International Journal of Innovative
Computing, Information and Control
Volume **10**, Number **3**, June **2014**

ICIC International ©2014 ISSN 1349-4198
pp. **933-946**

# CSPMDS-PREFIXSPAN: MINING CLOSED SEQUENTIAL PATTERNS OVER DATA STREAM SLIDING WINDOWS

Qiang Zeng[1,2], Gaowei Han[1,2], Dengxi Chen[1,2] and Jiadong Ren[1,2]

[1]College of Information Science and Engineering
Yanshan University
No. 438, Hebei Ave., Qinhuangdao 066004, P. R. China
{ zq80000; hangaowei2007 }@126.com; chenxi.1988.8.16@163.com; jdren@ysu.edu.cn

[2]The Key Laboratory for Computer Virtual Technology and System Integration of Hebei Province
Qinhuangdao 066004, P. R. China

Received May 2013; revised September 2013

Abstract. *Previous studies indicate incremental methods were not used sufficiently and data structures always consumed large memories and long execution time because of lack of pruning method in data stream algorithms. It has shown that obtaining closed patterns always need long time because of pattern comparison among all mining results. So a closed sequential pattern algorithm called CSPMDS-PrefixSpan based on sliding windows over data stream is presented in this paper. The sliding window is separated into several fragments to be mined by PrefixSpan algorithm respectively and the incremental method is developed. PFPS-Tree, FCPS-Tree and PatternTable are defined as novel summary data structures to store the potential frequent sequential patterns effectively. Using PatternTable, it assists in updating the potential frequent patterns in the PFPS-Tree and uses previous mining results effectively. And we prune the PFPS-Tree using the proposed closed detection pruning and window support threshold pruning to form a closed tree FCPS-Tree and output the closed patterns easily. The experiments show that CSPMDS-PrefixSpan algorithm has a good secondary support threshold pruning and closed detection performance. Under the same conditions, it overcomes the memory limit and the execution time is reduced compared with some previous algorithms.*

**Keywords:** Data stream, Sliding window, Closed sequential pattern, *FPS-Tree*, *PatternTable*

1. **Introduction.** The sequential pattern is used widely in the data analysis of DNA, weather, stock exchange and even software vulnerabilities [1,2]. Recently, the research on sequential pattern has been focused on the improvement of the algorithm performance. Optimizing data structures, improving pruning methods and the closed and incremental mining are main directions to optimize data stream algorithms [3,4].

The concept of sequential pattern [5] was first proposed for the analysis of supermarket shopping basket. After this, a lot of sequential pattern algorithms were proposed [6,7]. The PrefixSpan algorithm [8] was presented to solve the problem of generating a lot of projection databases. It does not take into account all possible frequent sub-sequences to project but constructs the projection databases based on frequent prefixes. With the development of computer about CPU and memory, the algorithm MEMISP [9] based on memory indexing approach was presented. It only needs to scan the database once and finds the patterns by indexing, which outperforms the PrefixSpan. However, if the sequences are long or the support threshold is low, PrefixSpan and MEMISP have low efficiency. In this case, mining closed sequential pattern is a good choice. Recently, Gomariz et al. put forward a closed sequential pattern mining algorithm ClaSP [10]. It uses

an effective space search and pattern pruning method as well as a vertical data structure, which proves that it has higher performances compared with CloSpan [11]. In addition, Kandpal and Agnihotril advanced a new closed sequential pattern algorithm SBLOCK [12] from another angle based on the concept that larger memory block can absorb the smaller memory blocks with the same type, abandoning the previous time-consuming method of candidate set generation and maintenance.

However, the above algorithms cannot be used for the mining of sequences in data stream directly. The reasons can be summarized in three aspects. They generally need to scan the database repeatedly; their data structures can consume too many memories and their execution time tends to be long when processing the large data sets, which cannot meet the requirements of real time processing in data streams; what is more, their patterns obtained are extremely large, which cannot be analyzed better. The use of data stream model can overcome some drawbacks above, which appeals to a lot of researches.

The SPEED [13] algorithm is the first algorithm to mine sequential pattern in data streams which is based on sliding windows. It uses a tree called *treereg* to store summary information and introduces the concept of region to update *treereg*. However, it can only process single items and stores all the arrival sequences in the tree before mining the sequences, which needs a lot of memories. In order to overcome these drawbacks, SeqStream [14] algorithm mines closed sequential patterns. It designs an inverse closed sequence tree to store closed sequential patterns decreasing the memory usage. After this, Yang et al. [15] designed an incremental algorithm IAspam in across-streams. It converts the data into bitmap representation to process the computation and overcomes the drawback of only handling single items. In addition, ICspan algorithm [16] is another algorithm developed by Yang et al. to mine closed sequential patterns. It adds to incremental method by extending MILE algorithm. Referring to these two algorithms, Lee et al. proposed PAlgorithm and PSAlgorithm [17]. They use a path tree to integrate the partial mining results efficiently. IncSPAM [18] extends item-sequences to itemset-sequences. It uses a bit-sequence representation of items to reduce the usage of memory and time. Inspired by this algorithm, Guyet and Quiniou [19] put forward a sequence pattern mining algorithm about incremental item set flows over sliding windows to avoid duplicate mining by expanding the PSP algorithm. It updates the tree structure with the incremental method and reduces the computing time of support significantly. However, its incremental particle is as small as an item set, which is inefficient because of frequent executing of the algorithm.

The related algorithms are more than what we have described above; they all focus on optimizing of data structures and pruning methods. Tanbeer et al. [20] developed a structure called CPS-Tree and introduced the concept of dynamic tree restructuring to produce a highly compact frequency-descending tree structure at runtime. SS-BE and SS-MB algorithm [21] discussed the problem of mining with precise error bounds. Both algorithms scale linearly in execution time as the number of sequences grows. SS-BE2, SS-LC and SS-LC2 [22] proposed by Koper and Nguyen are the extensions of SS-BE. They improved the tree pruning method to guarantee the high completeness and correctness of the results. SPAMS algorithm [23] uses an automaton-based structure and can output constantly on any user's specified thresholds. In addition, it took into account the characteristics of data streams and proposed a well-suited method to provide near and optimal results with a high probability. Top-DSW algorithm [24] is a research on web usage mining. It mines top-k sequential patterns over stream damped sliding windows. A structure TKP-DSW-list was developed and a pruning mechanism called TKP-list-Maintain was used to improve performances of the algorithm.

