A PROJECTION-BASED HYBRID SEQUENTIAL PATTERNS MINING ALGORITHM

Chichang Jou

Department of Information Mangement, Tamkang University, 151 Ying-Chuan Road, Tamsui, Taipei, Taiwan

Keywords: Hybrid sequential pattern, Pattern growth, Projected position array, Projected support array, Projected database.

Abstract: Sequential pattern mining finds frequently occurring patterns of item sequences from serial orders of items in the transaction database. The set of frequent hybrid sequential patterns obtained by previous researches either is incomplete or does not scale with growing database sizes. We design and implement a Projection-based Hybrid Sequential PAttern Mining algorithm, PHSPAM, to remedy these problems. PHSPAM first builds Supplemented Frequent One Sequence itemset to collect items that may appear in frequent hybrid sequential patterns. The mining procedure is then performed recursively in the pattern growth manner to calculate the support of patterns through projected position arrays, projected support arrays, and projected databases. We compare the results and performances of PHSPAM with those of other hybrid sequential pattern mining algorithms, GFP2 and CHSPAM.

1 INTRODUCTION

Data mining aims to extract implicit information inside voluminous electronic data, and has become more and more important for science, engineering, business, etc. Frequent sequential pattern mining obtains item sequence patterns satisfying the condition that the number of their occurrences in the database, called *support*, is greater than or equal to a given minimum threshold. It has been applied to applications like stock trend, genome sequencing, web page traversal, customer behavior analysis, etc.

Based on requirement of items' adjacency in a pattern, sequential patterns could be classified into three categories (Chen et al., 2002): (1) Continuous patterns, where the adjacent items in a pattern must be matched by consecutive items in the transactions. (2) Discontinuous patterns, where adjacent items in a pattern are matched by items separated by zero or more items. (3) Hybrid patterns, where discontinuity is represented by a special symbol '*' so that continuous and discontinuous adjacency requirements could be specified together. For example, for a transaction *T* to match the hybrid pattern <*AB**C*>, its matching items for *A* and *B* must be adjacent, and its matching items for *B* and *C* are not necessarily consecutive.

As far as we know, the most efficient hybrid sequential pattern mining algorithm is GFP2 (Chen et al., 2002). Its basic concept is the Apriori principle--candidate frequent patterns of length *n* could be generated by joining frequent patterns of length *n-1*. However, with repetitive items in a transaction, some frequent patterns might be missed by this technique. For example, suppose the database has only one transaction *T*=<*ABCBD*>, and the minimum threshold is 2. The only frequent pattern of length *1* is <*B*>. By joining frequent patterns of length *1*, the candidate frequent patterns of length 2 are <*B**B*> and <*BB*>. Supports for patterns <*A**B*> and <*B**D*> are 2, but they are not obtainable by filtering <*B**B*> and <*BB*>.

Jou (2006) proposed an algorithm, CHSPAM, to obtain the complete hybrid sequential patterns. However, CHSPAM's performance is no better than GFP2. We utilize the pattern growth methodology and propose Projection-Based Hybrid Sequential PAttern Mining algorithm, PHSPAM, to tackle this performance problem.

The rest of the paper is organized as follows: Section 2 surveys related work. Problem definition and PHSPAM are illustrated in Section 3. Experimentations and their results are discussed in Section 4. Section 5 concludes the paper and points out future work directions.

Jou C. (2009).
A PROJECTION-BASED HYBRID SEQUENTIAL PATTERNS MINING ALGORITHM.
In *Proceedings of the 11th International Conference on Enterprise Information Systems - Artificial Intelligence and Decision Support Systems*, pages 152-157
DOI: 10.5220/0001986001520157

2 RELATED WORK

Agrawal and Srikant (1995) extended the frequent itemset mining algorithm (Agrawal and Srikant, 1994; Agrawal et al., 1993) for non-serial transactions to discontinuous sequential pattern mining for serial transactions using the Apriori principle for sequential patterns. Pei et al. (2000) obtained discontinuous sequential web traversal patterns by constructing the access pattern tree from web access logs. Chen et al. (1998) extended the Apriori principle to obtain frequent continuous sequential web path traversal patterns.

