicity* function in Algorithm 5.3 locates periodicities presented in a list of positions and is applied to both starting positions and ending positions. With two scans of the list of possible starting positions, we can generate a potential span s of higher-level meta-pattern that X may participate in. Intuitively, s should be at least the minimum length of a valid subsequence of X (i.e., $span(X) \times min\_rep$, where $span(X)$ is the span of X). For example, in Fig. 2(a), the minimum length of a valid subsequence of $(a, *)$ is $2 \times 3 = 6$ if $min\_rep = 3$. In turn, the span of any valid meta-pattern that takes $(a, *)$ as a component has to be at least 6 since the component $(a, *)$ will occupy at least 6 positions. During the first scan, a counter is initialized for each possible span greater than or equal to $span(X) \times min\_rep$. While scanning through the list of starting positions, for each starting position x, consider the distance of x to any previous starting position y in the list. If it is larger than $span(X) \times min\_rep$, the corresponding counter is incremented. At the end of the scan, only spans whose corresponding counter reaches $min\_rep - 1$ are retained. During the second pass through the list, for each retained span s, the algorithm locates series of at least min_rep starting positions, such that the distance between any pair of consecutive ones is s. Note that there may be multiple series existing concurrently. As shown in Fig. 2(c), {1, 19, 37}, {3, 21, 39}, and {10, 28, 46} are three series of starting positions that exhibit periodicity of span 18. The same process is also performed on the ending positions. Figure 2(d) shows the result for span 18 in the above example.

It is easy to see that the first scan through the list takes $O(|list|^2)$ computations and the second scan may consume $O(|S| \times |list|^2)$ time, where $|list|$ and $|S|$ are the number of positions in $list$ and the number of retained spans after the first scan, respectively. Since the number of positions in $list$ is at most the length of the input sequence, the overall computational complexity is $O(|S| \times |list|^2) = O(|S| \times N^2)$, where N is the length of the input sequence.

Algorithm 5.3. Periodicity discovery

```
Periodicity($list$, $potential$)
{
    for each potential span $s$ ($s \geq min\_rep \times span(X)$) do
        initialize a counter $count[s]$ to zero
    for each position $x \in list$ do
        for each position $y \in list$ ($y < x$) do
            $distance \leftarrow x - y$
            increment $count[distance]$
    $S \leftarrow$ all spans $s$ where $count[s] \geq min\_rep - 1$
    /* $S$ holds spans of potential periodicities that may exhibit in $list$ */
    for each span $s \in S$ do
        for each position $x \in list$ do
            if there exists a series of positions $x, x + s, \ldots, x + i \times s$ in $list$
                and $i \geq min\_rep$ do
            /* In the case that multiple valid values of $i$ exist, the maximum
            one is chosen. */
                $potential[s] \leftarrow potential[s] \cup \{x, x + s, \ldots, x + i \times s\}$
                /* $potential$ stores the set of positions in $list$ that
                exhibit periodicity. */
  return
}
```

Then, in the function *Component* (Algorithm 5.4), for each retained span s, consider each pair of cyclic starting position series $x, x+s, \ldots, x+s \times j_x$ and cyclic ending position series $y, y+s, \ldots, y+s \times j_y$. If there exist two sub-series, $x', x'+s, \ldots, x'+\lambda \times s$ and $y', y'+s, \ldots, y'+\lambda \times s$ such that $min\_rep \times span(X) \leq y'-x' \leq s$ and $\lambda \geq min\_rep$, then X may participate in a meta-pattern of span s as a component that occupies $y'-x'$ positions. The subsequence from position x' to y', from position $x'+s$ to $y'+s$, and so on are potentially valid subsequences to support X as a component in a higher-level meta-pattern. For example, there are 3 series of cyclic starting positions and 3 series of cyclic ending positions corresponding to span 18 as shown in Fig. 2(c) and (d) respectively. Let us take a look at the starting position series 3, 21, 39 and ending position series 15, 33, 51. The starting position 3 and ending position 15 satisfy the above condition and the subsequences from position 21 to 33, and from position 39 to 51 are two additional potential valid subsequences to support $(a, *)$ as a component of a higher-level meta-pattern and such a component occupies 13 positions (Fig. 2(e)). Note that the subsequences in this example are not perfect repetitions of $(a, *)$. In fact, each of them consists of two perfect segments of $(a, *)$ separated by a disturbance of length 1. This example further verifies that the meta-pattern can indeed tolerate imperfection that is not allowed in basic periodic patterns.

