Dynamic FP-Tree Based Mining of Frequent Patterns Satisfying Succinct Constraints

Carson Kai-Sang Leung

The University of Manitoba
Winnipeg, MB, Canada
kleung@cs.umanitoba.ca

Abstract. Since its introduction, frequent-pattern mining has been generalized to many forms, which include *constrained data mining*. The use of *constraints* permits user focus and guidance, enables user exploration and control, and leads to effective pruning of the search space and efficient mining of frequent patterns. In this paper, we focus on the use of *succinct constraints*. In particular, we propose a novel algorithm, called *dyFPS*, for *dynamic FP-tree based mining of frequent patterns satisfying succinct constraints*. Here, the term "dynamic" means that, in the middle of the mining process, users are able to modify the succinct constraints they specified. In terms of functionality, our algorithm is capable of handling these modifications effectively by exploiting succinctness properties of the constraints in an FP-tree based framework. In terms of performance, the dyFPS algorithm efficiently computes all frequent patterns satisfying the constraints.

Keywords: Data mining, constraints, succinctness, dynamic changes, FP-trees

1 Introduction

The problem of mining association rules [1, 2] – and the more general problem of finding frequent patterns – from large databases has been the subject of numerous studies. Over the past decade, frequent-pattern mining has been generalized to many forms, which include constrained data mining. The use of *constraints* permits users to specify the patterns to be mined according to their intention, and thereby allowing user focus and guidance. Consequently, the computation is limited to what interests the users. In addition, constrained mining also enables user exploration and control. As a result, effective pruning of the search space and efficient mining of frequent patterns can be achieved.

To handle constraints in the process of mining frequent patterns from large databases, many different approaches have been proposed. The following are some examples. Srikant et al. [12] considered item constraints in association rule mining. Bayardo et al. [4] developed Dense-Miner to mine association rules with the user-specified consequent meeting "interestingness" constraints (e.g., minimum support, minimum confidence, minimum improvement). Garofalakis

B. Kuijpers and P. Revesz (Eds.): CDB 2004, LNCS 3074, pp. 112–127, 2004.

Auxiliary information about items:

Item	a	b	c	d	e	f	g
$Price$	36	12	24	28	20	32	16
Qty	700	300	400	500	800	600	200
$Type$	$soda$	$soda$	$snack$	$beer$	$beer$	$beer$	$meat$

$\boldsymbol{C_1}$: $min(S.Qty) \geq 500$
$\boldsymbol{C_2}$: $max(S.Price) \geq 28$
$\boldsymbol{C_3}$: $S.Type \supseteq \{snack, soda\}$
$\boldsymbol{C_4}$: $S.Price = 16$
$\boldsymbol{C_5}$: $S.Type \subseteq \{beer, snack\}$
$\boldsymbol{C_6}$: $soda \in S.Type$
$\boldsymbol{C_7}$: $max(S.Price)/avg(S.Price) \leq 7$

Fig. 1. Examples of various classes of constraints

et al. [5] developed SPIRIT to mine frequent sequential patterns satisfying regular expression constraints. Ng et al. [8, 10] proposed a constrained frequent-set mining framework within which users can use a rich set of constraints – including SQL-style aggregate constraints (e.g., C_1 and C_2 in Fig. 1) and non-aggregate constraints (e.g., C_3, C_4, C_5 and C_6) – to guide the mining process to find only those rules satisfying the constraints. In Fig. 1, constraint $C_1 \equiv min(S.Qty) \geq 500$ says that the minimum Qty value of all items in the set S is at least 500. Constraint $C_3 \equiv S.Type \supseteq \{snack, soda\}$ says that the set S includes some items whose $Type$ is $snack$ and some items whose $Type$ is $soda$; constraint $C_4 \equiv S.Price = 16$ says that all items in the set S are of $Price$ equal to 16. Ng et al. also developed the CAP algorithm in the constrained frequent-set mining framework mentioned above. Such an Apriori-based algorithm exploits properties of *anti-monotone constraints* and/or *succinct constraints* to give as much pruning as possible. Constraints such as $C_1, ..., C_6$ in Fig. 1 are **succinct** because one can directly generate precisely all and only those itemsets satisfying the constraints (e.g., by using a precise "formula", called a member generating function, that does not require generation and testing of itemsets not satisfying the constraints). For instance, itemsets satisfying constraint $C_2 \equiv max(S.Price) \geq 28$ can be precisely generated by combining at least one item whose $Price \geq 28$ with some possible optional items (whose $Price$s are unimportant), thereby avoiding the substantial overhead of the generate-and-test paradigm. It is important to note that *a majority of constraints in this constrained frequent-set mining framework is succinct*; for those constraints that are not succinct, many of them can be induced to weaker constraints that are succinct! Among the succinct constraints in Fig. 1, $C_1 \equiv min(S.Qty) \geq 500$ is also **anti-monotone** because any superset of an itemset violating the constraint (i.e., containing an item whose $Qty < 500$) also violates the constraint. Grahne et al. [6] also exploited the anti-monotone and/or succinct constraints, but they mined valid correlated itemsets. Pei et al. [11] developed the $\mathcal{FIC}$ algorithms, which integrate a tree-based mining framework with constraint pushing. Specifically, to enhance performance,

