Available online at www.sciencedirect.com
ScienceDirect
Information Sciences 178 (2008) 1498–1518
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

Efficient strategies for tough aggregate constraint-based sequential pattern mining

Enhong Chen [a], Huanhuan Cao [a], Qing Li [b,*], Tieyun Qian [c]

[a] *Department of Computer Science, University of Science and Technology of China, Hefei Anhui, PR China*
[b] *Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong*
[c] *Department of Computer Science, Wuhan University, Wuhan, Hubei, PR China*

Received 13 November 2006; received in revised form 2 July 2007; accepted 12 October 2007

Abstract

Frequent sequential pattern mining with constraints is the task of discovering patterns by incorporating the user defined constraints into the mining process, thus not only improving mining efficiency but also making the discovered patterns to better meet user requirements. Though many studies have been done, few have been carried out on the "tough aggregate constraints" due to the diffIculty of pushing the constraints into the mining process. In this paper we provide efficient strategies to deal with tough aggregate constraints. Through a theoretical analysis of the tough aggregate constraints based on the concept of total contribution of sequences, we first show that two typical kinds of constraints can be transformed into the same form and thus can be processed in a uniform way. We then propose a novel algorithm called PTAC (sequential frequent Patterns mining with Tough Aggregate Constraints) to reduce the cost of using tough aggregate constraints through incorporating two effective strategies. One avoids checking data items one by one by utilizing the features of promisingness exhibited by some other items and validity of the corresponding prefix. The other avoids constructing an unnecessary projected database through effectively pruning those unpromising new patterns that may, otherwise, serve as new prefixes. With these strategies, our algorithm obtains good performance in speed and space, as demonstrated by experimental studies conducted on the synthetic datasets generated by the IBM sequence generator, in addition to a real dataset.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Frequent sequential pattern; Tough aggregate constraints

1. Introduction

Sequential pattern mining is the task of discovering frequent subsequences as patterns [16] in a sequence database. It has been an active and important field of research and development since it was first introduced in [14]. Mining sequential patterns has found a variety of applications in analyzing genome sequences, capturing the

* Corresponding author. Fax: +852 2788 8292.
E-mail addresses: cheneh@ustc.edu.cn (E. Chen), caohuan@mail.ustc.edu.cn (H. Cao), itqli@cityu.edu.hk (Q. Li), qty@whu.edu.cn (T. Qian).

0020-0255/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2007.10.014

important relationship between network alarm signals in the form of frequent telecommunication alarm sequences, acquiring the information for medical diagnosis or preventive medicine by identifying frequent temporal patterns of symptoms and diseases exhibited by patients, and improving hyperlinked structure of e-commerce websites to promote the sales with frequent user browsing patterns discovered from web server logs [6]. However, early algorithms for mining frequent sequential patterns lack user-controlled focus in the pattern mining process, thus only a small part of the large number of returned sequential patterns is actually of interest to the users. As pointed out in [6], this kind of *unfocused* approach to sequential pattern mining suffers from the drawbacks of disproportionate computational cost for selective users and overwhelming volume of potentially useless results. Indeed, if user-defined constraints can be pushed into the mining process, we can not only improve mining efficiency but also make the discovered patterns meet user requirements better. Recently, a number of different kinds of constraints have been proposed for different applications. The typical examples of these constraints are item constraint, aggregate constraints, regular expression constraints, duration constraints, gap constraints, aggregate constraints, average value constraints [12].

Among the proposed constraints, aggregate constraints are used to express user requirements on the aggregate of items in a pattern. For example, a *Max_Min constraint* is used to express the requirement that the max item's value or the min item's value in a given sequence has to be in a certain range, and a *sum constraint without negative values* is used to denote that the sum of all items' values of a given sequence has to be in a certain range while all values are not negative. The above types of aggregate constraints are in fact easy to deal with and have available effective methods to deal with. Thus such types of aggregate constraints are called simple aggregate constraints. However, there exists another type of aggregate constraints which is more useful and, meanwhile, more difficult to deal with. For example, a marketing analyst may be interested in any sequential pattern whose average price of the contained items is over $100, which is actually an *average value constrain* especially useful for analyzing the retail order sequences. However, such a commonly used constraint is difficult to deal with and there are few relevant algorithms existing. Another example of aggregate constraint beneficial to marketing analysis is the *sum constraint with negative values* which is also difficult to be pushed into the process of mining the desired frequent sequential patterns.

These aggregate constraints which are common but more difficult to deal with are called tough aggregate constraints [12]. In this paper, we focus on tough aggregate constraint-based sequential pattern mining. First, let us introduce these two constraints in detail. Suppose that every item of each sequence in a given sequence database is associated with a negative or positive value, the *sum constraint with negative values* requires that the sum of every item in a desired sequential pattern should not be less than a given constant value. For instance, when mining telecommunication alarm sequences, we may want to assign those interesting items positive values and give the others negative values, thereby finding more important patterns via a *sum constraint with negative values*. Table 1 gives another example to show a sum constraint with negative values. Here, every sequence records the information of a football player's performance in one round of match. An item denotes an action such as goal, dangerous pass, assist, foul, offside, getting yellow card or red card. Each action is associated with a weight. In particular, the action of goal may be assigned with the highest weight 3, and the action of getting red card is of the lowest weight −3. An element records a player's performance in a period of time, and a match is divided into four sessions. In Table 1, the first sequence says that the player Ballack kicked a goal and made a misplay in the first session of the match, and had a dangerous pass in the second session. In the third session, he had an assist. There is no record of the fourth session because he had left the playing field during the third session. In order to analyze the player's performance in different segments, a professional

Table 1
A sequence database of football player's performance in one round of match

Sid	Sequence
Ballack	⟨(goal.3,foul.-1)(dangerouspass.1) (assist.2)⟩
Kaka	⟨(yellow card.-2) (assist.2) (offside.-1)⟩
Zidane	⟨(misplay.-1)(dangerous pass.1)(redcard.-3)⟩
Viera	⟨(goal.3)(dangerous pass.1) (dangerouspass.1) (red card.-3)⟩
Lampard	⟨(yellowcard.-2)(shoot.1,assist.2)(misplay.-1)⟩

Table 2
A sequence database that records the customers' retail orders

Sid	Sequence
Lee	⟨(pen.$10, battery.$1) (bedsheet.$5)⟩
James	⟨(pen.$10)(bedsheet.$5, shirt1.$25, shirt2.$30)⟩
Lily	⟨(bedsheet.$5)(skirt1.$50)(shoe1.$100)⟩
Sabrina	⟨(skirt1.$50)⟩

coach may want to find those patterns[1] whose support is at least 2 and whose sum of weights has to be greater than 0, which is actually an example *sum constraint with negative values*.

Similarly, given a sequence database, an *average value constraint* requires that, for every desired sequential pattern, the average of all its items should not be greater or less than a given constant value. The average value constraint is also very useful in applications. For example, consider the sequence database shown in Table 2. The database records the retail orders of four customers and the value of every item corresponding to its price. It is common that some retail organizations may like to get those sequences whose average price is higher than a given threshold, which is in fact an average aggregate constraint. For example, if the constraint requires that the average price of all items in the pattern is higher than $5, and that its support is not smaller than 2, then <(pen.$10) (bedsheet.$5)>, <(skirt1.$50)> would be a targeted sequential pattern.

As mentioned above, compared with simple aggregate constraints, tough aggregate constraints are more difficult to deal with. The first reason is that tough aggregate constraints, different from other classes of constraints, have different concrete forms and are hard to be tackled in a uniform way. This can be seen from the above two examples. Another reason is that unlike some typical constraints, it is difficult for these constraints to be directly used to prune useless candidate sequences. Compared with monotony and anti-monotony constraint, tough aggregate constraints are a lot more complex and difficult to be used to prune useless sequential patterns, because it is difficult to decide when to prune a pattern with this constraint. For example, given a pattern <(pen.$10)> in Table 2 and a tough aggregate constraint that requires the average value of a target pattern should be larger than 5, it is difficult to decide whether or not use <(pen.$10)> to construct longer sequential patterns. In fact, though <(pen.$10)> satisfies the constraint, some of its super sequences may violate the constraint, such as <(pen.$10) (battery.$1)>, and some of its super sequences may satisfy the constraint as itself, such as <(pen.$10) (battery.$1) (bedsheet.$5)>. Given the difficulty of taking advantage of tough aggregate constraint to prune useless sequential patterns, special designed pruning strategies are needed to be formulated.

In this paper we demonstrate that these typical kinds of tough aggregate constraints can be transformed into the same form and thus can be processed in a uniform way. So our first major contribution of this paper is to construct a framework to deal with the constraints uniformly. As the second contribution, we present more effective strategies than existing work for tough aggregate constraints. Different from existing work on pruning useless candidate sequential patterns, our strategies use divide-and-conquer technique to process candidate sequential patterns to avoid unnecessary checking. In addition, to prune useless sequential patterns, a more optimized partition approach is proposed to reduce the space cost. Moreover, with our strategies, the operations of removing unpromising items and counting the support of all items are performed in the same scan. Thus compared with existing work, our proposed strategies can fully utilize the properties of tough aggregate constraints by further exploiting the features of tough aggregate constraints. Last but not the least, we provide a uniform algorithm framework for dealing with tough aggregate constraints using our strategies. With these strategies, the resultant algorithm obtains better performance in terms of speed and space.