Since, for a given span s, the cardinalities of $start[s]$ and $end[s]$ are at most the input sequence length N, the computational complexity of $Component()$ is $O(|start[s]| \times |end[s]|) = O(N^2)$ for a given span s.

Algorithm 5.4. Find potential component

```
Component($s$, $start$, $end$, $candidates$)
{
  for each $start\_position \in start[s]$ with span $s$ do
      for each $end\_position \in end[s]$ with span $s$ do
          if ($end\_position - start\_position \leq s$) and
                  /* The total length of the component cannot exceed
                  the span of the periodicity. */
              ($end\_position - start\_position \geq min\_rep \times span(X)$) and
                  /* The length of the component has to be at least the minimum
                  length of a valid subsequence of $X$ */
              (Valid_Subsequence ($start\_position, end\_position$)) do
                  /* Valid_Subsequence returns $true$ if the subsequence between
                  $start\_position$ and $end\_position$
                  is a valid subsequence of $X$. */
          {
              $component\_length \leftarrow end\_position - start\_position$
              $new\_component \leftarrow (X, component\_length)$
              $candidates \leftarrow$
               $candidates \cup (start\_position, end\_position, new\_component)$
          }
}
```

Notice that the above identified potential subsequences are not guaranteed to be valid because we only consider the possible starting and ending positions and ignore whether the symbols in between form a valid subsequence. In fact, the identified subsequences might not be valid especially in the scenario where valid segments scatter sparsely throughout the data sequence. This can be observed from the example in Fig. 4. The three potential subsequences generated for component length 18 are invalid if $max\_dis = 5$. Therefore, it is necessary to validate these subsequences. We note that the set of maximum valid segments and their connectivity[5] can be organized into a graph and a depth-first traversal would be able to verify whether a subsequence between two positions is valid or not.

It is easy to see that each invocation of $Component\_generation()$ would take $O(|S| \times N^2)$ computation. Note that this is only a worst-case bound and the actual running time is usually much faster. We also want to mention that, without the component location property, an algorithm that employs the Apriori property would have to consume as much as $O(N^{2 \times min\_rep})$ time. Experimental studies in Section 7 also demonstrate the pruning power of the component location property.

6. Candidate Pattern Generation and Validation

6.1. Candidate Pattern Generation

For each combination of component P, span s, and component length p, the candidate singular pattern $(P[1, p], * : [p + 1, s])$ is constructed. In the previous example in Fig. 2, four candidate singular meta-patterns (starting at position 1) are constructed from the candidate component $(a, *)$ for span 18, one for each distinct component length shown in Fig. 2(e). They are $((a, *) : [1, 6], * : [7, 18])$, $((a, *) : [1, 8], * : [9, 18])$, $((a, *) : [1, 13], * : [14, 18])$, and $((a, *) : [1, 15], * : [16, 18])$. Note that any pattern of format $(* : [1, t], P[t + 1, t + p], * : [t + p + 1, s])$ is essentially equivalent to $(P[1, p], * : [p + 1, s])$ with a shifted starting position in the data sequence.

For the generation of candidate complex patterns, the Apriori property is employed. The candidate pattern $(x_1 : [1, t_1], x_2 : [t_1 + 1, t_2], \ldots, x_l : [t_{k-1} + 1, s])$ is constructed if all of $(x_2 : [1, t_2 - t_1], \ldots, x_l : [t_{k-1} - t_1 + 1, s - t_1], * : [s - t_1 + 1, s])$,[6] $(x_1 : [1, t_1], * : [t_1 + 1, t_2], \ldots, x_l : [t_{k-1} + 1, s])$, ..., and $(x_1 : [1, t_1], x_2 : [t_1 + 1, t_2], \ldots, * : [t_{k-1} + 1, s])$ are valid patterns. Referring back to the inventory replenishment example discussed previously, after we identify $((r, *) : [1, 24], * : [25, 52])$ and $((r, * : [2, 3]) : [1, 27], * : [28, 52])$ as valid patterns (through the process presented previously), two candidate patterns will be constructed via the Apriori property, and they are $((r, *) : [1, 24], *, (r, * : [2, 3]) : [26, 52])$ and $((r, *) : [1, 24], (r, * : [2, 3]) : [25, 51], *)$. In general, for a given set of valid patterns, multiple candidate patterns can be constructed, each of which corresponds to a possible layout of gaps (filled by *) between each pair of consecutive non '*' components. This is the primary difference between the application of the Apriori property in traditional models and in the discovery of meta-pattern.