Based on the approach to attack the sequential pattern mining problem, its solutions could be classified into two categories: 1. Candidate pattern generation and filtering; 2. Pattern growth.

The methods in the candidate pattern generation and filtering category extend the Apriori principle. The whole database needs to be scanned in each candidate pattern generation step. For large databases, this approach needs huge memory space to store the candidate patterns. GSP (Agrawal and Srikant, 1996), GFP2 (Chen et al., 2002), and SPADE (Zaki, 2001) all belong to this category. GSP added time constraints between adjacent items in a pattern, relaxed the restriction that elements of a pattern must be from the same transaction, and allowed items to cross taxonomy levels. GFP2 was the first to obtain hybrid sequential patterns. It utilized the containment relationship of supports between the patterns with '*' and those without '*' to reduce the number of database scans. SPADE utilized subset relationship of patterns in calculating the support of candidate patterns by checking the positions of each item in the transactions.

In the pattern growth methodology, to skip the time-consuming candidate patterns generation step, a candidate pattern is built by concatenating a pattern *p* with patterns obtained from its projected databases, which are collections of after-*p* sub-transactions. Supports of patterns with prefix *p* are counted by summing supports of patterns supported by sub-transactions in *p*'s projected database. FreeSpan (Han et al., 2000a; Pei et al., 2004) first utilized the pattern growth concept from FP-Growth (Han et al., 2000b). PrefixSpan (Pei et al., 2001; Pei et al., 2004) used bi-level projection to reduce the number of projected databases. It also recorded the transaction id and the position of sequential patterns, instead of the real data, to reduce the memory requirement. Jou (2006) discovered that some frequent patterns could not be obtained by the Apriori principle. He proposed the first complete hybrid sequential pattern mining algorithm, CHSPAM. Its mining procedure is performed recursively in the pattern-growth manner by the backward support counting method.

3 THE PHSPAM ALGORITHM

Our definition of hybrid sequential pattern follows that of GFP2 (Chen et al., 2002): Let $I = \{i_1, i_2, ..., i_m\}$ denotes the set of items in a database. A transaction $T=<t_1\ t_2\ ...\ t_k>$ is an item sequence, where for all i between 1 and k, $t_i \in I$. A database consists of a set of transactions. A special symbol '*' not in I is used in the pattern specification to denote the occurrence of 0 or more items. A sequence $<x_1\ x_2\ ...\ x_n>$ is called a *hybrid sequential pattern* if:

(1) $x_i \in I \cup \{ * \}$ for all $1 \le i \le n$

(2) $x_1 \in I$ and $x_n \in I$; and

(3) for all $1 < i < n$, if x_i ='*' then $x_{i-1} \in I$ and $x_{i+1} \in I$.

Condition (2) requires a pattern to start and end with an item. Condition (3) exclude consecutive *'s in a pattern. Hybrid sequential pattern will be abbreviated as pattern henceforth when no confusion arises. Thus, *<ABC>, <A*BC>* and *<A*B*C>* are legitimate patterns, while *<*AB>* and *<AB**C>* are not.

A pattern p is said to be contained in a transaction T, denoted as $p \subset T$, if:

(1) for all items $x \in p$, there is a matching item $x \in T$;

(2) for all items $x, y \in p$, if x precedes y in p, then x's matching item in T also precedes y's; and

(3) for all items $x, y \in p$, if x is adjacent to y in p, then x's matching item in T is also adjacent to y's.

The above conditions require that a matching instantiation of p in T must respect the serial order of items in p.