The rest of the paper is organized as follows: Section 2 introduces some related work about frequent sequential pattern mining. Section 3 introduces some background knowledge about sequential pattern mining, and theoretically demonstrates that both the average value constraint and the sum constraint with negative values can be processed using the same strategies. In Section 4, we present the framework of PTAC (*sequential frequent Patterns mining with Tough Aggregate Constraints*), and describe our new strategies and the optimization

[1] In fact, there are commercial companies that have provided the similar analyzing services for coaches in Europe.

scheme in detail. The experimental results and analysis are given in Section 5, and Section 6 concludes the paper with a summary.

2. Related work

For frequent sequential pattern mining, most existing works can be classified into two categories. One category is Apriori based algorithm. Its basic idea is to generate candidate sequential patterns by joining frequent sequential patterns that have been found and then check their frequencies. Thus, the algorithm uses all i-length patterns to generate $(i+1)$ – length patterns and expires if no candidates can be generated. These works include GSP [15], Apriori-all [14], SPADE [19], MSPS [9] and FSPAN [5]. The main drawback of these algorithms is that lots of useless candidates are generated, which dramatically increases the cost of main memory, when the *min_support* is small. The other category is FP-growth based algorithm. Its basic idea is to partition the original sequence database to smaller sub databases by some partition cells, and then to mine patterns in these sub databases. Unless no new patterns can be found, the partition is recursively performed with the growth of partition cells. These works include FreeSpan [8] and PrefixSpan [11]. There are mainly two advantages of these algorithms. The first one is that they can avoid generating numerous candidate sequential patterns and then decrease the cost of main memory. The second is that they reduce the searching space of mining and thus increase the efficiency of mining sequential pattern.

There are some studies focusing on specific application of sequential pattern mining. Chang and Lee [4] and Ho et al. [7] give solutions to mine sequential patterns over online data streams. In addition, in real life, the user of sequential patterns mining always query more than once. In order to mine sequential patterns efficiently, some information of the previous query is potentially useful. Ren and Zong [13] propose an algorithm called MIFSPM, which takes advantage of the information from the previous query to increase the efficiency of current query. Furthermore, some applications such as analysis of DNA sequences do not only focus on the frequency of a given pattern in the sequence dataset, but also take interest in the pattern's frequency in specific sequences. For this problem, [17] introduces the concept of M-seqeunce and gives a solution for this problem based on M-seqeunce. Also in the application of analysis of DNA sequences, sometimes the patterns are too long that most of algorithms of sequential pattern mining have difficulty in finding them in an acceptable time. For this problem, Disc-all [3] adopts DISC strategy and perform better than most of other algorithms when mining long patterns.

To deal with the problem of different constraint-based sequential pattern mining, much research has been done, which can be classified into two categories. The first category studies how to deal with diverse constraints in a general framework. For example, in [12], a framework called Prefixgrowth is built based on a prefix-monotone property. The framework can adopt different strategies for dealing with different constraints while assuring its high performance at the same time. Under this framework, frequent sequential patterns are found recursively in the projected database, and then those patterns satisfying the prefix-monotone constraint are output and used as new prefix to construct a new projected database. In this way, most constraints can be effectively used to prune those undesired patterns as early as possible. The second category is to design efficient mining algorithms to deal with the problem of specific forms of constraint-based sequential pattern mining. For example, for the monotony constraint, ExAnte [1] exploits the pruning power of monotony constraints through a pre-processing step by iteratively pruning infrequent items first, and pruning the sequences not satisfying the monotony constraints next.

For tough constraints, as we discussed above, they have different concrete forms and thus specific methods should be designed to deal with different forms of tough constraints. For example, for MaxGap constraint, the available work includes cSPADE [18], CBPSAlgm [2] and CCSM [10]. However, compared with MaxGap constraint, much less work on the other important classes of tough constraints, especially the tough aggregate constraint, has been done. The most related work for dealing with tough aggregate constraint-based sequential pattern mining is the specific strategies proposed in PrefixGrowth [12]. For tough aggregate constraints which lack prefix-monotone property, PrefixGrowth adopts specific strategies in order to take the advantage of the specific properties of these constraints in the process of mining desired sequential patterns. To handle the average value constraints, it adopts two pruning rules: one is *unpromising sequence pruning rule* and the other is *unpromising pattern pruning rule*, but the corresponding pruning strategies do not fully utilize the benefit of the constraints. As for the sum constraints with negative values, no specific strategies have ever been studied.

3. Sequential pattern mining and tough aggregate constraints

In this section, we first briefly introduce some basic concepts about sequential pattern mining, and then theoretically analyze the tough aggregate constraints.

3.1. Sequential pattern mining

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of *items*. An *element* is a subset of items and denoted as $e = (e_1, e_2, \ldots, e_i)$ where $e_k (k \in [1, i])$ represents an item in I. In an element, the same item can only occur at most once and we assume that all items are sorted in alphabetical order. A *sequence* is an ordered subset of elements and denoted as $s = \langle s_1, s_2, \ldots s_j \rangle$ where $s_k(k \in [1, j])$ is an element. The same element can occur more than once in a sequence. A sequence $\alpha = \langle a_1, a_2, \ldots, a_n \rangle$ is called a sub-sequence of another sequence $\beta = \langle b_1, b_2, \ldots, b_m \rangle$ and β a super-sequence of α, denoted as $\alpha \subseteq \beta$, if there exist integers $1 \leqslant j_1 < j_2 \ldots j_n \leqslant m$ such that $a_1 \subseteq b_{j1}$, $a_2 \subseteq b_{j2}, \ldots, a_n \subseteq b_{jn}$. A *sequence database* S is a set of tuples where a tuple $\langle sid, sequence \rangle$ contains a sequence and its ID. If a sequence a is a subsequence of f sequences in the tuples of S and $f > m$(*a given constant value*), we say that a is frequent in S and m is the *minimum support*.

Let α be a sequence $\langle a_1, a_2, \ldots, a_n \rangle$, β be a sequence $\langle b_1, b_2, \ldots, b_m \rangle$, where $m \leqslant n$. Sequence β is called a *prefix* of α if and only if (1) $b_i = a_i$ for $i \leqslant m - 1$; (2) $b_m \subseteq a_m$; and (3) all the items in $(a_m - b_m)$ are alphabetically sorted after those in b_m. Given sequences α and β such that β is a prefix of α, a subsequence α' of sequence α is called a *projection* of α w.r.t. prefix β if and only if (1)α' has prefix β and (2) there exists no proper super-sequence α' of α' such that α' is a subsequence of α and also has prefix β. Let $\alpha' = \langle e_1, e_2, \ldots, e_n \rangle$ be the projection of α w.r.t. prefix $\beta = \langle e_1, e_2, \ldots, e_{m-1}, e_{m'} \rangle$ $(m \leqslant n)$. Sequence $\gamma = \langle e_{m''}, e_{m+1}, \ldots, e_n \rangle$ is called the suffix of α w.r.t. prefix β, denoted as $\gamma = \alpha/\beta$, where $e_{m''} = e_m - e_{m'}$. A projected database Sp of a sequence database S w.r.t. prefix a is the set of tuples where every tuple $\langle sid, suffix \rangle$ includes its original sequence ID and the *suffix* of a sequence in S w.r.t prefix a. Note that each tuple's *suffix* is not empty, which means that a *suffix* of a sequence in S w.r.t. prefix a can not be put into Sp if it is empty. More details about the background knowledge of sequential pattern mining can be found in [11].

3.2. Theoretical analysis

In fact, both the average value constraints and the sum constraints with negative values can be expressed in a uniform way. To begin with, for a sum constraint with negative values, if an item's value is positive, we can say that it does positive contribution on the sum. Otherwise, we can say that it does negative contribution on the sum. In this way, we can get the sum of a sequence through calculating the contribution of its items by an appropriate method. More importantly, an average constraint is similar to a sum constraint with negative values from the following view: for an item whose value is greater than the constant value, we can regard that it does positive contribution to the required average value. For the purpose of designing a uniform strategy to deal with these different forms of tough constraints, we first give two definitions as follows.

Definition 1. Let s be a sequence and c be a constant value, and suppose that every item i of s is associated with a value v_i. The *positive contribution* of item i w.r.t. c is $\max(v_i - c, 0)$, the *negative contribution* of item i w.r.t. c is $\min(v_i - c, 0)$, and the *total contribution* of item i w.r.t. c is $v_i - c$.

Definition 2. Let s be a sequence and c be a constant value, and suppose that every item i of s is associated with a value v_i. The *positive contribution*, *negative contribution* and *total contribution* of sequence s w.r.t. c are, respectively, the sum of all its items' *positive contributions*, *negative contributions* and *total contributions* w.r.t. c.

For instance, let us consider the sequence database shown in Table 3. In the table each item is represented by its associated value. Let constant value c be 25. The negative contribution of Sequence 1 w.r.t. c is -30, while its positive contribution w.r.t. c is 25 and its total contribution is -5. For the sake of brevity, we denote positive contribution w.r.t. c for sequence s as $s.pc(c)$. Similarly negative contribution and total contribution w.r.t. c for sequence s are, respectively, denoted as $s.nc(c)$ and $s.tc(c)$.

