**Mining Frequent Closed Sequential Patterns with Time Constraints using Termination** **Criteria**

Van Nguyen Thi, Jim Z. C. Lai

Department of Computer Science and Engineering

National Taiwan Ocean University

Keelung, Taiwan 202

ABSTRACT

In this paper, a frequent closed sequential patterns with time constraints algorithm is developed. The proposed method uses the developed early termination criteria to speed up the process of generating patterns. This algorithm first determines scan sequential database and deletes infrequent items from all transactions of data sequence. By determining a set of valid transactions, whose frequent items are used for generating type-1 or type-2 patterns, this method perform checking support count of each item belongs to valid transaction to find frequent closed patterns*.* Using also the developed termination condition, the proposed method will significantly reduce the mining time. Compared with close time constrained sequential mining (CTSP), which is the unique algorithm presented to solve the problem of mining closed sequential patterns with time constraints in our best knowledge, the presented approach can reduce the computing time by 61.5% using the sequential databases, generated from T10I4D100k data set, with number of data sequences = 5,000, minimum gap = 12, maximum gap = 16, sliding window = 3 and minimum support threshold = 0.5%. Using the same data set, the proposed method can reduce the computing time of CTSP by 37.3% with number of data sequences = 5,000, minimum gap = 5, maximum gap = 16, sliding window = 3, and minimum support threshold = 1%. Experimental results show that this method is more remarkable when a larger data set with smaller minimum support threshold is used.

*Index Terms—Data mining*, *closed patterns*, *time constraints, mining frequent patterns***1. Introduction**

Data mining refers to extracting or mining knowledge from a large amount of data. Sequential pattern mining is one of the important issues in the research of data mining. The mining is to discover all the frequent sub-sequences in a sequence database, which contains some sequences. Each sequence consists of a list of events (or elements) and each event has a set of items. Note here that a transaction is an event. A sequence of transactions contains a pattern, if each element of the pattern belongs to some transactions of the sequence. Many studies have indicated that specifying constraints may increase the accuracy of mining results in practice. Various constraints, such as item, length, super-pattern, duration, and gap, can be specified to find more desirable patterns. Some constraints can be handled by a post-processing on the result of mining without constraints. However, time constraints affect the support computation of patterns so that they cannot be handled without adapting time attributes into the mining algorithms.

In addition to time-constrained sequential pattern mining [], the issue of mining sequential patterns has been extended. Various topics such as maximal sequential pattern mining [], closed sequential pattern mining [ ], top-k sequential pattern mining [ ] and so on are studied. A sequential pattern is maximal if it is not a subsequence of another frequent sequence. It is closed if no any frequent super-sequence has the same support. Although the closed patterns are more compact than the complete set of patterns, the same amount of information can be derived. Recently, mining closed patterns rather than complete patterns has attracted many researches is more helpful to users. Several algorithms [ ] were introduced to deal with the problem of mining closed sequential patterns. The CloSpan [ ] and BIDE [ ] algorithms discover patterns using pattern-growth methodology. The CloSpan algorithm needs to keep potential closed patterns and verify them afterward. The BIDE algorithm was presented to directly output the closed patterns. The Closed Time Constrained Sequential Pattern Mining(CTSP) algorithm [ ], which is a pattern-growth approach [ ], uses time-indexing to mine frequent patterns with time constraints of minimum gap, maximum gap and sliding window. As shown in reference [ ], CTSP is the first algorithm in our knowledge about mining frequent closed time constraint sequential pattern. Three time constraints including minimum gap, maximum gap and sliding time window are specified to enhance the semantics of sequence discovery. The minimum time interval between two adjacent transactions of a pattern is denoted as the minimum gap. Similarly, maximum gap refers to the maximum time interval between two adjacent transactions of a pattern. Sliding window is defined as the time interval that the items of a pattern may be from two adjacent elements occurring within this specified time interval from a sequence. Note here that when minimum gap equals maximum gap, the time gap is called exact gap.

The drawback of CTSP is that infrequent patterns cannot be detected without counting their supports. This will require more computing time of generating time-constrained frequent sequential patterns and more memory space of storing candidate frequent sequential patterns.

Since we are interested in frequent sequences instead of frequent items, a fast algorithm to mine frequentsequential *k*-sequences for *k* ≥ 2 will be developed in this paper. The proposed method first determines frequent items. This method uses a termination condition developed in this paper to stop the process of generating infrequent patterns of longer length from a frequent sequential pattern. Using this kind of approach, the computational complexity is reduced. The proposed method further uses the set of valid transactions for a pattern, which is not used by CTSP, to speed up the process of generating frequent sequential patterns.

Note here that the proposed approach does not generate candidate *k*-sequences. This paper is organized as follows. Section 2 describes related works. Section 3 presents the algorithms developed in this paper. Some experimental results are given in Section 4 and concluding remarks are presented in Section 5.