$\mathcal{FIC}$ uses an extended prefix-tree structure – called *Frequent Pattern tree* or *FP-tree* [7] – to capture the content of the transaction database. The algorithms exploit the so-called convertible constraints (e.g., C_7 in Fig. 1). Leung et al. [9] exploited succinct constraints, and developed the FPS algorithm to effectively mine frequent itemsets satisfying succinct constraints. However, their FPS algorithm does not handle dynamic changes to succinct constraints.

It is well-known that data mining is supposed to be a human-centered and exploratory process. Human involvement is not confined to user focus (i.e., constrained mining); it also includes *interactive mining*. With interactive mining, users can (i) monitor the mining process and (ii) make change dynamically during the mining process, and thus having a decisive influence on the mining process.

In this paper, our **key contribution** is the development of a novel algorithm, called *dyFPS*, for *dynamic FP-tree based mining of frequent patterns satisfying succinct constraints*. This algorithm can be considered as a non-trivial extension of the FPS algorithm [9]. In terms of functionality, the dyFPS algorithm allows users to modify the succinct constraints in the middle of the mining process, and it handles these modifications very effectively. In terms of performance, the dyFPS algorithm, like its static counterpart (i.e., the FPS algorithm), is very efficient because it avoids the generate-and-test paradigm by exploiting succinctness properties of the constraints in an FP-tree based framework.

This paper is organized as follows. In the next section, relevant background material is described. Section 3 presents an overview of our proposed dyFPS algorithm. Section 4 shows the experimental results. Finally, conclusions are presented in Section 5.

2 Background

In this section, we first give a definition of succinct constraints. We then provide an overview of the FPS algorithm [9], which is an FP-tree based mining algorithm for handling succinct constraints.

2.1 Succinct Constraints

Definition 1 (Succinctness [10]). *Define* $\text{SAT}_C(\texttt{Item})$ *to be the set of itemsets that satisfy the constraint* C*. With respect to the lattice space consisting of all itemsets,* $\text{SAT}_C(\texttt{Item})$ *represents the pruned space consisting of those itemsets satisfying* C*. We use the notation* 2^I *to mean the powerset of* I*.*

(a) $I \subseteq \texttt{Item}$ *is a* succinct set *if it can be expressed as* $\sigma_p(\texttt{Item})$ *for some selection predicate* p*, where* σ *is the selection operator.*
(b) $SP \subseteq 2^{\texttt{Item}}$ *is a* succinct powerset *if there is a fixed number of succinct sets* $\texttt{Item}_1, ..., \texttt{Item}_k \subseteq \texttt{Item}$ *such that* SP *can be expressed in terms of the powersets of* $\texttt{Item}_1, ..., \texttt{Item}_k$ *using union and minus.*
(c) A constraint C *is* **succinct** *provided that* $\text{SAT}_C(\texttt{Item})$ *is a succinct powerset.*

Consider constraint $C_3 \equiv S.Type \supseteq \{snack, soda\}$. Its pruned space consists of all those itemsets that contain at least one *snack* item and at least one *soda* item. Let $\texttt{Item}_1$, $\texttt{Item}_2$ and $\texttt{Item}_3$, respectively, be the sets $\sigma_{Type=snack}(\texttt{Item})$, $\sigma_{Type=soda}(\texttt{Item})$ and $\sigma_{Type\neq snack \wedge Type\neq soda}(\texttt{Item})$. Then, $\texttt{Item}_1$ contains all the *snack* items, $\texttt{Item}_2$ contains all the *soda* items, and $\texttt{Item}_3$ contains neither a *snack* item nor a *soda* item. Hence, C_3 is succinct because its pruned space $\text{SAT}_{C_3}(\texttt{Item})$ can be expressed as $2^{\texttt{Item}} - 2^{\texttt{Item}_1} - 2^{\texttt{Item}_2} - 2^{\texttt{Item}_1 \cup \texttt{Item}_3} - 2^{\texttt{Item}_2 \cup \texttt{Item}_3}$. Although $\text{SAT}_{C_3}(\texttt{Item})$ is a complicated expression involving several unions and minuses, itemsets satisfying the succinct constraint C_3 can be directly generate precisely (i.e., *without* generating and then excluding those itemset not satisfying C_3). More specifically, every itemset ν satisfying C_3 can be efficiently enumerated by combining (i) an item from $\texttt{Item}_1$ (i.e., a *snack* item), (ii) an item from $\texttt{Item}_2$ (i.e., a *soda* item), and (iii) some possible optional items from any of $\texttt{Item}_1, \texttt{Item}_2$ and $\texttt{Item}_3$.