Table 3
An example of sequence database

Sid	Sequence
1	⟨(50, 20)(20)(20, 10)⟩
2	⟨(20)(10, 50)⟩
3	⟨(30, 20)(30, 20)(30, 20, 10)(45)⟩
4	⟨(30) (20, 10)⟩

Theorem 1. *Given a sequence s and a constant value c, let s.sum be the sum of the values of all items in s, s.length be the length of s. We say that s satisfies the average value constraint s.sum/s.length* $\geqslant c$, *if and only if* $s.tc(c) \geqslant 0$.

Proof. Because sequence s satisfies the average value constraint $s.sum/s.length \geqslant c$, we have

$$s.sum \geqslant c \times s.length \Longleftrightarrow s.sum - c \times s.length \geqslant 0 \Longleftrightarrow \sum_{i=1}^{s.length} (\text{the } i\text{th item's value-}c) \geqslant 0 \Longleftrightarrow s.tc(c) \geqslant 0 \quad \square$$

Analogously, it is also true that s satisfies the average value constraint $(s.sum/s.length) \leqslant c$, if and only if $s.tc(c) \leqslant 0$. In the following all the conclusions are also true in the case that "$\geqslant$" (or "$\leqslant$") is substituted simultaneously by "$\leqslant$" (or "$\geqslant$"). For succinctness, we do not enumerate them here case by case.

Theorem 2. *Given a sequence s containing some items associated with negative values and a constant value c, let s.sum be the sum of the values of all items in s. We say that s satisfies the sum constraint with negative values* $s.sum \geqslant c$, *if and only if* $s.tc(0) \geqslant c$.

Theorem 2 is easy to prove because $s.sum$ equals to $s.tc(0)$. Therefore, for both the average value constraint and the sum constraint with negative values, we can use a uniform strategy to deal with them, i.e., for every desired sequential pattern p, its total contribution w.r.t c_1 is not smaller than c_2, i.e. $p.tc(c_1) \geqslant c_2$ where both c_1 and c_2 are the given constant values.

4. PTAC – a new algorithm for the tough aggregate constraints

In this section, we firstly give an overview of the framework of our proposed algorithm PTAC. Secondly, we discuss how to improve the pruning efficiency, and then introduce our new strategies for dealing with the tough aggregate constraints. At the end, some further optimization issues of these strategies are considered.

4.1. The framework

PTAC is a prefix-growth based algorithm for the tough aggregate constraints. Fig. 1 shows how PTAC works. After finding all frequent items, PTAC constructs projected databases w.r.t these frequent items as prefixes. For an α-projected database, PTAC finds all frequent and promising items and then partitions them into several subsets, which are at different promising levels. We say that some items in a sequence are *promising* only if they belong to some patterns that satisfy the constraint. For each subset, we use specific strategies to deal with them and extend truly promising items as new prefixes. Then a projected database is constructed for every new prefix and we recursively invoke PTAC.[2]

Fig. 2 presents the pseudo code of PTAC. The most important part of PTAC is pushing constraints into the process of finding promising and frequent items in the α-projected database. Another important part is how to divide and conquer all candidate items. In PTAC, we always try to extend the most promising items first, and then check other items with the important information obtained from the previous mining stage, therefore

[2] To improve the performance of the PTAC algorithm, the *pseudo database* technology [8] can be used in practice.

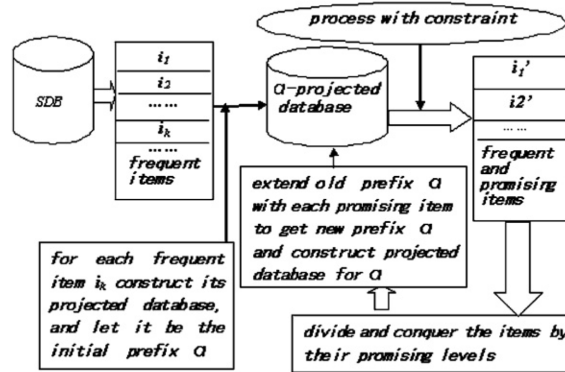

Fig. 1. The process of executing PTAC.

```
Input:    sequence database: SDB; minimum support: min_support; and the constraint C
Output:   all sequential patterns satisfying the constraint C.
Method: call PTAC(<>, SDB, G);
//G is a data structure to record the information obtained in the previous mining stage

procedure PTAC(a, S|a, G)
Var: Set, Set_1, Set_2, ......Set_k, item, a';
  1.    Set = GetAllValidFrequentItems(a, S|a)
  2.    partition Set into Set_1, Set_2, ......Set_k
  3.    for each Set_i
  4.        for each promising item in Set_i
  5.            a'= ExtendPrefix(a, item)
  6.            modify G if necessary
  7.            call PTAC (a', S|a', G)
  8.        end for
  9.    end for
 10. end procedure
```

Fig. 2. The pseudo Code of PTAC Framework.

avoiding constructing projected databases for those useless items. In the following section we give two specific strategies for dealing with these two issues.

4.2. Two new strategies

Generally speaking, in PrefixGrowth-based algorithms for sequential pattern mining, the cost of using tough aggregate constraints to prune invalid items comes from the following two aspects:

(1) checking all candidate sequences;
(2) constructing the projected database for each new pattern serving as a new prefix.

In order to reduce these costs, we propose two corresponding new strategies. For the ease of description of our strategies we first give some formal definitions related to the promising items.

Given an α-projected database, let x be an item belonging to suffix β, and the tough aggregate constraint $p.tc(c_1) \geqslant c_2$, where p is the desired pattern and $p.tc(c_1)$ stands for the total contribution w.r.t c_1. $Pre.x$ represents the sequence obtained by removing x and all the items behind x from sequence β. For example, for a sequence $\langle(20)(30, 10)\rangle$, $Pre.30$ is $\langle(20)\rangle$, and (β-$Pre.x$) represents the sequence constructed by removing $Pre.x$ from β.

Definition 3. If $[\alpha.\mathrm{tc}(c_1) + x.tc(c_1) + (\beta/x).pc(c_1)] \geqslant c_2$, x is called *promising*.

Under the condition that x is *promising*, we can certainly find those items behind x with which α can be used to extend them into a valid pattern.

Definition 4. If $[\alpha.tc(c_1) + (\beta - Pre.x).pc(c_1).] \geqslant c_2$, x is called *approximately promising*.

Under the condition that x is *approximately promising*, there must exist certain items behind x that can be used to construct a valid pattern with x and α, except that when $x.tc(c_1)$ is negative and $\alpha.tc(c_1)$ is negative. In the latter case $x.\ tc(c_1)$ has not been counted, so it is possible that x is not *promising*.

Definition 5. If $[\alpha.tc(c_1) + (\beta - Pre.x).nc(c_1)] \geqslant c_2$, x is called *extremely promising*.

Under the condition that x is *extremely promising*, all the items behind x can be used to construct a valid pattern with x and α.

For example, let us consider the (20)-projected database of Table 3. The projected database is illustrated in Table 4.

Notice that the underscore in the suffix $\langle(_10)\rangle$ means item 10 is in the same element as the prefix $\langle(20)\rangle$. Given a constraint $p.tc(15) \geqslant 0$ which means that $c_1 = 25$ and $c_2 = 0$, the first item (20) in Suffix 1 is *approximately promising*, since $[\langle 20\rangle.tc(15) + \langle(20)(20, 10)\rangle.pc(15)] = [5 + 10] = 15\rangle 0$. Meanwhile it is also *promising* and *extremely promising*, since $[\langle 20\ \rangle.tc(15) + (20).tc(15) + \langle(20,\ 10)\rangle.tc(15)] = 10 > 0$ and $[\langle 20\rangle.tc(15) + \langle(20)(20, 10)\rangle.nc(15)] = 0$. Similarly, the item (10) in Suffix 2 is *approximately promising* and *promising* but not *extremely promising*.

Let us start from analyzing how PrefixGrowth [12] decides whether an item is promising with tough aggregate constraints. Given a tough aggregate constraint $p.tc(c_1) \geqslant c_2$, this constraint is an average value constraint when c_2 is 0 according to the definition in Section 2.2, and it is a sum constraint with negative values when c_1 is 0. Moreover, let *s.valid_sum* be the sum of the values of all the valid items of a sequence s, and $s.n$ be the number of the occurrences of them in s. Here, we say that a sequence s is valid if it satisfies the given constraint, and an item x is valid if it satisfies the given constraint when we treat it as a sequence consisting of only one item. The decision of whether an item is promising with tough aggregate constraints in PrefixGrowth can be described with our proposed definitions as follows: with an average value constraint $p.tc(c_1) \geqslant 0$, for an invalid item x of s, if $(x.value + s.valid_sum)/(s.n + 1) < c_1$, x is considered as unpromising and should be removed; with a sum value constraint $p.tc(0) \geqslant c_2$, if $(x.value + s.valid_sum) < c_2$, for an invalid item x of s, x is also considered as unpromising and should be removed. This assertion is self-evident. In fact, we can design a uniform strategy to effectively decide whether an item should be removed on the basis of the concept of *item contribution* defined in Section 2.2.

Theorem 3. *Given a sequence s with a tough aggregate constraint* $p.tc(c_1) \geqslant c_2$, *an item x in s must be invalid and thus can be removed safely if* $(x.\ tc(c_1) + s.pc(c_1)) < c_2$.