**2. Related Works**

In this section, we will briefly describe CTSP algorithm [ ].

Let *I* = {*i*1, *i*2,…, *im*} be a set of items and *ds* be a data sequence, where each element (transaction) *t* in *ds* is a set of items such that *t* ⊆ *I*. Denote *DB* as the sequential database, which consists of |*DB*| data sequences. Each data sequence is identified by sequence identification (SID) and each transaction *t* in *ds* is associated with transaction identification (TID). Similarly each item in a transaction *t* has item identification (IID). The IID and TID, respectively, indicate the position of an item in a transaction and the location of a transaction in a data sequence. Let A be a set of items. A transaction *t* contains A, if and only if A ⊆ *t*. A data sequence *ds* with *n* elements is represented by *ds* = {*ot*1*t*1, *ot*2*t*2,…, *otntn*}, where *oti* (*i =* 1 to *n*) is the occurring time of the *i*th transaction *ti*. A data sequence with *k* items is called *k****-***sequence. A frequent sequential sequence is called a frequent sequential pattern.A sequential pattern *p* with *n* elements can be represented by *p* = [*b*1*t*1,*e*1, *b*2*t*2,e2,…, *bntn*,*en*], where *bi* and *ei*(*i* = 1 to *n*) denote the beginning time and ending time of the *i*th transaction *ti* and *bi* ≤ *ei*. If a sequential pattern *p* can be found from a data sequence *ds*, then we say the data sequence *ds* contains pattern *p*. The number of data sequences containing a pattern *p* is called the support of *p* and is denoted as *p*.*sup*.

If a sequential pattern *p* satisfies a minimum support (*min\_sup*) and three time constraints : minimum gap (*mingap*), maximum gap (*maxgap*) and sliding window (*swin*) then the pattern *p* is called a time-constrained frequent sequential pattern, where *p*.*sup* ≥ *min\_sup*×|*DB*| and |*DB*| is the number of data sequences in the sequential database *DB*. A data sequence *ds* is a closed time-constrained sequential pattern, abbreviated as closed time-pattern, if it is a time-pattern and is closed. The set of time-constrained frequent closed sequential patterns with *k*-items is denoted as *Lk*.

2.1 CTSP algorithm

We refer to the CTSP algorithm, which mines patterns within the pattern-growth framework. The techniques used are similar to the pseudo projection version of PrefixSpan algorithm [ ] and bi-direction closure checking in BIDE algorithm [ ], while it can handle constraints minimum/maximum gaps and sliding time-window. Assume that the *DB* can fit into the main memory, CTSP first loads *DB* into memory (as *MDB*) and scans *MDB* once to find all frequent items. With respect to each frequent item, CTSP then constructs a time-index set for the 1-sequence item and recursively forms time-patterns of longer length. The time-index set is a set of (data-sequence pointer, time-index) pairs. Only those data sequences containing that item would be included. The time-index indicates the list of last start time : last end time (lst : let) pairs.

Given a frequent pattern *p* and a frequent item *x*, *p*’is a type-1 pattern if it can be obtained by adding {*x*} after the last element of *p*. *p*’is called a type-2 pattern, if the last element of *p*’ is the union of {*x*} and the last element of *p*. Given a time-index of *p* in a data sequence *ds*, CTSP first determines the *forward extension periods* (FEP) to find the potential frequent item to form pattern *p*’. Let the time-index of *p* in *ds* be denoted as (lst1, let1, lst2, let2,…, lst*l*, let*l*). The FEP to generate type-1 pattern will satisfy the following condition : let*i* + *mingap* ≤ FEP ≤ lst*i* + *maxgap,* where *i* from1to *k* . For the type-2 pattern, the corresponding *forward extension periods* should satisfy : let*i* − *swin* ≤ FEP ≤ lst*i* + *swin,* where *i* from1to *k*. To form contiguous super sequence *p*’ from a pattern *p*, CTSP checks support count for each item belongs to FEPs. If any support count of an item is equal to the support count of *p,* then *p* is not closed*.* Otherwise, continue to backward checking. Similarly, we find the *backward extension periods* (BEP). Given a frequent pattern *p* = <e1…er…ew> contained in *dsi*.If e1 ⊆ est1∪…∪eet1, …, e*r* ⊆ est*r*∪…∪eet*r*, …, e*w* ⊆ est*w*∪…∪eet*w*, where est1, …, est*w*, eet1, …, eet*w* are elements in *dsi*, the list of timestamps [st1 : et1, …, st*r* : et*r*, …, st*w* : et*w*] where *str* : let*r* is called the timeline of *p* in *dsi*. The BEP satisfies either one of the following conditions : (1) et1 - *maxgap* ≤ BEP ≤ st1 - *mingap* (2) et*i* - *swin* ≤ BEP ≤ st*i* or et*i* ≤ BEP ≤ st*i* + *swin,* where *i* from1to *w*. CTSP also checks support count for each item belongs to BEPs. If any support count of an item is equal to the support count of *p,* then *p* is not closed. Otherwise, frequent closed time constrained patterns are output. Continue this process until no longer length frequent closed time constrained patterns are generated.