6.2. Canonical Pattern

The candidate patterns generated by the method above may be redundant. For instance, let us consider sequence ABABABAB. Both patterns (AB) and (ABAB) can be

[5] If the disturbance between two segments is at most max_dis, then we consider they are connected. Otherwise, they are considered not connected.

[6] This is equivalent to $(* : [1, t_1], x_2 : [t_1 + 1, t_2], \ldots, x_l : [t_{k-1} + 1, s])$.

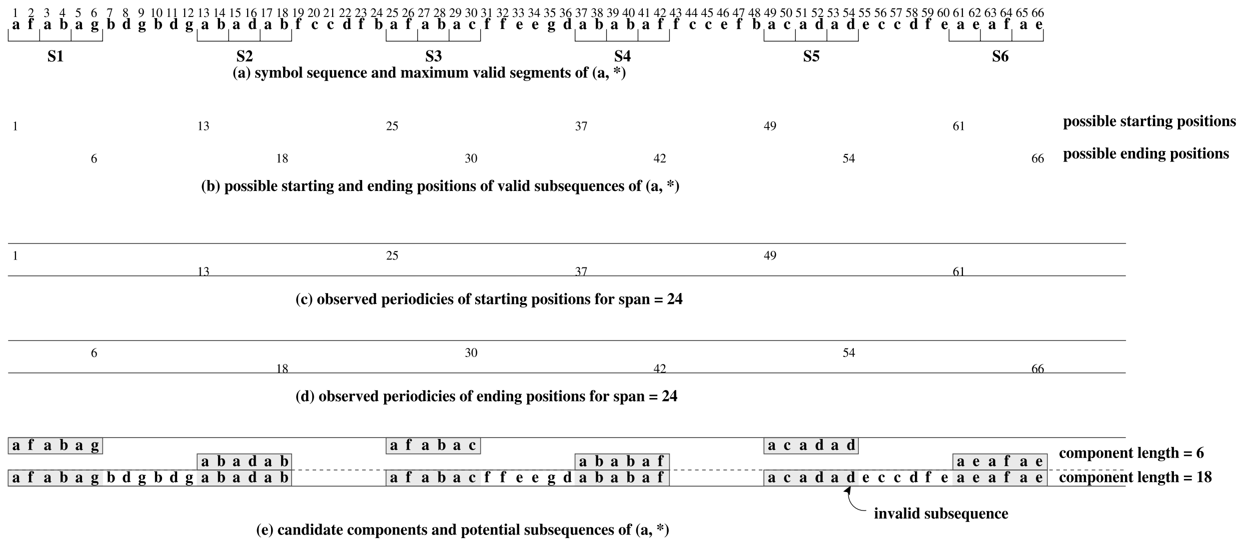

a f a b a g b d g b d g a b a d a b f c c d f b a f a b a c f f e e g d a b a b a f f c c e f b a c a d a d e c c d f e a e a f a e
S1
S2
S3
S4
S5
S6
(a) symbol sequence and maximum valid segments of (a, *)
possible starting positions
possible ending positions
(b) possible starting and ending positions of valid subsequences of (a, *)
(c) observed periodicies of starting positions for span = 24
(d) observed periodicies of ending positions for span = 24
component length = 6
component length = 18
invalid subsequence
(e) candidate components and potential subsequences of (a, *)

Fig. 4. An example of invalid potential subsequence.

valid patterns if $min\_rep \leq 3$. However, it is clear that the pattern (ABAB) is a redundant pattern of (AB). Thus, all valid patterns can be divided into two categories: derived patterns and canonical patterns.

Definition 6.1. A pattern P_1 is called a *derived pattern* if there exists another pattern P_2 ($P_2 \neq P_1$) and P_1 can be formed by appending multiple P_2 together.

We say P_1 is derived from P_2. In the previous example, (ABAB) is derived from (AB).

Definition 6.2. A pattern P_1 is called a canonical pattern if there does not exist any pattern from which P_1 is derived.

(AB) is a canonical pattern in the previous example.