For each pattern p and transaction T, the number of different matching instantiations for $p \subset T$ is called the *support* of p in T, denoted by $supp_{p,T}$. For example, suppose p=*<A*BC>* and T=*<BACABC>*. Then $supp_{p,T}$ = 2, where the first instantiation matches *A, B, C* to positions 2, 5, 6 of *T*, and the second matches them to positions 4, 5, 6. For a given minimum support threshold *minSupp*, we call pattern p a frequent pattern if p satisfies the condition:

$$\sum_{\forall T} supp_{p,T} \ge minSupp\ .$$

From short to long patterns, PHSPAM generates for each pattern p a projected position array, which keeps for all matching transactions the positions of the items matching *p*'s last item. Corresponding to each position in the position array, their contribution to the support for *p* is stored in *p*'s projected support array. The mining procedure is recursively applied to the projected databases by summing supports from their projected support arrays.

PHSPAM has four steps: Step 1 scans the database and constructs the Supplemented Frequent One Sequence itemset, denoted as *SFOS*, to collect items that may appear in frequent patterns. Step 2 transforms items not in *SFOS* into '+' to generate the reduced database, which helps decrease the number of comparisons in step 4. Step 3 generates the projected position arrays, projected support arrays, and projected databases for each *SFOS* item. Step 4 performs recursive support counting in the projected databases, and finally checks whether *SFOS* items are frequent patterns. We discuss each step in the following subsections.

3.1 Generating the SFOS Itemset

The *length* of a pattern *p* is defined as the number of items occurring in *p*. In hybrid sequential pattern mining, frequent patterns of length 1 are the same as the set of items with support greater than or equal to *minSupp.* As illustrated in Section 1, items with support less than *minSupp* could appear in frequent patterns of length longer than 1. PHSPAM remedies the problem by constructing *SFOS* in the first step as follows:

(1) All items with their total number of occurrence greater than or equal to minSupp. These are the traditional frequent one sequence items;
(2) All repetitive items, which are items occurring more than once in a single transaction; and
(3) All items before and after any pair of repetitive items in a transaction.

For the example in Section 1, Items *A* and *D* are added into *SFOS* since *A* occurs before the first *B,* and *D* after the second *B*. According to the definition of *SFOS*, if an item occurs more than two times in a transaction, then all items in that transaction will be added into *SFOS*.

3.2 Database Reduction

The second step transforms each series of consecutive non-*SFOS* items in the original database *DB* into a special symbol '+', to obtain *DB'*, the reduced database. Since the first and last items of a pattern are in *SFOS*, '+'s appearing in the prefix or suffix of a reduced transaction are deleted in this reduction. For example, suppose the sample database *DB* contains the 3 transactions in Table 1, and *minSupp* is 4. By scanning *DB* once, its *SFOS* could be constructed as {*A, B, D*} with supports 4, 5, 3 respectively. Note that the support of item *D* is less than *minSupp*. *D's* inclusion in *SFOS* is because *D* is after the repetitive items *B* in transactions 2 and 3. By scanning *DB* once more, the reduced database *DB'* in Table 2 could be constructed.

Table 1: The sample database.

trans. id	serial items
1	ABCED
2	BACBAD
3	BCABD

Table 2: Reduced sample database.

trans.id	serial items
1	AB+D
2	BA+BAD
3	B+ABD

3.3 Generating Projected Databases for Each SFOS Item

For pattern *p* and transaction *T*, projected position array $PPA_{p,T}$ contains the positions of *T*'s item matching *p*'s last item, and projected support array $PSA_{p,T}$ contains the supports for *p* at these positions. The collections of $PPA_{p,T}$ (similarly, $PSA_{p,T}$) over all transactions *T* is denoted as PPA_p (PSA_p). The third step constructs for each item *i* in *SFOS* PPA_i, PSA_i, and the projected database PDB_i from *DB'* as follows: if *i* occurs in transaction *T*, then push all pairs of *T*'s transaction id with *i*'s positions in *T* into PPA_i. Additionally, set the support for *T* as *1* for each pair in PSA_i. The projected database of pattern <*i*>, PDB_i, is the set of all after-*i* sub-transactions. For repetitive items in a transaction, we will have a sub-transaction for each occurrence of the item.