At this stage, we can present CTSP as follows :

CTSP Algorithm

1. Input a sequence database (*DB)*, minimum support (*min\_sup*), minimum gap (*mingap*), maximum gap (*maxgap*) and sliding time-window (*swin*) .
2. Load *DB* into memory (as MDB) and scan *DB* to determine the set of frequent items .

(3) For each frequent item *x* :

(3.1) form the sequential patttern *p* = [{*x*}];

(3.2) scan MBD once to construct *p*-Tidx, where *p*-Tidx is the time index of *p*;

(3.3) call CMine (*p*, *p*-Tidx) to mine the closed time patterns.

(4) Output the set of all closed time-constrained sequential patterns.

The subroutine CMine (*p*, *p*-Tidx) mines frequent type-1 and type-2 patterns and is presented below.

Subroutine CMine (*p*, *p*-Tidx)

Parameter : *p* = prefix with support count, *p*-Tidx = time-index set for *p*

1. for each data sequence *ds* in the *p*-*DB*, // *p*-*DB* : sequences indicated in *p*-Tidx
2. collect the FEPs of type-1 and type-2 patterns, respectively.
3. for each item in the FEPs of type-1 and type-2 patterns, add one to its support

count, respectively.

1. if any support count of a stem is equal to the support count of *p*, then *p* is not closed.

Otherwise, output *p* if Backward(*p*, *p*-Tidx) return “closed”. //Backward checking valid

1. for each item *x*’ found in the FEPs of type-1 pattern and <(*x*)>.sup ≥ minsup,
2. form the type-1 pattern *p*’ by extending stem *x*’.
3. construct *p*’-Tidx, time-index set of *x*’.
4. call CMine(*p*’, *p*’-Tidx).
5. for each item *x*’ found in the FEPs of type-2 pattern and <(*x*)>.sup ≥ minsup,
6. form the type-2 pattern *p*’ by appending stem *x*’.
7. construct *p*’-Tidx, time-index set of *x*’.
8. call CMine(*p*’, *p*’-Tidx).

Subroutine : Backward(*p*, *p*-Tidx)

Parameter : prefix *p* = <e1…e*r*…e*w*>, *p*-Tidx = time-index set

1. for each elemente*j*in *p*
   1. for each data sequence ds in the *p*-*DB*, // *p*-*DB* : sequences indicated in *p*-Tidx
      1. find the BEPs of e*j* in *ds*
      2. add one to the support count of the BEIs
   2. if any support count of a BEI is equal to the support count of *p*, return “not

closed”.

1. return “closed”

**3. Proposed Algorithm**

In this section, we will present a method, which speeds up the generation of frequent sequential patterns. This method makes use of the transaction identification (TID) and item identification (IID) to support the generation of valid transactions. This process will reduce the computing time of generating *p*’ of (|*p*|+1)-sequences from pattern *p*. From the experiments, we can find that the proposed method has the less computing time than CTSP, which is the only method mining closed frequent sequential pattern with time constraints to our knowledge.

**3.1. Frequent sequential pattern generation algorithm using valid transactions**

In this section, some basic definitions for the proposed method will be introduced. Denote *Lk* as the set of frequent *k*-sequences. Let *p*∈*Lk* be a frequent pattern of length *k*. That is, *p* has *k* items. Denote the last added item to *p* as *LIT*, which is an element in *tj* (the *j*th element of a data sequence *dsi* in *DB*). Suppose that pattern *p* can be found *r* times from a sequence *dsi*. The time interval record (Tir*i*) of pattern *p* with respect to *dsi* is denoted as *p*- Tir*i* and is represented by *p*-Tir*i* = {*i*, Tir*i* } = {*i*, (lst1, TID1, IID1), (lst2, TID2, IID2),…, (lst*r*, TID*r*, IID*r*)}, where *i* is the SID (sequence identification); lst*j* is last-starting time of the *j*th repetition of *p* in *dsi*. TID*j* and IID*j* (*j =* 1 to *r*) are the transaction identification and item identification, respectively, where TID*j* and IID*j* specify the last item *LIT* adding to *p*. The time interval record of pattern *p* is denoted as *p*-Tir = {*p-*Tir*i* }. Note here that the support of *p* equals | *p*-Tir|.