However, the insufficiently incremental methods make the algorithms execute inefficiently when faced with data streams. The pattern tree consumes large memories since they prune the candidate patterns by the support threshold only once. They tend to store the positions of the items in sequences into tree structure, so it often costs a lot of time to search for the tree nodes when deleting the old patterns. Therefore, a new sequential pattern algorithm CSPMDS-PrefixSpan over data stream is proposed to overcome these drawbacks. The major contributions of this study can be summarized as follows.

Firstly, the sliding window is defined and separated into several fragments; the incremental particle is increased to several stream fragments.

Secondly, three pruning methods of fragment support threshold, window support threshold and closed detecting are defined to prune the mining result.

Thirdly, we propose a data structure *PatternTable* to store the positions of patterns and to assist in searching for the tree nodes.

Fourthly, we only store the frequent patterns in the *PFPS-Tree*, which saves memories.

Lastly, a mechanism is proposed to get the closed patterns easily from the *PFPS-Tree*, whose time consumption is low.

The remaining paper is organized as follows. Section 2 gives the definitions and descriptions of problems. The algorithm and its complexity analysis are given in Section 3. Section 4 gives the experiments to analyze performances of the algorithm comparatively. Section 5 concludes the paper.

## 2. Definitions and Descriptions of Problems.

2.1. **Basic concepts.** The sliding window is defined as $W$ (*wid*, $|w|$), where *wid* stands for serial number of a window and $|w|$ represents the size of the window. $|w|$ is equal to the number of fragments contained within the window. The fragment is expressed as $P$ (*wid*, *pid*, $|p|$), in which *wid* means the serial number of window that contains the fragment, *pid* is the serial number of the fragment within the sliding window and $|p|$ represents the size of the fragment. $|p|$ is equal to the number of sequences in the fragment, then $0 < n \times |p| \leq |w|$ ($n$ is a positive integer). The fragment length is the number of time points it contains, represented as $L$ uniformly. In a sliding window, the *pid* equals 1 whose fragment is the furthest from the current time, on the contrary, the pid equals $|w|$. The size of each slide is the length of several fragments. The symbol *MIN_SUP* is used as the support threshold of the window, and *min_sup* as the support threshold of the fragment uniformly.

Generally, the sequence $S$ can be recorded as $\langle SID, e_1, e_2, e_3, e_4, \ldots, e_i \ldots \rangle$, where $e_k$ is expressed as $e_k$ $(ie_k, t_k)$, in which $ie_k$ is an item set or an event, $t_k$ is the occurring time point. *SID* represents the serial number of a sequence. The sequence can be abbreviated as $S = \langle e_1, e_2, e_3, e_4, \ldots, e_i \rangle$ ($i$ and $k$ are both positive integers). If a sequence contains $i$ elements, it is called *i-sequence*. In the same way, *1-pattern* is a sequential pattern with one element. A frequent sequential pattern is closed if there does not exist a frequent super sequence with the same support. The sequence in a fragment is expressed as $seq_{number}$ $(0 < number \leq |p|)$. If the sequence occurs in the nearest time point in the fragment, *SID* equals 1, on the contrary, *SID* equals $|p|$. Table 1 shows an example of sequences with time point. Assuming that $L = 6$, these sequences in data streams can be seen in Figure 1.

**Definition 2.1.** *Support*$(s)_p$. *In a fragment, the support count of sequence s is the number of sequences that contain s, which is expressed as* $P_i$ $(seq_{number})$, *wherein i indicates the serial number of the fragment. The ratio of support count of sequence s with the fragment size is support* $(s)_p$ *and formally expressed as follows.*

$$support(s)_p = \{S | s \subseteq S \wedge S \in P \wedge P \in W\}/|P| \tag{1}$$

TABLE 1. A sequence database instance

| *SID* | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ | $t_9$ | $t_{10}$ | $t_{11}$ | $t_{12}$ | $t_{13}$ | $t_{14}$ | $t_{15}$ | $t_{16}$ | $t_{17}$ | $t_{18}$ | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $a$ | $abc$ | $ac$ | $d$ | $cf$ | | $ef$ | $ab$ | $df$ | $c$ | $b$ | | $f$ | $db$ | $df$ | $d$ | $g$ | | |
| 2 | $ad$ | $c$ | $bc$ | $ae$ | | | $e$ | $g$ | $af$ | $c$ | $b$ | $c$ | $c$ | $gc$ | $f$ | $c$ | $b$ | $e$ | |
| 3 | $d$ | $ef$ | $ac$ | $g$ | $f$ | $a$ | $a$ | $bg$ | $cf$ | $eg$ | $ab$ | $d$ | $e$ | $f$ | $df$ | $g$ | $cg$ | $f$ | |

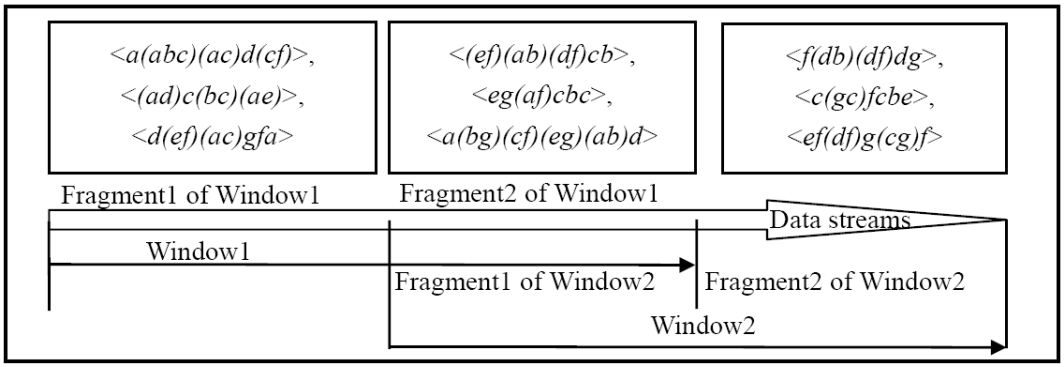


FIGURE 1. The sequences in data streams

**Definition 2.2.** *Potential frequent sequential pattern. If* $support(s)_p \geq min\_sup$*, the sequence* $s$ *is called a potential frequent sequential pattern.*