Table 3: Projected position array, projected support array, projected database of pattern <*B*>.

subtr. id	trans. id	PPA_B	PSA_B	PDB_B
1	1	2	1	+D
2	2	1	1	A+BAD
3	2	4	1	AD
4	3	1	1	+ABD
5	3	4	1	D

To simplify the explanation, a tabular form of PPA_p, PSA_p, PDB_p will be indexed by the same sub-transaction id. Table 3 illustrates the tabular form of PPA_B, PSA_B, and PDB_B for the reduced sample database. These projected data structures for *SFOS* items are stored in the hard disk. Since the support counting procedure uses one projected database for an *SFOS* item at a time, at any time PHSPAM loads only one set of these projected data structures into memory.

3.4 Recursive Traverse and Count

Figure 1 demonstrates the fourth step, *TraverseAndCount,* for a pattern *p*. Let I_p denote the set of items in the projected database PDB_p. This step first calls *GetSupport(p)* to obtain the support count mapping, M_p, which maps, for all items *i* in I_p, patterns <*p*i*> and <*pi*> to the their supports. Details of *GetSupport* will be explained in the next paragraph. The supports M_p (<*p*i*>) and M_p (<*pi*>) are compared to *minSupp* to decide whether <*p*i*> and <*pi*> are frequent patterns. Then, it calls *BuildProjectedArrays(p,i)* to build projected position arrays (PPA_{p*i} and PPA_{pi}), and projected support arrays (PSA_{p*i} and PSA_{pi}). Suppose the first item of *p* is *x*. The projected database PDB_{p*i} (similarly, PDB_{pi}) could be obtained on the fly from PDB_x and PPA_{p*i} (similarly, PDB_x and PPA_{pi}). *TraverseAndCount* will then be recursively applied to <*p*i*> and <*pi*>.

```
/* <p>  denotes a pattern */
        /* $PDB_p$ is the projected database of <p> */
/* FP is the set of frequent patterns */
Procedure TraverseAndCount(p)
Begin
        $M_p$ = GetSupport(p)
        /* $M_p$ maps <p*i> and <pi>, i being one of the
           items in $PDB_p$, to their supports */
         for each item i ∈  $I_p$
            if $M_p$ (<p*i>) ≥  minSupp then
                 add <p*i> to FP
           endIf
           if $M_p$ (<pi>) ≥  minSupp then
               add <pi> to FP
           endIf
           BuildProjectedArrays(p, i)
            /* build projected position/support arrays, for
               patterns <p*i> and <pi> */
           TraverseAndCount(<p*i>)
            TraverseAndCount(<pi>)
        endFor
End Procedure
```

Figure 1: Pseudo code of *TraverseAndCount.*

For a pattern *p*, Figure 2 shows the construction of the support count mapping M_p in *GetSupport(p)* by scanning the sub-transactions in PDB_p as follows: Suppose the index of sub-transaction T_s in PDB_p is *j*. Out of all matching instantiations of <*p*> in T_s's original transaction *T,* $PSA_{p,T}[j]$ matching instantiations satisfy the condition that the last element of *p* is in position $PPA_{p,T}[j]$. Thus, for all items *i* in T_s*, T* would contribute $PSA_{p,T}[j]$ to the support of <*p*i*>. That is, each occurrence of item *i* in T_s would increase M_p*'s* final supports of <*p*i*> by $PSA_{p,T}[j]$. Similarly, if an item *i* in T_s satisfies the condition that the last element of *p* in the matching instantiation is adjacent to *i*, verified by checking whether the position of *i* is equal to $PPA_{p,T}[j]+1$, each occurrence of *i* would increase the support count mapping of <*pi*> by $PSA_{p,T}[j]$. In M_p, the support counts for <*p*i*> and <*pi*> are the accumulated sum of supports of <*p*i*> and <*pi*> over all sub-transactions in PDB_p. It is clear that M_p maps each pattern to their support count in the original database.