Denote let*j* as the last-ending time of the *j*th repetition of *p* in *dsi*. Note here that let*j* equals , where  is occurring time of the transactionwith TID = TID*j* in *dsi*. In the case that *LIT* is the last element of  and pattern *p*satisfies one of the following conditions : (a) (– lst*j*) > *swin* and (– lst*j*) > *maxgap* (b) (– lst*j*) > *swin* and (–) < *mingap* as shown in figure 1, then a pattern *p*’ of length (*k*+1) cannot be formed, where  is the occurring time of the transaction *tnext*, whose transaction identification is (TID*j*+1), next to . This is due to *p*’ will not satisfy (1) the maximum gap or sliding window constraint if condition (a) is satisfied, (2) the sliding window or minimum gap constraint if condition (b) is held. If one of the above two conditions holds, no valid transaction exists for *p*’, where a valid transaction for *p*’ is used to form sequential patterns of length (*k*+1).

Denote the support of a pattern *p* as *p*.*sup*, where *p*.*sup* = |*p*-Tir|. If there exist *k* sequences with SVT (*p*) = ∅ and (*p*.*sup* - *k*) < *min\_sup*×|*DB*|, then no frequent sequential pattern *p*’ of length (|*p*|+1) can be formed, where SVT (*p*) is the set of valid transactions for a pattern *p*. This is because *p*’.*sup* ≤ (*p*.*sup* - *k*) < *min\_sup*×|*DB*|. The proposed method will use the early termination condition : (there exist *k* sequences with SVT (*p*) = ∅ and (*p*.*sup* - *k*) < *min\_sup*×|*DB*|), to stop the process of generating *p*’ from *p*. This early stopping criterion is not used by CTSP. The early termination criterion is implemented in the pattern generation algorithm presented in the following sections.

The proposed method first determines frequent items and all transactions of a data sequence are lexicographically sorted. Next, this method deletes all infrequent items from all transactions of a data sequence. Note here that the frequency of an infrequent item is less than *min\_sup*×|*DB*|.

*ds*

ot*tnext*

*swin*

lst*j*

(a)

*maxgap*

*ds*

(b)

ot*tnext*

*swin*

lst*j*

ot*TIDj*

*mingap*

Figure 1: The condition that *p*’ cannot be formed : (a) (– lst*j*) > *swin* and (– lst*j*) > *maxgap*, (b) (– lst*j*) > *swin* and (– ) < *mingap*

Let *L*1 consist of all frequent items. For each *x*∈*L*1, set *p* = [{*x*}]. For a frequent pattern *p* of length *k*, two types of potential patterns (type-1 pattern and type-2 pattern) of length (*k*+1) will be generated. The generated pattern of length (*k*+1) is denoted as *p*’. Given a frequent pattern *p* and item *x*, *p*’is a type-1 pattern if it can be obtained by adding {*x*} after the last element of *p*. *p*’is called a type-2 pattern, if the last element of *p*’ is the union of {*x*} and the last element of *p*. Given a time interval record of pattern *p* in a data sequence *dsi*, the proposed method perform forward checking valid by determining a set of valid transactions, whose frequent items are to be used for generating the type-1 or type-2 pattern *p*’ of *p*. This set of valid transactions for a pattern *p* is denoted as SVT*i* (*p*) with respect to a data sequence *dsi*. Let the time interval record of pattern *p* with respect to *dsi* be denoted as *p*-Tir*i* = {*i*, (lst1, TID1, IID1), (lst2, TID2, IID2),…, (lst*r*, TID*r*, IID*r*)}. To generate type-1 pattern, a transaction *t*SVT*i* (*p*) should satisfy the following condition: (+ *mingap*)lst*j*+ *maxgap* for *j* = 1 to *r*, where is the occurring time of transaction *t* in *dsi*. If the item *LIT* indicated by IID*j* is not the last element of , then all elements in transaction  from (IID*j*+1) to |*t*| can be used form the type-2 pattern *p*’. For all items of a transaction *t*, which fulfills the following requirement can also be used to generate the type-2 pattern *p*’: for *j* = 1 to *r*. Note here that the proposed method will not look back to find frequent items to form type-2 patterns and CTSP will. From valid transactions, we count support of each item. If any item has support count equal to support of pattern *p*, then *p* is not closed. Otherwise, we continue to backward valid checking. Given a frequent pattern *p* = <e1… e*r*…e*w*> contained in *dsi*, the proposed method determines a set of valid transactions. This set of valid transactions for a pattern *p* is denoted as SVT*i* (*p*) with respect to a data sequence *dsi*. For each timeline [st1 : et1, …, st*r* : et*r*, st*w* : et*w*] of *p* in *dsi*, the BEP satisfies either one of the following conditions. To generate type-1 pattern, a transaction *t*SVT*i* (*p*) should satisfy the following condition: , where  is the occurring time of transaction *t* in *dsi*. If the item LIT indicated by IID*j* is not the last element of , then all elements in transaction from (IID*j*+1) to |*t*| can be used form the type-2 pattern *p*’. For all items of a transaction *t*, which fulfills the following requirement can also be used to generate the type-2 pattern *p*’:  or  for *j* = 1 to *w*, where *ott* is the occurring time of transaction *t* in *dsi* . From valid transactions, we count support of each item. If any item has support count equal to support of pattern *p*, then *p* is not closed. Otherwise, *p* is outputted.

