JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 29, 1151-1169 (2013)

DOI:10.1688/JISE.2013.29.6.6

PTree: Mining Sequential Patterns Efficiently in Multiple Data Streams Environment

GUANLING LEE, YI-CHUN CHEN AND KUO-CHE HUNG
Department of Computer Science and Information Engineering
National Dong Hwa University
Hualien, 974 Taiwan
E-mail: guanling@mail.ndhu.edu.tw

Although issues of data streams have been widely studied and utilized, it is nevertheless challenging to deal with sequential mining of data streams. In this paper, we assume that the transaction of a user is partially coming and that there is no auxiliary for buffering and integrating. We adopt the *Path Tree* for mining frequent sequential patterns over data streams and integrate the user's sequences efficiently. Algorithms with regards to accuracy (*PAlgorithm*) and space (*PSAlgorithm*) are proposed to meet the different aspects of users, although *GAlgorithm* for mining frequent sequential patterns with a gap limitation is proposed. Many pruning properties are used to further reduce the space usage and improve the accuracy of our algorithms. We also prove that PAlgorithm mine frequent sequential patterns with the approximate support of error guarantee. Through thoughtful experiments, synthetic and real datasets are utilized to verify the feasibility of our algorithms.

Keywords: data mining, multiple data streams, sequential patterns, frequent patterns, knowledge discovery

1. INTRODUCTION

In recent years, emerging applications, such as sensor network data analysis, network traffic analysis, and Web-click stream mining, have called for a study of a new kind of environment, called data streams. For data stream applications, the volume of data is usually too huge to be stored in persistent devices or to be scanned more than once [14]. In this paper, we focus on the problem of frequent sequence mining in data streams. Study of frequent sequence mining of offline fashion has been extensively done in a sequence database. A sequence database is a collection of ordered data items or events *i.e.*, a DNA sequence, network flow logs and Web-click stream logs. Frequent sequences mining can be classified, based on the pattern they produce, into two categories: consecutive sequence [4, 7] and generalized sequence. A generalized sequence is a sequence that allows wildcards. A sequential pattern is similar to a generalized sequence without the consideration of an itemset. The sequential patterns mining problem was first discussed in [1]. Thereafter, many efficient mining methods have been studied extensively. Since conventional sequential pattern mining methods may generate a huge number of short and trivial patterns, Kum *et al.* aim at mining sequential patterns roughly by using a cluster technique [9]. Pei *et al.* introduce *PrefixSpan* to avoid hung memory usage during the mining process [15]. Yu and Chen propose an algorithm for mining a sequential pat-

Received March 2, 2013; accepted April 19, 2013.
Communicated by Ruay-Shiung Chang and Sheng-Lung Peng.

tern from the multidimensional sequence data [18]. Chen *et al.* first proposed an off-line algorithm for sequential pattern mining across multiple streams [3]. Zhao *et al.* first proposed a prefix-projection based algorithm, U-PrefixSpan, for probabilistically frequent sequential patterns mining [19]. Due to compression based pattern mining has been applied to many data mining tasks, Lam *et al.* proposed an efficient greedy algorithm, Go-Krimp, to directly mine compressing sequential patterns [10].

The key characteristics of existing algorithms in sequences database are the multiple scans and secondary memory accesses involved. The characteristics are diametrically opposed to the limitations of the data stream environment: one pass scheme, short response time and limited system resources. Consequently, on-line fashion algorithms are needed to tie in the data stream environment. Li *et al.* introduced an on-line algorithm, *StreamPath*, to mine frequent sequence patterns [12]. In the proposed algorithm, a majority method is used for the guarantee of memory usage. In recent years, top-k sequences mining have been discussed in [6, 13]. The issues of mining closed sequential patterns in a data stream environment are introduced in [2, 5, 17]. Moreover, the problem of mining sequential patterns in the multiple data streams environment is introduced in [16], and the application of sequential patterns with B2B is discussed in [8].

Sensor ID					
	s_1	{A, C}		{A, B}	{C}
	s_2	{B}	{A, B, C}	{C}	{A, B}
Time Period		1	2	3	4

Fig. 1. Data of sensor gathering in each time period.

User ID					
	A	s_1	s_2	s_1	s_2
	B	s_2	s_2	s_1	s_2
	C	s_1	s_2	s_2	s_1
Time Period		1	2	3	4

Fig. 2. The database after integration.

In this paper, the problem of finding frequent sequential patterns across multiple streams in an on-line fashion is introduced and discussed. We illustrate our motivation by the following application. Suppose biologists are interested in the mice's sequences of reactions (events) after giving mice certain triggers. To collect those sequences, biologists tag each mouse with an ID and set sensors for every event. The sensors will output an ID sequence periodically. Each output represents who has passed by in the corresponding time period. Biologist can mine recently frequent sequences of events by mining the outputs generated by sensors; for example, there are 3 mice, {A, B, C}, and 2 sensors, $\{s_1, s_2\}$. The data reported by the sensors is shown in Fig. 1. In the traditional way, a buffer is needed to buffer and integrate temporal outputs. The integrated database is shown in Fig. 2. Upon analyzing Fig. 2, it is easy to see that sequences $\{s_1, s_2\}$ and $\{s_2, s_2, s_1\}$ are frequently accessed. A user's transaction gradually becomes complete under this environment of data streams, which is practical in many real applications, such as routing optimization by using logs of routers and transportation monitoring. With the growing technique of RFID, monitoring a vast number of objects is becoming more practical. However, it is hard to mine underlying knowledge from separate records in real time and update it incrementally as time advances.

The problem of mining frequent sequential patterns in data streams with incremental transactions has not been well discussed to date. In [12, 13], they aimed to mine frequent

sequences, called maximal forward references from data streams. A maximal forward reference (MFR) is a consecutive forward reference sequence without any backward reference, for example, the MFR of sequence <CABCDCA> is <ABCD>. They adapt FP-Tree based structures to achieve mining a frequent sequence in one pass; however, they do not take the problem of incremental transaction into account. In our previous research [11], we developed a notation of sequential patterns in multiple streams and proposed two algorithms, PAlgorithm and PSAlgorithm, for mining sequential patterns without any auxiliary buffer. An approximation for counting the algorithms with a guaranteed error boundary was proposed. Furthermore, we also proposed an algorithm with compressed space. Upon these two algorithms, the paths maintained by Ptree are within the current window. However, people may be interested in other types of frequent paths. Therefore, in this paper, we propose a variational algorithm, GAlgorithm, to deal with the problem of mining frequent paths with a gap limitation for different application based on the same structure. Moreover, we will discuss some optimizations to improve the speed of algorithms.

The rest of the paper is organized as follows. The problem statement is provided in section 2. In section 3, the structure and algorithms are explained, followed by the performance result with thoughtful experiments in section 4. Conclusions are discussed in section 5.

2. PRELIMINARIES

2.1 Stream Environment

Let $i = \{i_1, i_2, \ldots, i_n\}$ be the monitored items and n be the total number of items that can be a variable or fixed value depending on applications. An itemset $e = \{i_{j_1}, \ldots, i_{j_k}\}$ is a subset of I. The monitoring sensors set is $S = \{s_1, s_2, \ldots, s_m\}$ where m is the total number of monitoring sensors. Taking Fig. 1 as an example, the item set is {A, B, C} and the monitoring sensor set is $\{s_1, s_2\}$. The data stream generated by s_j is a sequence with consecutive and boundless data items and has the form $\{e_{s_jt_1}, e_{s_jt_2}, \ldots, e_{s_jt_k}\}$ where $e_{s_jt_k}$ is the set of items detected by s_j during the time period t_k and named as an *appearance*. Referring to Fig. 1, $e_{s_1t_1}$ = {A, C} means item A and C appear at monitoring sensor s_1 in time period t_1. The collection of appearances during a specific time period t_k is a *tuple*, $w_k = \{e_{s_1t_k}, e_{s_2t_k}, \ldots, e_{s_mt_k}\}$. In this paper, we assume that an item can only appear once during a time period, but multiple times in different periods.

