Sequential Pattern Mining in Multiple Streams

Gong Chen Xindong Wu Xingquan Zhu
Department of Computer Science, University of Vermont, Burlington, VT 05405, USA
{gchen, xwu, xqzhu}@cs.uvm.edu

Abstract

In this paper, we deal with mining sequential patterns in multiple data streams. Building on a state-of-the-art sequential pattern mining algorithm PrefixSpan for mining transaction databases, we propose MILE[1], an efficient algorithm to facilitate the mining process. MILE recursively utilizes the knowledge of existing patterns to avoid redundant data scanning, and can therefore effectively speed up the new patterns' discovery process. Another unique feature of MILE is that it can incorporate some prior knowledge of the data distribution in data streams 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 balance the memory usage and time efficiency in memory constrained environments.

1. Introduction

Many real-world applications involve data streams. Discovering structures of interest in multiple data streams is an important problem. For example, the knowledge from data streams in ICU (such as the oxygen saturation, chest volume and heart rate) may indicate or predicate the state of a patient's situation.

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 heart rate. We are interested in knowledge in the form of frequent sequential patterns across data streams. Das et al. [2] studied the problem of rule discovery from discretized data streams. A rule here is in the form of the occurrence of event A indicating the occurrence of event B within time T. This type of causal rule can be treated 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 [3] 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 [3] is only a special case of sequential patterns. Similar to [2] and [3], our work deals with a period history of data streams (for example, one day or one hour) which can afford multiple scans.

[1] MIning in muLtiple strEams.

2. Problem Statement

Definition 2.1 *Given*

- *a set of streams* $\{S^1, S^2, ..., S^k, ..., S^{|V|}\}$ *where* S^k=$S^k_1 S^k_2 ... S^k_q ... S^k_m$ *is a stream of events in which* S^k_q *is an event in the set of events for stream* S^k *and occurs at the time point q,* $|V|$ *is the number of streams, 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 at the time point ***m****:*

- *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* $i \in [0, \mathbf{w}-1]$ *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,* $p^{k_1}_{i_{j_1}}$ *and* $p^{k_2}_{i_{j_2}}$ $(1 \leq k_1 \leq |V|, 1 \leq k_2 \leq |V|, i_{j_1} < i_{j_2})$, $S^{k_1}_{t+i'_{j_1}}$ *and* $S^{k_2}_{t+i'_{j_2}}$*, 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*
- *the number of each pattern's occurrences in the set of data streams until the time point m (inclusive) is greater than* ***minSup****.*

Consider the following example (where a dot denotes an anonymous event) with 3 data streams and 12 time points. If *minSup* = 2 and *w* = 4, we can find the pattern {(33 22 *)(* * 11)}. We put pattern literals at the same time point in parentheses and put all pattern literals in braces.

	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

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 either happen immediately after (33 22 *) or several time points later within the same window. To be concise, we ignore wild cards in a pattern 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 streams have different sets of tokens, this representation causes no confusion.

For an arbitrary pattern P=$\alpha\tilde{t}\beta$ where α and β are subpatterns of P and $\tilde{t}$ is a token in P, we define $suffix(\tilde{t})$=β, and $prefix(\tilde{t})$=α. Assuming that there are a total of *n* patterns having $\alpha\tilde{t}$ as a prefix: $\alpha\tilde{t}\beta_1$, $\alpha\tilde{t}\beta_2$, $\ldots$, $\alpha\tilde{t}\beta_n$. 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) [4], we first outline its basic steps that can be applied to solve our problem.

1. Scan data streams to locate tokens whose frequency is greater than *minSup*, and output them (each of which is a frequent pattern of 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 data streams at the same window to locate token *b* whose frequency is greater than *minSup*; append *b* to *a*; output *ab*; let *a* = *ab*, and goto step 2. If no frequent token exists, return.

At the same time point, a token in a lower-numbered stream is scanned after tokens in streams with higher numbers; and at different time points in the same stream, a token at a later time point is scanned after earlier tokens.

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 $\tilde{t}$ at $\tilde{t}$.endLoc,
update the frequency for $\tilde{t}$ at $\tilde{t}$.freq;
6 for every token $\tilde{t}$ and if($\tilde{t}$.freq>minSup)
7 if($suffixes(\tilde{t})$ in s)
8 $suffixesSet(t)$←**SuffixAppend**($\tilde{t}$, $suffixes(\tilde{t})$, idx);
9 else
10 $suffixesSet(t)$←**PrefixExtend**($\tilde{t}$, $suffixesSet(t)$, nIdx);
11 $suffixes(t)$←append $\tilde{t}$ to (_);
12 $suffixes(t)$←append $suffixes(\tilde{t})$ in $suffixesSet(t)$ to (_ $\tilde{t}$);
13 return $suffixes(t)$;
}

