New Generation Computing, 26(2008)75-96
Ohmsha, Ltd. and Springer

Mining Sequential Patterns across Time Sequences

Gong CHEN
Department of Statistics, University of California, Los Angeles,
8951 Mathematical Sciences Building, Los Angeles, CA 90095, USA
gchen@stat.ucla.edu
Xindong WU
Department of Computer Science, University of Vermont,
33 Colchester Avenue, Burlington, VT 05405, USA
xwu@cs.uvm.edu
Xingquan ZHU
Department of Computer Science and Engineering,
Florida Atlantic University, Boca Raton, FL 33431, USA
xqzhu@cse.fau.edu

Received 07 February 2006
Revised manuscript received 06 February 2007

Abstract In this paper, we deal with mining sequential patterns in multiple time sequences. Building on a state-of-the-art sequential pattern mining algorithm PrefixSpan for mining transaction databases, we propose MILE (MIning in muLtiple sEquences), an efficient algorithm to facilitate the mining process. MILE recursively utilizes the knowledge of existing patterns to avoid redundant data scanning, and therefore can effectively speed up the new patterns' discovery process. Another unique feature of MILE is that it can incorporate prior knowledge of the data distribution in time sequences into the mining process to further improve the performance. Extensive empirical results show that MILE is significantly faster than PrefixSpan. As MILE consumes more memory than PrefixSpan, we also present a solution to trade time efficiency in memory constrained environments.

Keywords: Data Mining, Sequential Patterns, Time Sequences.

§1 Introduction

Many real-world applications involve time sequences. Examples include data flows in medical ICU (Intensive Care Units), network traffic data and stock exchange rates. Discovering structures of interest in multiple time sequences

is an important problem. For example, the knowledge from time sequences in ICU (such as the oxygen saturation, chest volume and heart rate) may indicate or predicate a patient's situation, and an intelligent agent with the ability to discover knowledge in the data from multiple sensors can automatically acquire and update its environment model.[11)]

In this paper, we deal with categorical data consisting of tokens. A token stands for an event at a certain abstraction level, for example, a steady state of heart rate or a rising in stock price. We are interested in knowledge in the form of frequent sequential patterns across time sequences. Such a pattern can look like "the price of Sun stock and the price of IBM stock go up at the same time, and within two days Microsoft stock's price goes down and one day later Intel stock's falls as well." Mining such a sequential pattern across multiple time sequences (e.g., the stock prices of different companies) is a more challenging task than previous studies on rule discovery. The challenges come from the following two aspects. (1) In sequential pattern mining, there are many candidates across multiple sequences. A permutation of any subset of tokens in a time sequence can possibly be a sequential pattern from that time sequence and multiple time sequences complicate the situation even further. (2) The occurrence of a sequential pattern complicates the mining procedure even if the order of the pattern literals is the same. That is, a matching instance of a pattern can occur with noisy tokens at different time points, which makes it hard to count the number of patterns' occurrences. Practical and efficient solutions are needed to generate frequent patterns and count their occurrences for multiple sequences.

The contributions of this paper are as follows.

- We define the problem of discovering structures of interest in multiple sequences of data as mining sequential patterns across time sequences.
- We design an efficient algorithm MILE to solve this problem based on PrefixSpan, an existing sequential pattern mining algorithm. Extensive empirical results show that MILE is significantly faster than PrefixSpan.
- One unique feature of MILE is that it can incorporate prior knowledge of the data distribution in the sequences into the mining process to further improve the efficiency.
- We also propose a solution to balance the memory usage and time efficiency in memory limited environments.

The remainder of the paper is organized as follows. Our sequential pattern mining problem is formally defined in Section 2. In Section 3 we describe the design of our MILE algorithm. Empirical comparisons are presented in Section 4. Section 5 reviews related work and discusses the difference between our problem and previous studies. In Section 6 we discuss the effects of different window models in the problem settings. Finally, we conclude our work in Section 7.

§2 Problem Statement

We assume that time sequences consist of categorical data and real-valued data have been discretized. Each data entry is a token associated with a time

point, standing for an event happening at that time. We only consider patterns that span no more than a constant number, w, of consecutive time points, i.e., each pattern occurs within a time window of width w. We employ *consecutive* windows where a window of data is *non-overlapping* with other windows of data and each window *immediately follows* its previous window. We are interested in a pattern if the number of its occurrences is more than a threshold *minSup*. Both the *minSup* and w are user-specified parameters. We use parentheses to group pattern literals at the same time point and enclose the entire pattern with braces. Consider the following example (where a dot denotes an anonymous event) with 3 time sequences and 12 time points. If $minSup = 2$ and $w = 4$, we can find the pattern {(33 22 *)(* * 11)}.

	1	2	3	4	5	6	7	8	9	10	11	12
s^3	33	.	.	.	.	33	.	.	.	33	.	.
s^2	22	.	.	.	.	22	.	.	.	22	.	.
s^1	.	.	11	.	.	.	11	.	.	.	.	11

We call pattern literals at the same time point $(p^V ... p^k ... p^2 p^1)$ an **intra-pattern** where V is the number of time sequences. p^k either matches an event in sequence S^k or is a wild card which can match any event. A **pattern** consists of intra-patterns. The maximum number of intra-patterns in a pattern cannot be greater than w. Any two intra-patterns in a pattern cannot occur at the same time point in the sequences. Intra-patterns in a pattern have a *flexible temporal order* between them. In the above example, the pattern {(33 22 *)(* * 11)} requires intra-pattern (* * 11) to appear after intra-pattern (33 22 *). But (* * 11) can happen either immediately after (33 22 *) or several time points later within the same window. Our problem is formally defined in Definition 2.1.

Definition 2.1
Given

- a set of sequences $\{S^1, S^2, ..., S^k, ..., S^V\}$ where $S^k = S_1^k S_2^k ... S_q^k ... S_m^k$ is a sequence of events in which S_q^k is an event in the set of events for sequence S^k and occurs at the time point q, V is the number of sequences, and m is the total number of time points,
- the width of a consecutive time window w, and
- the threshold value *minSup*,

a complete set of patterns satisfying the following conditions are discovered:

- each pattern is in the form of the concatenation of intra-patterns;
- an intra-pattern $(p_i^V ... p_i^k ... p_i^2 p_i^1)$ consists of pattern literals at the same time point where $i \in [0, w-1]$ and p_i^k is either a literal matching an event in S^k or a wild card * matching any event;
- for any two literals from different intra-patterns in a pattern, $p_{i_{j_1}}^{k_1}$ and $p_{i_{j_2}}^{k_2}$ $(1 \le k_1 \le V, 1 \le k_2 \le V, i_{j_1} < i_{j_2})$, $S_{t+i'_{j_1}}^{k_1}$ and $S_{t+i'_{j_2}}^{k_2}$, the corresponding

matching events ($S^{k_1}_{t+i'_{j_1}} = p^{k_1}_{i_{j_1}}$ and $S^{k_2}_{t+i'_{j_2}} = p^{k_2}_{i_{j_2}}$) for a time point t, should preserve the *temporal* condition $i'_{j_1} < i'_{j_2}$ and should satisfy the condition that $i'_{j_2} - i'_{j_1} < w$;

- the number of each pattern's occurrences in the set of time sequences is greater than *minSup*.

To be concise, we ignore wildcards in a pattern in the following description. For example, we use {(33 22)(11)} instead of {(33 22 *)(* * 11)}. Since we can always encode tokens in such a way that different sequences have different sets of tokens, this representation causes no confusion.