In many applications users are only interested in the valid canonical patterns. To remove the derived patterns from consideration, we use the following observation. Let P_1 be a pattern derived from P_2. P_1 is valid if P_2 is valid by definition. As a result, we only need to consider the canonical patterns. During the candidate generation phase, we take a look at a candidate to see whether it is canonical. If so, we keep it as a candidate. Otherwise, it is removed from further consideration.

6.3. Candidate Pattern Validation

For a given candidate meta-pattern X, we need to find the longest valid subsequence for X if there exists any. This is similar to the longest subsequence identification algorithm presented in Yang et al (2000), with one exception: a component in a meta-pattern may correspond to a valid subsequence of some lower-level pattern, while each component in Yang et al (2000) is restricted to a symbol. This brings some difficulty to the identification of occurrence of X in the symbol sequence. Fortunately, the problem can be easily solved since we keep in $candidate$ the set of maximum valid segments that may support a lower-level pattern as a component of X in the step of candidate component generation. To verify whether a component Y matches a certain portion of the sequence, we can go back to verify Y against the maximum valid segments stored in $candidate$. To further improve the efficiency of the algorithm, the segments in $candidate$ can be organized in some data structure, e.g., a tree.

The rest of the algorithm deals with how to stitch occurrences of a meta-pattern together to generate the longest valid subsequence. This part of the algorithm is exactly the same as that in Yang et al (2000) where, for a given candidate meta-pattern X, the longest valid subsequence can be located in linear computation time with respect to the length of the symbol sequence. it. Note that this procedure does not need to resort to the raw symbol sequence and therefore can be accomplished very efficiently.

At the same time as generating the longest valid subsequence, a separate data structure Seg_X is maintained simultaneously to store all maximum valid segments. Every time a match of the meta-pattern X is discovered, if there exists a segment seg in Seg_X such that the last repetition of X in seg is adjacent to the current match, then we extend seg to include the current match. Otherwise, a new segment that consists of the current match is created. After a scan of the input symbol sequence, segments that contain less than min_rep repetitions of X are removed from Seg_X. The remaining segments are the maximum valid segments of X, which will be used in generating candidate components of higher-level meta-patterns.

Each invocation of $Maximum\_Valid\_Segment()$ takes $O(N)$ time to finish. The set Seg_X can be indexed according to the ending position of each segment to facilitate the process. Since there are at most $\frac{N}{span(X) \times min\_rep}$ segments of X, the space required to store Seg_X is $O(\frac{N}{span(X) \times min\_rep})$. Note that we only need to store the starting and ending positions of each segment.

Algorithm 6.1. Maximum valid segment discovery

```
Maximum_Valid_Segment()
{
  for each match $M$ of $X$ discovered in the symbol sequence do
      if there exist a segment $seg \in Seg_X$ s.t. $seg$ is adjacent to $M$ do
          extend $seg$ to include $M$
      else
          $newseg \leftarrow M$
          $Seg_X \leftarrow Seg_X \cup \{newseg\}$
  for each $seg \in Seg_X$ do
      if $seg$ has less than $min\_rep$ repetitions do
          $Seg_X \leftarrow Seg_X - \{seg\}$
  return
}
```

7. Experimental Results

The meta-pattern discovery algorithm is implemented in C on an AIX workstation with 300 MHz CPU and 128 MB main memory. A real trace log from the search engine *scour.net* is employed to evaluate the benefits of the meta-pattern model while four synthetically generated sequences are used to measure the performance of our algorithm.

7.1. Scour Traces

Scour is a web search engine specialized in multimedia content search whose URL is 'http://www.scour.net'. Since early 2000, the average daily number of hits on Scour has grown to over one million. A trace of all hits on Scour between March 1 and June 8 (total 100 days) (V. Busam, personal communication, 2000) were collected. The total number of accesses is over 140 million. Then the entire trace is summarized into a sequence as follows. The trace is divided into 30-minute intervals. The number of hits during each 30-minute interval is calculated. Finally, we label each interval with a symbol. For example, if the number of hits is between 0 and 9999, then this interval is labeled as a, if the number of hits is between 10,000 and 19,999, then this interval is labeled as b, and so on. The summarized sequence consists of 4800 occurrences of 43 distinct symbols.