At this stage, we would like to present the pattern generation algorithm and the proposed algorithm, which is referred to as MFCTCSP, below.

Proposed Algorithm

1. Input a sequence database (*DB)*, minimum support (*min\_sup*), minimum gap (*mingap*), maximum gap (*maxgap*) and sliding time-window (*swin*).
2. Load *DB* into memory (as MDB) and scan *DB* to determine the frequencies of all items in *DB* and delete the infrequent items from each transaction of all data sequences.
3. Sort all transactions in data sequences lexicographically. For each SID in *DB* and each frequent item *x* of a transaction *t* :
   1. set *p* = [{*x*}]
   2. construct *p*-Tir where *p*-Tir is the time interval record of *p*.
   3. use the pattern generation algorithm to mine the closed time patterns.
4. Output the set of all closed time-constrained sequential patterns.

Pattern Generation Algorithm

1. Input pattern *p*, the time interval record *p*-Tir and timeline *p*-Til. Set *j* = 0, S1temp = S2temp = ∅, and S’ = *p*’-Tir = ∅. Let *support* = | *p*-Tir|.
2. Fetch the next SID in *p*-Tir until no SID can be fetched :

Generate the set of valid transactions of *p* for type-1 and type-2 patterns, which are denoted as SVT1 and SVT2, respectively. Let SVTSID (*p*) = SVT1 ∪ SVT2. If SVTSID (*p*) = ∅ : (a) set *support* = *support* – 1; (b) if *support* < *min\_sup*×|*DB*|, return and output S’ = ∅ , otherwise go to step (3)

1. Checking valid
   1. If SVT1 ≠ ∅, for each transaction *t*∈SVT1 and count the support each item *x*∈*t* : (a) if *x.sup* ≠ *p*. *sup* and *x* ∉S1temp : set S1temp = S1temp∪{*x*} and go to step (3.3), otherwise *p* is not closed.
   2. If SVT2 ≠ ∅, for each *t*∈SVT2 :

(3.2.1) If *t*=, for each element *x* of *t* from (IID*l*+1) to |*t*| : If *x*∉S2temp and *x.sup*≠ *p*. *sup*: set S2temp = S2temp∪{*x*} and go to step (3.3), otherwise *p* is not closed.



(3.2.2) If *t*≠, for each item *x*∈*t* : If *x*∉S2temp and *x.sup* ≠ *p*.*sup*: set S2temp = S2temp∪{*x*} and go to step (3.3), otherwise *p* is not closed.



* 1. Backward checking valid for *p* = <e1… e*r*…e*w*>

Fetch the element from 1 to *w*: use “check1SVT”, “check2SVT” variable to determine whether stop generating item. At beginning, assign “check1SVT = true; check2SVT = true. If check1SVT = false, stop generating valid item for type1. It implies similarly to check2SVT.

Fetch the next SID *i* in *p*-Til until no SID can be fetched : Assign fmax = -1,where fmax is the maximum frequency of stems in S1temp.

* + 1. Operate with e1.

Generate the set of valid items of *p* for type-1 and type-2 patterns, SVI1 and SVI2, respectively.

* + - 1. If check1SVI = true

for each item *x*∈SVI1:

(a) if *x* ∉ S1temp

If (*1+( p*.*sup*-*i)* ≥ *min\_sup*×|*DB*|),

set S1temp = S1temp∪{*x*} and add one to support count of *x* or *x.sup* = 1.

if 1 > fmax, fmax = 1

else if 1 > fmax

check1SVI = false; break;

(b) if *x* ∈S1temp

If (*x.sup+1)+( p*.*sup*-*i)* ≥ *min\_sup*×|*DB*|*,*

set S1temp = S1temp∪{*x*} and add one to support count of *x* or *x.sup* = *x.sup+*1;

If *x.sup* > fmax, fmax = *x.sup*

else if *x.sup* > fmax

check1SVT = false; break;

if fmax *+( p*.*sup*-*i)* < *min\_sup*×|*DB*|

check1SVT = false, break;

* + - 1. If check2SVT = true

for each item *x*∈SVT2:

(a) if *x* ∉ S2temp

If (*1**+( p*.*sup*-*i)* ≥ *min\_sup*×|*DB*|),

set S2temp = S2temp∪{*x*} and add one to support count of *x* or *x.sup* = 1.

if 1 > f2max, f2max = 1

else if 1 > f2max

check1SVT = false; break;// stop generating

(b) if *x* ∈S2temp

If (*x.sup+1)+( p*.*sup*-*i)* ≥ *min\_sup*×|*DB*|*,*

set S2temp = S2temp∪{*x*} and add one to support count of *x* or *x.sup* = *x.sup+*1;

If *x.sup* > f2max, f2max = *x.sup*

else if *x.sup* > f2max

check1SVT = false, break;

if f2max *+( p*.*sup*-*i)* < *min\_sup*×|*DB*|

check2SVT = false, break;

If check1SVI = false and check2SVI = false, *p* is closed (support of any item in SVI always is smaller than support of *p*).