Notations For an arbitrary pattern P = $\alpha\tilde{t}\beta$ where α and β are sub-patterns of P and $\tilde{t}$ is a token in P, we define $suffix(\tilde{t}) = \beta$, and $prefix(\tilde{t}) = \alpha$. Assuming that there are n patterns having $\alpha\tilde{t}$ as a prefix: $\alpha\tilde{t}\beta_1$, $\alpha\tilde{t}\beta_2$, ..., $\alpha\tilde{t}\beta_n$, we define $suffixes(\tilde{t}) = \cup_{i=1}^{n}\{\beta_i\}$ for the prefix α. $suffixes(\tilde{t})$ should share the same prefix, although we may not explicitly show it. For example, assuming two patterns {(33 20 10)(22 15)(32 21 11)} and {(33 20 10)(22 16)(34 25 11)}, $suffixes(22)$ = {(_ 15)(32 21 11), (_ 16)(34 25 11)} for the shared prefix (33 20 10). Assuming we have $suffixes(\tilde{t}_1)$, $suffixes(\tilde{t}_2)$,..., $suffixes(\tilde{t}_n)$ for the shared prefix α, we define $suffixesSet(t)$ where t is the last token of α in the form of {$\tilde{t}_1$:$suffixes(\tilde{t}_1)$; $\tilde{t}_2$: $suffixes(\tilde{t}_2)$;...; $\tilde{t}_n$:$suffixes(\tilde{t}_n)$}. $suffixesSet(t)$ should share some prefix α. For example, if we have two more patterns {(33 20 10)(21 18)(32 27 11)} and {(33 20 10)(21 19)(34 25 11)}, - $suffixesSet(10)$={21:{(_ 18)(32 27 11), (_ 19)(34 25 11)}; 22:{(_ 15)(32 21 11), (_ 16)(34 25 11)}} for the prefix (33 20 10).

§3 Algorithm Description

3.1 Description of PrefixSpan

Since our algorithm builds upon an existing algorithm PrefixSpan (PseudoProjection), [13] we first outline its basic steps that can be applied to solve our problem.

1. Scan time sequences to locate tokens whose frequency is greater than the *minSup*, and output them (each of which is a frequent pattern with a single value). If no frequent token exists, return.
2. For each pattern a, from each of its ending locations (the time point when the last token occurs): scan time sequences at the same window to locate token b whose frequency is greater than the *minSup*; append b to a; output ab; let $a = ab$, and repeat step 2. If no frequent token exists, return.

At the same time point, a token in a lower-numbered sequence is scanned after tokens in sequences with higher numbers; and at different time points in the same sequence, a token at a later time point is scanned after earlier tokens. This is a systematic way to scan tokens into PrefixSpan and MILE, and the behavior of PrefixSpan and MILE does not depend on the order in which streams are

scanned.

Let us apply PrefixSpan to the following example which has 3 time sequences and 11 time points. $w = 3$ and $minSup = 2$. According to the parameters, we have 4 windows of data (the last window has only two time points of data), we want to find every sequential pattern that appears in at least 3 windows.

	1	2	3	4	5	6	7	8	9	10	11
s^3	33	32	39	33	31	38	33	30	35	36	37
s^2	21	22	23	24	22	25	26	22	27	28	29
s^1	10	12	11	13	14	11	15	16	11	17	18

Scan data once, and 11, 22 and 33 are found to be frequent patterns with a single value. {(11)}, {(22)} and {(33)} are output. Scan data after {(11)}, no frequent token is found since there is no data after {(11)} at the same window. Scan data after {(22)}, 11 is found to be frequent so {(22)(11)} is output. Scan data after {(22)(11)}, no frequent token is found. Scan data after {(33)}, 11 and 22 are found to be frequent so {(33)(11)} and {(33)(22)} are output. Scan data after {(33)(11)}, no frequent token is found. Scan data after {(33)(22)}, 11 is found to be frequent so {(33)(22)(11)} is output. Scan data after {(33)(22)(11)}, no frequent token is found.

3.2 Description of MILE

The above example demonstrates that when PrefixSpan mines {(33)(22)}, {(22)(11)} has been mined out at the previous stage as a pattern with 22 as prefix. Can we append this pattern directly to {(33)(22)} to form the pattern {(33)(22)(11)} without scanning the data after {(33)(22)}? In more general cases, can we append some mined patterns with b as prefix to pattern cb to form all the patterns with cb as prefix without scanning the data after cb? If this is possible, we can avoid scanning the data over and again and speed up the mining process. In the above example, the data after 22 has been scanned twice: one is for mining patterns with 22 as prefix, and the other is for mining patterns with (33)(22) as prefix. Another advantage is when a very long pattern β with b as prefix is mined out and β also appears after cb, we will get long patterns with cb as prefix directly by appending β to b rather than recursively scanning the data after cb. How can we recursively utilize the knowledge from mined patterns to speed up the mining process? We describe below our algorithm MILE to manage this process efficiently.

We explain the mining process of MILE with a part of a pattern tree in Fig. 2. One concatenation of literals on edges from the root to any node forms a pattern. For example, {11} is a pattern and so are {11 44}, {11 44 β_1} and {33 22 11 55 44 β_2}. Here we can ignore the parentheses in patterns to understand the main idea of MILE smoothly. We use β_i to denote a suffix of a pattern which contains tokens on the corresponding edge. Similarly, we use "55 44 β_i" to label the edge which denotes the concatenation of tokens 55, 44 and the suffix β_i. From the description of PrefixSpan, we can see that it performs

```
MILE(){
1  token t = ();
2  t.endLoc←start time points of every window;
3  suffixesSet(t) = ();
4  index idx = ();
5  pattern set←PrefixExtend(t, suffixesSet(t), idx);
}

PrefixExtend(token t, suffixesSet s, index idx){
1  index nIdx = ();
2  suffixesSet(t) = ();
3  for e in t.endLoc
4    /**scanning process**/
5    scan from e to the end of window starting at e,
     register locations for every token t̃ at t̃.endLoc,
     update the frequency for t̃ at t̃.freq;
6  for every token t̃ and if(t̃.freq>minSup)
7    if(suffixes(t̃) in s)
8      suffixesSet(t)←SuffixAppend(t̃, suffixes(t̃), idx);
9    else
10     suffixesSet(t)←PrefixExtend(t̃, suffixesSet(t), nIdx);
11   suffixes(t)←append t̃ to (_);
12   suffixes(t)←append suffixes(t̃) in suffixesSet(t) to (_ t̃);
13 return suffixes(t);
}

SuffixAppend(token t̃, suffixes s_t̃, index idx){
1  if(idx has no idx_t̃ for s_t̃)
2    /**building index**/
3    idx←build idx_t̃ for s_t̃ with information in s_t̃;
4  /**hitting process**/
5  Use every e in t̃.endLoc to hit idx_t̃,
   update frequency for a hitted suffix in s_t̃,
   register the hitted location for a hitted suffix;
6  /**choosing the desired suffixes**/
7  suffixes(t̃)←suffixes in s_t̃ whose frequency>minSup;
8  return suffixes(t̃);
}
```

Fig. 1 Pseudo code for MILE

a depth-first-search like discovery along this pattern tree. It mines patterns in the following order: ⟨ {11}, {22}, {33} ⟩ → ⟨ {11 44}, {11 55} ⟩ →...→{11 44 β_1}→...→{11 44 β_2}→...→{11 44 β_3}→...→{11 55 44 β_3}→{22 11}→...→{22 11 44 β_2}→...→{33 22 11 55 44 β_2}. A function PrefixExtend in MILE (Fig. 1) explores the pattern tree similarly. But when it comes to {11 55 44}, it finds that $suffixes(44)$ for prefix 11 has been mined, so it calls SuffixAppend to select the desired suffixes (which will be explained in the next paragraph) from $suffixes(44)$ and append them directly to {11 55 44} instead of performing a depth-first search to scan overlapping data in PrefixSpan (because all data after {11 55 44} must appear after {11 44}). Similarly, when it comes to {22 11}, it finds that $suffixes(11)$ for prefix {} has already been discovered so it calls

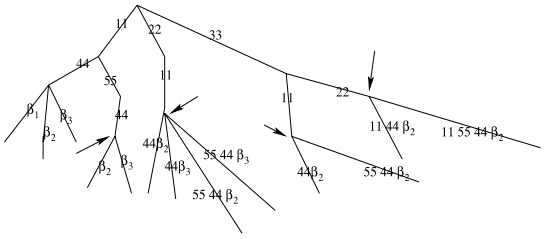


Fig. 2 Part of a pattern tree showing the mining process of MILE

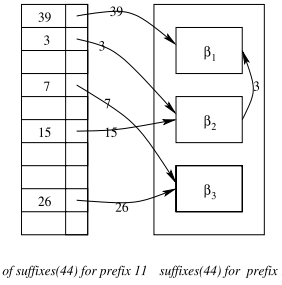


Fig. 3 Index of $suffixes(44)$ for prefix 11

SuffixAppend to select the desired suffixes from $suffixes(11)$ and append them to {22 11}. We use arrows to mark each place where SuffixAppend occurs in Fig.2. We can see that SuffixAppend is embedded in the mining process to make new patterns' discovery fast.