Table 1 shows the number of patterns discovered from the Scour sequence with respective thresholds. There exist some interesting patterns. When min_rep and max_dis are set to 3 and 200, respectively, there is a level 3 meta-pattern. This level 3 pattern describes the following phenomenon. On a weekday between 4 am and 12 pm EST, there exists a pattern (b,b) where b stands for the number of hits is between 10,000 and 19,999; and during 5 pm to 1:30 am EST, we found the pattern (e, *), which means that the number of hits is between 40,000 and 49,999. Furthermore, this pattern repeated itself during each weekday within a week (i.e., level 2 pattern) and it also exhibits a weekly trend (i.e., level 3 pattern). This observation confirms the cyclical behavior of the Internet traffic discovered in Thompson et al (1997). Furthermore, various studies

Table 1. Patterns discovered in scour trace

min_rep	3	10	20
max_dis	200	200	200
Level 1 patterns	107	31	15
Level 2 patterns	12	5	2
Level 3 patterns	1	0	0
Meta only Patterns	10	5	2

(Crovella and Bestavros, 1997) have shown that the traffic on the World Wide Web exhibits self-similarity, which also confirms our findings. In addition, we also compute the meta-patterns that cannot be expressed in the form of basic patterns. We call these patterns *meta only* patterns and the number of these patterns is also shown in Table 1. From this table, we can see that most of the discovered level 2 and 3 patterns cannot be expressed in the form of basic patterns, and thus can only be represented as meta-patterns.

To further understand the behavior of our proposed meta-pattern mining algorithm, we constructed four long synthetic sequences and the performance of our algorithm on these sequences is presented in the following section.

7.2. Synthetic Sequences

The four synthetic sequences are generated as follows. Each sequence consists of 1024 distinct symbols and 20M occurrences of symbols. The synthetic sequence is generated as follows. First, at the beginning of the sequence, the level of pattern is determined randomly. There are four possible levels, i.e., 1, 2, and 3, 4. Next, the number of segments in this pattern is determined. The length l of each segment is selected based on a geometric distribution with mean μ_l. The number of repetitions of a lower-level pattern in a segment is randomly chosen between min_rep and $\lfloor \frac{l}{p} \rfloor$, where p is the span of the lower-level pattern. The number of symbols involved in a pattern is randomly chosen between 1 and the span p. The number of valid segments is chosen according to a geometrical distribution with mean μ_s. After each valid segment, the length of the disturbance is determined based on a geometrical distribution with mean μ_d. This process repeats until the length of the sequence reaches 20M. Four sequences are generated based on values of μ_l, μ_s, μ_r, and μ_d in Table 2.

7.2.1. Effects of Component Location Property Pruning

In our proposed algorithm, we use the component location property pruning to reduce the candidate patterns. Table 3 shows the pruning power of our algorithm. The pruning power is measured as the fraction of candidate patterns. We can see that the candidate patterns in our algorithm are around 10^{-2} to 10^{-4} of the overall patterns. This means that on average less than 1% of all patterns need to be examined in our algorithm. In addition, the pruning power increases (i.e., the fraction decreases) with larger min_rep because fewer patterns may be qualified by a larger (more restricted) min_rep parameter. In this experiment, the max_dis is fixed to 20.

We also study the effects of the parameter max_dis on our algorithm. Table 4 shows the effects of max_dis. The pruning power of our algorithm is evident. More than 99% of patterns are pruned. The pruning power decreases with larger max_dis threshold

Table 2. Parameters of synthetic data sets

Data Set	μ_l	μ_s	μ_r	μ_d
$DS1$	5	5	50	50
$DS2$	5	5	1000	1000
$DS3$	100	1000	50	50
$DS4$	100	1000	1000	1000

Table 3. Effects of min_rep on pruning power

min_rep	Scour trace	$DS1$	$DS2$	$DS3$	$DS4$
10	0.008	0.002	0.002	0.001	0.003
20	0.002	0.0008	0.0007	0.0002	0.0009
30	0.0008	0.0002	0.0003	0.00008	0.0002
40	0.0003	0.00004	0.00007	0.00001	0.00005

because more patterns may be qualified. Overall, the component location property can prune away a significant number of patterns and thus reduce the execution time of the pattern discovery process. We fix min_rep to 20.