**Definition 2.3.** *Support*$(s)_w$*. Assuming* $P_i(pfsp_i)$ *represents the potential frequent sequential pattern that contains sequence* $s$ *and is obtained from fragment* $i$ *(*$i$ *is a positive integer), the ratio of the number of potential frequent sequential patterns that contain* $s$ *with the number of all the sequences in the window, is support*$(s)_w$*. It is formally represented below.*

$$support(s)_w = (P_1(pfsp_1) + P_2(pfsp_2) + \ldots + P_{|p|}(pfsp_{|p|}))/(|w| * |p|) \tag{2}$$

**Definition 2.4.** *Frequent sequential pattern. A sequence* $s$ *is called a frequent sequential pattern if* $support(s)_w \geq MIN\_SUP$*.*

2.2. **Data structures.** What follow are four summary data structures proposed in the algorithm. The first is a table called *PatternTable*. The second is a potential frequent prefix sequence tree named *PFPS-Tree*. The third is a frequent prefix sequence tree shorted by *FPS-Tree*. The forth is a frequent closed prefix sequence tree signified by *FCPS-Tree*.

**Definition 2.5.** *PatternTable. The columns of the table are named batch, pattern and count, representing the serial number of a fragment, sequential pattern and support count of the pattern respectively. The PatternTable is used to store potential frequent sequential patterns and assist in updating PFPS-Tree effectively.*

**Definition 2.6.** *Old pattern. A pattern is an old pattern, if its batch equals the serial number of the fragment to be deleted. Old pattern is a collection of these patterns.*

**Definition 2.7.** *New pattern. We mine the patterns in the new arrival fragments; new pattern is a collection of these patterns.*

**Definition 2.8.** *PFPS-Tree. The potential frequent sequential patterns are stored in the PFPS-Tree, which is a multi-tree. The node of the tree contains three regions, item, count and node-link. They mean sequence pattern, sequence pattern support count and pointer*

*pointing to its child node respectively. The root node of the tree is named by "root". In the PFPS-Tree, except the root node, item of a node is a sequential pattern, and the item of parent node is the longest prefix of each item in its child nodes.*

*FPS-Tree* and *FCPS-Tree* have the same internal structures with the *PFPS-Tree*. If we remove all the nodes whose supports are smaller than *MIN_SUP* from the *PFPS-Tree*, we will get an *FPS-Tree*. If any node whose support is equal to its any parent node support is deleted from the *FPS-Tree*, an *FCPS-Tree* is obtained. There exit two conversions, from *PFPS-Tree* to *FPS-Tree* and from *FPS-Tree* to *FCPS-Tree*, which contain two pruning methods. Both *PFPS-Tree* and *FCPS-Tree* have a property.

***MIN_SUP pruning.*** *If the node whose support is less than MIN_SUP in the PFPS-Tree is found, we delete this node and its descendant nodes recursively.*

***Closed detection pruning.*** *If the node whose count equals its parent node count in the FPS-Tree is found, the pattern of its parent node is not closed. Then we delete this node and its descendant nodes recursively to* guarantee that the pattern of its parent node is closed *in the tree.*

**Property 2.1.** *The child node pattern support is not greater than the support of its parent node pattern in the PFPS-Tree.*

**Proof:** Assume that the child node pattern is $s_1$ and the parent node pattern is $s_2$. According to Formula (1),

$$support(s_1)_p = \{S|s_1 \subseteq S \wedge S \in P \wedge P \in W\}/|p|$$

and

$$support(s_2)_p = \{S|s_2 \subseteq S \wedge S \in P \wedge P \in W\}/|p|$$

are gotten. Because $s_2$ is the longest prefix of $s_1$ in the *PFPS-Tree*, there will be $s_2 \subseteq s_1$. So $support(s_1)_p \leq support(s_2)_p$.

**Property 2.2.** *Except the root node in the FCPS-Tree, the counts of nodes in any path from the root to the leaf become a decreasing trend.*

**Proof:** Assume the support of the parent node is $\text{support}_1$, the support of its child node $\text{childnode}_1$ is $\text{support}_2$. And the $\text{childnode}_2$ is a child node of $\text{childnode}_1$, its support is $\text{support}_3$. Since the pattern in *FCPS-Tree* is closed, there must be $\text{support}_1 > \text{support}_2$ and $\text{support}_2 > \text{support}_3$, so it will be $\text{support}_1 > \text{support}_2 > \text{support}_3$, which is a decreasing trend.

3. **CSPMDS-PrefixSpan Algorithm.** In this section, the algorithm CSPMDS-PrefixSpan (Closed Sequential Pattern Mining in Data Streams extended by PrefixSpan) is proposed, which contains sub-algorithms named by Build-FPS-Tree and Update-FPS-Tree.

3.1. **The novel ideas of incremental mining and obtaining closed patterns.** The window is divided into several fragments. The fragments are mined by PrefixSpan algorithm respectively and we store mining results into the *PFPS-Tree* and the *PatternTable*. When the number of fragments that have been mined reaches to the capacity of the window, CSPMDS-PrefixSpan uses the *MIN_SUP pruning* and *closed detection pruning* to prune the *PFPS-Tree*, and then an *FCPS-Tree* is formed, which contains shortly closed patterns. The execution of the two pruning methods is in Algorithm 3.2.3. The good pruning performance can be seen from Figure 2 to Figure 3.