Now we describe the selection process of SuffixAppend. It has three steps which are commented in the pseudo code in Fig.1: building index, hitting process and choosing the desired suffixes. We also use one part of the pattern tree in Fig. 2 to show how these three steps work. Assuming MILE is currently running at point {11 55 44} with ending locations (or time points when 44 in this pattern occurs in the time sequences) (3, 7, 15, 26) and finds that $suffixes(44)$ for prefix 11 has been mined, it calls SuffixAppend. Ending locations are collected in the scanning process commented in the pseudo code of PrefixExtend: t.endLoc (on line 5 of PrefixExtend) contains the collected ending locations that are associated with token t.

Assume $minSup = 1$ and start locations (or time points when 44 in the corresponding suffixes occurs) of suffixes in $suffixes(44)$ for prefix 11 are as follows: (_ β_1): (3, 39); (_ β_2): (3, 15); (_ β_3): (7, 26). We assume that no index has been built for $suffixes(44)$, so the building process is started. To speed up the hitting process at a later stage, we use a hash table indexed by start locations of suffixes in $suffixes(44)$. Scan suffixes $suffixes(44)$ and their start locations once to insert each suffix into the corresponding bucket according to their start locations. Since (_ β_1) and (_ β_2) share the same start location 3, they are put into a linked list indexed by 3. The resulting hash table is shown in Fig. 3.

Now the hitting process begins. Every ending location of {11 55 44} is used as a key to search the hash table for the suffixes in $suffixes(44)$ for prefix 11 appearing after 11 55 44. When ending location *3* is used, the frequencies of (_ β_1) and (_ β_2) are increased by 1. When ending location *7* is used, the frequency of (_ β_3) is increased by 1. After all ending locations have been consumed, the choosing process will store every suffix whose frequency is greater than the *minSup* in $suffixes(44)$ for prefix {11 55} for possible future appending. Also, the selected suffixes will be appended to prefix {11 55 44} in PrefixExtend. In this case, (_ β_2) and (_ β_3) are selected for appending to prefix {11 55 44}, which

also can be seen from Fig. 2. Note that the constructed index for $suffixes(44)$ with 11 as prefix is stored for future use to avoid a repeated building process. For example, if we have a pattern {11 66 44}, then this index will be used again for appending suffixes to that pattern. This index will be dropped when all patterns with {11} as prefix are discovered. At this point, the readers might think that if no suffixes can be appended, this building process will be pure overhead. Actually, this is not true. If no suffixes can be appended, we only need to check ending locations of a prefix to decide whether there is any suffix to be appended if we have this index in hand. Otherwise, we need to scan the data during the scanning process in PrefixExtend to make the decision. In the case of relatively small numbers of ending locations, suffixes and their start locations, and a relatively large amount of data to be scanned, this indexing can still speed up the mining process even if no suffixes are appended. This will be demonstrated in the experimental results.

3.3 Optimization Techniques

[1] Incorporating prior knowledge

If some prior knowledge of the data distribution in time sequences is available, we can further improve the efficiency of the mining process based on our suffix appending approach. Assume that the users know in advance the frequency of one token's occurrence in some sequence is higher than others'. That means it will have more chance to get more suffixes appended if the mining process of patterns with this token as prefix can be delayed to a later stage. In this way, MILE will avoid more expensive depth-first search. The strategy we employ is to encode such a sequence with larger values and the largest value is assigned to the token with the highest frequency. We show this encoding strategy with the following example.

s^1	x	y	z	z	y	x	y	x	z
s^2	e	f	g	e	f	e	g	f	g
s^3	a	a	a	b	c	a	a	a	a

In s^3, token a occurs more frequently than the other two tokens (and tokens in the other time sequences are random). So we encode a with the largest value 33 and the sequences as follows.

s^3	33	33	33	32	31	33	33	33	33
s^2	20	21	22	20	21	20	22	21	22
s^1	10	11	12	12	11	10	11	10	12

In PrefixExtend, we can control MILE in the way that patterns with a smaller value as prefix are mined earlier than the ones with a larger value as prefix. It is understandable that subtrees starting with smaller values are searched first in the pattern tree and those subtrees with larger values will use SuffixAppend to explore instead of depth-first search. In general cases, we assign

higher encoding values to the tokens of higher frequencies in one data sequence and assign a higher encoding value to the sequence that contains the token of the highest frequency. Empirical results in Section 4 show that this heuristic can further improve the performance of MILE.

[2] Balancing memory usage and performance

MILE uses more memory than PrefixSpan since it records down previously mined suffixes and builds corresponding indexes if needed. With advances in computer engineering, the sizes of main memory for computers are growing fast and the price of memory is cheap. Several gigabytes are simply normal with a regular computing server. If the users are more concerned with time efficiency, MILE is clearly a good choice. If the users are also concerned with the memory a data mining system consumes, we now describe a solution to balance the memory usage and time performance of MILE.

In a normal situation, the number of shorter patterns is greater than the number of longer patterns, and the locations (frequencies) of shorter patterns are much more than the locations (frequencies) of relatively longer patterns. Similar situations exist for mined suffixes. If MILE only records down and builds indexes for mined suffixes whose length exceeds a predefined parameter l, and uses PrefixExtend to grow shorter patterns which will not be mined by SuffixAppend due to unrecorded short suffixes, it will use less memory than the original algorithm. But the efficiency will degrade at the same time. For example, if the predefined parameter $l = 1$, suffix $\{\_\ 44\}$ for prefix 11 will not be recorded down in the pattern tree in Fig. 2, but $\{\_\ 44\ \beta_1\}$ will (assuming that β_1 contains at least one token). Since the information about suffix $\{\_\ 44\}$ for prefix 11 is not available at a later stage, patterns {22 11 44} and {33 11 44} will be mined in PrefixExtend rather than in SuffixAppend in the original design. Empirical results in Section 4 show that this solution can significantly save memory and, in the meanwhile, can maintain a reasonable efficiency. After all, we can see that the longer suffixes are appended, the more benefits the mining process gets from our suffix appending approach. MILE still works well without using relatively short suffixes.

§4 Experimental Evaluation

In this section, we compare PrefixSpan and MILE with data sets under different parameter settings. We also analyze those factors that impact the performance of MILE.

Experiment Environment All experiments were conducted on a server with four 1GHz SPARC CPUs and 8 gigabyte memory. The operating system is Solaris 9. We implemented MILE and PrefixSpan[13] in Java. The JVM version is 1.5.0_01-b08. All outputs are turned off.