If check1SVI = true

for each item *x*∈SVI1 If *x.sup* > *p*.*sup, p* is closed .Otherwise, *p* is not closed

If check2SVI = true

for each item *x*∈SVI2 If *x.sup* > *p*.*sup,* *p* is closed . Otherwise, *p* is not closed

1. Output frequent closed pattern *p;* from each item in S1temp, S2temp, update type-1 and type-2 closed pattern *p*’, repeat from step 3 for *p*’.

**3.2. An illustrative example**

In the following, we will give an example to illustrate the proposed method. The major differences between the proposed method and CTSP are : (a) the proposed method will not look back to find frequent items to form type-2 patterns and CTSP will; (b) a termination criterion is used by the proposed method to stop the process of generating infrequent patterns in forward checking and stop earlier closed checking in backward checking; (c) CTSP checks the time index for each item to form *p*’ from *p*; while the proposed method uses each item from a transaction in the set of valid transactions to speed up the generation of *p*’. To illustrate the process of the proposed method, an example of sequential database from reference [] as shown in table 1 is used. This sequential database consists of four sequences. In table 1, “5{*a*, *f*}” with SID = 1 means that the occurring time of the transaction {*a*, *f*} in *ds*1 is 5. Similar meanings are applied for other transactions. Table 1 presents the mined frequent sequential patterns with *min\_sup* = 50%, *mingap* = 3, *maxgap* = 15 and *swin* = 2 using the proposed method.

Step 1 : *Load DB into memory, then find all frequent items and delete all infrequent items from all transaction in data sequences*.

MFCTSP first reads the *DB* into memory and scans the in-memory *DB* (abbreviated as *MDB*) once to find frequent items. Then, we have the frequent items in this data set are “*a*,” “*b*”, “*c*”, “*d* ” with frequencies of 3 and “*e*” with frequencies of 2.

Step 2 : *Construct the time interval record set of the sequential patterns.*

Take [{a}] for example, MFCTSP scans MDB and constructs the time interval record set [{a}]-Tir, as shown in Figure 5(a). Using the proposed method, we can find that the set of time interval record of pattern for pattern [{*a*}] is [{*a*}]-Tir = {{*a*}-Tir1, {*a*}-Tir2, {*a*}-Tir4}, where [{*a*}]-Tir1 = {1, (5, 2, 1), (31, 4, 1)}. The time interval records for C2 and C4 are processed accordingly. Note here that [{*a*}]-Tir1 = {1, (5, 2, 1), (31, 4, 1) implies that item “*a*” is in the first element of the second transaction and in the first element of the fourth transaction *ds*1. The same implications hold for [{*a*}]-Tir2 and [{*a*}]-Tir4. The [{*a*}]-Tir is used in subroutine CMine, which searches potential type-1and type-2 stems.

*Step 3 : Using valid transactions to select stems in FEP from the time interval record set*

With respect to prefix [{*a*}], the set of valid transactions for a pattern [{*a*}] is denoted as [{*a*}]-SVT= {{*a*}-SVT1, {*a*}-SVT2, {*a*}-SVT4}. In *ds*1*,* to generate type-1, a transaction *t*{{*a*}-SVT1 should have occurring time belong to [8:20, 34:46] from [lst1 +*mingap* : ot1+*maxgap*, lst2+*mingap*: ot2+*maxgap*], where lst1 = ot1 = 5 and lst2 = ot2 = 31. The SVTs of C2 and C4 are obtained correspondingly. Now, all items are selected from valid transaction and counted the support from all data sequences. Item b passes the threshold to be a stem. Similarly, the type-2 SVTs contained all transactions belongs to [3:7, 29:33] in C1 and in C2 is [4:8], and in C4 is [3:7]. Items c then has enough support to be a stem.

*Step 4 : Check the closure of sequential pattern*

Prefix [{a}] has support 3 but the support of stems c (and b) is 2, so CTSP calls subroutine Backward to count the supports of potential BEIs. The BEPs corresponding to [{a}] are shown in Figure 5(d). No BEI is found to satisfy the time constrains. Consequently, [{a}] : 3 is outputted as a closed time-pattern.