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

Figure 1. Pseudo code for MILE

3.2. Description of MILE

We describe MILE with a pattern tree in Figure 2. A concatenation of literals on the edges from the root to any node forms a pattern. Here we can ignore the parentheses in patterns to understand the main idea of MILE smoothly. β_i denotes a suffix of a pattern. From the description of PrefixSpan, we can see that it performs a depth-first-search like discovery along this pattern tree. It mines patterns in the following order: $\langle$ {11}, {22}, {33} $\rangle$ → $\langle$ {11 44}, {11 55} $\rangle$ →...→{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}. PrefixExtend in MILE 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 from $suffixes(44)$ and append them directly to {11 55 44} instead of performing a depth-first search to scan overlapping data (since all data after {11 55 44} must appear after {11 44}) as PrefixSpan does. We use arrows to

mark each place where SuffixAppend occurs in Figure 2. We can see that SuffixAppend is embedded in the mining process to make new patterns' discovery fast.

Now we describe the selection process in SuffixAppend. It has three steps which are commented in Figure 1: building index, hitting process and choosing the desired suffixes. We show how these three steps work by using one part of the pattern tree in Figure 2. Assuming MILE is currently running at point $\{11\ 55\ 44\}$ with ending locations (time points when 44 in this pattern occurs) (3, 7, 15, 26), it finds that $suffixes(44)$ for prefix 11 has been mined, so SuffixAppend is called.

Assume (1) *minSup*=1; (2) start locations (time points when 44 in the corresponding suffixes occurs) of suffixes in $suffixes(44)$ for prefix 11 are (_ β_1): (3, 39), (_ β_2): (3, 15), and (_ β_3): (7, 26); and (3) no index has been built for $suffixes(44)$ thus far, so the building index process starts. The resulting hash table is shown in Figure 3.

Now the hitting process begins. Every ending location of $\{11\ 55\ 44\}$ is hashed into the hash table to update corresponding suffixes' frequencies. Then, the choosing process caches every frequent suffix in $suffixes(44)$ for prefix $\{11\ 55\}$ for possible future appending. Every frequent suffix is also appended to prefix $\{11\ 55\ 44\}$ in PrefixExtend. In this example, (_ β_2) and (_ β_3) are frequent which can also be seen from Figure 2. 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.

3.3. Optimization Techniques

Incorporating Prior Knowledge If some prior knowledge of the data distribution in data streams is available, the performance of MILE can be further improved. If the users, for example, know in advance the frequency of one token's occurrence in some data stream is higher than others, that token might have more chance to get more suffixes appended if the discovery for patterns with that token as prefix is conducted at a later stage than the one for patterns

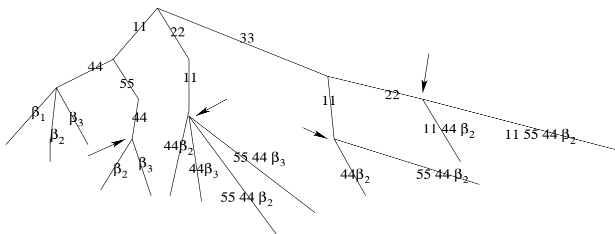

Figure 2. Part of a pattern tree showing the mining process of MILE

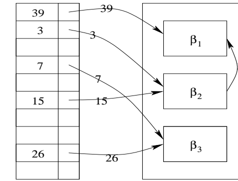

Figure 3. Index of $suffixes(44)$ for prefix 11

with other tokens as prefix. In this way, MILE will avoid more expensive redundant data scanning. Our general strategy in MILE is to discover patterns with tokens of a lower frequency as prefix earlier than the ones with tokens of a higher frequency as prefix under an encoding mechanism. The encoding method is presented in [1].

Balancing Memory Usage and Performance If MILE only records down and builds indices for mined suffixes whose length exceeds a predefined parameter *l*, and uses PrefixExtend to grow shorter patterns, it will use less memory than the original algorithm although the efficiency will degrade at the same time. In [1], we have provided more discussions about this issue and the experimental results show that MILE can save a significant amount of memory while maintaining a reasonable efficiency through the memory balancing procedure.

4. Experimental Evaluation

Experiment Environment All experiments were conducted on a server with four 1GHz SPARC CPUs and 8 gigabyte memory. OS is Solaris 9. We implemented MILE and PrefixSpan (according to [4]) in Java. The JVM version is 1.5.0_01-b08. All outputs are turned off.

Data Generation We generated data sets with uniform and multinomial distributions with specified probabilities. The uniform distribution is used unless otherwise explicitly explained. Three parameters are used in each data set name to indicate its settings: *s* denotes the number of streams, *t* the number of time points, and *v* the number of different tokens per stream. The window size is set to 4.

Performance Comparisons When Varying Time Points and Window Sizes First we compare the performance of MILE with PrefixSpan on data sets of different time points. Relative *minSup* is used. Figure 4 and more, similar results in [1] show that MILE runs consistently faster than PrefixSpan. When the *minSup* becomes less and less, more and more patterns appear and more computation is involved. It is at that point the difference between MILE and PrefixSpan becomes large. When we vary the window size and fix the other factors, [1] shows a consistent performance improvement of MILE over PrefixSpan.