A *path* of an item i_l, $path_{i_l} = <s_{t_1}, s_{t_2}, \ldots, s_{t_k}>$ where $s_{t_j} = \{s_\beta$, if $i_l \in s_{s_\beta t_j}\}$, is a time ordered sequence and is recorded by the sensors where it passes by. And, s_{t_1} is the *head* of the $path_{i_l}$. Moreover, k increases as time advances. A $path_{i_l} = <\alpha_1, \alpha_2, \ldots, \alpha_j>$ is a *sub-path* of a $path_{i_p} = <\beta_1, \beta_2, \ldots, \beta_k>$, denoted by $path_{i_l} \subseteq path_{i_p}$, if there exist integers $1 \le l_1 \le \ldots \le l_j \le k$ such that $\alpha_1 = \beta_{l_1}, \alpha_2 = \beta_{l_2}, \ldots, \alpha_j = \beta_{l_j}$. We also call $path_{i_p}$ a *super-path* of $path_{i_l}$. The support count of a path x is the number of items whose path contains x. And the support of x, denoted by $support(x)$, is defined by the ratio of the support count of x to the total number of items. Taking Fig. 1 as an example, the support count of path $<s_1s_2>$ is three.

2.2 Problem Definition

Since people have more interest in recent data, we can define a *stream window*, $W_j = \{w_{j-D+1}, \ldots, w_{j-2}, w_{j-1}, w_j\}$, $j - D + 1 \geq 0$, to delimit which data we are interested in, where j is the current time period and D is the user defined window size. The timestamp of a path formed by item i is denoted by $timestamp(i, path_i)$, that is the time that i appeared in the head. For example, referring to Fig. 2, the value of timestamp(A, $<s_1s_2s_1>$) is one. A path formed by item i is expired whenever its timestamp is out of a current time window. Our idea is to treat every path equally within the current window and ignore expired paths as soon as the window slides.

The input of our system is an (infinite) set of tuples which are coming one by one at every end of time period. Our goal is to mine the paths with support that exceed a *minimum support* within the current window. Minimum support is a user-defined parameter $min\_sup \in (0, 1)$. Given a user-defined error threshold $\varepsilon \in (0, 1)$ and $\varepsilon << \text{min\_sup}$, the answer of our algorithm is required to have the following guarantees:

1. All paths whose support exceeds min_sup are output. There are no *false negatives*.
2. No paths whose support has a value of less than (min_sup-ε) is output.

3. ALGORITHMS

In this section, we first explain our basic idea of frequent paths mining by answering the following two questions: How can we enumerate the complete set of frequent paths and, since the inputs of our environment are continuous tuples, how can we concatenate an item's path without the help of auxiliary buffer. Then, we propose two algorithms to come to the compromise between precision (*PAlgorithm*) and space (*PSAlgorithm*).

3.1 Path Tree

Assume that there is a lexicographical ordering < among the set of sensor ids S (*i.e.*, in Fig. 1, one possible sensor ordering can be $S_1 < S_2$). Conceptually, the complete search space of path mining forms a *path tree* (*PTree*), which can be constructed in the following way: The root node $node^0_{\varnothing}$ of the PTree is at level 0 and labeled ∅. From root node $node^0_{\varnothing}$ but excluding $node^0_{\varnothing}$, each node $node^0_{s_j}$ PTree forms a path $path(node^l_{s_j}) = <s_x, \ldots, s_j>$, where $node^1_{s_x}$ is in the path from $node^0_{\varnothing}$ to $node^l_{s_j}$ and s_x is the label of the node following $node^0_{\varnothing}$. A set of items denoted by $item\_set(node^l_{s_j})$ maintained in node $node^l_{s_j}$ is used to store the relationship between paths and items, and assist the recognition of the item's path whenever the new arrival tuple is inserted into the PTree. Moreover, each item i maintained in node $node^l_{s_j}$ is associated with a timestamp $timestamp(i, node^i_{s_j})$ which is set to $timestamp(i, path(node^l_{s_j}))$. Fig. 3 shows the PTree built from Fig. 1. In the PTree, each node represents a path from root to the node and contains a set of items. According to Fig. 3, we see that item A has passed paths $<s_1>$, $<s_1, s_2>$ and $<s_2>$ during the time window of time period 2, since A is maintained by each node accordingly. It is obvious that PTree stores all the information of items path and sub-paths. The PTree construction process is illustrated by the following example: in Figs. 1 and 3, $w_2 = \{e_{s_2t_2=\{A,B,C\}}\}$ is incoming, item A will be inserted into $node^1_{s_2}$ with timestamp 2, since $node^1_{s_2}$ already

exists. In addition, because A has visited $node^1_{s_1}$ before and $node^1_{s_1}$ has no child with label s_2, a new child $node^2_{s_2}$ is inserted into $node^1_{s_1}$ to maintain the path $<s_1, s_2>$ formed by A. Moreover, A is inserted into *item_set*($node^2_{s_2}$) and *timestamp*(A, $node^2_{s_2}$) is set to *timestamp*(A, $node^1_{s_1}$) since *timestamp*(A, $node^1_{s_1}$) is the timestamp of path $< s_1, s_2>$ formed by A. Recursively, we can extend a node $node^l_{s_j}$ at level l with label s_j in PTree by adding sensor id s_m as its child node $node^{l+1}_{s_m}$ at the next level $l + 1$. This is called the *path extension* and the extended path is denoted by *path*($node^l_{s_j}$)$\cdot s_m$ (the corresponding node is denoted by $node^l_{s_j} \cdot s_m$). Taking Fig. 1 as an example, after the update of appearance $e_{s_2t_2}$ = {A, B, C}, $node^1_{s_1}$ and $node^1_{s_2}$ are identified and extended to generate *path*($node^l_{s_1}$)$\cdot s_2$ and *path*($node^l_{s_2}$)$\cdot s_2$, since paths $<s_1s_2>$ and $<s_2s_2>$ are formed.

In the PTree, the item i and its *timestamp*(i, $node^l_{s_j}$) is set accordingly whenever i was inserted into node $node^l_{s_j}$ and the timestamp is used to identify which items' sub-path *path*($node^l_{s_j}$) has expired in the next time window. An item does not contain *path*($node^l_{s_j}$) in the time period of t_k whenever the *timestamp*(i, $node^l_{s_j}$) + D is no greater than t_k. We can take Fig. 3 as an example and assume that the size of the stream window is 3. As the stream window slides from time period 3 to 4, the item i whose *timestamp*(i, $node^l_{s_j}$) equals 1 is expired and dropped from $node^l_{s_j}$, *i.e.*, item A in $node^2_{s_1}$ and B in $node^3_{s_1}$. We remove item i and its corresponding timestamps from a $node^l_{s_j}$ and $node^l_{s_j}$'s sub-tree whenever the *timestamp*(i, $node^l_{s_j}$) is expired, since valid sub-paths of i are still maintained in the PTree. Obviously, the support count of a path formed by node $node^l_{s_j}$ (we will use the node to represent the path formed by itself for short), *sup_count*($node^l_{s_j}$), is the number of items it maintains. By traversing the PTree and identifying the node whose support count exceeds min_sup × n, we can enumerate all frequent paths. The PTree has the following properties:

Property 1: In the process of path extension, the possible itemset whose path contains *path*($node^l_{s_j}$)$\cdot s_m$ is the intersection of the items maintained by $node^l_{s_j}$ and its sibling $node^l_{s_m}$.