*Step 5 : Recursively grow (discover) pattern*

We will generate three frequent patterns [{*a*}, {*b*}], [{*a*, *c*}] from pattern [{*a*}] and CMine is recursively called. Considering type-1 FEPs of [{*a*}, {*b*}], which are [21 : 33] for C1 and [13 : 25] for C2, no stem can be found so that no pattern of prefix [{*a*}, {*b*}] is formed. Similarly, the type-2 FEPs from [let1−*swin* : lst1+*swin*] for C1 and C2 are [16 : 20] and [8:12], respectively. No pattern of prefix [{*a*}, {*b*}] also can be formed. Subroutine backward is then called to check the closure of [{*a*}, {*b*}].

The timelines of prefix [{*a*}, {*b*}] are [5 : 5, 18 : 18] for C1 and [6 : 6, 10 : 10] for C2. The BEPs corresponding to type-2 extension of element [{b}] are [16 : 20] for C1 and [8 : 12] for C2. No items are found to extend element [{b}]. Similarly, the BEPs corresponding to type-2 extension and type-1 extension of element [{a}] can be found as shown in Figure 5(e). The item c appears in both BEPs of C1 and C2 (type-2 extension) so that [{*a*}, {*b*}] is not a closed time-pattern. That is, [{*a*}, {*b*}]’s contiguous super sequence [{*a,c*}, {*b*}] has the same support. Since [{*a*}, {*b*}]is processed, CMine then continues on the type-1 and type-2 stems of [{*a*, *c*}].

With respect to prefix [{*a*, *c*}], the set of time interval records of pattern for pattern [{*a*, *c*}]-Tir = {[{*a*, c}]-Tir1, [{*a*, *c*}]-Tir2}, where [{*a*, *c*}]-Tir1 = {1, (3, 2, 1), (31, 4, 1)} and [{*a*, *c*}]-Tir2 = {2, (6, 1, 1), (18, 4, 1)}. Applying the proposed method for pattern [{*a*, *c*}], we will obtain the frequent pattern [{*a*, *c*}, {*b*}] with support = 2 and the process need not to call backward closure checking. The process continues to mine with prefix [{*a*, *c*}, {*b*}]. We have [{*a*, *c*}, {*b*}] -Tir = {{1, (3, 3, 1)}, {2, (6, 2, 1)}}. Since SVT1 ([{*a*, *c*}, {*b*}]) = ∅, we will have (|[{*a*, *c*},{*b*}]-Tir| -1 }<*min*\_*sup*×|*DB*| = 2. The proposed method will stop the mining process using [{*a*, *c*}, {*b*}] to find frequent patterns of length 4. However, CTSP will continue the mining process of using [{*a*, *c*}, {*b*}] to generate patterns of length 4. This will explain the proposed method can reduce the computing time of CTSP. No more pattern is formed and the mining of prefix [{a}] now stops.

This method then applies steps on [{a}], [{c}], [{d}], and [{e}]. The complete set of closed time-patterns is listed in the rightmost column in Table1.

Table 1 : An example of sequential database

|  |  |  |
| --- | --- | --- |
| Sid | Sequence | Closed time-constrained sequential patterns  (*minsup =* 50%, *mingap* = 3, *maxgap* = 15, *swin* = 2) |
| C1 | <3(c)5(a, f)18(b)31(a)45(f)> | {{a}} : 3, {{a}{d}} : 2, {{a, c}{b}{a}}: 2, {b} : 3, {b}{e}{d}} : 2, {{b}{d}} : 2, {c} : 3, {c d} : 2 , {d} : 3, {e}: 3 |
| C2 | <6(a, c)10(b)17(e)18(a)24(c, d)> |
| C3 | <1(b)20(b, g)27(e)34(d, g)35(g)> |
| C4 | <5(a)10(d)21(c, d)26 (e)> |



Figure 5: Time interval record set and BEPs of patterns

**4. Experimental Results**

To evaluate the performance of the proposed algorithm, several synthetic databases, the T10I4D100k data set and Retail data set are used. The synthetic sequential databases are generated using the similar method described in reference []. For the synthetic databases, denote *S*ave as the average number of items in a data sequence. For each transaction *t*, its items are from a set of items (*I*s). For each *x in I*s, there exits a weight *witm*, which is generated randomly, associated with it. The range of *witm* is from 1 to *P*max. The weight *witm* models the probability of *x* being picked to generate a transaction in a data sequence *ds.* An extended set of items (*I*ext) determined from *I*s is used to generate transactions. That is, for each item *x* in *I*s, *x* is repeated *witm* times in *I*ext.

The transaction size |*t*| for transaction *t* is generated randomly from *T*min to *T*max, where *T*min and *T*max are the minimum and maximum transaction sizes, respectively. To generate a transaction *t*, |*t*| different items are picked randomly from *I*ext. That is, a transaction *t* is generated using the Transaction Generation Algorithm (TGA), which is presented below.