From projected database PDB_B of pattern <*B*> in Table 3, I_B is {*A, B, D*}. Take item *A* as an example. The detailed support counting of the mapping of <*B*A*> in M_B is demonstrated in Table 4. Note that the support of <*B*A*> contributed from transaction 2 position 5 is 2, where both sub-transactions 2 and 3 in PDB_B contribute one support. Since sum of supports of <*B*A*> in PDB_{B*A} is 4, <*B*A*> is added into the set of frequent patterns. However, since only 2 of the 4, both from transaction 2, satisfy the condition that item *A* is immediately after *B*, support of <*BA*> is 2, and <*BA*> is not a frequent pattern. The function *BuildProjectedArrays(<B>,A)* then builds projected position arrays (PPA_{B*A} and PPA_{BA}), and projected support arrays (PSA_{B*A} and PSA_{BA}). *TraverseAndCount* is then recursively performed on <*B*A*> *and* <*BA*> separately.

For the projected database PDB_{B*A} in Table 4, item *D* occurs once in sub-transactions 1, 2 and 3.

```
/* $PDB_p$ : the Projected Database of pattern <p> */
/* $PPA_{p,T}$ : the Projected Position Array  for sub-transaction
           T in $PDB_p$ */
/* $PSA_{p,T}$: the Projected Support Array  for sub-transaction
           T in $PDB_p$ */
Function GetSupport(p)
Begin
   for each item i ∈ $I_p$
      $M_p$(<p*i>)=0; $M_p$(<pi>)=0
   endFor
   for each T ∈ $PDB_p$
       j = index of T in $PDB_p$
       for each item i ∈ T
                    /* do not count '+' as an item */
          if i != '+' then
             $M_p$(<p*i>)=$M_p$(<p*i>)+ $PSA_{p,T}$[j]
             if ($PPA_{p,T}$[j]+1==position of i
                    in T) then
                  $M_p$(<pi>)=$M_p$(<pi>)+
                    $PSA_{p,T}$[j]
       endFor
   endFor
   return $M_p$
```

Figure 2: Pseudo code of *GetSupport.*

Table 4: Projected position array, projected support array, and projected database of pattern <*B*A*>.

subtr. id	trans. id	PPA_{B*A}	PSA_{B*A}	PDB_{B*A}
1	2	2	1	+BAD
2	2	5	1+1	D
3	3	3	1	BD

The supports for <*B*A*D*> contributed by sub-transaction 1, 2, and 3 are 1, 2, and 1, respectively. Thus, M_{B*A}(<*B*A*D*>)=4. Since only the case in sub-transaction 2 satisfies the condition that *D* is immediately after *A*, M_{B*A}(<*B*AD*>)=2. The support count accumulation details of <*B*A*i*> and <*B*Ai*> for all items *i* in I_{B*A} could be computed in the same fashion. Thus, the sample database has the following frequent patterns: *<A>, <B>, <A*D>, <B*A>, <B*D>,* and *<B*A*D>*.

4 EXPERIMENTS

We implement GFP2, CHSPAM and PHSPAM in VB.NET 2005. The first experiment compares numbers of obtained frequent patterns and execution time with respect to database size and average length of frequent patterns. The second experiment checks the sensitivity of PHSPAM regarding the minimum support. The experiments are performed in the Windows Server operating system, Intel Xeon 3100 Dual-Core CPU, and 1024MB DDR memory. The databases, generated by IBM Quest Synthetic Data Generator, are stored in Microsoft SQL Server 2005. The fixed parameters for generating the databases are: the number of distinguished items is 10000; the number of items in the frequent one sequence itemset is 1500; the average transactions length is 15. The varied parameters are: |D|: the number of transactions in the database, and |I|: the average length of frequent patterns.