7.2.2. Effects of Computation-Based Approach

In our approach, we only store the maximum valid segments for each pattern. We compare the computation-based approach with the match-based approach. In the match-based approach, for each pattern, all valid subsequences are stored and used to mine higher-level meta-patterns. For each level of pattern, we track the CPU time consumed by the computation-based approach and the match-based approach. (We assume that all information can be stored in main memory. Since the number of possible subsequences is much larger than that of maximum valid segments, the match-based approach has more advantages with this assumption.) The ratio of the CPU time of the computation-based approach over that of the match-based approach is calculated and presented in Table 5. It is obvious that the computation-based approach can save at least 95% of the CPU time compared to the match-based approach. This is due to the fact that the number of maximum valid segments is far less than that of valid subsequences, as we explained in Section 5.

7.2.3. Overall Response Time

The overall response time is one of the most important criteria for evaluation of an algorithm. We mine the meta-patterns with different min_rep threshold. For a given min_rep, we mine the patterns on all four data sets and the average response time over

Table 4. Effects of max_rep on pruning power

max_dis	Scour trace	$DS1$	$DS2$	$DS3$	$DS4$
10	0.0009	0.0002	0.0002	0.0001	0.0003
20	0.002	0.0008	0.0007	0.0002	0.0009
30	0.008	0.002	0.003	0.0008	0.002
40	0.01	0.007	0.009	0.002	0.005

Table 5. CPU time ratio of computation-based approach versus match-based approach

Level	$DS1$	$DS2$	$DS3$	$DS4$
2	0.02	0.02	0.04	0.01
3	0.05	0.03	0.01	0.02

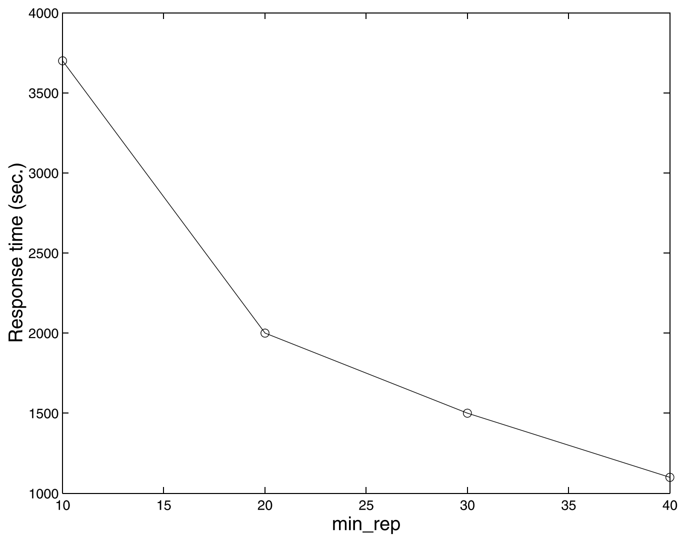


Fig. 5. Response time of meta-pattern mining algorithm.

the four data sets are taken and shown in Fig. 5. The average response time decreases exponentially as *min_rep* increases. Although the average response time is a bit long when *min_rep* is small, it is still tolerable (around 1 hour) due to the pruning effects of component location property and the computation-based candidate pattern generation. The meta-pattern mining algorithm is also applied to symbol sequences with different lengths. We found that the response time of our algorithm is linearly proportional to the length of the symbol sequence.

8. Conclusion

Meta-pattern is proposed to capture the hierarchical cyclic behavior exhibited in a data sequence. A meta-pattern itself can serve as a component of some higher-level meta-patterns. To accommodate noises in the data sequence, two parameters *min_rep* and *max_dis* are used to qualify a subsequence. The number of candidates of meta-patterns could be very large. To minimize the response time of the pattern mining process, a pruning algorithm based on the component location property and Apriori property is proposed which can greatly reduce the number of candidate patterns. In addition, a computation-based algorithm is designed to identify potential meta-pattern candidates. We use the meta-pattern mining algorithm on some real traces and some very interesting results are discovered.

References

Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In Proceeding of 20th international conference on very large data bases, pp 487–499

Agrawal R, Srikant R (1995) Mining sequential patterns. In Proceedings of international conference on data engineering (ICDE), pp 3–14

Agrawal R, Psaila G, Wimmers E, Zait M (1995) Querying shapes of histories. In Proceedings of 21st international conference on very large data bases, pp 502–514

Berger G, Tuzhilin A (1998) Discovering unexpected patterns in temporal data using temporal logic. Temporal databases: research and practice. Lecture notes on computer sciences 1399, pp 281–309

Berndt D, Clifford J (1996) Finding patterns in time series: a dynamic programming approach. Advances in Knowledge Discovery and Data Mining, pp 229–248