Data Generation We generate data sets with uniform distribution and also multinomial distribution with specified probabilities. Unless explicitly explained otherwise, the data are uniformly distributed. Three parameters are used in the name of each data set to indicate the data set's settings: s denotes

the number of sequences, t the number of time points, and v the number of different tokens per sequence. For example, *s3t200v3* means that the data set has 3 sequences, 200 time points, and each sequence has 3 different tokens. The window size is set to 4 as default. All experimental results are averaged values over 10 repetitions.

Performance Comparisons with Varying Time Points and Window Sizes First we compare the performance of PrefixSpan and MILE on small (s9t200v4), medium (s9t2000v4) and large (s9t20000v4) data sets with a fixed window size of 4 and various *minSup* values. Here we use relative values. For example, if there are 50 windows of data and $minSup = 50\%$, the frequency of a pattern is required to be greater than 25. Time sequences have more values in the time dimension than in other dimensions. So this group of comparisons reflects the normal situation. Figures 4, 5 and 6 show that MILE runs consistently faster than PrefixSpan. Note that when the *minSup* is increased, the number of patterns is decreased and the performance of MILE becomes similar to PrefixSpan. When the number of patterns is very small (for example, less than 10), MILE may be less efficient than PrefixSpan. However, when the *minSup* becomes less and less, the number of patterns becomes more and more and the performance of MILE is consistently much better than PrefixSpan. In the largest data set, MILE can achieve a 46.01% improvement (by (PrefixSpan's CPU time-MILE's CPU time) / PrefixSpan's CPU time, which is denoted by (Pt-Mt) / Pt hereafter) over PrefixSpan when $minSup = 7\%$. When the *minSup* is small, the number of patterns becomes very large and much more computation is involved than when the *minSup* is large. It is at that point the difference between MILE and PrefixSpan becomes important. When we vary the window size and fix the other factors, Fig. 7 shows the consistent performance of MILE.

Performance on A Dense Data Set This experiment shows that MILE's performance is not dependent on whether the given data set is dense or not. A dense data set presents a good number of patterns even when the *minSup* is very high. We notice that the data set *s9t2000v4* is a sparse data set since patterns can only be found when the *minSup* is low. We generate a dense data set *s9t2000v4d* as follows: with four unique tokens (1,2,3,4), we first form a sequence by repeating a longest sequential pattern (1234) (when the window size is 4) 500 times and replicate this sequence 9 times to form 9 sequences[*1]; then randomness is injected into these sequences in the way of a token mutating into another token with probability 0.02 and being the same token with probability 0.94. We can see that numerous patterns are formed in this way—even when the *minSup* is 76%, 218,799 patterns are found. Figure 8 shows that MILE runs consistently faster than PrefixSpan. The performance difference becomes larger when the *minSup* is decreased. This result demonstrates that whether a data set is dense or not does not affect MILE's performance.

The Relationship between Efficiency and the Suffix Appending Approach Intuitively, the larger the number of patterns formed by suffix

[*1] Stream ID is concatenated to the token value. For example, sequence 1 has tokens {11,12,13,14} and sequence 9 has tokens {91,92,93,94}

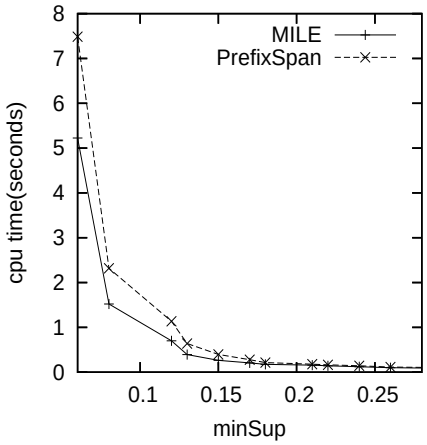


Fig. 4 CPU time comparison when minSup is varied, data set s9t200v4

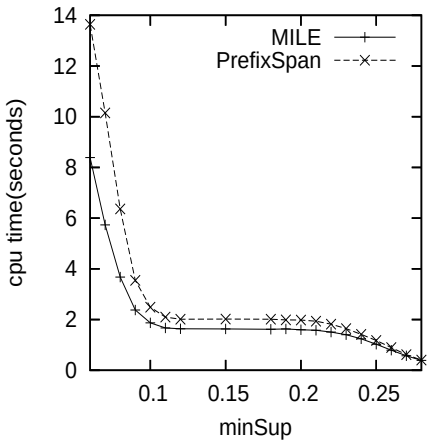


Fig. 5 CPU time comparison when minSup is varied, data set s9t2000v4

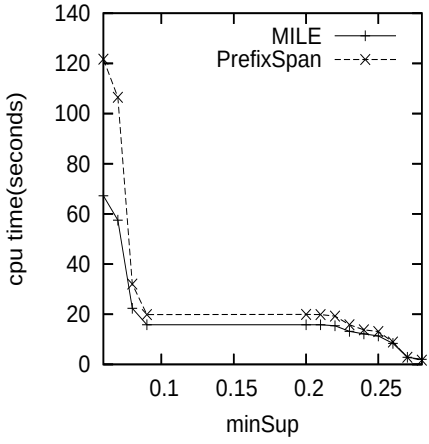


Fig. 6 CPU time comparison when the minSup is varied, data set s9t20000v4

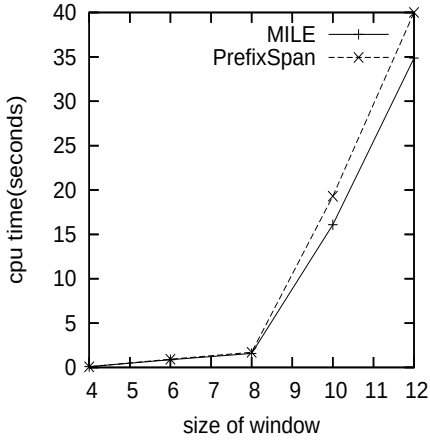


Fig. 7 CPU time comparison when window size is varied, data set s6t2000v6 and minSup=20%

appending, the faster MILE runs in comparison with PrefixSpan. Figure 9 illustrates this by putting two ratios together: one is (Pt-Mt)/Pt (explained in the last paragraph) standing for the efficiency of MILE; the other is Sn/Tn which is the ratio of the number of patterns formed by suffix appending over the number of all patterns. From this figure, we see two points. First, the trends of the two curves show that when suffix appending occurs more frequently, the mining process will be faster. Second, even if no suffix appending happens (Sn/Tn=0), the constructed index is not just pure overhead and can actually speed up the mining process as explained in Section 3.2.

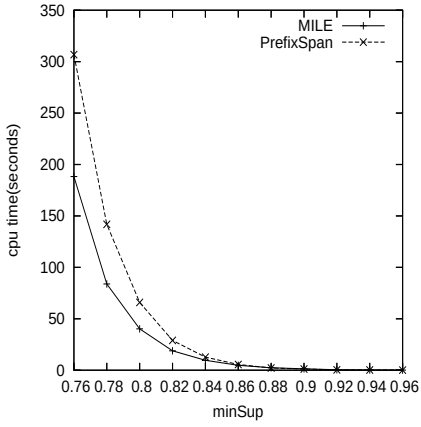


Fig. 8 CPU time comparison when minSup is varied, a dense data set s9t2000v4d

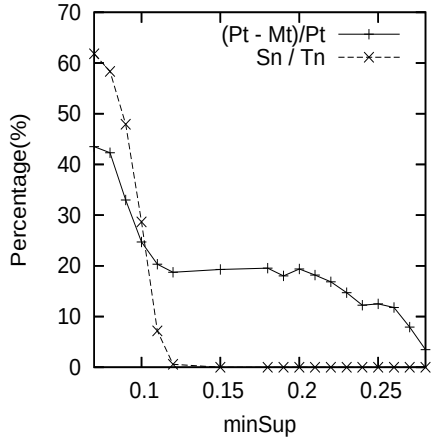


Fig. 9 Relationship between efficiency and the number of patterns formed by suffix appending, data set s9t2000v4