The incremental methods are novel and the incremental particle is suitable. When the new fragments arrive, firstly, the old patterns in the *FCPS-Tree* are deleted by the

assisting of *the PatternTable*, which can be seen in Figure 4, and then we remove the old patterns from the *PatternTable*; secondly, the new patterns are added to the *PFPS-Tree* and the *PatternTable*. When the number of fragments reaches to the capacity of the window again, the *PFPS-Tree* is pruned and an *FCPS-Tree* is gotten in the same way. After these, we mine like the above continuously. Algorithm 3.3.1 describes the execution of incremental mining.

3.2. **Build-FPS-Tree.**

**Algorithm 3.2.1.** Build-FPS-Tree(Sequences, $|p|$, $|w|$, *min_sup*, *PFPS-Tree*).

Input: Sequences of the fragment, $|p|$, $|w|$, *min_sup*, *PFPS-Tree*.

Output: *FCPS-Tree*.

Build-FPS-Tree(Sequences, $|p|$, $|w|$, *min_sup*, *PFPS-Tree*)

{

(1) $n = 0$; //$n$ represents the number of sequences that have arrived

(2) while ($n < |w|$){

(3) for ($i = 0$; $i < |w|$; $i = i + m * |p|$) {//$m$ is an positive integer

(4) FS = PrefixSpan(Sequences);//FS is a frequent sequence

(5) CreatePatternTable(FS); // add frequent sequences to PatternTable

(6) *PFPS-Tree* = Insert(FS, *PFPS-Tree*); // insert patterns to PFPS-Tree

}

(7) $n$ + +;

}

(8) If ($n == |w|$){

(9) *FCPS-Tree* = Traverse(*PFPS-Tree*);

}

}

Build-FPS-Tree algorithm contains four parts, mining in fragments, setting up *PatternTable*, building *PFPS-Tree* and transforming *PFPS-Tree* into *FCPS-Tree*. The algorithm contains two detailed algorithms, Insert ($FS_i$, *PFPS-Tree*) and Traverse (*PFPS-tree*). The former is to insert potential frequent patterns into *PFPS-Tree* and the latter is to transform the *PFPS-Tree* into *FCPS-Tree*.

**Example 3.1.** *For the instance streams in Figure 1, assume min_sup = 50%. We mine two fragments respectively in window 1, and then form the PatternTable with the potential frequent sequential patterns shown as Table 2.*

Table 2. *PatternTable* (some columns are not shown)

| *batch* | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| *pattern* | <a> | <b> | <c> | <d> | <e> | <f> | <ab> | <(ac)> | <(bc)> |
| *count* | 3 | 2 | 3 | 3 | 2 | 2 | 2 | 2 | 2 |
| *batch* | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| *pattern* | <cf> | <a> | <b> | <c> | <e> | <f> | <g> | <ab> | <gc> |
| *count* | 2 | 3 | 3 | 3 | 3 | 3 | 2 | 3 | 2 |

**Algorithm 3.2.2.** Insert ($FS_i$, *PFPS-Tree*).

Input: $FS_i$, $FS_i$ is $i$-th frequent sequence; empty *PFPS-Tree*.

Output: *PFPS-Tree*.

Insert ($FS_i$, *PFPS-Tree*)

{

```
(1) If (PFPS-Tree == NULL)
      {
(2)     CreateTree(root);//build a tree with a node named "root"
(3)     PFPS-Tree.getroot().Insert (FS_i, count, node-link);
        }
(4)     else
          {
(5)         If (Same(FS_i, PFPS-Tree.node.Item))
            {
(6)             node.count = node.count + FS_i.count;
            }
(7)          else if (prefix(FS_i, PFPS-Tree.node.Item))
               {
(8)              node .Insert(FS_i, count, node-link);
               }
(9)              else
                 {
(10)              PFPS-Tree.getroot().Insert(FS_i, count, node-link);
                 }
          }
}
```

In the inserting process, if the tree is empty, we create a tree with the root named "root", then the first pattern is linked to the root node directly. If the tree is not empty, and if a node pattern the same with the inserted pattern can be searched for, we increase the node count directly. If the same node pattern cannot be found, we will else search for the longest prefix node pattern in the tree, and link the inserted pattern to the prefix node. If the same node and the prefix node of the inserted pattern cannot be found, the pattern has to be linked to the root.

**Example 3.2.** *For instance streams in Figure 1, assume min_sup = 50%. Then, we find frequent patterns in the fragment 1 and fragment 2 of window 1 respectively, and build PFPS-Tree as shown in Figure 2.*

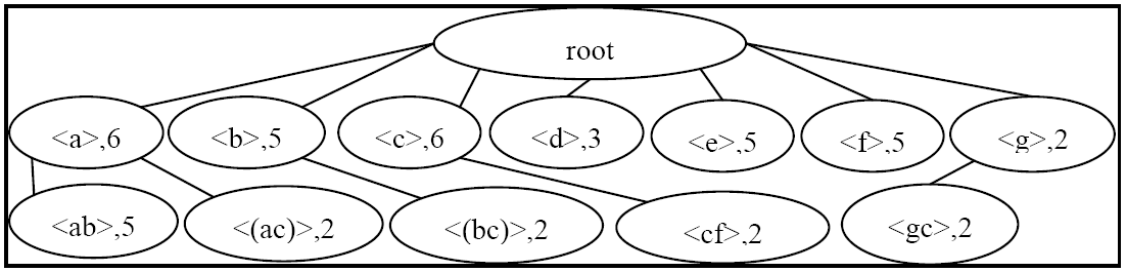


FIGURE 2. *PFPS-Tree* (some of tree nodes are not shown)

The shortly closed patterns can be easily obtained by comparing the *count* of parent node with its child nodes, which is a novel method to get closed patterns. The pattern need not compare with all mining results but only with its child nodes, which is high efficient. By two pruning methods, a lot of redundant patterns are removed, which is friendly with the memory and execution time. The detail is in Algorithm 3.2.3.