Proof. (1) For the average value constraints, c_2 is 0. According to [12], we have $(x.value + s.valid_sum)/(s.n + 1) < c_1$. Thus $(x.value + s.valid_sum) < c_1 \times (s.n + 1)$. It follows that $(x.value + s.valid_sum - c_1 \times (s.n + 1)) < 0$, i.e., $((x.value - c_1) + (s.valid_sum - c_1 \times (s.n))) < 0$. So we have $(x.tc(c_1) + s.pc(c_1)) < 0$.

(2) For the sum constraints with negative values, c_1 is 0. Since $(x.value + s.valid_sum) < c_2$ thus we also have $(x.tc(0) + s.pc(0)) < c_2$. □

Table 4
(20)-projected database based on the sequence database in Table 3

Sid	Suffix
1	$\langle(20)(20, 10)\rangle$
2	$\langle(10, 50)\rangle$
3	$\langle(30, 20)(30, 20, 10)(45)\rangle$
4	$\langle(_10)\rangle$

Theorem 4. *For a suffix β in theα-projected database with a tough aggregate constraint $p.tc(c_1) \geqslant c_2$, an invalid item x in β can be removed safely if $(\alpha.tc(c_1) + x.tc(c_1) + (\beta/x).pc(c_1)) < c_2$.*

Proof. (1) For the average value constraints, $c_2 = 0$. Since $(x.value + (\alpha\beta).valid_sum)/(\alpha\beta.n + 1) < c_1$, thus $(\alpha.valid_sum + x.value + \beta.valid_sum) < c_1(\alpha.n + 1 + \beta.n)$. Due to that $\alpha.valid_sum \leqslant \alpha.sum$ and $\alpha.n \leqslant \alpha.length$, we have

$$
\begin{aligned}
&(\alpha.sum + x.value + \beta.valid_sum) < c_1(\alpha.length + 1 + \beta.n)\\
\Longleftrightarrow\ &[\alpha.sum + x.value + \beta.valid_sum - c_1(\alpha.length + 1 + \beta.n)] < 0\\
\Longleftrightarrow\ &[(\alpha.sum - c_1 \times \alpha.length) + (x.value - c_1) + (\beta.valid_sum - c_1 \times \beta.n)] < 0\\
\Longleftrightarrow\ &(\alpha.tc(c_1) + x.tc(c_1) + \beta.pc(c_1)) < 0\\
\Longleftrightarrow\ &(\alpha.tc(c_1) + x.tc(c_1) + (\beta/x).pc(c_1)) < 0
\end{aligned}
$$

(2) For the sum constraints with negative values, $c_1 = 0$. Since $(x.value + (\alpha\beta).valid_sum) < c_2$, thus $(\alpha.valid_sum + x.value + \beta.valid_sum) < c_2$. So $(\alpha.tc(0) + x.tc(0) + \beta.pc(0)) < c_2$. Due to that $\beta.pc(0) \leqslant (\beta/x).pc(0)$, we have $(\alpha.tc(0) + x.tc(0) + (\beta/x).pc(0)) < c_2$. □

With these two theorems we can assert that an item can be safely removed if it is *unpromising*. Furthermore, both the average value constraints and the sum constraints with negative values can be processed in a uniform way.

4.2.1. Pruning candidate sequences

When tackling the problem of constraint-based sequential pattern mining, we should prune the unpromising candidate sequences as early as possible so that the cost of checking all candidate sequences can be reduced as much as possible. Here, "candidate sequences" refer to those potentially frequent patterns obtained by extending an old pattern with frequent items in PatternGrowth-based algorithms. For example, in Table 3, we find that item (20), viewed as an old pattern, is frequent. Considering its (20)-projected database illustrated in Table 4, we find that both items (20) and (10) are frequent, so both (20) (10) and (20) (20) are candidate sequences. For a PatternGrowth-based algorithm, if unpromising items are pruned when counting all items' supports in the α-projected database, we can assure that all the items used to extend α are both frequent and promising. Therefore we can avoid generating and checking those unpromising sequences. However, it is not efficient to check all items one by one. In fact, for many items in a projected database, it is possible for us to decide, on the basis of the features of some other items and the prefix, whether they can be removed or not without checking. Therefore in order to prune those unpromising items effectively, we adopt *divide-and-conquer* to deal with these two situations separately, depending on whether α is valid or not.

For the α-projected database, our strategy is as follows: if α is invalid, then for every suffix β, we check all its items from the first to the last. While an item x of suffix β is approximately promising, x is counted if it is also at the level of promising; otherwise item x and all the items behind it are omitted. On the other hand, if α is valid, then for every suffix β, its all valid items are counted and its invalid items are checked from the first to the last: when an invalid item x is extremely promising, we count it in along with all the items behind it; otherwise we count it in only if it is promising. In this way we avoid the checking of the valid items in the case that the prefix is valid.

For example, considering the (20)-projected database illustrated in Table 4 with the constraint $p.tc(25) \geqslant 0$, we know that the prefix (20) is invalid and its total contribution is -5. For $\langle(20)(20, 10)\rangle$, we see that the first item (20) is not approximately promising because the sum of the total contribution of prefix (20) and the positive contribution of $\langle(20)\ (20, 10)\rangle$ equals to -5. So it is unnecessary for us to check all the remaining items, i.e., (20) and (10) in (20, 10) of Sequence 1. Table 5 shows in italics those items that are avoided from checking. If the constraint is $p.tc(15) \geqslant 0$, the total contribution of the prefix is 5. For $\langle(20)\ (20, 10)\rangle$, we can see that the first (20) is extremely promising because the sum of the negative contribution of $\langle(20)(20, 10)\rangle$ and the total contribution of the prefix is 0. As a result, we can count all the remaining items without checking.

Table 5
Items (in italics) avoided from checking for the given constraint $p.tc(25) \geqslant 0$

Sid	Suffix
1	⟨(20) (*20*, *10*)⟩
2	⟨(10, *50*)⟩
3	⟨(30, 20) (*30*, *20*, *10*)(*45*)⟩
4	⟨(_10)⟩

4.2.2. Pruning new patterns before constructing projected databases

Though all frequent items that we get are promising, the new patterns obtained by extending an old pattern (i.e. prefix) with them may be unpromising. For example, consider again the sequence database shown in Table 3 and its (20)-projected database illustrated in Table 4. Given an average constraint $p.avg \geqslant 25$, from the (20)-projected database we can find that (10) is a frequent and promising item and, plausibly, we should generate a new pattern ⟨(20) (10)⟩ as a new prefix and construct a projected database for the new prefix. However, we find that the new pattern can not be the prefix of any sequential pattern, so the construction of its projected database and the pattern mining from the projected database is absolutely a waste. To solve this problem on the basis of PrefixGrowth, we first describe and prove a new theorem below.

Theorem 5. *Given a tough aggregate constraint and* α*-projected database, for any item x that violates the constraint, if* $\{\alpha x\}$ *violates the constraint and there is a valid sequential pattern* $\{\alpha x\beta\}$*, then there must be a valid sequential pattern* $\{\alpha\beta'\}$ *so that the first item of* β' *is a valid item and* $\beta' \subseteq \beta$.

Proof. Suppose that β is denoted as $\langle I_1, I_2, I_3, \ldots, I_n\rangle$, where $I_i (i = 1, 2, \ldots, n)$ is an item. Let $\bigcup_{i=1}^{k}\langle I_i\rangle = \langle I_1, I_2, \ldots, I_k\rangle$. It is apparent that for a valid sequence s, if an invalid item I is removed from s, then the result sequence $(s - \langle I\rangle)$ is still valid. This leads us to the following observation:

- There must exist at least one valid item in β: Assume otherwise (i.e. there does not exist a valid item I_j in β), then $(\langle\alpha x\beta\rangle - \bigcup_{i=1}^{n}\langle I_i\rangle)$ is valid, which means that $\langle\alpha x\rangle$ is valid. This obviously contradicts with the assumption.

So without losing generality, let I_k be the first valid item in β. From the above observation we know that $(\langle\alpha x\beta\rangle - \bigcup_{i=1}^{k-1}\langle I_i\rangle - \langle x\rangle)$ is valid. Let $\beta\prime = \beta - \bigcup_{i=1}^{k-1}\langle I_i\rangle$, we can conclude that Theorem 5 is true. □

Based on Theorem 5, we present here a strategy for pruning new patterns with tough aggregate constraints as follows. In an α-projected database, we find all frequent and promising items and categorize them into three classes: (1) the set of the valid items (denoted as I_1); (2) the set of the items that are invalid but can construct a valid sequential pattern with α (denoted as I_2); and (3) the set of the items that are invalid and can not construct a valid sequential pattern with α (denoted as I_3). Then our strategy works as follows: firstly, we extend α with items in I_1 and construct projected databases for new patterns. When mining these projected databases, if we find a frequent and valid pattern $\langle\alpha\beta\rangle$ while the first item of β is valid, we check whether there exists certain item x_i that belongs to I_3 and can be added to $\langle\alpha\beta\rangle$ to make the result pattern $\langle\alpha x_i\beta\rangle$ valid. If so, all such items will be *marked*. After finding all sequential patterns $\langle\alpha\beta\rangle$ in these projected databases, α will be extended with the items belonging to I_2 to obtain new patterns as new prefixes, and then for each new prefix its corresponding projected database is constructed. Now, in these newly constructed projected databases, we directly mine sequential patterns without going through the kind of checking as we do for the items in I_1. Finally, we extend α with the marked items belonging to I_3, thus we can avoid constructing unpromising patterns and corresponding (useless) projected databases.