Bettini C, Wang X, Jajodia S, Lin J (1998) Discovering frequent event patterns with multiple granularities in time sequences. IEEE Transactions on Knowledge and Data Engineering, 10(2):222–237

Chakrabarti S, Sarawagi S, Dom B (1998) Mining surprising patterns using temporal description length. In Proceedings of international conference on very large data bases, pp 606–617

Chan K, Fu A (1999) Efficient time series matching by wavelets. In Proceedings of 15th international conference on data engineering, pp 126–133

Crovella M, Bestavros A (1996) Self-similarity in world wide web traffic: evidence and possible causes, SIGMETRICS 160–169

Das G, Lin K, Mannila H, Renganathan G, Smyth P (1998) Rule discovery from time series. In Proceedings of international conference on knowledge discovery and datamining, pp 16–22

Feldman R, Aumann Y, Amir A, Mannila H. Efficient algorithms for discovering frequent sets in incremental databases. In Proceedings of ACM SIGMOD workshop on research issues on data mining and knowledge discovery (DMKD), pp 59–66

Garofalakis M, Rastogi R, Shim K (1999) SPIRIT: sequential pattern mining with regular expression constraints. In Proceedings of international conference on very large data bases (VLDB), pp 223–234

Guralnik V, Wijesekera D, Srivastava J (1998) Pattern directed mining of sequence data. In Proceedings of ACM SIGKDD, pp 51–57

Guralnik V, Srivastava J (1999) Event detection from time series data. In Proceedings of ACM SIGKDD, pp 33–42

Han J, Gong W, and Yin Y (1998) Mining segment-wise periodic patterns in time-related databases. In Proceedings of international conference on knowledge discovery and data mining, pp 214–218

Han J, Dong G, Yin Y (1999) Efficient mining partial periodic patterns in time series database. In Proceedings of international conference on data engineering, pp 106–115

Han J, Pei J, Mortazavi-Asl B, Chen Q, Dayal U, Hsu M (2000) FreeSpan: frequent pattern-projected sequential pattern mining. In Proceedings of international conference on knowledge discovery and data mining, pp 355–359.

Jagadish H, Koudas N, Muthukrishnan S (1999) Mining deviants in a time series database. In Proceedings of 25th international conference on very large data bases (VLDB), pp 102–113

Keogh E, Smyth P (1997) A probabilistic approach to fast pattern matching in time series databases. In Proceedings of international conference on knowledge discovery and datamining, pp 24–30

Korn F, Jagadish H, and Faloutsos C (1997) Efficiently supporting Ad Hoc queries in large datasets of time sequences. In Proceedings of ACM conference on management of data (SIGMOD), pp 289–300

Lin L, Risch T (1998) Querying continuous time sequences. In Proceedings of 24th international conference on very large data base (VLDB), pp 170–181

Mannila H, Toivonen H, Verkamo A (1997) Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery, 1(3):259–289

Mannila H, Meek C (2000) Global partial orders from sequential data. In Proceedings of ACM SIGKDD, pp 161–168

Oates T (1999) Identifying distinctive subsequences in multivariate time series by clustering. In Proceedings of ACM SIGKDD, pp 322–326

Ozden B, Ramaswamy S, Silberschatz A (1998) Cyclic association rules. In Proceedings of 14th international conference on data engineering, pp 412–421

Padmanabhan B, Tuzhilin A (1996) Pattern discovery in temporal databases: a temporal logic approach. In Proceedings of ACM KDD, pp 351–354

Padmanabhan B, Tuzhilin A (1998) A belief-driven method for discovering unexpected patterns. In Proceedings of ACM KDD, pp 94–100

Qu Y, Wang C, Wang X (1998) Supporting fast search in time series for movement patterns in multiple scales. In Proceedings of 7th ACM internatinal conference on information and knowledge management, pp 251–258