Incorporating Prior Knowledge on Data Distributions Figure 5 demonstrates the performance (Pt-Mt)/Pt ((PrefixSpan's CPU time-MILE's CPU time) / PrefixSpan's CPU time, which is denoted as (Pt-Mt) / Pt hereafter) of

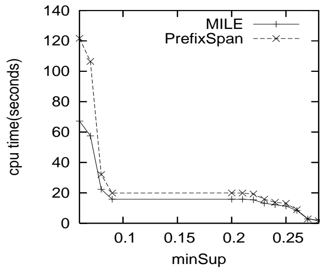

Figure 4. CPU time comparison, s9t20000v4

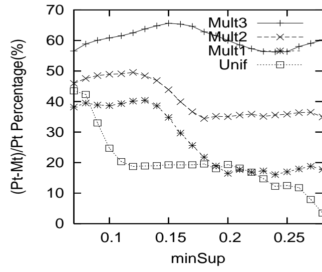

Figure 5. Various distributions, s9t2000v4

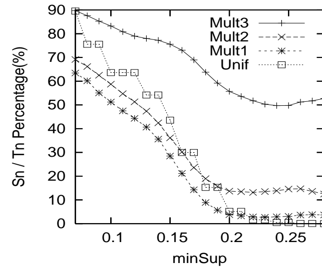

Figure 6. The ratio Sn / Tn, s9t2000v4

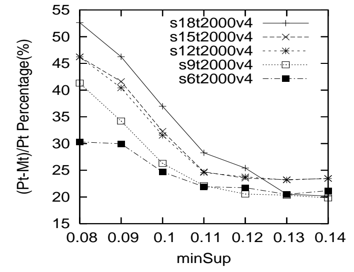

Figure 7. Varying the number of streams

MILE when some prior knowledge about data distributions is incorporated into the mining process as described in Section 3.3. We generated data sets in such a way that (1) data set *Mult1* has one stream 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 streams each of which contains a token (with a probability of 0.55) that happens more frequently than others; and (3) data set *Mult3* has two streams each of which contains a token (with a probability of 0.75) that happens more frequently than others. From Figure 5, 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 the suffix appending approach.

Intuitively, the larger the number of patterns formed by suffix appending, the faster MILE runs in comparison with PrefixSpan. From Figure 6, we can see that the performance of MILE is indeed consistent with the ratio of the number of patterns formed by suffix appending over the number of all patterns Sn/Tn. Sn/Tn in these three data sets is in the order of *Mult3*>*Mult2*>*Mult1*.

Tokens in data set *Unif* are uniformly distributed. In this case, the average performance of MILE is minimized when no prior knowledge can be incorporated. However, the discussion from the previous paragraphs in this section shows that MILE still outperforms PrefixSpan when dealing with this kind of data sets. Note that although in Figure 6 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. Why does this happen? In [1], statistics on suffixes of different lengths were collected which show that *Mult3*, *Mult2* and *Mult1* have more mined suffixes of longer lengths than *Unif*. This indicates that more expensive depth-first search for repeatedly scanning overlapping parts of data is avoided by the suffix appending approach in the multinomial distribution cases. The longer the length of appended suffixes, the better the MILE's performance.

Performance Comparison When Varying the Number of Streams Figure 7 shows the scalability of MILE when the number of data streams 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 streams is increased. Although [1] shows that the increase in the number of streams does not change much the length of appended suffixes, the ratio Sn/Tn is increased when the number of streams is increased. This explains why MILE gains more improvement over PrefixSpan as the number of streams becomes larger.

5. Conclusion

In this paper, we have provided an efficient algorithm MILE to manage the discovery of sequential patterns in multiple data streams. Although MILE was built upon PrefixSpan, the unique suffix appending approach in MILE has made it distinct from PrefixSpan by avoiding redundant data scanning and speeding up new patterns' discovery. Furthermore, two optimization techniques, incorporating prior knowledge and a memory balancing procedure, further improve MILE's performance. Extensive empirical results have shown that MILE is significantly faster than PrefixSpan.

References

[1] G. Chen, X. Wu, and X. Zhu. Mining sequential patterns across data streams. *Computer Science Technical Report CS-05-04, University of Vermont*, 2005.

[2] G. Das, K.-I. Lin, H. Mannila, G. Renganathan, and P. Smyth. Rule discovery from time series. In *Proceedings of the 4th International Conference of Knowledge Discovery and Data Mining*, pages 16–22, 1998.

[3] T. Oates and P. R. Cohen. Searching for structure in multiple streams of data. In *Proceedings of the 13th International Conference on Machine Learning*, pages 346–354, 1996.

[4] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Trans. Knowl. Data Eng.*, 16(11):1424–1440, 2004.