Although succinct constraints can be of various forms, they can be categorized into the following subclasses [9], depending on whether or not they are also anti-monotone:

- **SAM constraints.** Succinct anti-monotone constraints, such as $C_1 \equiv min(S.Qty) \geq 500$, $C_4 \equiv S.Price = 16$, $C_5 \equiv S.Type \subseteq \{beer, snack\}$ in Fig. 1; and
- **SUC constraints.** Succinct non-anti-monotone constraints, such as $C_2 \equiv max(S.Price) \geq 28$, $C_3 \equiv S.Type \supseteq \{snack, soda\}$, $C_6 \equiv soda \in S.Type$.

2.2 The FPS Algorithm

Like many FP-tree based algorithms, the FPS algorithm [9] consists of two main operations: (i) the construction of an FP-tree, and (ii) the growth of frequent patterns. The FPS algorithm first generates valid items (i.e., items satisfying succinct constraints) by using a member generating function, and then scans the transaction database to check the frequency of each valid item. As a result, valid frequent singleton itemsets can be found. The FPS algorithm then builds an initial FP-tree, which captures the content of the transaction database. By using this tree, a projected database can be formed for each valid item X. Here, an X-projected database is a collection of transactions having X as prefix. By *recursively* applying this FP-tree based mining process to each of these projected databases, valid frequent itemsets can be found. More specifically, suppose $X_1, X_2, X_3, ...$ are valid items. Then, itemsets containing X_1 can be computed from the $\{X_1\}$-projected database (and subsequent projected databases having X_1 as prefix), itemsets containing X_2 but not X_1 can be computed from the $\{X_2\}$-projected database, itemsets containing X_3 but not X_1 or X_2 can be computed from the $\{X_3\}$-projected database, and so on. Therefore, the entire mining process can be viewed as a divide-and-conquer approach of decomposing both the mining task and the transaction database according to the frequent patterns obtained so far. This leads to a focused search of smaller datasets.

The FPS algorithm consists of two components – **FPSam** and **FPSuc** – for handling SAM and SUC constraints, respectively. The key differences between these two components are as follows. *FPSam* divides the domain items into two groups (i.e., valid and invalid groups). Among the two groups, *only* items from the valid group are considered when forming the initial FP-tree and subsequent projected databases because any frequent itemset satisfying a SAM constraint is composed of only valid items. For example, any frequent itemset satisfying $C_1 \equiv min(S.Qty) \geq 500$ is composed of only items whose $Qty \geq 500$. In contrast, *FPSuc* divides the domain items into mandatory and optional groups[1]. It uses items from the mandatory and optional groups because any frequent itemset satisfying a SUC constraint is composed of mandatory items and possibly some optional items. For example, any frequent itemset satisfying $C_2 \equiv max(S.Price) \geq 28$ is composed of at least one item whose $Price \geq 28$ and possibly some optional items (whose $Price$s are unimportant). During the mining process, items are ordered in such a way that mandatory items appear before optional ones (i.e., all mandatory items appear below any optional ones in the FP-tree). Projected databases are formed only for the mandatory items, with the aforementioned item ordering. Consequently, all and only those frequent itemsets having appropriate mandatory items as prefix can be computed. In other words, all computed itemsets are valid (i.e., guaranteed to contain mandatory items and may contain some optional items). While more details can be found in the work of Leung et al. [9], we use the following example to illustrate an execution of the (static) FPS algorithm.