Rafiei D (1999) On similarity-based queries for time series data. In Proceedings of 15th international conference on data engineering, pp 410–417
Ramaswamy S, Mahajan S, Silberschatz A (1998) On the discovery of interesting patterns in association rules. In Proceedings of 24th international conference on very large data bases (VLDB), pp 368–379
Sheng M, Hellerstein L (2001) Mining partially periodic event patterns with unknown periods. In Proceedings of 17th IEEE international conference on data engineering, pp 205–214
Spiliopoulou M (1999) Managing interesting rules in sequence mining. In Proceedings of european conference on principles and practice of knowledge discovery in databases, pp 554–560
Srikant R, Agrawal R (1996) Mining sequential patterns: generalizations and performance improvements. In Proceedings of 5th international conference on extending database technology (EDBT), pp 3–17
Thomas S, Sarawagi S (1998) Mining generalized association rules and sequential patterns using SQL queries. In Proceedings of 4th international conference on knowledge discovery and data mining (KDD98), pp 344–348
Thompson K, Miller G, Wilder R (1997) Wide-area Internet traffic patterns and characteristics. IEEE Network 11(6):10–23
Wetprasit R, Sattar A (1998) Temporal Reasoning with qualitative and quantitative information about points and durations. In Proceedings of 15th national conference on artificial intelligence (AAAI-98), pp 656–663
Yang J, Wang W, Yu P (2000) Mining asynchronous periodic patterns in time series data. In Proceedings of ACM SIGKDD international conference on knowledge discovery and data mining (SIGKDD), pp 275–279
Zaki M (2000) Sequence mining in categorical domains: incorporating constraints. In Proceedings of 9th international conference on information and knowledge management, pp 422–429
Zaki M (2001) SPADE: an efficient algorithm for mining frequent sequences. Machine Learning Journal (special issue on unsupervised learning) 42(1/2):31–60

Author Biographies

Jiong Yang is a visiting assistant professor at the Computer Science Department, UIUC. His current research interests include data mining, bio-informatics, Internet traffic engineering, Internet content retrieval, and multimedia systems over the Internet. Dr Yang received the B.S. degree from the University of California at Berkeley in 1994, and the M.S. and Ph.D. degrees in Computer Science from the University of California, Los Angeles in 1996 and 1999, respectively. He is the author of more than 30 research papers.

Wei Wang is currently an assistant professor in the Computer Science Department at the University of North Carolina, Chapel Hill. Her current research interests include data mining, database systems, and bio-informatics. Dr Wang received the M.S. degree from the State University of New York at Binghamton in 1995, and the Ph.D. degree in Computer Science from the University of California, Los Angeles, in 1999. She is a member of the editorial board of the *Journal of Data Management* and has published more than 40 research papers.

Philip S. Yu received the B.S. degree in E.E. from National Taiwan University, the M.S. and Ph.D. degrees in E.E. from Stanford University, and the M.B.A. degree from New York University. He is with the IBM Thomas J. Watson Research Center and currently manager of the Software Tools and Techniques group. His research interests include data mining, bioinformatics, Internet applications and technologies, database systems, multimedia systems, parallel and distributed processing, performance modeling and workload analysis. Dr Yu has published more than 320 papers in refereed journals and conferences. He holds or has applied for 254 US patents. Dr Yu is a Fellow of the ACM and a Fellow of the IEEE. He is the Editor-in-Chief of *IEEE Transactions on Knowledge and Data Engineering*. He is also an associate editor of *ACM Transactions on Internet Technology* and that of *Knowledge and Information Systems*. He is a member of the IEEE Data Engineering steering committee and is also on the steering committee of IEEE Conference on Data Mining. He was an editor and advisory board member of *IEEE Transactions on Knowledge and Data Engineering* and also a guest co-editor of the special issue on mining of databases. In addition to serving as program committee member on various conferences, he was the program co-chair of the 11th International Conference on Data Engineering and the program chairs of the 2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing, the PAKDD Workshop on Knowledge Discovery from Advanced Databases, and the 2nd international Workshop on Advanced Issues of E-Commerce and Web-based Information Systems. He served as the general chair of the 14th International Conference on Data Engineering and is currently serving as the general co-chair of the 2nd IEEE International Conference on Data Mining. He has received several IBM and external honors including Best Paper Award, two IBM Outstanding Innovation Awards, an Outstanding Technical Achievement Award, two Research Division Awards and the 70th plateau of Invention Achievement Awards. He also received an IEEE Region 1 Award for ‘promoting and perpetuating numerous new electrical engineering concepts’ in 1999. Dr Yu is an IBM Master Inventor and was recognized as one of the IBM’s ten top leading inventors in 1999.

Correspondence and offprint requests to: Jiong Yang, Computer Science Department, UIUC, Urbana, IL 61801, USA. Email: jioyang@cs.uiuc.edu