Incorporating Prior Data Distributions Knowledge Figure 10 demonstrates the performance of MILE when some prior knowledge about data distributions is incorporated into the mining process as described in Section 3.3. We generate data sets in such a way that (1) data set *Mult1* has one sequence containing a token (with a probability of 0.55) that happens more frequently than others (each of which is associated with a probability of 0.15); (2) data set *Mult2* has two sequences each of which contains a token (with a probability of 0.55) that happens more frequently than others (each of which is associated with a probability of 0.15); and (3) data set *Mult3* has two sequences each of which contains a token (with a probability of 0.75) that happens more frequently than others (each of which is associated with a probability of 0.05). From Fig. 10, we can see that the performance of MILE in these three data sets is in the order of *Mult3*>*Mult2*>*Mult1*. This result shows that when prior knowledge of data distributions is available, we can use the encoding mechanism in Section 3.3 to get more benefits from our suffix appending approach. From Fig. 11 the performance is consistent with the ratio Sn/Tn (which indicates how often suffix appending happens). That is, Sn/Tn in these three data sets is in the order of *Mult3*>*Mult2*>*Mult1*.

Tokens in data set *Unif* are uniformly distributed. This type of data set is the base line. In this case, the average performance of MILE is minimized when no prior knowledge is incorporated. However, the discussion from the previous paragraphs in this section shows that MILE still consistently outperforms PrefixSpan when dealing with such kinds of data sets. Note that although in Fig. 11 the ratio Sn/Tn in data set *Unif* is sometimes greater than both *Mult2* and *Mult1* and is even close to the ratio Sn/Tn in *Mult3*, MILE's performance in *Unif* is the lowest (still better than PrefixSpan). Figures 12, 13, 14 and 15 show why this happens. In these four figures, statistics on suffixes of different

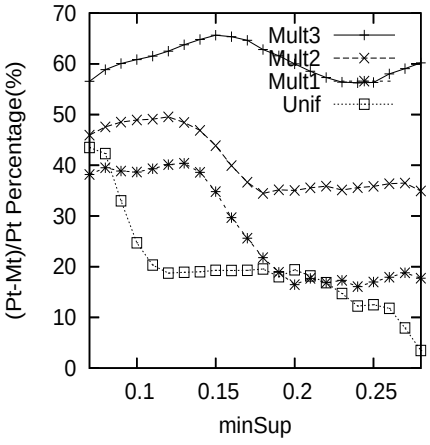


Fig. 10 Efficiency ((Pt-Mt)/Pt) of MILE under different data distributions, data set s9t2000v4 with different distributions

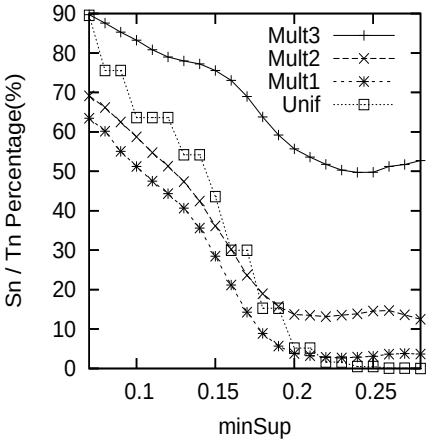


Fig. 11 The ratio Sn / Tn, data under different distributions, data set s9t2000v4

lengths were collected at the first level of a pattern tree (the suffixes with a single pattern literal as prefix). From these four figures, we can see that *Mult3*, *Mult2* and *Mult1* have more mined suffixes of longer lengths than *Unif*, which roughly indicates that more expensive depth-first search is avoided by our suffix appending approach. The longer the length of appended suffixes, the better the MILE's performance.

To further demonstrate the effectiveness of the encoding strategy that the most frequent token is encoded by the the largest value in Section 3.3, we compare it with two different encoding strategies with a data set of the same distribution *Mult1*: 1. the smallest value is used to encode the most frequent token so that patterns with that token as prefix will be mined first; and 2. a random value is used to encode the most frequent token so that patterns with that token as prefix will be mined in random order. Figure 16 shows that our encoding strategy, which allows patterns with the most frequent token as prefix to be mined last, is superior to the other two. Also, we can see that the reversed mining order is worse than the random order.

Performance on A Real World Data Set A real data set, cad, is used for performance comparison. The data are a subset of the JR-CAD data set collected by CENS (Center for Embedded Networked Sensing) at UCLA. It is the air temperature data recorded by a small scale of sensor networks in March 2006. Since the sensors are geographically close to each other, they tend to record correlated data at the same time range. 9 sensors' data form 9 time sequences with 8,928 data points in each. The sampling rate is one data point every 5 minutes. Since the original data are continuous, we discretize the data as follows: for each sequence, we calculate 5 quantiles min, 1st quartiles, median, 3rd quartiles, and max; they form 4 ranges; data points in different

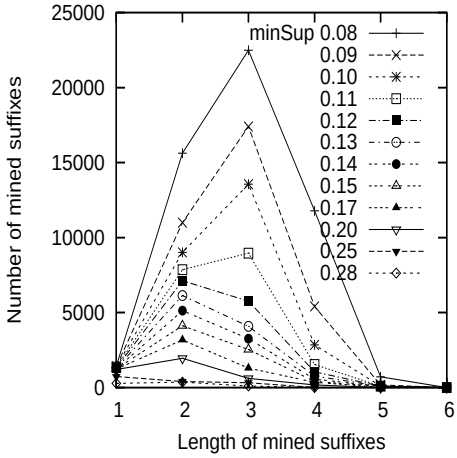


Fig. 12 Approximate distribution of lengths of mined suffixes (collected at the first level of a pattern tree), data set s9t2000v4, Mult3 distribution

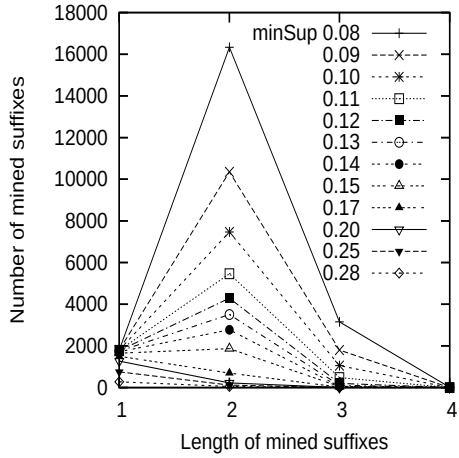


Fig. 13 Approximate distribution of lengths of mined suffixes (collected at the first level of a pattern tree), data set s9t2000v4, Mult2 distribution

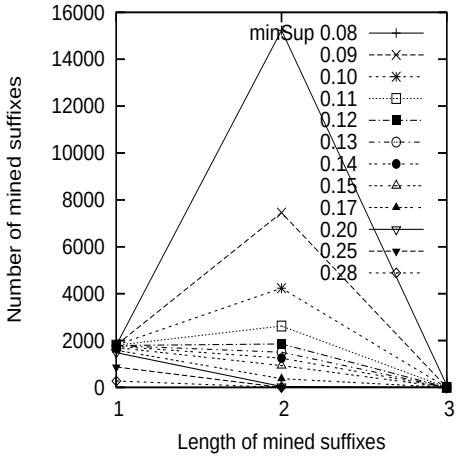


Fig. 14 Approximate distribution of lengths of mined suffixes (collected at the first level of a pattern tree), data set s9t2000v4, Mult1 distribution