**Algorithm 3.2.3.** Traverse(*PFPS-Tree*).

Input: *PFPS-Tree*.

```
Output: FCPS-Tree.
Traverse(PFPS-Tree)
{
(1) for (each node in the PFPS-Tree except the root)
    {
(2)     if (node.count == node.node-link.count)
          {
(3)        delete (node)
                {
(4)             delete node.childnode; node = node.node-link;
                 }//if the node is not closed, delete the child nodes recursively
          }
(5)     if ((node.count/(|w|*|p|)) <MIN_SUP)
          {
(6)        delete (node) {
(7)                        delete node; node = node. node-link;
                             } //delete the node and its children recursively
          }
(8)     else
            {
(9)        FCPS-Tree.node = PFPS-Tree.node;
            }//copy node
    }
}
```

In the algorithm Traverse(*PFPS-Tree*), the *MIN_SUP pruning* and *closed detection pruning* are used to delete the nodes that do not meet the conditions except the root node. In this way, an *FCPS-Tree* is formed at last.

**Example 3.3.** *Let MIN_SUP = 50%, MIN_SUP pruning and closed detection pruning are executed in the PFPS-Tree of Figure 2, then an FCPS-Tree is obtained shown as Figure 3. It can be seen clearly that a lot of potential patterns are removed from the tree because their support is smaller than MIN_SUP and they cannot guarantee their parent node is closed.*

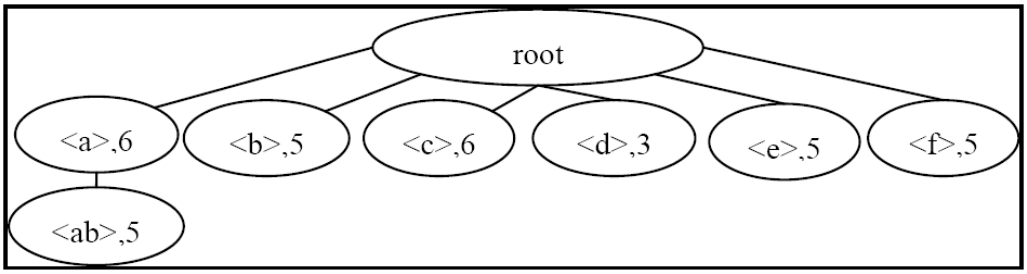


FIGURE 3. *FCPS-Tree* (some of tree nodes are not shown)

3.3. **Update-FPS-Tree.**

**Algorithm 3.3.1.** Update-FPS-Tree(|*w*|, *PatternTable*, *PF PS-Tree*).

Input: |*w*|, |*p*|, *PatternTable*, *PFPS-Tree*.

Output: *FCPS-Tree*.

Update-FPS-Tree(|*w*|, *PatternTable*, *PFPS-Tree*)

```
{
(1) PFPS-tree = DeleteTree(PatternTable,PFPS-Tree);//delete old patterns
(2) DeletePatternTable();//delete old patterns in the PatternTable
(3) for (each sequence in the new fragment)
    {
(4)   FS = PrefixSpan(Sequences); //FS are new patterns
(5)   PFPS-Tree = Insert(FS, PFPS-Tree); // insert new patterns
(6)   Insert(FS, PatternTable); // insert new patterns into the PatternTable
    }
(7) FCPS-Tree = Traverse(PFPS-Tree);
}
```

Update-FPS-Tree algorithm contains six steps, which are deleting old patterns from the *PFPS-Tree*, deleting old patterns from the *PatternTable*, mining new fragments, inserting new patterns into the *PatternTable*, inserting new patterns into the *PFPS-Tree* and calling the algorithm Traverse(*PFPS-Tree*) again to transform the *PFPS-Tree* into the *FCPS-Tree*. Just deleting old patterns from the *PatternTable* and deleting old patterns from the *PFPS-Tree* are new processes, others have been given in Section 3.2. So we only give the algorithm Deletetree(*PatternTable*, *PFPS-Tree*) below.

**Algorithm 3.3.2.** DeleteTree(*PatternTable*, *PFPS-Tree*).

Input: *PatternTable*, *PFPS-Tree*.

Output: *PFPS-Tree* without old patterns.

```
Deletetree(PatternTable, PFPS-Tree)
{
(1) for (each pattern in the PatternTable)
    {
(2) if (pattern.batch == 1)
        {
(3)       if (same(pattern, PFPS-Tree))
            {
(4)           node.count = node.count − pattern.count;
(5)           if (node.count == 0)
                {
(6)               delete (node){
(7)                                   delete node; node = node.node-link;
                                  }//delete the node and its child nodes recursively
                }
            }
        }
    }
}
```

The processes of Deletetree(*PatternTable*,*PFPS-Tree*) are as follows. The pattern whose *batch* equals 1 in the *PatternTable* is searched for in the tree. If a node pattern the same with the pattern is found, the result of the node count subtracting the count of the pattern is the new count of this node. After this, if the count of the node is not bigger than 0, we remove this node and its descendant nodes. In the end, deleting old patterns from the *PFPS-Tree* is finished.

**Example 3.4.** *We execute Algorithm 3.3.2 in the PFPS-Tree of Figure 2 by the assisting of the PatternTable in Table 2, and then the PFPS-Tree after deleting old patterns is formed and shown as Figure 4.*

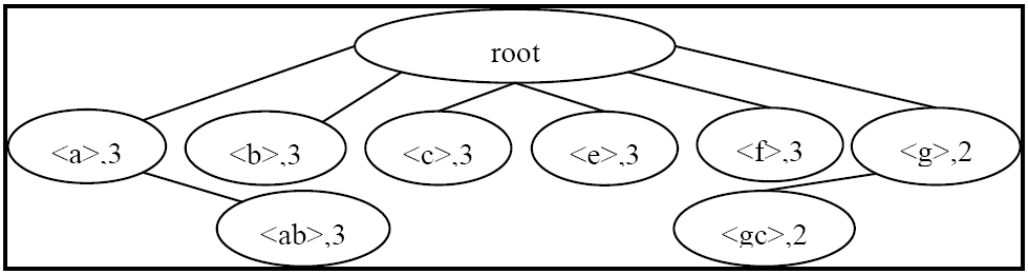


FIGURE 4. *PFPS-Tree* (some of tree nodes are not shown)