PrefixGrowth [12] adopts a similar strategy. But it simply categorizes the candidate items into two classes: the valid items and invalid items. As a result, if we find a frequent and valid pattern $\langle\alpha\beta\rangle$ and the first item of β is valid, we have to check all the invalid items including those that are apparently promising. Even worse, when adopting this kind of strategy, we need a data structure to record the information of marked items for every recursive level. Furthermore, PrefixGrowth requires recording at every recursive level the information of the items that need not to be checked, which incurs a lot of space and also the performance is lowered.

Let us consider the sequence database shown in Table 3 again and try to mine all the desired sequential patterns by PTAC with both strategies. Assume that the adopted constraint is $p.tc(25) \geqslant 0$ and the min_support is 2.

Step 1: Scan the database once and we find (10), (30) as the frequent and promising items with the first strategy. Then (30) and (10) are categorized into I_1 and I_3, respectively.

Step 2: According to the second strategy, we firstly process (30)-projected database as shown in Table 6. With the first strategy, we find the only frequent and promising item (20). With the second strategy, we try to check whether $\langle(30)(10)(20)\rangle$ is valid. Obviously it is not, so (10) for the original sequence database in Table 3 is not marked.

Step 3: According to the second strategy, (20) is classified into I_2 and no items can be classified into I_1 neither I_3. Then $\langle(30)(20)\rangle$-projected database is constructed as shown in Table 7. In this database, no item is both frequent and promising.

Step 4: Now we go back to step 1. Since (10) in the original database is not marked, its projected database is not constructed. The mining process terminates, and the final result is $\langle(30)(20)\rangle$.

Fig. 3 gives the pseudo code of GetAllValidFrequentItems(a, $S|a$) which is the most important function of PTAC. Based on that, Fig. 4 illustrates the pseudo code of PTAC which adopts all foregoing strategies. Here

Table 6
(30)-projected database based on the sequence database of Table 3

Sid	Suffix
3	⟨(_20)(30, 20)(30, 20, 10)(45)⟩
4	⟨(20, 10)⟩

Table 7
⟨(30)(20)⟩-projected database based on the sequence database of Table 3

Sid	Suffix
3	⟨(30, 20, 10)(45)⟩
4	⟨(_10)⟩

```
Input: prefix a; S|a, the projected database of a
Output: promising and frequent items in the a-projected database
procedure GetAllValidFrequentItems(a, S|a)
var: item, seq;
1.   if (a is invalid) then
2.     for each seq in S|a
3.       for each item in seq
4.       if (item is approximately promising) then
5.           if (item is promising) then
6.               call Count(item)
7.           else break
8.         endfor
9.       endfor
10.  else for each seq in S|a
11.        for each item in seq
12.       if (item is valid ) then call Count(item)
13.       else if (item is extremely promising) then
14.             call Count(items behind item);
15.             break
16.       else if (item is promising) then
17.           call Count(item)
18.       end for
19.    endfor
```

Fig. 3. The pseudo code of GetAllValidFrequentItems.

```
Input: sequence database: SDB; minimum support:
       min_support; and the aggregate constraint Ca
Output: all sequential patterns that satisfy the constraint Ca
Method: call PTAC (<>, SDB, G);
procedure PTAC (a, S|a, G)
// a is prefix; S|a is the projected database of a
var: item, Set, ValidItems, IPItems, ICItems, a'
1. Set = GetAllValidFrequentItems(a, S|a)
2. Divide Set into ValidItems, IPItems, ICItems
 3. Add ICItems into    G;
4. for each item in ValidItems
5.            a' = ExtendPrefix(a, item);
6.            if (a'   satisfies Ca) then
7.                 {   call WriteIntoPatternTbl (a');
8.                     call Modify(a', G); }
9.            call PTAC (a', S|a', G)
10. endfor
11. for each item in IPItems
12.           a' = ExtendPrefix(a, item)
13.    if (a'   satisfies Ca) then
14.           call WriteIntoPatternTbl(a');
15.         call PTAC (a', S|a', G)
16. endfor
17. for each item in G and ICItems)
18.           if ( item is marked ) then
19.                { a' = ExtendPrefix(a, item);
20.                    if (a'   satisfies Ca) then
21.                        call WriteIntoPatternTbl(a');
22.                    call PTAC(a', S|a', G)}
23    endfor
24.   Remove ICItems from G
25.   end procedure
```

Fig. 4. The pseudo code of PTAC for dealing with the tough aggregate constraint.

we briefly explain some newly introduced variables and functions: G is a data structure for recording which invalid and unpromising items have been marked, and it is empty at the beginning. IPItems denotes all *Invalid* but *Promising* items and, correspondingly, ICItems denotes all the items which are *Invalid* and need to be *Checked*. The function Modify(newPrefix, GlobalICitemInfo) finds and marks those IC items that can be used to construct valid sequential patterns with newPrefix in GlobalICitemInfo.

4.3. Room for further optimization

In our above-presented strategy for pruning candidate sequences, for every item x to be checked in sequence s, it is needed to compute the positive contribution or negative contribution of the suffix of s with respect to (w.r.t.) x, so as to decide whether it is necessary to count x or not. If these contribution values are computed and recorded in advance, they can be directly used when checking items and thus the efficiency of the algorithm can be improved. For this purpose, two data structures called *GPCset* (*Global Positive Contribution*) and *GNCset* (*Global Negative Contribution*) can be introduced. *GPCset* is a 1-dimension vector. Assume that s_i is the ith sequence of the original sequence database, x_j is its jth item and MaxLength is the max length of all sequences, $GPCset[(i-1)*MaxLength+j-1]$ records the positive contribution of the suffix of s_i w.r.t. x_{j-1}. Similarly *GNCset* is also a 1-dimension vector and its element $GNCset[i][j]$ stores the negative contribution of the suffix of s_i w.r.t. x_{j-1}. Every element of *GPCset* and *GNCset* is computed in the step of preprocessing the sequence database. *GPCset* and *GNCset* can be repeatedly used even when the min_support has been changed. Certainly, if the constant value c of the aggregate constraint changes to c', *GPCset* and *GNCset* for c' need to be computed again.

5. Experiment and analysis

5.1. Experimental datasets

To evaluate the performance of the (plain) PTAC, the optimized version of PTAC and PrefixGrowth [12], we have conducted extensive experiments both on two synthetic datasets and a real dataset. The formers are generated by the IBM sequence generator [14], including (1) the sequence dataset C5T4S2.0I1.25D50K containing 50,000 sequence records, whose average length is 5, with each element averagely having 4 items, and the size of the item set being 5000; (2) the long sequence dataset C10T10S2.0I1.25D10K containing 10,000 sequential records whose average length is 100, and the size of the item set is 5000. In addition, for scalability testing purpose, we also generate the datasets C5T4S2.0I1.25D5K, C5T4S2.0I1.25D10K, C5T4S2.0-I1.25D20K, C5T4S2.0I1.25D35K, which have the same settings as C5T4S2.0I1.25D50K except the number of sequences. The sizes of these four datasets are 5000, 10,000, 20,000 and 35,000, respectively. The real dataset is 10,000 VIP customers' purchase records of a supermarket. Each sequence represents a customer's purchase record in the March of 2006, an element represents the customer's purchase record in a week and an item represents a kind of commodity with its price as value. In this dataset, each sequence has just four elements, and 1008 kinds of commodity have been recorded.

In order to impartially evaluate the performance of these algorithms, an identical bottom level implementation is used to represent sequences and the same preprocessing is performed. In these experiments, we adopt the same constraint that the total contribution w.r.t. c of every desired sequential pattern has to be no smaller than 0. Because this constraint is equivalent to an average value constraint and c is the minimum average value of a target sequential pattern, we give constant c a meaning name, i.e. *minimum average value* (*minAvg* for short). For the synthetic datasets, the value of an item is set to be

$$item.value = item.subscript \text{ Mod } 100 + 1.$$

From this we know that the average value of a pattern must be an integer in the range of [1, 100]. So in our experiments *minAvg* should be less than 100.

5.2. Experimental platform

All experiments were performed on a PC with Pentium IV 2.0 GHz CPU and 512M main memory running Microsoft Windows XP Professional Edition. PTAC, its optimization, and PrefixGrowth are all performed using Microsoft Visual C++ 6.0 and Microsoft SQL Server 2000.

5.3. Experiments on synthetic datasets

In this subsection, we report our experiments on the two synthetic data sets and analyze the results.

5.3.1. Comparing the running time and scalability

Fig. 5 illustrates the comparison of the efficiency of PTAC, its optimization (optimized version), and PrefixGrowth with a constant minimum support 0.2% and variable *minAvg* values. From the figure we can see clearly that both PTAC and its optimization outperforms PrefixGrowth when the minimum average value *minAvg* changes from a very small value 5 to a very large value 95, and their performance are almost the same when the minimum average value approaches to its maximum value 100. When the minimum average value is extremely big, the number of sequential patterns that satisfy the constraint is very small. Therefore most of the candidate sequences and items can be pruned easily. Under this condition, PrefixGrowth can also mine the desired sequential patterns effectively.