Property 2: The *timestamp*(i, $node^l_{s_j}\cdot s_m$) is always no greater than *timestamp*(i, $node^l_{s_j}$).

Property 3: When *timestamp*(i, $node^l_{s_j}$) is expired, item i maintained by the nodes of the sub-tree rooted at $node^l_{s_j}$ can also be dropped.

And according to property 1, we can get the following lemma.

Lemma 1: The support count of *path*($node^l_{s_j}$)$\cdot s_m$ is no greater than the number of items within the intersection of the items maintained by $node^l_{s_j}$ and its sibling with label s_m.

Proof: In the process of path extension, the possible itemset whose path contains *path*-($node^l_{s_j}$)$\cdot s_m$ is the intersection of the items maintained by $node^l_{s_j}$ and its sibling $node^l_{s_m}$. Therefore, the support count of *path*($node^l_{s_j}$)$\cdot s_m$ cannot exceed the number of items within the intersection of the items maintained by $node^l_{s_j}$ and its sibling with label s_m.

3.2 Algorithms

If we can record all the paths whose support count is no smaller than one in the

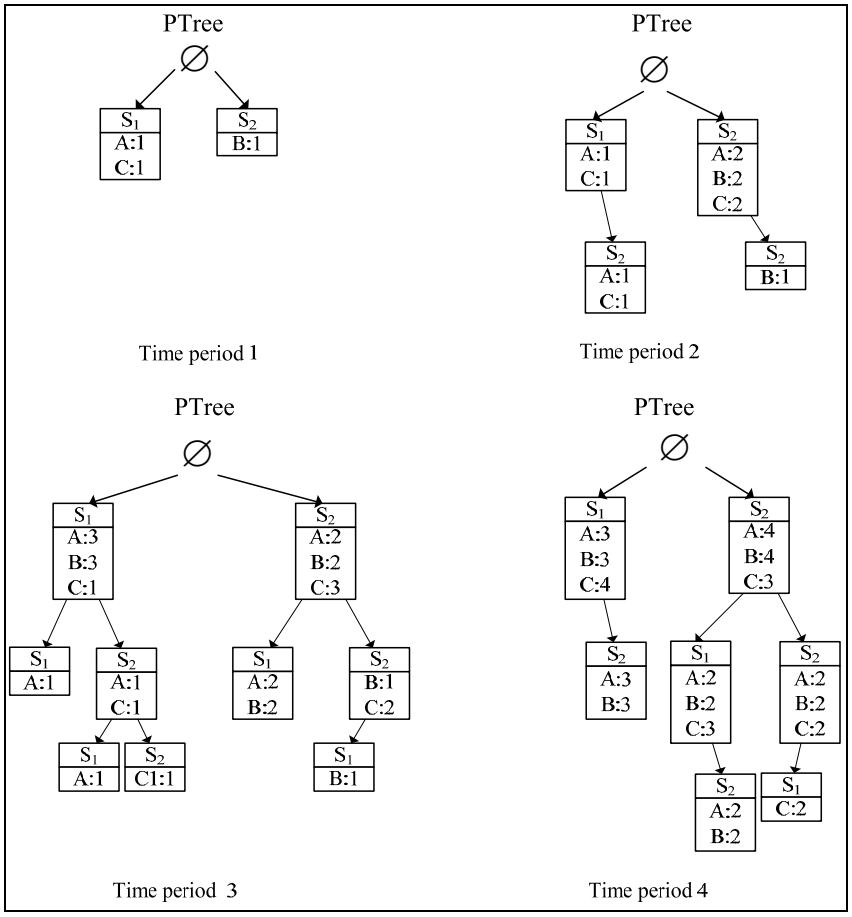


Fig. 3. PTree constructed from Fig. 1.

PTree, we can solve the problem by traversing the tree and reporting the $path(node^l_{s_j})$ as a frequent path if and only if the number of items maintained by $node^l_{s_j}$ is no smaller than min_sup × n. However, the rapidly increasing number of paths can cause severe problems which include prohibitive space and computing overheads. To address this problem, Teng *et al.* hold the generation of a candidate pattern until all its sub-patterns of size-1 are frequent. However, the problem of this generation mechanism is delayed pattern recognition. To solve the problem, a user-specified error parameter ε is used as the candidate generation threshold, *ε-generation*. After an update iteration, nodes with support count exceeding ε are identified. These identified nodes can be taken into account to generate new candidates, the candidates of which are then counted in later iterations.

Because $path(node^l_{s_j}) \cdot s_m$ would not be generated until the support counts of $node^l_{s_j}$ and its sibling $node^l_{s_m}$ exceed $\varepsilon \times n$, we could not determine the items whose path contains $path(node^l_{s_j})$ and $path(node^l_{s_m})$ forms $path(node^l_{s_j}) \cdot s_m$ or $path(node^l_{s_m}) \cdot s_j$ whenever the path extension is performed. Therefore, we set the items maintained by extended nodes $node^l_{s_j} \cdot s_m$ and $node^l_{s_m} \cdot s_j$ to the possible maximum set of items. According to property 1, the pos-

sible items whose path contain $path(node^l_{sj}) \cdot s_m$ is the intersection of the items maintained by $node^l_{sj}$ and its sibling $node^l_{sm}$. Similarly, the sub-tree rooted at node $node^l_{sj}$ can be removed whenever the sup_count($node^l_{sj}$) is less than $\varepsilon \times n$.

Lemma 2: All nodes whose true support exceeds min_sup can be recognized in the PTree constructed by ε-generation.

***Proof*:** Assume t' is the support count of a node γ accumulated so far and t is the true support of γ, we have $t' \geq t \geq t' - \varepsilon$, since sup_count($\gamma$) is always set to ε whenever it was generated. If γ's true support exceeds min_sup, we have $t' \geq t \geq$ min_sup. Thus, nodes with true support count that exceed the min_sup will be recognized. ❑

Lemma 3: Each node γ with estimated support t' no smaller than the min_sup in PTree constructed by using ε-generation satisfies: $t' - t \leq \varepsilon$.

***Proof*:** Because the over estimate of γ is $t' - t$, where t is the true support of γ, and we know $t \geq t' - \varepsilon$. Therefore, $t' - t \leq t' - (t' - \varepsilon)$. And we get $t' - t \leq \varepsilon$. ❑

From Lemmas 2 and 3, we can examine that the PTree constructed by using ε-generation is satisfying those guarantees outlined in the previous section.

Moreover, we can further reduce the overestimated support count of a node by the assistance of timestamp. In the following, we introduce two algorithms to enumerate the complete set of frequent paths with regards to the accuracy and space. The first algorithm, PAlgorithm, keeps the full information of the timestamp in nodes to meet the requirements of the guarantee discussed in section 2. The second algorithm, PSAlgorithm, keeps less information of the timestamp in nodes. It calculates an item's timestamp through an estimation function.

Before explaining the PAlgorithm, we will now outline some properties of the PTree:

Property 4: Whenever the path extension of $path(node^l_{sj}) \cdot s_m$ is performed, the *timestamp*(i, $node^l_{sj} \cdot s_m$) is equal to *timestamp*(i, $node^l_{sj}$), when *timestamp*(i, $node^l_{sj}$) ≤ *timestamp*(i, $node^{l'}_{sm}$). Otherwise, it is equal to *timestamp*(i, $node^l_{sm}$).