Transaction Generation Algorithm

1. Input the extended set of items *I*ext.
2. Generate the transaction size |*t*| for transaction *t*. Set *s* = 1 and *t* = ∅, where ∅ is an empty set.
3. Pick an item *x* from *I*ext randomly. If *x*∉*t*, update *t* = *t*∪*x* and go to step (5).
4. Repeat step (3).
5. Set *s* = *s* + 1. If *s* ≤ |*t*|, go to step (3); otherwise output *t*.

The number of items in a data sequence *ds* is generated from a Poison distribution with mean *S*ave. Note here that |*ds*| represents the number of items in *ds*. A transaction in *ds* is picked randomly from a set of transactions S*trn*. |S*trn*|, which is the cardinality of S*trn*, presents the number of transactions in S*trn*. The transaction generation algorithm (TGA) is used to generate transactions for S*trn*. Let Δot be the difference of the occurring time between two neighboring transactions in a data sequence *ds*. Denote Δot*min* and Δot*max* as the minimum and maximum values of Δot. At this stage, we would like to present the sequential database generation algorithm (SDBGA) as follows :

Sequential Database Generation Algorithm

1. Input the set of transactions S*trn* and the cardinality of sequential database |*DB*|, where *DB* = {*dsj*} is the sequential database.
2. For *j* = 1 to |*DB*| :
   1. Generate |*dsj*| from a Poison distribution with mean *S*ave, where |*dsj*| represents the number of items in *dsj* and set *dsj* = ∅, ot = 0 and *cur\_len* = 0.
   2. Pick randomly a transaction *t* from S*trn*. Generate Δot randomly between Δot*min* and Δot*max*. Update ot = ot +Δot and assign ot as the occurring time of transaction *t*.
   3. If (*cur\_len*+|*t*|)<|*dsj*|, update *dsj*= *dsj*∪{*t*}*,* set *cur\_len* = *cur\_len* + |*t*| and go to step (2.2); otherwise delete last (*cur\_len* + |*t*| - |*dsj*|) items from *t* and update *dsj*=*dsj*∪{*t*}.
3. Output *DB*.

All computing is performed on an Intel(R) Core(TM) i7 CPU 930 @2.80GHz with memory of 12 GB. To evaluate the performance of the proposed method, the proposed algorithm MFCTCSP is compared only with CTSP in terms of computing time, as in related work, allow only three time constraints.

**4.1. Example 1 : The synthetic databases with short transactions**

In this example, several synthetic databases are obtained using the similar method described in reference []. In this example, |*I*s| = 1,500, *Tmin* = 1, *Tmax* =10, *P*max = 100, |S*trn*| = 2000, Δot*min* = 1, Δot*max* = 40 and *S*ave = 1000 are used, where |*I*s| is the size of a set of items; *T*min and *T*max are the minimum and maximum transaction sizes, respectively; *P*max is the maximum value of *witm*, which models the probability of an item being picked to generate a transaction in a data sequence; |S*trn*| presents the number of transactions in S*trn*; Δot*min* and Δot*max* are the minimum and maximum values of the difference of the occurring time between two neighboring transactions in a data sequence; and *S*ave is the average sequence size. The chart below performs presents the computing time of mining frequent sequences with various values of |*D*|, *min\_sup* = 0.5%, *mingap* = 5, *maxgap* = 16 and *swin* = 3 where |*D*| is from 1,000 to 5,000.

|  |  |  |
| --- | --- | --- |
| Number of data sequences (|D|) | CTSP | MFCTCSP |
| 1K | 1647.37 | 440.783 |
| 2K | 4671.263 | 825.486 |
| 3K | 9131.658 | 1255.093 |
| 4K | 15613.997 | 1683.112 |
| 5K | 23981.169 | 2172.11 |

* 1. **Example 2: The** **T10I4D100k data set**

In this example, five sequential databases with |*D*| = 1,000, 2,000, 3,000, 4,000 and 5,000 are generated from a subset of the T10I4D100k data set [19] using the sequential database generation algorithm. Twenty thousand transactions are picked randomly from the T10I4D100k data set to form S*trn*. In the T10I4D100k data set, the maximum length of transaction is 20 and the number of items is 3,568. In this example, |*I*s| = 3,568, *Tmin* = 1, *Tmax* =20, |S*trn*| = 2000, Δot*min* = 1, Δot*max* = 40 and *S*ave = 2000 are used. Compared with the databases in example 1, the sequential databases in this example have longer transactions and larger data sizes with various values of |*D*|, *min\_sup* = 0.5%, *mingap* = 5, *maxgap* = 16 and *swin* = 3.

|  |  |  |
| --- | --- | --- |
| Number of data sequences (|DB|) | CTSP | MFCTCSP |
| 1K | 314.714 | 153.613 |
| 2K | 595.546 | 311.813 |
| 3K | 879.108 | 472.103 |
| 4K | 1166.874 | 637.354 |
| 5K | 1454.011 | 800.484 |