3.4. **Algorithm complexity.** The CSPMDS-PrefixSpan algorithm only maintains a *PatternTable* and a *PFPS-Tree* with the size of a window. *FCPS-Tree* is not saved in the memory but to be output. So it reduces the memory consumption, that is to say, the space complexity is low. The time complexity of Algorithm 3.2.1 is mainly reflected on the processes of insertion and traversal of the *PFPS-Tree*. It is connected with the depth represented as $d$ and the width represented as $e$ of the tree. Because the sizes of the sliding windows and fragments are not infinite, then $d$ and $e$ are not infinite. The time cost of these two processes is $O(d \times e) \times 2$, that is to say, the time complexity is $O(n^2)$. In Algorithm 3.3.1, the time complexity is mainly reflected on the traversal process of the *PFPS-Tree*, deleting old patterns, inserting new patterns and traversing the *PFPS-Tree* to translate it into the *FCPS-Tree*. The time cost of these three processes is $O(d \times e) \times 3$, that is to say, the time complexity is $O(n^2)$. In all, the time complexity of CSPMDS-PrefixSpan algorithm is low.

4. **Experiments.** Experiments are conducted on the computer with Windows 7 operating system, CPU of 2.90GHZ and memory of 2G by NetBeans platform and java language. We remove the incremental method from the CSPMDS-PrefixSpan and mine each window respectively, and then the algorithm named CSPM-Stream is gotten. The algorithms PrefixSpan [6], MEMISP [7], GSP [3] and CSPM-Stream are introduced as contrasts. The experiment data are produced by IBM data generation-D100C10T5S4N0.5I 1.25, which was used in paper [5,7]. The data set is input as a data stream. We express the minimum support threshold of PrefixSpan, MEMISP and GSP as *minsup*. Because *MIN_SUP* plays the role of pruning the number of patterns in the window, so it can be equal to *min_sup* or slightly smaller.

4.1. **Performance of execution time.**

**Experiment 1. Execution time comparison at different amounts of data.** The *minsup*, *min_sup* and *MIN_SUP* are all set to 0.1, $|w| = 4$. The time cost comparison is shown in Figure 5. The time cost of CSPMDS-PrefixSpan is less than GSP, PrefixSpan and CSPM-Stream. The reason is that PrefixSpan and GSP process all data at the same time, which costs a lot of time in reading data into memories and calculating the support count by projection or connection with large data; while the CSPMDS-PrefixSpan only mines a fragment once, its time costs equal to the sum of execution time in each fragment approximately. The time costs of CSPMDS-PrefixSpan at $|p| = 1000$ and $|p| = 5000$ are quite similar, which proves that CSPMDS-PrefixSpan has a steady performance. CSPM-Stream without incremental method processes the stream fragments repeatedly, which consumes much time. When the sequences are about more than 20000, the GSP and PrefixSpan are out of memory according to our experimental results. In all, it proves that data stream sequential pattern algorithms are more efficient than the traditional

sequential pattern algorithms.

**Experiment 2. Execution time comparison at different support thresholds.**

The number of sequences to be processed is 10000. The comparison of efficiency at different support thresholds is shown in Figure 6. From the figure, it shows that the time cost of CSPMDS-PrefixSpan is quite lower than PrefixSpan and CSPM-Stream at the support threshold from 8% to 12%. CSPMDS-PrefixSpan outperforms CSPM-Stream obviously, which proves that the incremental method in this paper is quite effective.

### 4.2. Performance of memory usage, closed detection and MIN_SUP pruning.

**Experiment 3. Comparison of computer memory usage.**

In this experiment, we set the experimental parameters the same as Experiment 1, the memory consumptions are shown in Figure 7. The memory usage of CSPMDS-PrefixSpan becomes a horizontal line approximately and is less than PrefixSpan and MEMISP who have ascending lines at any data amount. The amount of data processed in the memory by CSPMDS-PrefixSpan is equal while it increases in PrefixSpan and MEMISP every time, which causes the case. The increasing of data amount does not make the memory usage grow continuously in CSPMDS-PrefixSpan, but do in PrefixSpan and MEMISP. MEMISP is based on the memory indexing, which is an algorithm that reduces the time cost but consumes a lot of memories. When the sequences are about more than 20000, the MEMISP and PrefixSpan are out of memory and do not work according to our experiment results. Therefore, CSPMDS-PrefixSpan outperforms PrefixSpan and MEMISP in memory usage.

**Experiment 4. Performance of closed detection *pruning* and *MIN_SUP pruning*.**

The number of sequences to be processed is 10000. The *closed detection pruning* and *MIN_SUP pruning* performance is measured and the results are shown as Figure 8. It can be seen the patterns are significantly reduced when we use the two pruning methods so that we avoid a lot of useless patterns. CSPMDS-PrefixSpan without executing pruning obtains a lot of potential patterns, which are more than PrefixSpan and CSPMDS-PrefixSpan after pattern pruning.