Property 5: If item i is a newly added item in $node^l_{sj}$ at time t_k, i would not be inserted into $node^l_{sj} \cdot s_m$ during the path extension of $node^l_{sj}$ with sibling $node^l_{sm}$ at t_k.

The pseudo code of PAlgorithm is shown in Fig. 4, and the PTree constructed by PAlgorithm shown in Fig. 5 is referred to in Fig. 1. Through PAlgorithm, PTree persistently maintains a root and a set of child nodes $\{node^1_{sm} | s_m \in S\}$, only the monitored nodes are counted in each update. Generally, PAlgorithm traverses PTree in a DFS manner. For an incoming item i of appearance e_{sjtk}, PAlgorithm will check each node $node^1_{sm}$ at level one and update i recursively if item i exists in $node^1_{sm}$. After the recursive updating of i, PAlgorithm inserts or updates i with timestamp t_k into node $node^l_{sj}$. Referring to Fig. 5, when item A of $e_{s_2t_2}$ is coming, the nodes' information rooted at $node^1_{s_1}$ with A is updated, since A is maintained in $node^1_{s_1}$. After the update, A is inserted into $node^1_{s_2}$, and *timestamp*(A, $node^1_{s_2}$) is set to the current time period t_2. By checking the items maintained by nodes, PAlgorithm recognizes the sub-paths of an item i and updates nodes through call-

ing subroutine *PUpdate*. The sense of recursively calling PUpdate is to recognize each i's sub-paths, to concatenate each i's sub-paths with s_j and to update each i's sub-paths to the latest timestamp. The item's timestamp of a node will be updated if the node's label equals the sensor id of the current appearance. Moreover, the *timestamp*(i, $node^l_{s_m}$) should be updated by the *timestamp*(i, $node^l_{s_m}$*'s parent*) instead of the current time period.

PAlgorithm()
Input: a stream of tuples gathered by sensors, a user-specified min_sup, error parameter ε and window size D.
Output: Frequent path set F.
1: **while**(1) **do**
2: w_{t_k} = tuple of time period t_k
3: **foreach** appearance $e_{s_j t_k} \in w_{t_k}$ **do**
4: **foreach** item $i \in e_{s_j t_k}$ **do**
5: **foreach** node $node^1_{s_m}$ contains i **do**
6: $node^1_{s_m}$.*PUpdate*(i, s_j, (i, $node^1_{s_m}$));
7: **if** $node^1_{s_j}$ contains i **then**
8: update *timestamp*(i, $node^1_{s_j}$) to t_k;
9: **else**
10: insert i into $node^1_{s_j}$ with timestamp t_k;
11: **if** sup_count($node^1_{s_j}$) $\geq \varepsilon n$ **then**
12: Path Extension;
13: Pruning Process;
PUpdate(i, s_j, ts)
Input: an item ID i, label of current appearance s_j and timestamp of i in parent node ts
1: **if** exist a child with label s_j then
2: **foreach** child $node^{l+1}_{s_m}$ contains i **do**
3: $node^{l+1}_{s_m}$.*PUpdate*(i, s_j, i's timestamp in current node);
4: **if** $node^{l+1}_{s_j}$ contains i **then**
5: update *timestamp*(i, $node^{l+1}_{s_j}$) to ts;
6: **else**
7: insert i into $node^{l-1}_{s_j}$ with timestamp ts;
8: **if** $node^{l+1}_{s_j}$ has no child && support($node^{l+1}_{s_j}$) $\geq \varepsilon \cdot N$ **then**
9: Path Extension;

Fig. 4. PAlgorithm.

After the insertion, PAlgorithm will check nodes for the path extension by calling the subroutine *Path Extension*. The subroutine is triggered when the sup_count($node^l_{s_j}$) is no less than $\varepsilon \times n$. Moreover, the height of PTree is at most D and we can avoid the problem of delayed recognition. If the path extension process of a node $node^l_{s_j}$ is triggered, siblings with a support count of no less than ε (including $node^l_{s_j}$) will be identified and extended with each other to form nodes at the next level. According to property 1, the default itemset of $path(node^l_{s_j}) \cdot s_m$ can be assigned. Moreover, according to properties 4 and 5, the item's timestamp can be estimated more precisely.

Before processing the next tuple, *Pruning Process* has two purposes:

(1) to identify which paths an item does contain in the next time window.
(2) to delete the sub-tree rooted at nodes with a support count of less than εn.

An item i would not contains $path(node^l_{s_m})$ in time period of t_{k+1} whenever $timestamp(i, node^l_{s_m}) + D$ is no greater than t_{k+1}. Thus we drop items in nodes if their corresponding timestamp are expired. Moreover, according to property 3, we can delete item i from's sub-tree safely whenever $timestamp(i, node^l_{s_j})$ is expired in $node^l_{s_j}$. The sub-tree rooted at node $node^l_{s_j}$ would be deleted whenever sup_count($node^l_{s_j}$) is less than εn, owing to the path extension of ε-generation.

Upon PAlgorithm, at any time point, we can enumerate the complete set of frequent paths in the current time window by identifying those nodes with a support count that exceeds min_sup × n. The experimental results indicate that the PAlgorithm has dramatic accuracy. Even under a large setting value of ε, the accuracy of the PAlgorithm is also near perfect. However, the increasing space is unsuitable in some circumstances since the PAlgorithm stores all timestamps of items in nodes. To avoid the space overhead of storing timestamps, we propose another algorithm, PSAlgorithm, to enumerate frequent paths with an estimated timestamp. The necessary information is timestamps of items in nodes at level one.

The procedure of PSAlgorithm is almost the same as PAlgorithm. The difference between PAlgorithm and PSAlgorithm is the estimated timestamp, since PSAlgorithm gets an item's timestamp through an estimated method. Therefore, the PTree can be modified as follows: the nodes maintain a set of items instead of items and timestamps except nodes at level one. Because we evaluate an item's timestamp through an estimated method, it results in a loose guarantee of node's support count. The loose guarantee is a tradeoff to meet the compromise between accuracy and space.

Property 6: The $timestamp(i, node^l_{s_j})$ can be estimated as $timestamp(i, node^l_{s_m}) - k + 1$, where $timestamp(i, node^1_{s_m})$ is the minimum value of $\{timestamp(i, node^1_{s_m}) | s_m \in \{path(node^l_{s_j})\}\}$ and k is the last index of s_m in $path(node^l_{s_j})$.

For example, referring to Fig. 5, the $timestamp(A, node^3_{s_2})$ is estimated as 5 − 2 + 1, since the minimum timestamp of item A in $path(node^3_{s_2})$ is five ($timestamp(A, node^1_{s_1})$) and the last index of s_1 in $path(node^3_{s_2})$ is two. The pseudo code of PSAlgorithm is given in Fig. 6. Consequently, whenever updating a tuple, the update process of PSAlgorithm is to recognize and to concatenate an item's path without timestamps assignment and update. The timestamps assignment and update are only involved in nodes at level one.

3.3 The Variation: GAlgorithm

In the previous section, two algorithms are proposed to compute frequent paths over data streams within a sliding window. Upon those algorithms, the paths maintained by Ptree are within the current window. However, people may be interested in other types of frequent paths, for example, mine out the complete set of frequent paths among each path $<s_{t_1}, s_{t_2}, \ldots, s_{t_k}>$ that $t_l - t_{l\text{-}1} \leq$ gap, where the gap is a user specified parameter. In this section, we propose a variational algorithm, GAlgorithm, to deal with the problem of mining frequent paths with a gap limitation.