As mentioned above, the performance comparison is meaningless when the minimum average value is too large. So we perform the experiments to evaluate the performance of PTAC and its optimization with small or medium minimum average values *minAvg*. Fig. 6 and 7 present the performance of PTAC, its optimization and PrefixGrowth on the same dataset with the constant minimum average value set to be 5 and 50, respectively, and the *min_support* varies form 0.2% to 1%. From these two figures we can see that both PTAC and its

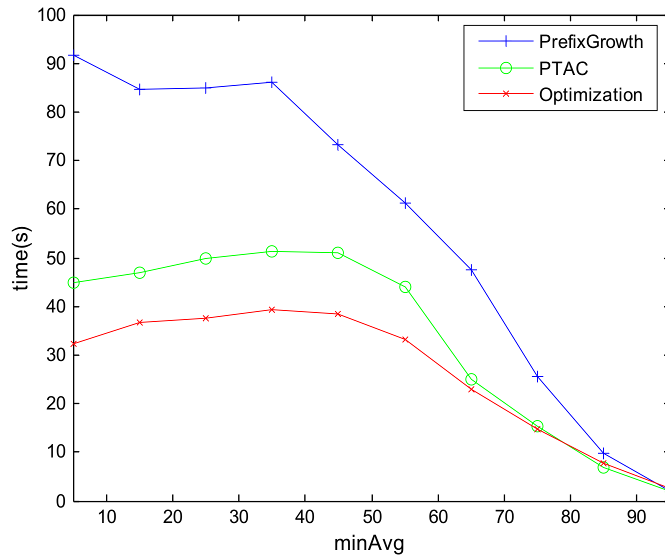

Fig. 5. The performance of PTAC, its optimization and PrefixGrowth on dataset C5T4S2.0I1.25D50K with the minimum support 0.2%.

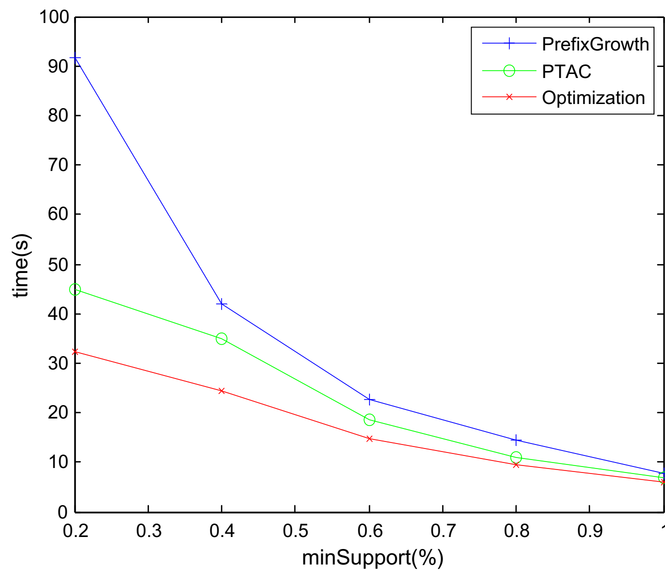

Fig. 6. The performance of PTAC, its optimization and PrefixGrowth on dataset C5T4S2.0I1.25D50K with the minimum average value 5.

optimization perform better. Moreover, compared with that of PrefixGrowth, the smaller the minimum average value is, the better the performance of PTAC and its optimization is.

Fig. 8 shows the performance of PTAC, its optimization and PrefixGrowth with the minimum support fixed to be 1% and variable *minAvg* values on long sequence dataset C10T10S2.0I1.25D10K. The result shows that PTAC and its optimization still largely outperform PrefixGrowth when tackling long sequences.

In the experiment on C10T10S2.0I1.25D10K, we also mainly care about the performance of PTAC and its optimization with small or medium minimum average values *minAvg*. Fig. 9 presents the performance of PTAC, its optimization and PrefixGrowth on C10T10S2.0I1.25D10K. The constant *minAvg* is set to be 5

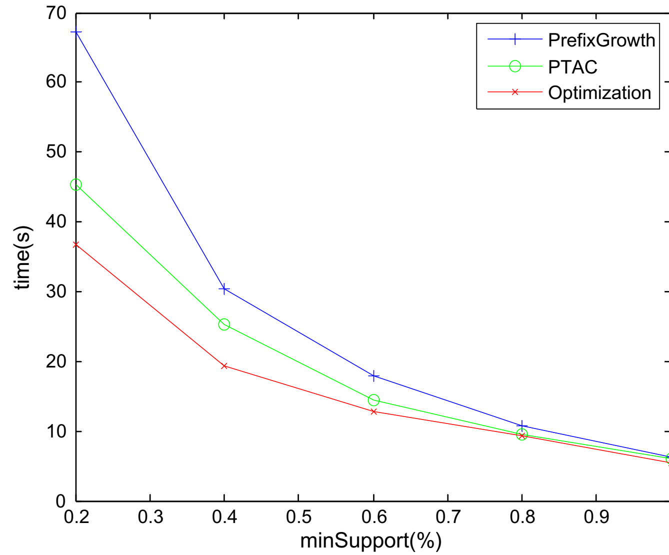

Fig. 7. The performance of PTAC, its optimization and PrefixGrowth on dataset C5T4S2.0I1.25D50K with the minimum average value 50.

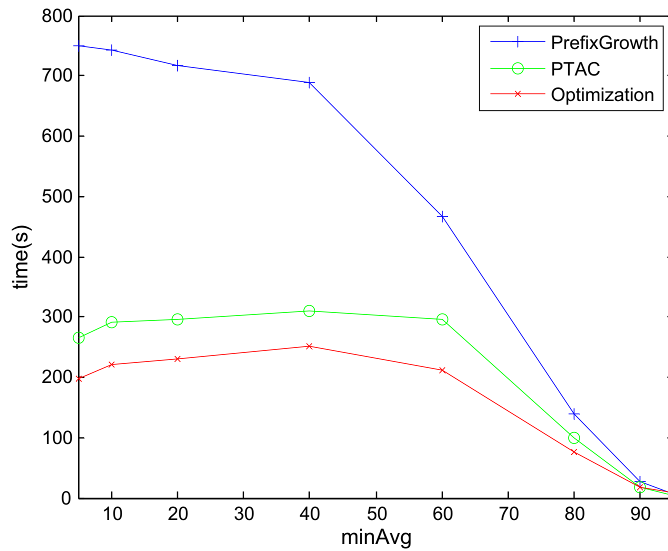

Fig. 8. The performance of PTAC, its optimization and PrefixGrowth on dataset C10T10S2.0I1.25D10K with the minimum support 1%.

and the *min_support* varies from 1% to 5%. From this figure we can see that both PTAC and its optimization perform better. Moreover, compared with that of PrefixGrowth, the smaller the minimum average value is, the better the performance of PTAC and its optimization is. Limited by the space, we omit the experimental results when the constant minimum average values are medium. In fact, under the same condition, it is similar to the result obtained on C5T4S2.0I1.25D50K.

To evaluate the scalability of PTAC, its optimization and PrefixGrowth, we perform experiments on the datasets C5T4S2.0I1.25D5K, C5T4S2.0I1.25D10K, C5T4S2.0I1.25D20K, C5T4S2.0I1.25D35K and C5T4S2.0I1.25D50K, respectively. The *min_support* is set to be 0.2% and the *minAvg* is set to be 5. Fig. 10 shows the

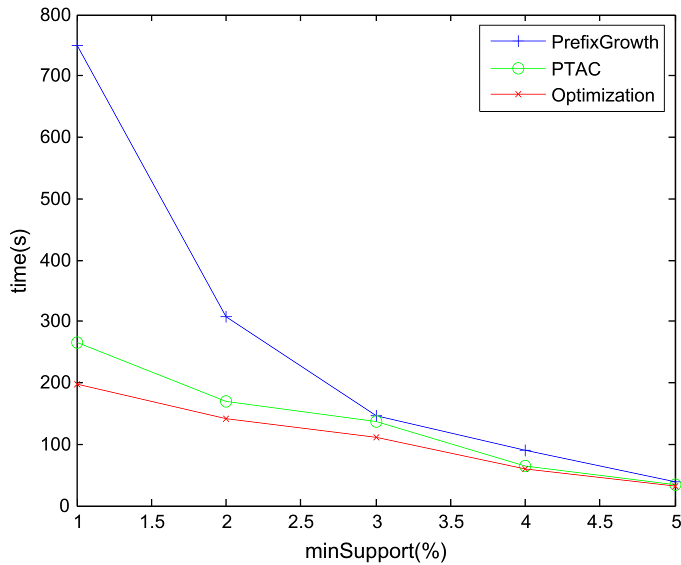

Fig. 9. The performance of PTAC, its optimization and PrefixGrowth on dataset C10T10S2.0I1.25D10K with the minimum average value 5.