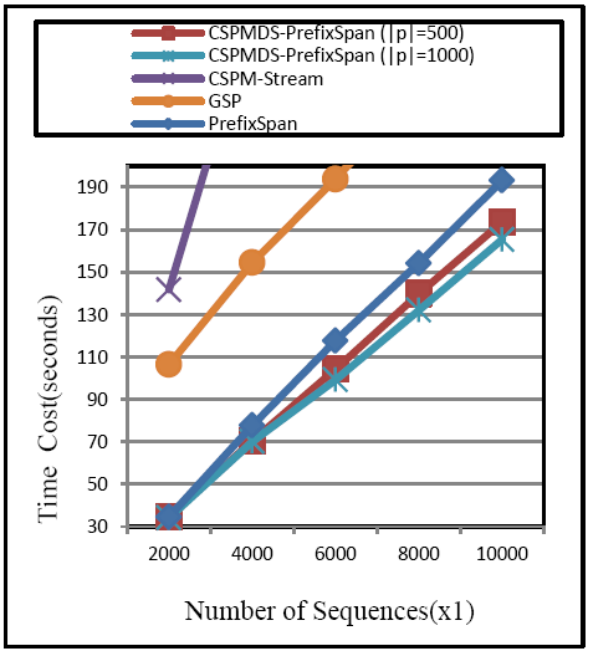


FIGURE 5. Execution time comparison with various amounts of data

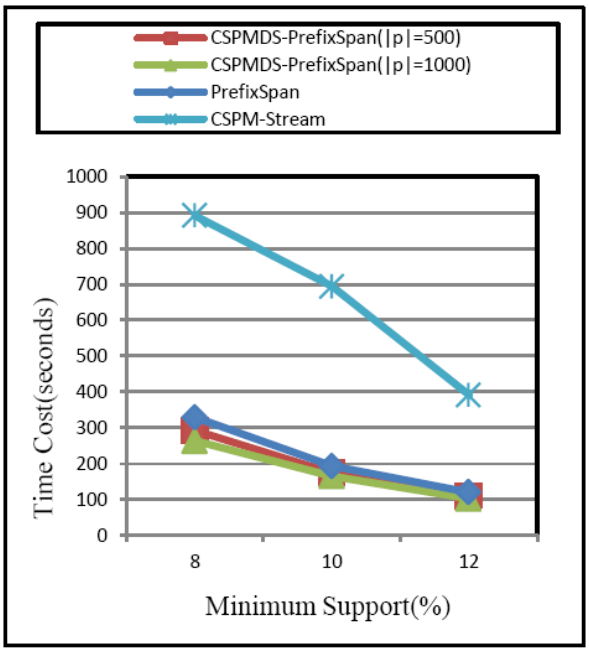


FIGURE 6. Execution time comparison with various support thresholds

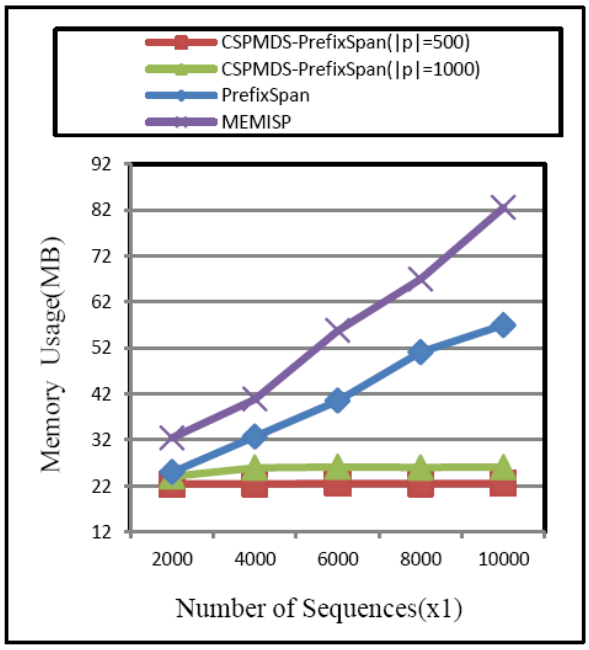


FIGURE 7. Memory usage comparison

4.3. **Synthesis analysis and discussion of results.** The experiment results have shown that CSPMDS-PrefixSpan is better than PrefixSpan, MEMISP, GSP and CSPM-Stream in some aspects, especially in the memory usage and pattern pruning. The incremental approach makes CSPMDS-PrefixSpan mine stream quickly and only scan the stream once. The time cost of CSPMDS-PrefixSpan is increased linearly when the data become larger and larger due to the good use of incremental method. When the data set reaches to about 20000 sequences, PrefixSpan, MEMISP and GSP are all out of memory and cannot work. In this case, CSPMDS-PrefixSpan has a good advantage over these

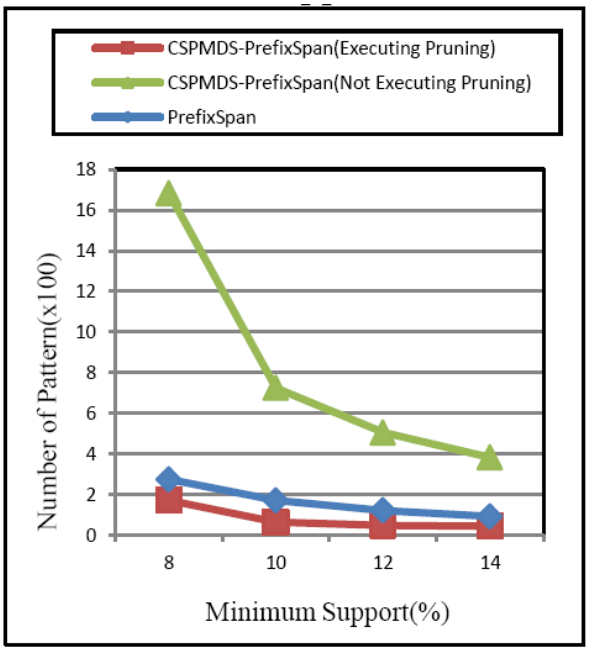


FIGURE 8. Efficiency of pattern pruning

algorithms; it can work well at any amount of data because of its strong point of saving memory. The pattern pruning not only makes the algorithm friendly to time usage but also decreases the hardness to analyze the mining results, in which CSPMDS-PrefixSpan outperforms PrefixSpan and CSMP-Stream a lot.

In all, CSPMDS-PrefixSpan is comparatively outstanding. However, accuracy of mining may be lower than traditional algorithms of sequential pattern. It may lose some patterns in translating potential pattern into frequent pattern in CSPMDS-PrefixSpan. Since the data streams are unbounded, high-speed and continuous, improving the accuracy of mining is a hard and long-term task.