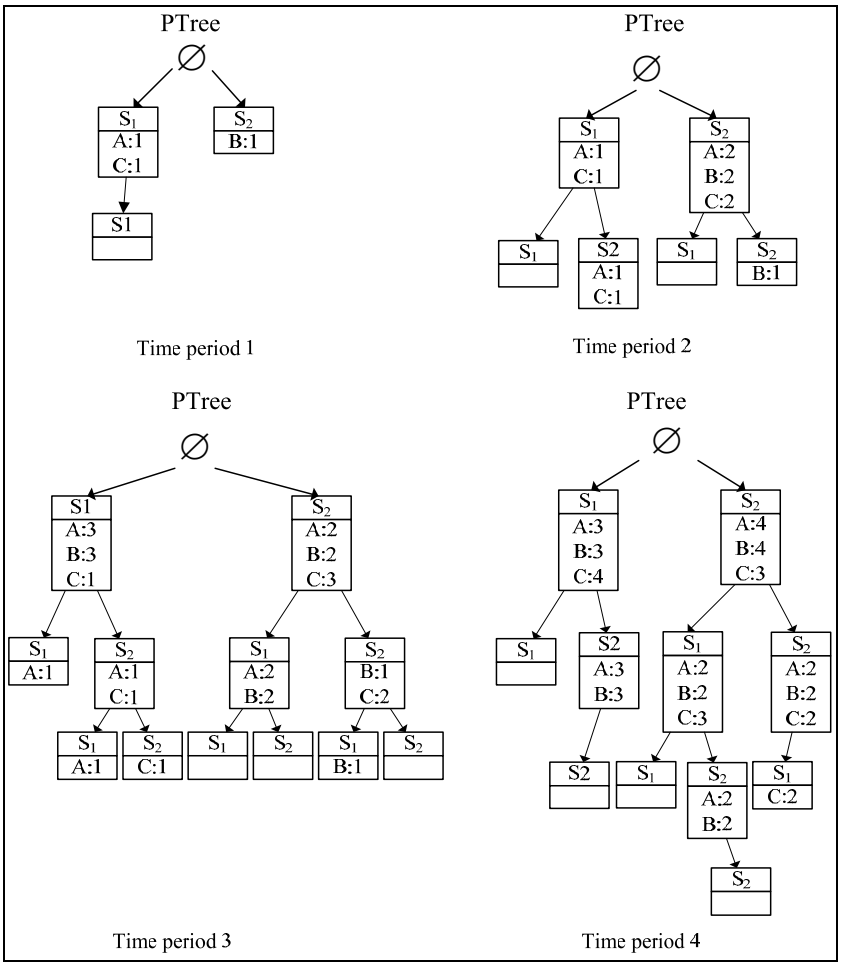


Fig. 5. PTree constructed by PAlgorithm.

PSAlgorithm()
Input: a stream of tuples gathered by sensors, a user-specified min_sup, error parameter ε and window size D.
Output: Frequent path set F.
1: **while**(1) **do**
2: w_{t_k} = tuple of time period t_k
3: **foreach** appearance $e_{s_j t_k} \in w_{t_k}$ **do**
4: **foreach** item $i \in e_{s_j t_k}$ **do**
5: **foreach** node $node^1_{s_m}$ contains i **do**
6: $node^1_{s_m}.PSUpdate(i, s_j)$;
7: **if** $node^1_{s_j}$ contains i **then**
8: update $timestamp(i, node^1_{s_j})$ to t_k ;
9: **else**
10: insert i with timestamp t_k into $node^1_{s_j}$;

11: **if** sup_count($node^1_{s_j}$) ≥ εn **then**
9: PathExtension;
10: Pruning Process;

PSUpdate(i, s_j)
Input: an item ID i and label of current appearance s_j.
1: **foreach** child $node^{l+1}_{s_m}$ contains i **do**
2: $node^{l+1}_{s_m}$.PSUpdate(i, s_j);
3: **if** !($node^{l+1}_{s_j}$ contains i)**then**
4: insert i into $node^{l+1}_{s_j}$;
5: **if** sup_count($node^{l+1}_{s_j}$) ≥ εn **then**
6: path extension;

Fig. 6. PSAlgorithm.

Sensors ID												
	s_1	{A}				{A}	{A}				{A}	
	s_2		{A}									{A}
Time Period		1	2	3	4	5	6	7	8	9	10	11

Fig. 7. Data of sensors gathered in each time period.

We refer to Fig. 7, and assume the value of gap is three. The paths of $path_A$ are $<s_1>$ and $<s_1s_2>$ after time periods 1 and 2 respectively. Accordingly, $path_A$ is $<s_1s_2s_1s_1>$ after time period 6. The $path_A$ would be maintained in PTree until the end of time period 9, since the deviation between time period six and ten exceeds the gap. In this case, the $path_A$ $<s_1s_2s_1s_1>$ would be dropped and refreshed by $<s_1>$. According to the problem, discovering the complete set of frequent paths with gap limitation, we have the following property.

GAlgorithm()
Input: a stream of tuples gathered by sensors, a user-specified min_sup , error parameter ε and gap D.
Output: Frequent path set F.
1: ts = {}; // set of item's timestamp
2: **while**(1) **do**
3: w_{t_k} = tuple of time period t_k
4: **foreach** appearance $e_{s_jt_k} \in w_{t_k}$ **do**
5: **foreach** item $i \in e_{s_jt_k}$ **do**
6: **foreach** node $node^1_{s_m}$ contains i **do**
7: $node^1_{s_m}$.GUpdate(i, s_j);
8: **if** !($node^1_{s_j}$ contains i)**then**
9: insert i into $node^1_{s_j}$;
10: **if** sup_count($node^1_{s_j}$) ≥ εn **then**
11: PathExtension;
12: **if** $i \in ts$ **then**
13: update *timestamp*(i) to t_k;
14: **else**

15: insert *timestamp*(*i*) into *ts* with value of t_k;
16: pruning process;

GUpdate(i, s_j)
Input: an item ID i and label of current appearance s_j.
1: **foreach** child $node_{s_m}^{l+1}$ contains i **do**
2: $node_{s_m}^{l+1}$.GUpdate(i, s_j);
3: **if** !($node_{s_j}^{l+1}$ contains i)**then**
4: insert i into $node_{s_j}^{l+1}$;
5: **if** sup_count($node_{s_j}^{l+1}$) ≥ εn **then**
6: path extension;

Fig. 8. GAlgorithm.

Property 7: The whole $path_i$ of an item i <s_{t_1}, s_{t_2}, …, s_{t_k}> can be dropped whenever t_k + *gap* < current timestamp.

According to Property 7, we drop an item's path as long as the last timestamp of the path satisfies the drop condition. Thus, we monitor an item's path upon one timestamp instead of timestamps in nodes. The pseudo code of GAlgorithm is shown in Fig. 8. Basically, GAlgorithm is similar to the algorithms discussed in the previous section. GAlgorithm enumerates the frequent paths from the PTree and keeps one timestamp for each item. Whenever a path extension is performed, GAlgorithm extends a node $node_{s_m}^l$ with sibling s_j according to Property 1 and Property 5, since there are no more timestamps to perform pruning checks. Finally, in the pruning process, GAlgorithm traverses the PTree in a DFS manner and drops items away from the PTree upon Property 7.

3.4 Optimizations

Basically, our algorithms have three stages: Update, Path Extension and Pruning Process. In this section, we will discuss some optimizations to improve the speed of algorithms.