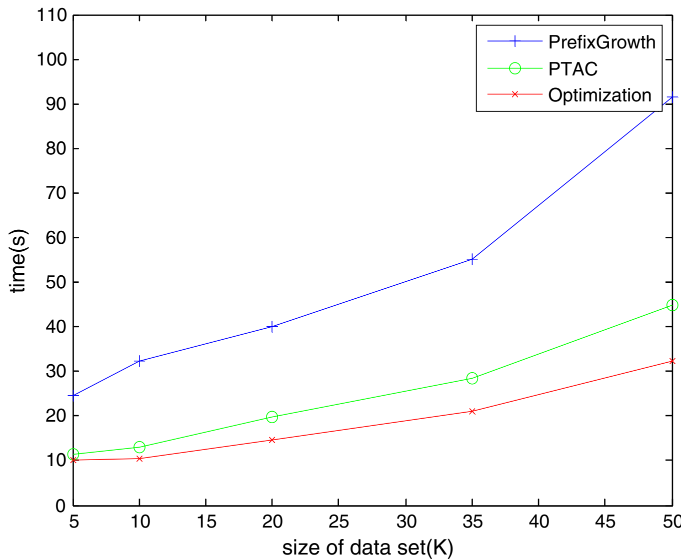

Fig. 10. The scalability of PTAC, its optimization and PrefixGrowth on dataset C5T4S2.0I1.25.

performance of PTAC, its optimization and PrefixGrowth when the size of dataset increases form 5K to 50K. From Fig. 10 we can see that even though the running time of all three algorithms increases almost linearly, the time cost of PrefixGrowth is always larger and increases faster than those of the others. It means that the larger the sequence database size is, the better PTAC and its optimization perform, which implies that our methods have better scalability than that of PrefixGrowth. Moreover, the time cost of PTAC increases a little faster than that of its optimization and becomes higher than the latter when the dataset size reaches to a certain degree, which means that the optimization of PTAC is more effective in dealing with a larger sized dataset.

5.3.2. Comparing the effectiveness of pruning strategies and the cost of space

The number of checked items reduced is an important factor used to evaluate the effectiveness of a pruning strategy. A well-designed pruning strategy should avoid checking useless items as many as possible. To evaluate the pruning effectiveness of PTAC and PrefixGrowth (because the optimized PTAC adopts the same strategies as the plain PTAC, its pruning effectiveness is the same as the plain PTAC), we perform experiments on the datasets C5T4S2.0I1.25D5K, C5T4S2.0I1.25D10K, C5T4S2.0I1.25D20K, C5T4S2.0I1.25D35K and C5T4S2.0I1.25D50K, respectively. Fig. 11 shows, with the constant *min_support* 0.2%, *minAvg* 5 and the variable number of sequences, the number of the checked items when pruning candidate sequences under PTAC and PrefixGrowth. The figure illustrates that the number of the checked items under both PTAC and PrefixGrowth increase linearly with the increase of the sequence database size. However, compared with that under PrefixGrowth, the number of the checked items under PTAC increases much more slowly. It is due to that our strategies avoid lots of unnecessary checking for the item promisingness, which confirms that our strategies of pruning candidate sequences are more effective than the counterparts in PrefixGrowth.

When mining the projected databases whose prefixes are constructed with those items with the highest promising level, both PrefixGrowth and PTAC have an associated data structure to store some promisingness information for not yet mined invalid items. In PTAC, the associated data structure is *GlobalICitemInfo*, and we call the total space of this data structure occupied in all iterative processes as *total associated space*. For instance, the algorithm iterates five times and the spaces of the associated data structure occupied in these iterative steps are, respectively, 4KB, 3KB, 2KB, 1KB, 0.8KB. Thus the total associated space is 10.8KB. Because we adopt different strategies to partition candidate items under their promising level, for each iterative process, the size of this associated data structure is smaller than that under PrefixGrowth. Therefore, the total associated space under PTAC should also be smaller than that under PrefixGrowth in the same environment. In fact, our experiment validates our assumption, as Fig. 12 shows. Experiments are performed on the datasets C5T4S2.0I1.25D5K, C5T4S2.0I1.25D10K, C5T4S2.0I1.25D20K, C5T4S2.0I1.25D35K and C5T4S2.0I1.25D50K, respectively. Overall, the total associated spaces for both two algorithms, under the same environment as that in Fig. 11 (with the constant minimum support 0.2%, the constant minimum average value 5 and variable number of sequences), increase slowly and sometimes decrease for the noise reason. But it is obvious that the associated space under PTAC is always much smaller. Meanwhile, the total associated space for the optimized PTAC is the same as that for PTAC, due to the reason that they adopt the same strategies.

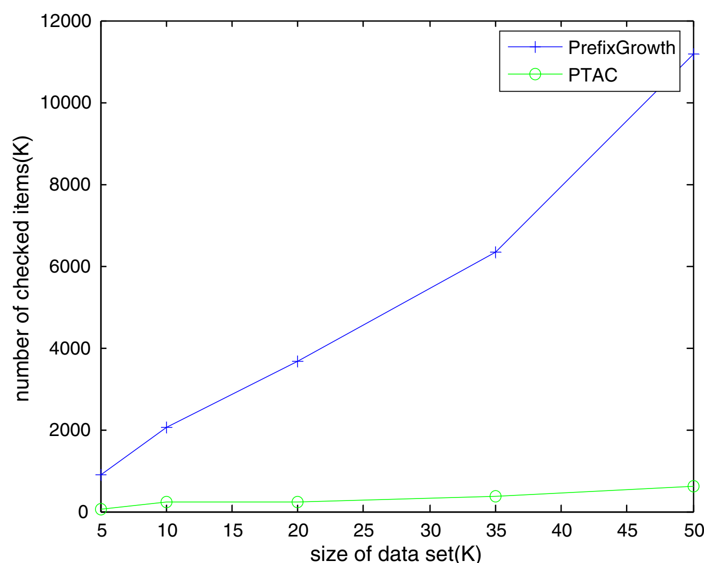

Fig. 11. The number of the checked items under PTAC and PrefixGrowth on dataset C5T4S2.0I1.25.

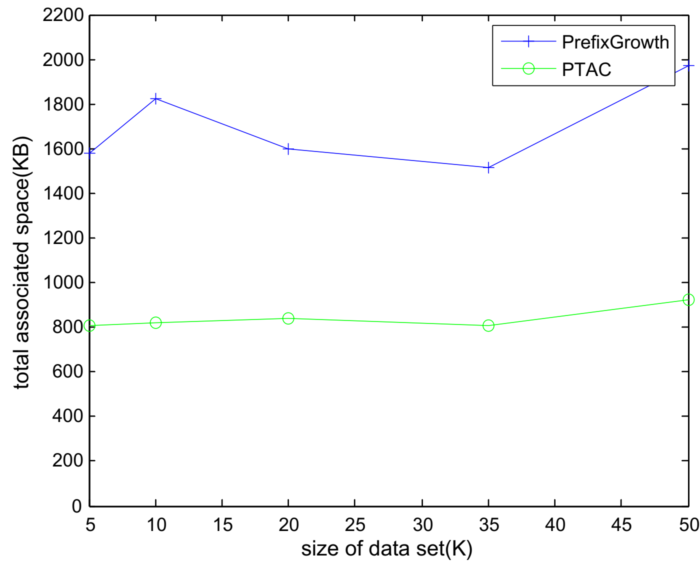

Fig. 12. The total associated space under PTAC and PrefixGrowth on dataset C5T4S2.0I1.2.

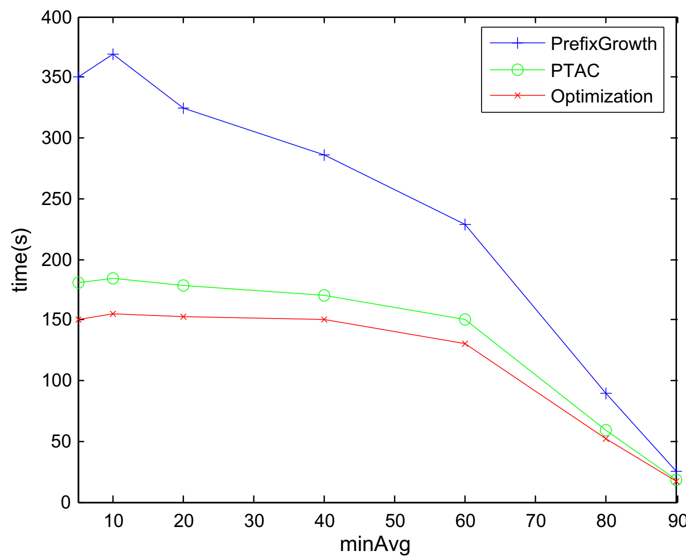

Fig. 13. The performance of PTAC and PrefixGrowth on the real dataset with the minimum support 1%.

5.4. Experiments on the real dataset

In this subsection, we report our experiments on the real dataset mentioned previously. Fig. 13 and 14 show the efficiency comparison of PTAC, its optimization and PrefixGrowth with variable *minAvg* on the real dataset. In the experiment illustrated in Fig. 13, the minimum support is fixed to be 1%, and in the experiment illustrated in Fig. 14, the minimum support is fixed to be 2%. The results obtained on these two experiments show that PTAC and its optimization perform better than PrefixGrowth on this dataset.

In the experiment on the real data, we also mainly care about the performance of PTAC with a small or medium minimum average value *minAvg*. Fig. 15 and 16 present the performance of PTAC, its

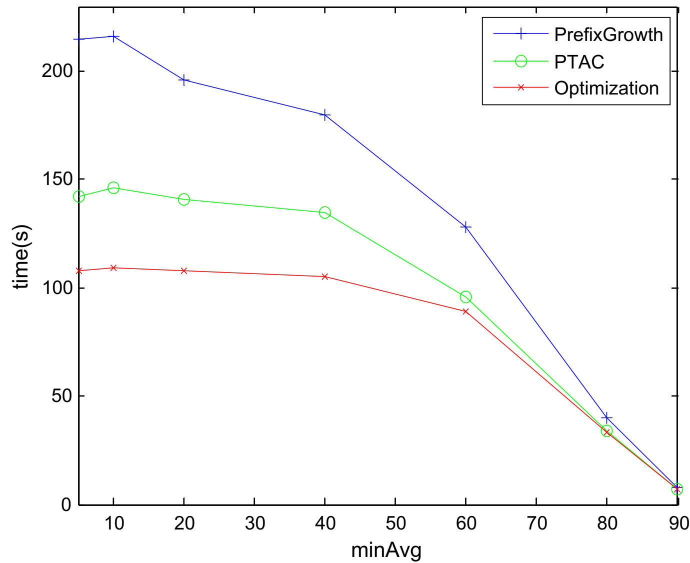

Fig. 14. The performance of PTAC and PrefixGrowth on the real dataset with the minimum support 2%.