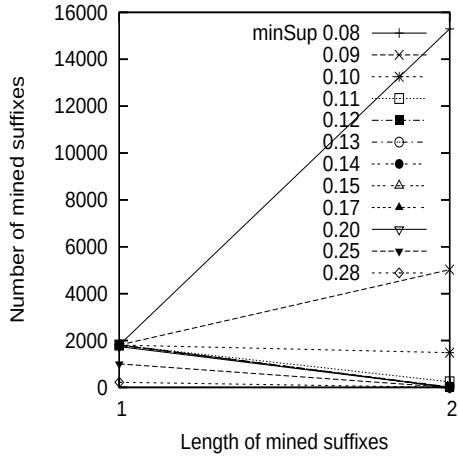


Fig. 15 Approximate distribution of lengths of mined suffixes (collected at the first level of a pattern tree), data set s9t2000v4, Unif distribution

ranges are assigned with different tokens. Therefore, each data sequence has 4 unique tokens. A half an hour time window is used. Figure 17 shows that MILE's performance is consistently superior to PrefixSpan when the *minSup* is less than 22% where a large amount of patterns exist (for example, 28,484 patterns are returned when the *minSup* is 20% and there are only 43 patterns returned when the *minSup* is 27%). As mentioned before, when the number of

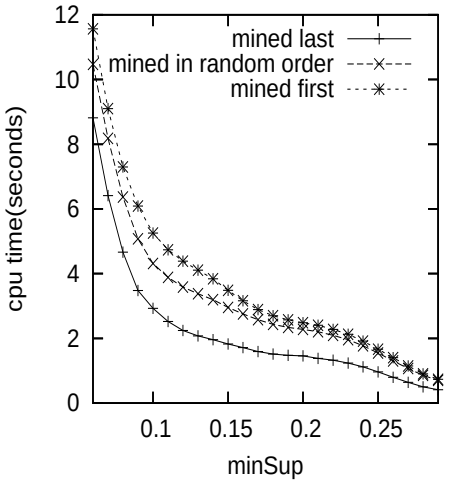


Fig. 16 CPU time comparison when patterns starting with the most frequent token are mined in three different orders, data set s9t2000v4, Mult1 distribution

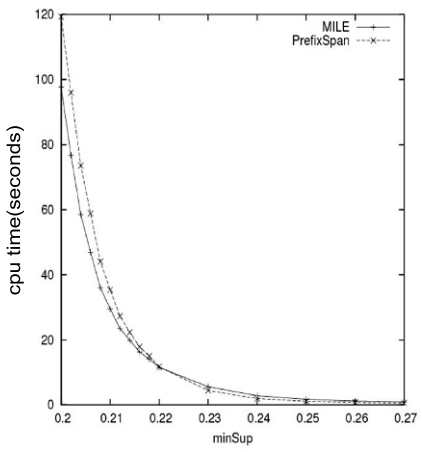


Fig. 17 CPU time comparison when the minSup is varied, data set cad

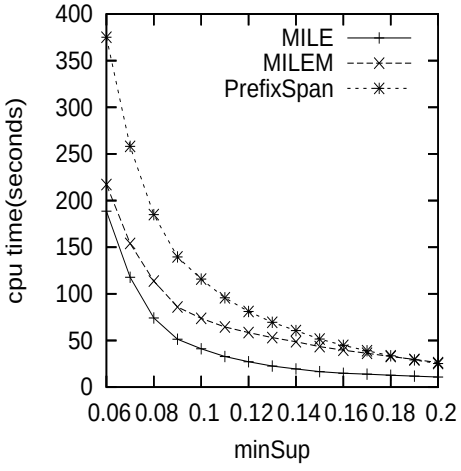


Fig. 18 CPU time comparison when minSup is varied, data set s15t2000v4, Mult3 distribution

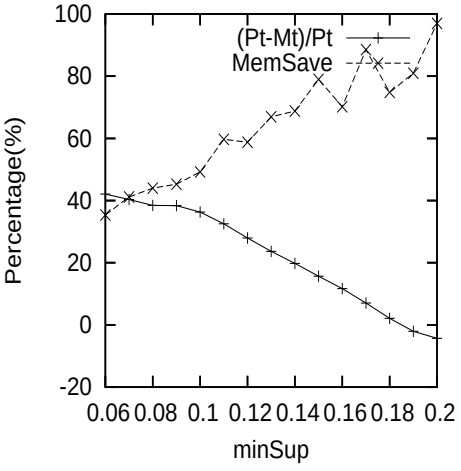


Fig. 19 CPU time vs. memory usage, data set s15t2000v4, Mult3 distribution

patterns is small, MILE's performance is less efficient than PrefixSpan due to the overhead of building index structures.

Balance between Memory Usage and Efficiency of MILE Figure 18 illustrates that the efficiency of our proposed solution in Section 2 when memory usage is of the concern of the users. If we need to save memory, MILE neither

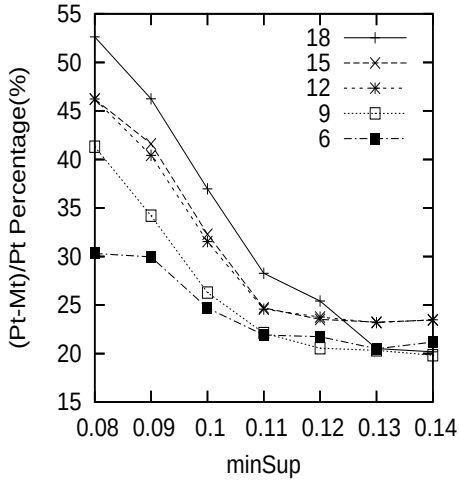


Fig. 20 Efficiency comparison of MILE when the number of sequences is varied, data set sXt2000v4 (X is the number of sequences labeled in the figure)

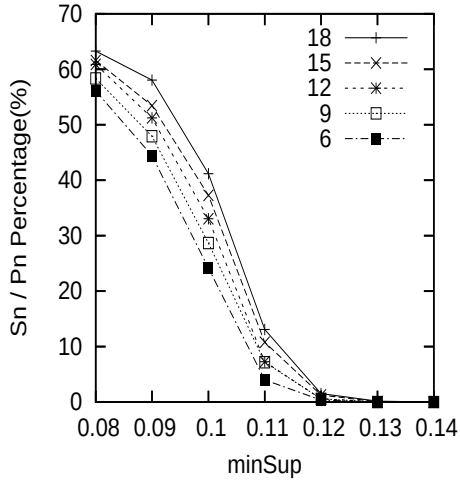


Fig. 21 The ratio Sn / Tn when the number of sequences is varied, sXt2000v4 (X is the number of sequences labeled in the figure)

records down short suffixes nor builds their corresponding location indexes. We use MILEM to denote this version of MILE. In Fig. 18, the information about suffixes shorter than 2 is not recorded. Since patterns are mostly short in the data set of uniform distribution and this distribution does not hold for most situations, we use multinomial distribution and various lengths of patterns to show the performance of the proposed memory saving solution. Figure 18 shows that the performance of MILE, MILEM and PrefixSpan is in the order of MILE > MILEM > PrefixSpan. Figure 19 compares the amount of memory saved by MILEM over MILE ((Memory used by MILE - Memory used by MILEM) / Memory used by MILE) and the efficiency of MILEM ((Pt-Mt)/Pt). We can see that in most cases MILEM can save a significant amount of memory while maintaining a reasonable efficiency. On average, it can save 64% memory over MILE and maintain a 21% improvement over PrefixSpan. When *minSup* is increased, the number of relatively long suffixes becomes less and the performance of MILEM degrades. However, usually the users are more interested in patterns across several time sequences to find correlations among them, and these patterns are relatively long in multiple time sequences like the distribution indicated by Fig. 12 rather than Fig. 15. So we can conclude that MILEM works well in a normal situation.