Update: This stage repeatedly checks children nodes and recursively updates children by calling self-function. According to the algorithms, the timestamp *timestamp*($node_{s_j}^l$) will be updated whenever the label s_j equals the sensor's id of the current appearance. Moreover, the nodes with label s_j of $node_{s_m}^l$'s sub-tree is existent if and only if the support counts of $node_{s_m}^l$'s sibling with label s_j and $node_{s_m}^l$ are exceeding εn. We determine whether the Update of a child is performed or not upon the following property:

Property 8: The update of a child $node_{s_m}^l$ is performed if and only if the support counts of $node_{s_m}^l$ and $node_{s_m}^l$'s sibling with label s_j are exceeding εn, where s_j is the sensor's id of the current appearance.

Path Extension: A Path extension is triggered whenever the support count of a node exceeds εn after the insertion of an item. To avoid redundant checks, we set a node to passive state, when the node has triggered a path extension before. Nodes with a passive state cannot trigger any path extension during later insertions, although they are still in-

volved in path extensions triggered by a sibling. The passive state of a node would be set to an active state whenever its support count is less than εn in that its support count has the possibility to exceed εn in later insertions.

Pruning Process: The pruning process is the bottleneck of our algorithms, since it traverses the whole PTree and checks each timestamp of the items. To reduce the overhead of the pruning process, we traverse the PTree in a DFS manner and employ a top-down policy. During the traversing process, we check the items' timestamps first and prune the items whose timestamps are expired. We do not further prune the node's sub-tree whenever the support count of the nose is less than εn, since its sub-tree would be dropped owing to the characteristics of ε-generation.

4. EXPERIMENTAL RESULTS

The simulation is implemented in JAVA, and performed on dual Xeon 2.80GHz with 4 GB RAM. In the experiments, synthetic datasets and real datasets are used to evaluate the efficiency, accuracy and scalability of our algorithms. We measured the *accuracy*, the number of actual frequent paths as a percentage of the number of the frequent paths mined by our algorithms. Since the PTree constructed by ε-generation is proven to possess no false negative, our three algorithms achieve *recall* of 1.

The synthetic data is generated by an IBM synthetic data generator. Table 1 summarizes the parameters used to generate synthetic datasets. For example, M100N50KTS-400W20 means the dataset is generated with regards to 100 sensors, fifty thousand items, during four hundred time periods and within a window size of 20. We identify that a node is frequent if and only if its support is no smaller than $\lceil \text{min\_sup} \times n \rceil$, where n is the total number of items. We define the total space of an algorithm as the value of total items and counters maintained by the algorithm.

The two real datasets are the MSNBC Anonymous Web Data Set obtained from KDD CUP and the Kosarak click-stream data of a Hungarian on-line news portal gathered from http://fimi.cs.helsinki.fi/data/. Each dataset consists of a collection of sessions where each session has a sequence of page references. The MSNBC dataset has 989818 sessions and the Kosarak has 990002 sessions, with each session containing one to thousands of page references. One difference between two datasets is the number of distinct pages. The MSNBC has only 17 distinct categories of pages, while the Kosarak has 41270 distinct pages. The parameter settings of the real datasets are summarized in Table 2.

Table 1. Parameter settings of synthetic datasets.

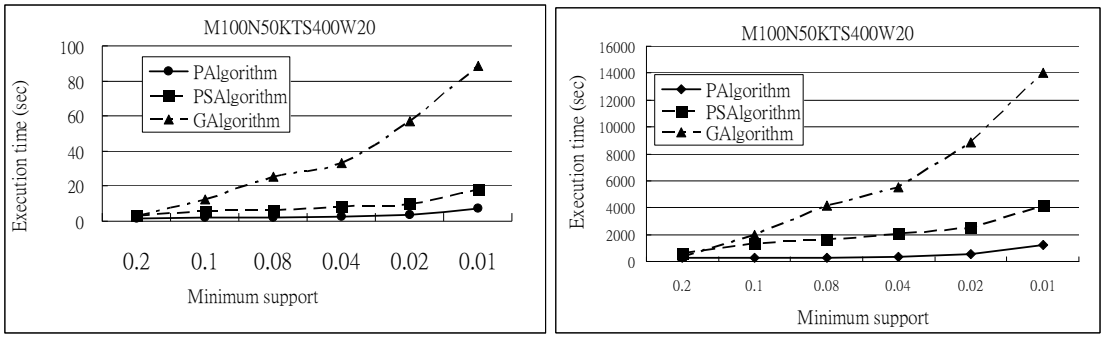
Notation	Meaning	Default	Range
M	Number of sensors	100	10~200
N	Number of items	50K	10K~200K
TS	Number of time periods	400	–
W	Window size (time period)	20	10~40
G	GAP size	5	3~7
min_sup	Minimum support threshold	0.04	0.01~0.2
ε	Error parameter	0.8 × min_sup	0.2~1

Table 2. Parameter settings of real datasets.

Notation	Meaning	Range
M	Number of pages	17 and 41270
N	Number of users	10K~800K
TS	Number of time periods	300
W	Window size (time period)	20
G	Gap size	5
min_sup	Minimum support threshold	0.04
ε	Error parameter	0.5 × min_sup

4.1 Synthetic Dataset

In the first simulation, the relationship between execution time and minimum support is conducted. As shown in Fig. 9, the execution time grows as the minimum support decreases. Clearly, in Fig. 9 (a), the execution time of PAlgorithm increases slightly in the range from min_sup = 0.2 to 0.02. Moreover, PAlgorithm outperforms PSAlgorithm in both datasets. However, the execution time of PSAlgorithm increases significantly as minimum support decreases, since the estimation approach is time consumed. Therefore, the execution time of the PAlgorithm is a quarter of that of the PSAlgorithm. Because the gap limitation has the possibility to form long paths, the execution time is much higher than the others.

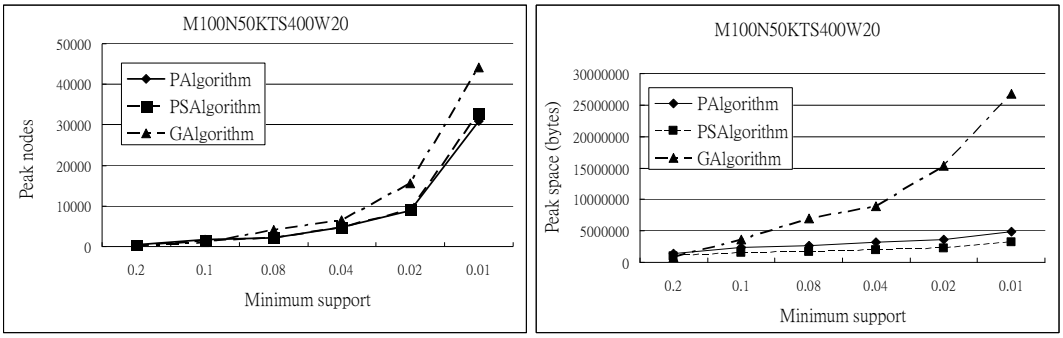


(a) (b)

Fig. 9. Execution time: (a) peak execution time of time periods; (b) total execution time.