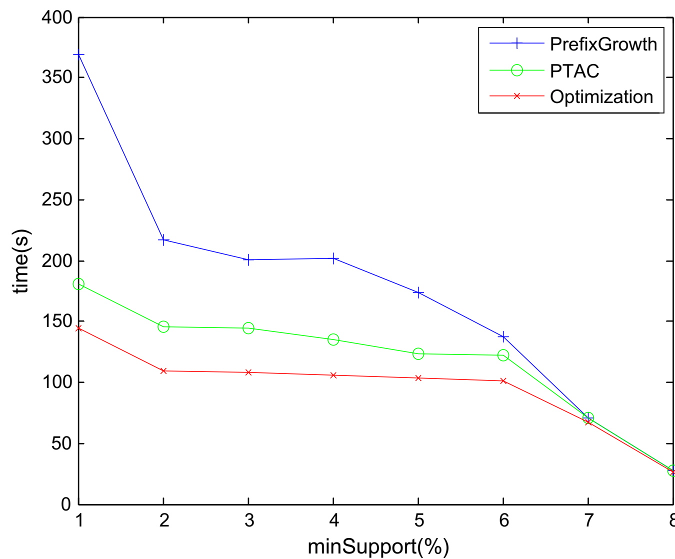

Fig. 15. The performance of PTAC and PrefixGrowth on the real dataset with the minimum average value 10.

optimization and PrefixGrowth with variable *min_support* on the real dataset. In the experiment illustrated in Fig. 15, *minAvg* is set to be 10 and *min_support* varies from 1% to 8%. In the experiment illustrated in Fig. 16, *minAvg* is set to be 35 and *min_support* varies from 1% to 6%. In this experiment, the performance of PTAC is too close to its optimization that their result curves are hardly separated. So in Fig. 16 only the performances of PTAC and PrefixGrowth are showed without that of PTAC's optimization. From these figures we can see that PTAC and its optimization perform better again. Moreover, compared with Prefix-Growth, the smaller the minimum average value *minAvg* is, the better the performances of PTAC and its optimization are.

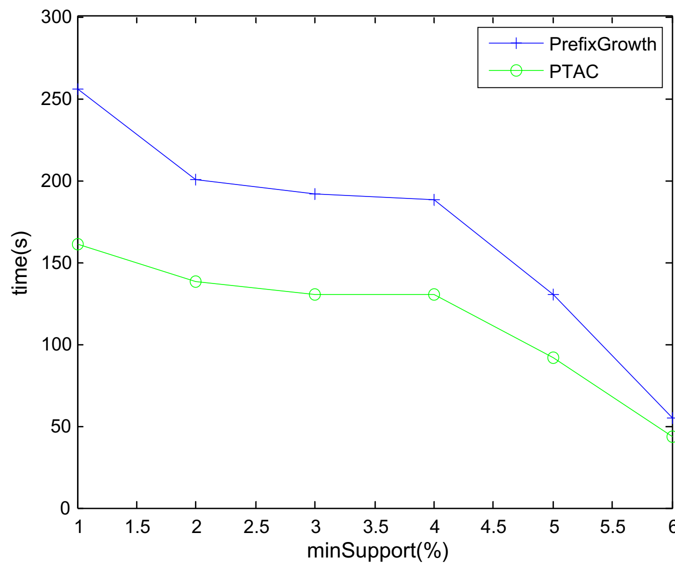

Fig. 16. The performance of PTAC and PrefixGrowth on the real dataset with the minimum average value 35.

6. Conclusion

In this paper, we have presented efficient strategies to deal with two typical kinds of tough aggregate constraints in a uniform way. Based on the notion of sequence contribution, we have theoretically demonstrated that the two typical kinds of tough aggregate constraints can be converted to a uniform form, and can thus be processed by the same strategies. We have come up with a new tough aggregate constraint-based sequential pattern mining algorithm called PTAC, in which two effective strategies for improving the constraint-related processing efficiency are presented. Finally, we conducted experiments to evaluate the performance of PTAC and its optimization on the datasets generated by the IBM sequence generator as well as a real dataset. It is shown that PTAC and its optimization outperform PrefixGrowth based on the experiments.

In PTAC, items are classified into three promising levels, i.e. *approximately promising*, *promising*, and *extremely promising*, and corresponding judging rules are designed to reduce the number of the checked items and the searching space. Aiming at the applications in which a quick overview of approximate results is required without considering the completeness of the results, we plan to move on to mining sequential patterns with aggregate constraints approximately by exploiting further the classification of promising levels. In particular, the approximate results may be discovered more quickly, even though the precision of the results may decrease a little bit. It is an interesting problem to strive a good balance between the precision and efficiency, which will mainly depend on appropriate strategies to speed up the mining process with an acceptable precision.

Acknowledgements

This work was supported by Natural Science Foundation of China (No. 60573077), Program for New Century Excellent Talents in University (No. NCET-05-0549), City University of Hong Kong under strategic research Grants (Nos. 7001956 and 7001997).

References

[1] Francesco Bonchi, Fosca Giannotti, Allessio Mazzanti, Dino Pedreschi, ExAnte: Anticipated data reduction in constrained patterns mining, in: *PKDD*, Cavtat-Dubrovnik, Croatia, 2003, pp. 59–70.

[2] E.H. Chen, T.S. Li, Phillip C-y SHEU, A general framework for monotony and tough constraint based sequential pattern mining, in: DaWak05, Copenhagen, Denmark, 2005, pp. 458–467..

[3] Ding-Ying Chiu, Yi-Hung Wu, Arbee L.P. Chen, An efficient algorithm for mining frequent sequences by a new strategy without support counting, in: ICDE'04, Boston, USA, 2004, pp. 375–386.
[4] Joong Hyuk Chang, Won Suk Lee, Efficient mining method for retrieving sequential patterns over online data streams, Journal of Information Science 31 (2005) 420–432.
[5] Xiaoyu Chang, Chunguang Zhou, Zhe Wang, Ping Hu, A novel method for mining sequential patterns in datasets, in: ISDA'06, Jinan, China, 2006, pp. 611–615.
[6] Minos Garofalakis, Rajeev Rastogi, Kyuseok Shim, Mining sequential patterns with regular expression constraints, TKDE 14 (2002) 530–552.
[7] Chin-Chuan Ho, Hua-Fu Li, Fang-Fei Kuo, Suh-YinLee, Incremental mining of sequential patterns over a stream sliding window, data mining workshops, in: ICDM Workshops, Hong Kong, China, 2006, pp. 677–681.
[8] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, M.-C. Hsu, Freespan: Frequent pattern-projected sequential pattern mining, in: Knowledge Discovery and Data Mining (KDD'00), Boston, USA, 2000 pp. 355–359.
[9] Congnan Luo, Soon M. Chung, A scalable algorithm for mining maximal frequent sequences using sampling, in: TCTAI(04), Boca Raton, USA, 2004, pp. 156–165.
[10] Salvatore Orlando, Raffaele Perego, Claudio Silvestri, A new algorithm for gap constrained sequence mining, in: ACM Symposium on Applied Computing, Lisbon, Portugal, 2004, pp. 540–547.
[11] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth, in: ICDE, Heidelberg, Germany, 2001, pp. 215–226.
[12] Jian Pei, Jiawei Han, Wei Wang, Mining sequential patterns with constraints in large databases, in: CIKM, McLean, USA, 2002, pp. 18–25.
[13] Jia-Dong Ren, Jun-Sheng Zong, A fast interactive sequential pattern mining algorithm based on memory indexing, in: Machine Learning and Cybernetics, Guangzhou, China, 2006, pp. 1082–1087.
[14] R. Srikant, R. Agrawal, Mining sequential patterns, in: ICDE, Taipei, Taiwan, 1995, pp. 3–14.
[15] R. Srikant, R. Agrawal, Mining sequential patterns: Generalizations and performance improvements, in: EDBT, Avigon, France, 1996, pp. 3–17.
[16] J.T.-L. Wang, G.-W. Chirn, T.G. Marr, B. Shapiro, D. Shasha, K. Zhang, Combinatorial pattern discovery for scientific data: some preliminary results, in: SIGMOD, Minneapolis, USA, 1994, pp. 115–125.
[17] Yun Xiong, Yang-yong Zhu, A multi-supports-based sequential pattern mining algorithm, in: CIT05, New York, USA, 2005, pp. 170–174.
[18] Mohammed J. Zaki, Sequence mining in categorical domains: incorporating constraints, in: CIKM, Washington, DC, USA, 2000, pp. 422–429.
[19] Mohammed J. Zaki, SPADE: an efficient algorithm for mining frequent sequences, Machine Learning 1 (2000) 1–31.