Performance Comparison with Varying the Number of Sequences Figure 20 shows the performance of MILE when the number of time sequences is increased. The results show that MILE runs consistently faster than PrefixSpan. Furthermore, the efficiency of MILE compared with PrefixSpan will become more significant when the number of sequences is increased. Actually, from the previous discussions, we can see that the performance of MILE is re-

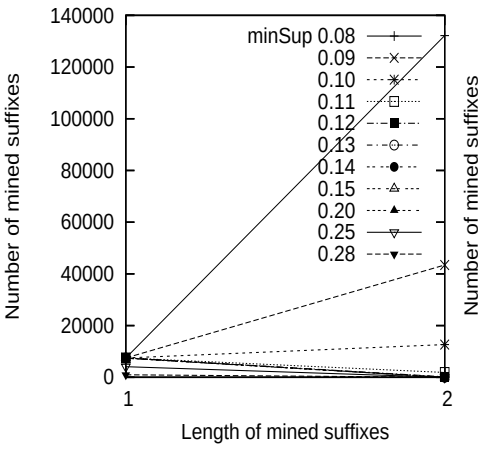


Fig. 22 Approximate distribution of lengths of mined suffixes (collected at the first level of a pattern tree), data set s18t2000v4

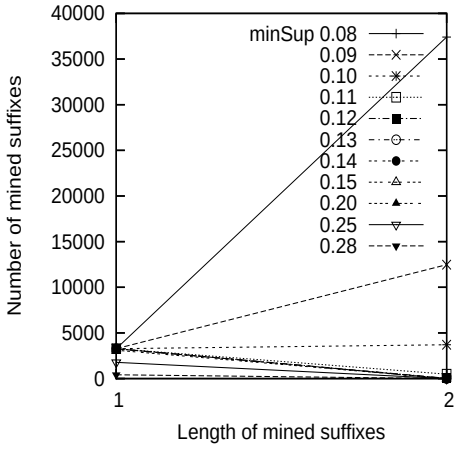


Fig. 23 Approximate distribution of lengths of mined suffixes (collected at the first level of a pattern tree), data set s12t2000v4

lated to the ratio of the number of patterns formed by suffix appending over the number of all patterns, and also related to the length of suffixes appended. For the second factor, we can see from Figs. 22, 23 and 24 that increasing the number of sequences has little impact on the length of suffixes appended. For the first factor, however, Fig. 21 illustrates that the ratio Sn/Tn is increased when the number of sequences is increased. This explains why MILE gains more improvement over PrefixSpan as the number of sequences becomes larger.

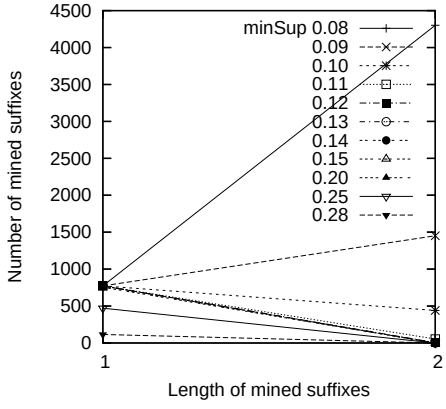


Fig. 24 Approximate distribution of lengths of mined suffixes (collected at the first level of a pattern tree), data set s6t2000v4

§5 Related Work

In this paper, we assume that real-valued data have been discretized into tokens. Yang et al.[16] reviewed various discretization methods. Keogh and Lin[7] presented a discussion on the effectiveness of an existing discretization method and introduced a novel approach for clustering time sequences.

Sequential pattern mining in transaction databases has been well studied in.[1, 2, 12, 14, 18 and 19] *PrefixSpan*[13] is an efficient sequential pattern mining algorithm. The merits of PrefixSpan come from the fact that it recursively projects the original data set into smaller and smaller subsets, from which patterns can be progressively mined out. PrefixSpan does not need to generate candidate patterns and identify their occurrences but grows patterns as long as the current item is frequent in the projected data set. This property makes PrefixSpan extremely efficient. However, when PrefixSpan recursively projects the original data set into overlapping subsets, it may scan the same part of the data over and again as analyzed in Section 3. This disadvantage can be overcome by our proposed approach, namely suffix appending (which is embedded in MILE).

One can see the semantic difference between sequential pattern mining in transaction databases and time sequences. For example, there might be no transactions, customer-ids and items purchased in time sequences. However, if we treat each time window of data as one customer's transactions (and each time point of the data as one transaction), then the problem of sequential pattern mining in time sequences can be generalized as sequential pattern mining in transaction databases and any sequential pattern mining algorithm can be used to solve the problem. This is why we can employ PrefixSpan to solve our problem and do fair comparisons between PrefixSpan and MILE. It is natural that the suffix appending approach we have proposed in MILE can also be adopted for sequential pattern mining in transaction databases, although MILE is designed to handle sequential pattern mining in time sequences.

Mannila et al.[10] studied the problem of mining frequent episodes. An episode consists of a set of events and a user-specified order over the events. Different sets of episodes are returned according to the order which can be parallel or serial. A parallel order specifies no temporal constraints on the set of events; whereby a serial order requires temporal constraints on each pair of events. The temporal order between each pair of intra-patterns in our problem (in Section 2) can be defined as a serial order. However, their work is to mine frequent episodes from a single sequence of events while our work aims to discover frequent patterns across multiple data sequences. Bettini et al.[3] considered the mining task where an event structure is given and some of its variables are instantiated. Possible instances for the other variables are discovered based on the frequency on which the corresponding events occur in an event sequence matching the structure. In their event structures the order of each pair of events is constrained by a different time granularity and some events in the structure are specified by the users in advance. These specific constraints reduce a large amount of computation. In contrast, our sequential pattern form involving no such specific user-defined constants is more general. This essentially makes our

problem more complex.

Das et al.[5] considered the problem of rule discovery from time sequences. A rule here is in the form of the occurrence of event A indicating the occurrence of event B within time T. We can treat this type of causal rule as a simplified sequential pattern of two events. A pattern in our problem involves an arbitrary number of events which make the problem much more complicated. Oates and Cohen[11] searched rules in the form of x indicating y within time δ where x is a set of events within a window and y is also a set of events within another window. However, in each of x and y, the order in which events happen is fixed. An x, for example, is like: after event A happens, *exactly* two time points later event B happens, and *exactly* three time points later event C happens. In our problem definition in Section 2, after event A happens, *within* two time points event B happens, and *within* three time points later event C happens. This more flexible temporal order makes our mining problem more challenging. Also, the rule form in the above study is only a special case of sequential patterns. The search for rules in the restricted form is unable to find our patterns.

Zhu and Shasha[20] found high correlations between all pairs of data streams based on Discrete Fourier Transforms. Yi et al.[17] studied an entire set of streams as a whole to predict the last "current" values based on a multi-variate linear regression. These two studies tried to build global models between two entire streams or among the entire set of streams while our focus in this paper is on mining local patterns across time sequences. By local, we mean that we are interested in the patterns of events in different sequences that happen within a time window in a flexible temporal order. In addition, the online algorithms in the above two papers deal with streaming data, while our algorithm, like the previous studies,[5,11] works offline on static sequences.

Another line of related research is to efficiently identify a pattern out of a set of patterns when that pattern appears in the data streams. Gao and Wang[6] proposed Fast Fourier Transforms based optimization techniques for this pattern matching process. Keogh and Smyth[8] attacked fast pattern matching with a probabilistic approach. Wang and Wang[15] monitored the occurrences of patterns in the form of conjunctive correlations among multiple data streams. We can see that before the online pattern matching process the users need data mining algorithms to discover interesting patterns from historical time sequences, which is the topic of this paper.