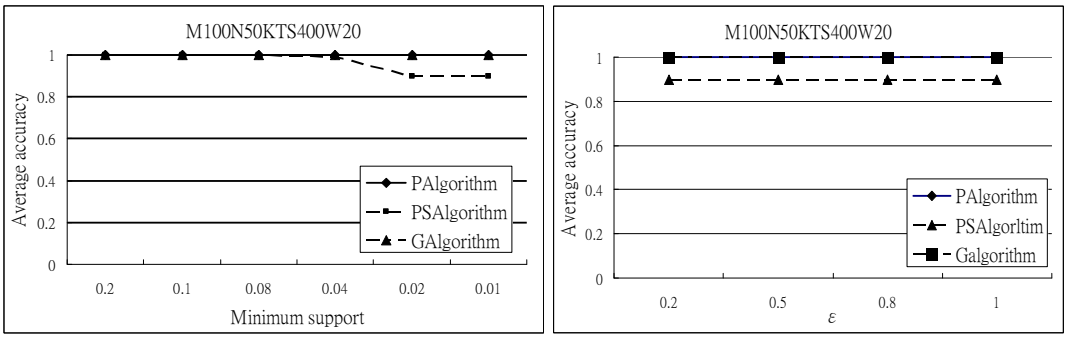
Fig. 10 shows the maximum nodes and space during the mining process maintained by PAlgorithm, PSAlgorithm and GAlgorithm. Note that the maximum nodes maintained by PSAlgorithm almost keep up with PAlgorithm as the minimum support threshold is decreasing, although PSAlgorithm has a loose support count guarantee. In Fig. 10 (b), the maximum space maintained by the PSAlgorithm is two thirds of the PAlgorithm, since the maximum counters maintained by the PSAlgorithm are steadily bounded. The maximum space and nodes maintained by GAlgorithm are large, because the length of the paths may be long.

Fig. 11 (a) shows the average accuracies of frequent algorithm mined temporal paths over data streams. The average accuracy of the PAlgorithm is admirably perfect,



(a) (b)

Fig. 10. Space usage: (a) nodes; (b) bytes.



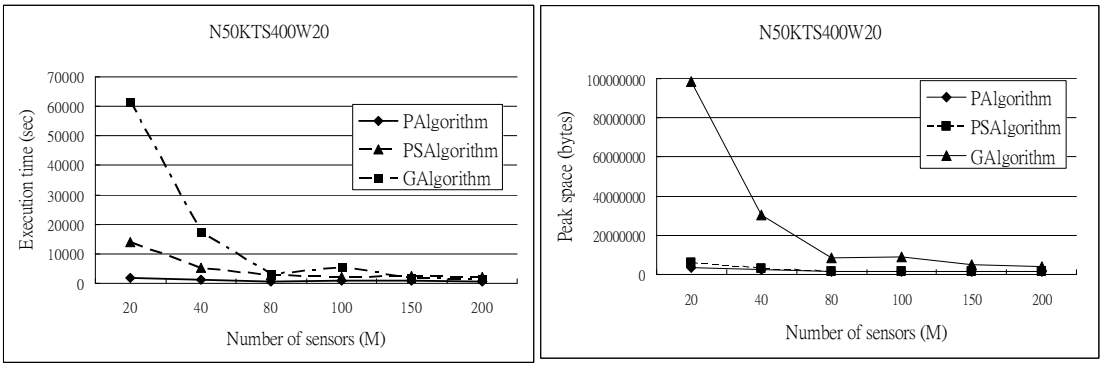
(a) (b)

Fig. 11. Average accuracy: (a) min_sup; (b) ε.

since it nearly stores every timestamp of an item. The PSAlgorithm shows its exceptional competence with regard to memory usage and accuracy. However, the accuracy and speed of the PSAlgorithm is influenced by the distribution of the dataset. We will explain how the distribution of the dataset affects the PSAlgorithm in a later paragraph. The accuracy of the GAlgorithm is perfect since its timestamps of each item always match the real timestamp. Referring to Fig. 11 (b), the effect of ε is shown. As shown in the result, the accuracy is very stable as the error parameter increases. The stable curve of the PSAlgorithm shows that the PSAlgorithm has competitive behavior under situations of silent distribution.

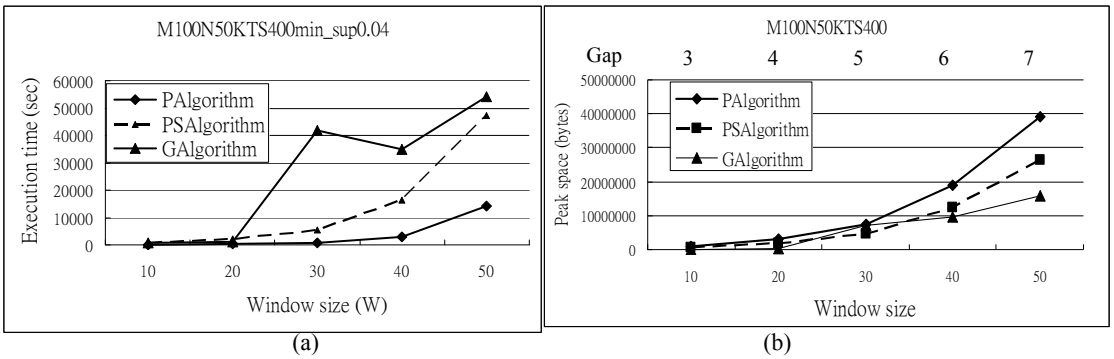
Referring to Fig. 12, the total execution time and peak space decreases as the number of sensors increases. The reason is that the shape of PTrees will become thin and high as the number of sensors decreases. As a result, the probability of timestamps getting updated becomes higher. On the other hand, the support count of a node is easier to exceed εn; therefore, we need to maintain more items in PTree and to check the pruning process. Moreover, the paths reported by GAlgorithm would be much longer when the number of sensors is small as shown in Fig. 12 (b).

Fig. 13 (a) shows the total execution times of algorithms under different window sizes. It is clear that the total execution times increase as the window or gap increases. Moreover, the peak spaces increase as the window size increases.



(a) (b)

Fig. 12. Different number of sensors (a) total execution time; (b) peak space.



(a) (b)

Fig. 13. Different window size (a) total execution time; (b) peak space.

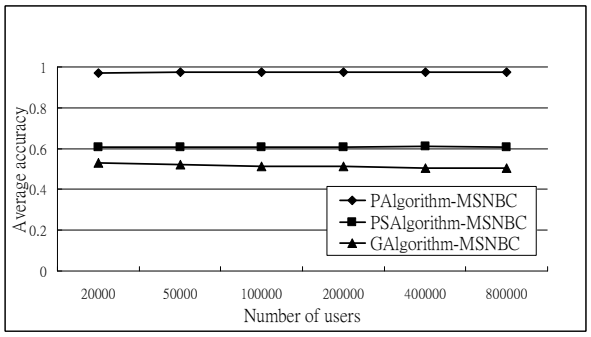
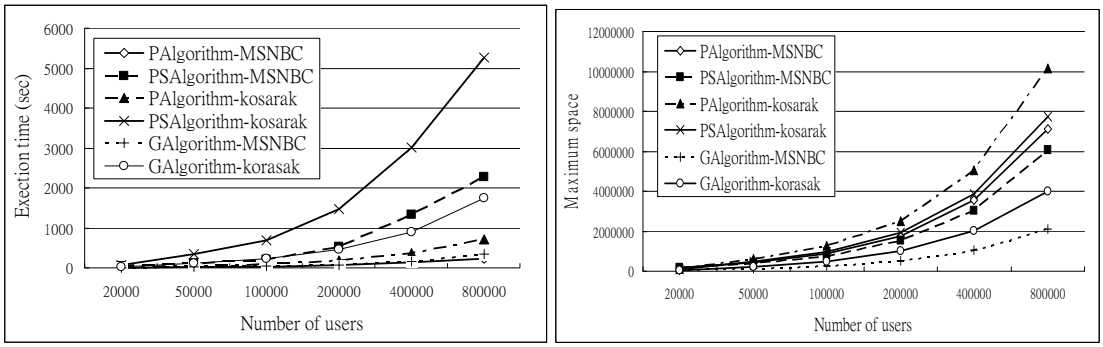


Fig. 14. Mining accuracy on MSNBC.