The first experiment is with *minSupp* set as 0.015% of |D|. The parameters of |D| are 100K, 200K, 300K, 400K and 500K, and the parameters of |I| are 2 and 4. Number of frequent patterns and execution time results are displayed in Table 5. In all 10 settings, the numbers of frequent patterns obtained by CHSPAM and PHSPAM are the same. They are 0.08% to 14.67% more than those by GFP2, but the percentage fluctuates with the database size. This shows that the behavior of PHSPAM is dependent on the probabilistic population of the repetitive items. Figure 3 displays the execution time in these 10 settings for GFP2 and PHSPAM only. With the projected data structures to accumulate support counts, PHSPAM runs faster than GFP2 and

Table 5: Results and performances for *minSupp* equal to 0.015% of |D|.

database	no. of freq patterns		execution time (second)		
	GFP2	CHSPAM/ PHSPAM	GFP2	CHSPAM	PHSPAM
I2D100K	968	973	212	1958	102
I2D200K	1506	1509	564	2625	352
I2D300K	1059	1062	803	6164	467
I2D400K	1787	1814	1222	10389	829
I2D500K	1306	1335	1277	13027	893
I4D100K	1315	1326	205	3053	200
I4D200K	2108	2117	521	4089	502
I4D300K	1535	1760	771	9856	687
I4D400K	4495	4995	1498	20638	1268
I4D500K	6146	6151	1862	29741	1441

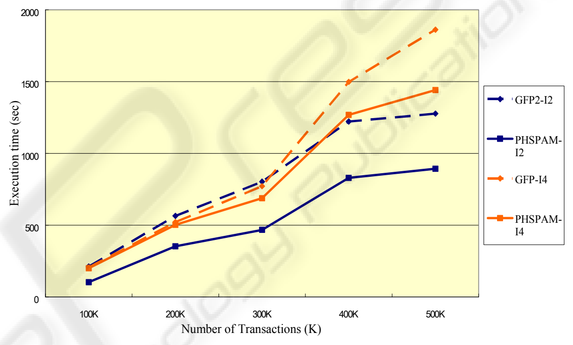

Figure 3: Execution time comparison of GFP2 and PHSPAM.

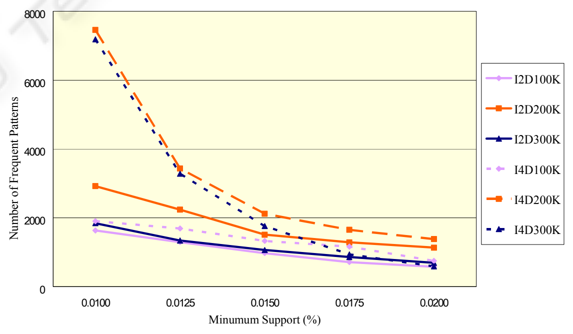

Figure 4: Number of frequent patterns with respect to the minimum support rate.

CHSPAM.

The second experiment runs PHSPAM with *minSupp* set as 0.0100%, 0.0125%, 0.0150%, 0.0175% and 0.0200% of |D|. The parameters of |D| are 100K, 200K, and 300K, and the parameters of |I| are 2 and 4. Number of frequent patterns and execution time results are displayed in Table 6. Both consistently grow with the decrease of the minimum support. Figure 4 shows the trend of number of frequent patterns regarding to the minimum support.

Note that the growth rate is sharper for the number of frequent patterns than that of the execution time. Thus, by decreasing the minimum support, the extra execution time is worthy of the insights gained from the extra frequent patterns.