§6 Discussion

In this paper, we use *consecutive* or *non-overlapping* time windows to model locations of patterns. That is, each window of data is not overlapped by other windows of data. Another option could be *sliding* or *overlapping* time windows. In this case, the users can specify the length of every sliding step. For example, after each time point a time window may slide one step (time point), which drops off the first time point of data in the original window and appends a new time point of data to the rest of data in the original window to form a new window of data. Likewise, a time window can slide two steps by updating

two new time points of data. If the length of the sliding step equals to the window size, a sliding window becomes a consecutive window. A sliding window allows a mining algorithm to discover patterns across two consecutive windows, while consecutive windows may break a pattern into two shorter patterns. One problem of a sliding window is that the redundancy of overlapping data may cause a mining algorithm to return patterns of inaccurate frequency. That is, a short pattern may exist in the overlapping part of several sliding windows and its one occurrence may be counted several times. One possible solution is to push the association of pattern literals into the discovery process. For example, if the length of the sliding step is 1, then for the data of a sliding window, only patterns starting at the first time point of that window are considered and other patterns will be considered in next sliding windows. In this case, the overlapping part of data can only be counted once as a part of some independent pattern (the overlapping part of the data can be counted many times as parts of different patterns). Note that different types of time windows may return different sets of patterns and may have impacts on the counting or scanning process of a mining algorithm, but they do not affect the application of our suffix appending approach. In every situation, the spirit of suffix appending can be used to utilize the knowledge from discovered patterns to speed up the mining process. Other techniques employed in MILE can be used as well under different types of time windows.

§7 Conclusion

Discovering interesting structures in multiple time sequences is a nontrivial task for many real-world applications. These patterns can be used to explore event correlations across time sequences and assess their causal relationships. In this paper, we have defined such structures as sequential patterns, which essentially involve more computational cost than association rule mining and thus become more challenging. MILE, the proposed algorithm, recursively utilizes the knowledge of the mined patterns from the previous mining procedures to accelerate the new pattenn's discovery. We have also applied a state-of-the-art sequential pattern mining algorithm PrefixSpan upon which MILE is built to solve our problem. Extensive empirical results have shown that MILE is significantly faster than PrefixSpan. MILE's performance can be further improved by incorporating prior knowledge of the data distribution into the mining process. In memory limited environments, we have proposed a solution to trade time efficiency in memory. The method effectively saves a significant amount of memory and meanwhile maintains a reasonable performance in terms of CPU time.

References

1) Agrawal, R. and Srikant, R., "Mining Sequential Patterns," in *Proc. of the 11th Int'l Conf. on Data Engineering*, pp. 3-14, 1995.

2) Ayres, J., Flannick, J., Gehrke, J. and Yiu, T., "Sequential Pattern Mining Using a Bitmap Representation," in *Proc. of the 8th Int'l Conf. on Knowledge*

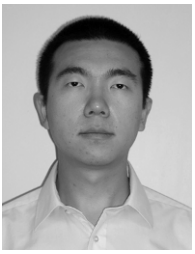
Discovery and Data Mining, pp. 429-435, 2002.

3) Bettini, C., Wang, X.S., Jajodia, S. and Lin, J., "Discovering Frequent Event Patterns with Multiple Granularities in Time Sequences", *IEEE Transactions on Knowledge and Data Engineering, 10- 2*, pp. 222-237, 1998.
4) Charikar,M., Chen, K. and Farach-Colton, M., "Finding Frequent Items in Data Streams," in *Proc. of Int'l Colloquium on Automata, Languages and Programming*, pp. 508-515, 2002.
5) Das, G., Lin, K., Mannila, H., Renganathan, G. and Smyth, P., "Rule Discovery from Time Series," in *Proc. of the 4th Int'l Conf. of Knowledge Discovery and Data Mining*, pp. 16-22, 1998.
6) Gao, L., and Wang, X.S., "Continually Evaluating Similarity-based Pattern Queries on a Streaming Time Series," in *Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 370-381. 2002.
7) Keogh, E. and Lin, J., "Clustering of Time Series Subsequences is Meaningless: Implications for Previous and Future Research," *Knowledge and Information Systems, 8- 2*, pp. 154-177, 2005.
8) Keogh, E., and Smyth, P., "A Probabilistic Approach to Fast Pattern Matching in Time Series Databases," in Proc. of the 3rd Int'l Conf. of Knowledge Discovery and Data Mining, pp. 16-22, 1997.
9) Manku,G. S., and Motwani, R., "Approximate Frequency Counts over Data Streams," in *Proc. of the 28th Int'l Conf. on Very Large Data Bases*, pp. 346-357, 2002.
10) Mannila, H., Toivonen, H. and Verkamo, A.I., "Discovery of Frequent Episodes in Event Sequences," *Data Mining and Knowledge Discovery, 1-3*, pp. 259-289, 1997.
11) Oates, T. and Cohen, P.R., "Searching for Structure in Multiple Streams of Data," in *Proc. of the 13th Int'l Conf. on Machine Learning*, pp. 346-354, 1996.
12) Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H.,Chen, Q., Dayal, U. and Hsu, M., "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix Projected Pattern Growth," in *Proc. of the 17th Int'l Conf. on Data Engineering*, pp. 215-226, 2001.
13) Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U. and Hsu, M., "Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach," *IEEE Transactions on Knowledge and Data Engineering, 16-11*, pp. 1424-1440, 2004.
14) Srikant, R. and Agrawal, R., "Mining Sequential Patterns: Generalized and Performance Improvements," in *Proc. of the 5th Int'l Conf. on Extending Database Technology*, pp. 3-17, 1996.
15) Wang, M. and Wang, X.S., "Efficient Evaluation of Composite Correlations for Streaming Time Series," in *Proc. of the 4th Int'l Conf. on Web-Age Information Management*, pp. 369-380, 2003.
16) Yang, Y., Webb, G. and Wu, X, "Discretization Methods," in *Data Mining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers* (O. Maimon and L. Rokach eds.), Kluwer Academic Publishers, 2005.
17) Yi, B., Sidiropoulos, N., Johnson, W., Jagadish, H.V., Faloutsos, C. and Biliris, A., "Online Data Mining for Co-Evolving Time Sequences," in *Proc. of the 16th Int'l Conf. on Data Engineering*, pp. 13-22, 2000.

18) Zaki, M. J., "Efficient Enumeration of Frequent Sequences," in *Proc. of the 7th Int'l Conf. on Information and Knowledge Management*, pp. 68-75, 1998.

19) Zaki, M. J., "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Machine Learning, 42-1/2*, pp. 31-60, 2001.

20) Zhu, Y. and Shasha, D., "StartStream: Statistical Monitoring of Thousands of Data Streams in Real Time," in *Proc. of the 28th Int'l Conf. on Very Large Data Bases*, pp. 358-369, 2002.



Gong Chen: He received the BEng degree from the Beijing University of Technology, China, and the MSc degree from the University of Vermont, USA. Both degrees are in Computer Science. He is currently a PhD student in the Department of Statistics at the University of California, Los Angeles, USA. His research interests include statistical computing, data mining, algorithm analysis and design, and database management.

Xindong Wu, Ph. D: He is a Professor and the Chair of the Department of Computer Science at the University of Vermont. He holds a PhD in Artificial Intelligence from the University of Edinburgh, Britain. His research interests include data mining, knowledge-based systems, and Web information exploration. He has published extensively in these areas in various journals and conferences, including IEEE TKDE, TPAMI, ACM TOIS, DMKD, KAIS, IJCAI, AAAI, ICML, KDD, ICDM, and WWW, as well as 14 books and conference proceedings.

Xingquan Zhu, Ph. D: He is an Assistant Professor in the Department of Computer Science and Engineering at Florida Atlantic University, Boca Raton, FL. From Feb. 2001 to Oct. 2002, he was a Postdoctoral Associate in the Department of Computer Science, Purdue University, West Lafayette, IN. From Oct. 2002 to July 2006, He was a Research Assistant Professor in the Department of Computer Science, University of Vermont, Burlington, VT. His research interests include data mining, machine learning, bioinformatics, multimedia systems, and information retrieval.