5. **Conclusions.** The characteristics of data streams make the traditional sequential pattern algorithms unable to be applied directly. In this paper, CSPMDS-PrefixSpan algorithm was proposed. It extended the PrefixSpan algorithm. The sliding windows were divided into several fragments to be mined respectively. The incremental particle was increased to several stream fragments. We designed the *PatternTable* and *PFPS-Tree* as summary data structures to store the key compressed information. The *PatternTable* assisted in the updating of *PFPS-Tree*, avoiding duplication of mining. The *closed detection pruning* and *MIN_SUP pruning* were presented to translate the *PFPS-Tree* into the *FCPS-Tree*. The experimental results have shown it overcomes the memory limit, reduces the execution time and has good performances of pattern pruning and incremental mining compared with some previous algorithms. However, there are some mining errors in the CSPMDS-PrefixSpan algorithm, the error handling is not sufficient and it is to be investigated further.

**Acknowledgment.** This work is supported by the Natural Science Foundation of P. R. China under Grant No. 61170190 and the Natural Science Foundation of Hebei Province P. R. China under Grant No. F2012203062. The authors are also appreciated to the valuable comments and suggestions of the reviewers.

## REFERENCES


[1] J. Ren, Y. Xie, A. Zhang, C. Hu and Y. Chen, A closed sequential pattern mining algorithm for discovery of the software bugs feature, *Journal of Computational Information Systems*, vol.7, no.7, pp.2322-2329, 2011.

[2] I-H. Li, J.-Y. Huang and I-E. Liao, Predicting sequential pattern changes in data streams, *International Journal of Innovative Computing, Information and Control*, vol.8, no.1(A), pp.285-302, 2012.

[3] A. Mala and F. R. Dhanaseelan, Data stream mining algorithms: A review of issues and existing approaches, *International Journal on Computer Science and Engineering*, vol.3, no.7, pp.2726-2732, 2011.

[4] H. He, S. Chen, K. Li and X. Xu, Incremental learning from stream data, *IEEE Trans. on Neural Networks*, vol.22, no.12, pp.1901-1913, 2011.

[5] R. Agrawal and R. Srikant, Mining sequential patterns, *Proc. of the 11th International Conference on Data Engineering*, Taiwan, pp.3-14, 1995.

[6] J. K. Febrer-Hern'andez and J. Hern'andez-Palancar, Sequential pattern mining algorithms review, *Intelligent Data Analysis*, vol.16, no.3, pp.451-466, 2012.

[7] H. Liu, Y. Lin and J. Han, Methods for mining frequent items in data streams: An overview, *Knowledge and Information Systems*, vol.26, no.1, pp.1-30, 2011.

[8] J. Pei and J. Han, PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth, *Proc. of the 17th International Conference on Data Engineering*, Heidelberg, Germany, pp.215-224, 2001.

[9] M. Lin and S. Lee, Fast discovery of sequential patterns through memory indexing and database partitioning, *Journal of Information Science and Engineering*, vol.21, no.1, pp.109-128, 2005.

[10] A. Gomariz, M. Campos, R. Marin and B. Goethals, ClaSP: An efficient algorithm for mining frequent closed sequences, *Knowledge Discovery and Data Mining, Lecture Notes in Computer Science*, vol.7818, pp.50-61, 2013.

[11] X. Yan, J. Han and R. Afshar, CloSpan: Mining closed sequential patterns in large datasets, *Data Mining*, vol.16, no.5, pp.40-45, 2003.

[12] K. C. Kandpal and R. Agnihotril, Sblock – A closed sequential pattern mining algorithm, *International Journal of Computer Applications in Engineering Sciences*, vol.1, no.3, pp.296-299, 2011.

[13] C. Raïssi, P. Poncelet and M. Teisseire, Need for SPEED: Mining sequential pattens in data streams, *Actes des 21iemes Journees Bases de Donnees Avancees*, 2005.

[14] L. Chang, T. Wang, D. Yang and H. Luan, SeqStream: Mining closed sequential patterns over stream sliding windows, *The 8th IEEE International Conference on Data Mining*, Pisa, Italy, pp.83-92, 2008.

[15] S.-Y. Yang, C.-M. Chao, P.-Z. Chen and C.-H. Sun, Incremental mining of across-streams sequential patterns in multiple data streams, *Journal of Computers*, vol.6, no.3, pp.449-457, 2011.

[16] 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, no.5, pp.728-735, 2011.

[17] G. Lee, K.-C. Hung and Y.-C. Chen, Path tree: Mining sequential patterns efficiently in data streams environments, *Intelligent Systems and Applications, SIST*, vol.20, pp.261-268, 2013.

[18] C.-C. Ho, H.-F. Li, F.-F. Kuo and S.-Y. Lee, Incremental mining of sequential patterns over a stream sliding window, *Proc. of the IWMESD*, Hong Kong, China, pp.677-681, 2006.

[19] T. Guyet and R. Quiniou, Incremental mining of frequent sequences from a window sliding over a stream of itemsets, *Actes IAF*, 2012.

[20] S. K. Tanbeer, C. F. Ahmed, B.-S. Jeong and Y.-K. Lee, Sliding window-based frequent pattern mining over data streams, *Information Sciences*, vol.179, no.22, pp.3843-3865, 2009.

[21] L. F. Mendes, B. Ding and J. Han, Stream sequential pattern mining with precise error bounds, *The 8th IEEE International Conference on Data Mining*, Pisa, Italy, pp.941-946, 2008.

[22] A. Koper and H. S. Nguyen, Sequential pattern mining from stream data, *Advanced Data Mining and Applications*, vol.7121, pp.278-291, 2011.

[23] L. Vinceslas, J.-E. Symphor, A. Mancheron and P. Poncelet, SPAMS: A novel incremental approach for sequential pattern mining in data streams, *Advances in Knowledge Discovery and Management*, vol.292, pp.201-216, 2010.

[24] H.-F. Li, Mining top-k maximal reference sequences from streaming web click-sequences with a damped sliding window, *Expert Systems with Applications*, vol.36, no.8, pp.11304-11311, 2009.