Table 6: Execution time for *minSupp* equal to 0.0100% 0.0125%, 0.0150%, 0.0175%, 0.0200% of |D|.

database	no. of freq patterns / execution time (second)				
	0.0100%	0.0125%	0.0150%	0.0175%	0.0200%
I2D100K	1633/125	1295/114	973/102	714/95	579/88
I2D200K	2920/394	2235/371	1509/352	1287/327	1133/304
I2D300K	1843/557	1340/511	1062/467	859/426	692/391
I4D100K	1902/237	1688/211	1326/200	1163/193	747/180
I4D200K	7460/794	3429/586	2117/502	1648/453	1381/425
I4D300K	7185/1206	3280/826	1760/687	937/452	589/415

5 CONCLUSIONS

We designed the PHSPAM algorithm to remedy the problems that the set of frequent hybrid sequential patterns obtained by previous researches is incomplete and that the execution time does not scale with growing database sizes. PHSPAM obtains the complete set by first collecting items that might appear in the frequent patterns. PHSPAM then uses the pattern growth techniques to calculate the support of patterns. PHSPAM was implemented and compared with GFP2 and CHSPAM. The experiments demonstrated that PHSPAM indeed obtained more frequent patterns than GFP2. In addition, achieving the same completeness result, the execution time of PHSPAM is better than GFP2 and much better than CHSPAM, due to the accumulated counts preserved in the projected data structures.

Our future research regarding hybrid sequential pattern mining includes:

(1) Apply PHSPAM in real world applications, like web page traversal paths mining through web logs.

(2) Examine the effect of replacing the support definition such that a transaction could contribute at most one in the support counting of a pattern.

(3) Consider application specific constraint of the patterns, like timing limitations.

ACKNOWLEDGEMENTS

This work was supported in part by Taiwan's National Science Council under Grant NSC ：97-2221-E-032-046.

REFERENCES

Agrawal, R., Imielinski, T., Swami, A., 1993. Mining association rules between sets of items in large databases. In *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data, Washington D.C., U.S.A., pp. 207-216.*

Agrawal, R., Srikant, R., 1994. Fast algorithm for mining association rules. In *Proc. of the 20th International Conference on VLDB, Santiago, pp. 487-499.*

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

Agrawal, R., Srikant, R., 1996. Mining sequential patterns: generalizations and performance improvements. In *Lecture Notes in Computer Science, Vol.1057, pp. 3-17.*

Chen, M., Park, J.S., and Yu, P.S., 1998. Efficient data mining for path traversal patterns. *IEEE Trans. Knowledge Data Engineering, Vol. 10(2), pp. 209-221.*

Chen, Y.L., Chen, S.S., Hsu, P.Y., 2002. Mining hybrid sequential patterns and sequential rules. *Information Systems, Vol. 27(5), pp. 345-362.*

Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., Hsu, M.C., 2000a. Freespan: frequent pattern-projected sequential pattern mining. In *Proc. of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Boston, U.S.A., pp. 355-359.*

Han, J., Pei, J., Yin, Y.W., 2000b. Mining frequent patterns without candidate generation. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data, New York, U.S.A. pp. 1-12.*

Jou, C., 2006. Mining Complete Hybrid Sequential Patterns. In *Proc. of the DMIN 2006 International Conference on Data Mining, pp. 218-223, Las Vegas, USA, June 26-29.*

Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C., 2001. PrefixSpan: mining sequential patterns efficiently by prefix projected pattern growth. In *Proc. of the 17th International Conference on Data Engineering, Heidelberg, Germany, pp. 106-115.*

Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C., 2004. Mining sequential patterns by pattern growth: the PrefixSpan approach. *IEEE Trans. on Knowledge and Data Engineering, Vol. 16(10), pp. 1-17.*

Pei, J., Han, J., Mortazavi-Asl, B., Zhu, H., 2000. Mining access patterns efficiently from web logs. In *Proc. of the 4_{th} Pacific-Asia Conference on Knowledge Discovery and Data Mining, Kyoto, Japan, pp. 396-407.*

Zaki, M. J., 2001. SPADE: an efficient algorithm for mining frequent sequences. *Machine Learning, Special Issue on Unsupervised Learning, Vol.42(1-2), pp. 31-60.*