4.2 Real Dataset

Fig. 14 shows the average accuracy of the PAlgorithm and PSAlgorithm. As shown in the results, the accuracy of PSAlgorithm is not as high as shown in Figs. 11 (a) and (b). This is because the number of sensors in MSNBC is much smaller than that of dataset

$N_{100}C_{50K}TS_{400}W_{20}\varepsilon_{0.8}$, which increases the probability of timestamps getting updated. As a result, the estimated timestamp tends to be overestimated, and decreases the accuracy of the PSAlgorithm.



(a) (b)

Fig. 15. Simulations on two real dataset: (a) execution time; (b) maximum space.

The execution times for the two real datasets are shown in Fig. 15 (a). As shown in the result, the execution times of both algorithms increase as the number of users increase. Moreover, the PAlgorithm performs better than the PSAlgorithm in the two different real datasets. Fig. 15 (b) shows the maximum spaces for two algorithms.

5. CONCLUSIONS

In this paper, the problem of mining temporal frequent paths over data streams is introduced. Furthermore, a different type of problem, mining frequent paths with a gap limitation over data streams, is also discussed. A data structure, PTree, is introduced to maintain the possible paths of users. Based upon the ε-generation technique, the PTree is guaranteed to achieve a recall of 1 and support count with error bounded. Using the PTree, three algorithms are proposed to deal with these problems of frequent path mining. According to the PTtree, some pruning mechanisms are proposed to reduce an unexpected error whenever a path candidate is generated. PAlgorithm provides an efficient way for mining results with dramatic accuracy, owing to the error bounded PTree. To further ease the maintenance overhead, the PSAlgorithm is proposed to mine frequent paths with estimated timestamp. The Experiment's results indicate that the PSAlgorithm can achieve an acceptable behavior of accuracy through the estimation. The third algorithm, GAlgorithm, shows that the PTree can be utilized to cope with different problems without modification. The PTree exhibits an ability to handle a diversity of problems.

REFERENCES

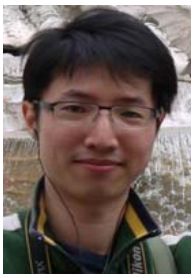
1. R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of ACM International Conference on Data Engineering*, 1995, pp. 3-14.

2. Y. C. Chen and G. Lee, "Mining sequential association rules efficiently by using prefix projected databases," *Special Issue on Design and Mining for Social-Aware Services, Journal of Computers*, Vol. 22, 2011, pp. 33-47.
3. G. Chen, X. Wu, and X. Zhu, "Sequential pattern mining in multiple streams," in *Proceedings of IEEE International Conference on Data Mining*, 2005, pp. 27-30.
4. M. Chen, J. S. Park, and P. S. Yu, "Efficient data mining for path traversal patterns," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, 1998, pp. 209-221.
5. L. Chang, T. Wang, D. Yang, and H. Luan, "SeqStream: Mining closed sequential patterns over stream sliding windows," in *Proceedings of the 8th IEEE International Conference on Data Mining*, 2008, pp. 83-92.
6. B. R. Dai, H. L. Jiang, and C. H. Chung, "Mining top-K sequential patterns in the data stream environment," in *Proceedings of International Conference on Technologies and Applications of Artificial Intelligence*, 2010, pp. 142-149.
7. M. El-Sayed, C. Ruiz, and E. A. Rundensteiner, "FS-Miner: Efficient and incremental mining of frequent sequence patterns in web logs," in *Proceedings of ACM International Workshop on Web Information and Data Management*, 2004, pp. 128-135.
8. Y. H. Hu, Y. L. Chen, and K. Tang, "Mining sequential patterns in the B2B environment," *Journal of Information Science*, Vol. 35, 2009, pp. 677-694.
9. H. Kum, J. Pei, W. Wang, and D. Duncan, "ApproxMAP: Approximate mining of consensus sequential patterns," in *Proceedings of SIAM International Conference on Data Mining*, 2003, pp. 311-315.
10. H. T. Lam, F. Moerchen, D. Fradkin, and T. Calders, "Mining compressing sequential patterns," in *Proceedings of the SIAM 12th International Conference on Data Mining*, 2012, pp. 319-330.
11. G. Lee, K. C. Hung, and Y. C. Chen, "Path tree: Mining sequential patterns efficiently in data streams environments," in *Proceedings of International Computer Symposium Workshop on Database*, *Data Mining*, *and Information Retrieval*, 2013, pp. 261-268.
12. H. Li, S. Lee, and M. Shan, "On mining webclick streams for path traversal patterns," in *Proceedings of ACM International World Wide Web Conference on Alternate*, Track Papers and Posters, 2004, pp. 404-405.
13. H. Li, S. Lee, and M. Shan, "DSM-TKP: Mining top-*K* path traversal patterns over web click-streams," in *Proceedings of IEEE/WIC/ACM International Conference on Web Intelligence*, 2005, pp. 326-329.
14. S. Muthukrishnan, "Data streams: Algorithms and applications," in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2003, pp. 413-413.
15. J. Pei, J. Han, B. Mortazavi-Asi, J. Wang, H. Pinto, Q. Chen, and M. Hsu, "Mining sequential patterns by pattern-growth: The PrefixSpan approach," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, 2004, pp. 1424-1440.
16. S. Y. Yang, C. M. Chao, P. Z. Chen, and C. H. Sun, "Incremental mining of across-treams sequential patterns in multiple data streams," *Journal of Computers*, Vol. 6, 2011, pp. 449-457.
17. S. Y. Yang, C. M. Chao, P. Z. Chen, and C. H. Sun, "Incremental mining of closed sequential patterns in multiple data streams," *Journal of Networks*, Vol. 6, 2011, pp.

728-735.
18. C. Yu and Y. Chen, "Mining sequential patterns from multidimensional sequence data," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, 2005, pp. 136-140.
19. Z. Zhao, D. Yan, and W. Ng, "Mining probabilistically frequent sequential patterns in uncertain databases," in *Proceedings of the ACM 15th International Conference on Extending Database Technology*, 2012, pp. 74-85.

Guanling Lee received the B.S., M.S., and Ph.Dd degrees, all in Computer Science, from National Tsing Hua University, Taiwan, in 1995, 1997, and 2001, respectively. She joined National Dong Hwa University. Taiwan, as an Assistant Professor in the Department of Computer Science and Information Engineering in August 2001, and became an Associate Professor in 2005. Her research interests include resource management in the mobile environment, data scheduling on wireless channels, search in the P2P network and data mining.

Yi-Chun Chen received the B.S. degree in Applied Mathematics from Feng Chia University, Taichung, Taiwan, in 2005 and the M.S. degree in computer science from National Dong Hwa University, Hualien, Taiwan, in 2007. He is currently pursuing the Ph.D. degree in the same department. His research interests include data mining and search in the P2P network.



Kuo-Che Hung received the B.S. and M.S. degree, all in Computer Science, from National Dong Hwa University, Taiwan, in 2004 and 2006, respectively. He is a Software Engineer at HTC Corporation, Taipei, Taiwan. His research interests include data mining and database.