Example 1. Consider the following transaction database:

Transaction ID	Contents: *itemset*
t_1	$\{a, b, d\}$
t_2	$\{a, b, c, d, e, g\}$
t_3	$\{a, c\}$
t_4	$\{a, b, c, d, e, g\}$
t_5	$\{c\}$

with the auxiliary information from Fig. 1:

`Item`	a	b	c	d	e	f	g
$Price$	36	12	24	28	20	32	16
Qty	700	300	400	500	800	600	200
$Type$	*soda*	*soda*	*snack*	*beer*	*beer*	*beer*	*meat*

Let the minimum support threshold be 2 (i.e., 40%). The FPS algorithm first generates items satisfying the succinct constraint $C_2 \equiv max(S.Price) \geq 28$ by

[1] More precisely, FPSuc divides the domain items into k mandatory groups and one optional group. Although constraint $C_3 \equiv \{snack, soda\}$ uses two mandatory groups (one for *snack* items and another for *soda* items), *most of SUC constraints require only one mandatory group*. For lack of space, we focus on the latter (i.e., $k = 1$) in this paper.

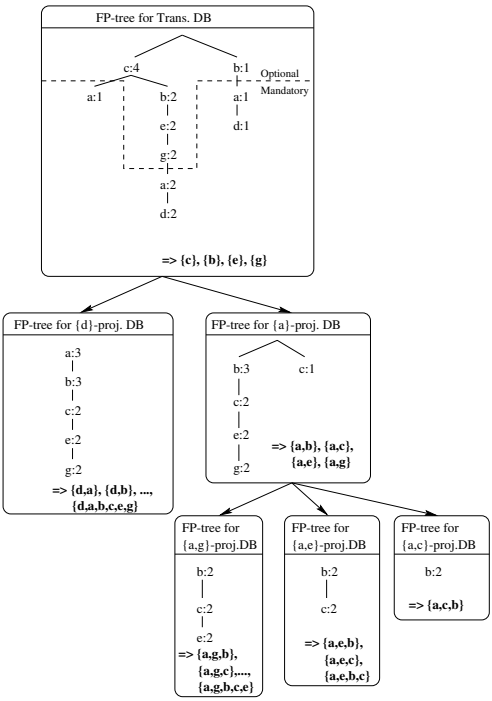


Fig. 2. The FPS algorithm mines frequent itemsets satisfying SUC constraint C_2

using a member generating function, and then scans the transaction database to check the frequency of each valid items. As a result, it finds valid frequent singleton itemsets $\{a\}$ and $\{d\}$. Then, FPS builds an initial FP-tree, as shown in Fig. 2. (The frequency of each item is shown in the figure, e.g., "c:4" in the tree indicates that the frequency of $\{c\}$ is 4.) Afterwards, FPS forms a projected database for each valid item (i.e., items a and d). This FP-tree based mining process is applied to the $\{d\}$-projected database to find all 31 valid frequent itemsets containing d. Similarly, the mining process is recursively applied to the $\{a\}$-projected database (and projected databases of the supersets of $\{a\}$). As a result, all the 15 valid frequent itemsets containing a but not d are found. □

3 Handling Dynamic Changes to Succinct Constraints

After reviewing the related work, let us start our discussion on the contribution of this paper – the development of the dyFPS algorithm. Like FPS, our dyFPS algorithm consists of two main components: *dyFPSam* and *dyFPSuc* for handling a dynamic change (e.g., a tightening change, a relaxing change) to a SAM constraint and a SUC constraint, respectively. Here, we assume that, at any point in time, there is at most one succinct constraint being modified. Of course, during the entire mining process, many different constraints can be changed. The

base case of our discussion below is on how to deal with changes to the constant *const*. When *const* is modified by the users in the direction of restricting the new solution space (denoted as $\boldsymbol{V_{new}}$) to be a *subset* of the old space (denoted as $\boldsymbol{V_{old}}$), we call this a *tightening change*. Otherwise, whenever the change to *const* corresponds to the situation where the new solution space contains the old space, we call this a *relaxing change*. For example, if the original succinct constraint is $max(S.Price) \geq 28$, then (i) changing the constant *const* from 28 to 32 corresponds to a tightening change and (ii) changing *const* from 28 to 24 corresponds to a relaxing change. Clearly, inserting a new constraint is a special case of a tightening change, and deleting an old constraint is an extreme case of a relaxing change. Thus, while our discussion below is confined to changing the constant *const*, any other modification (e.g., modifying $max(S.Price) \geq 28$ to $max(S.Price) \leq 28$) can be dealt with as a pair of constraint deletion and insertion.

A **naïve approach** of handling dynamic changes to succinct constraints is to simply ignore all valid frequent itemsets that have been produced so far with respect to the old constraint C_{old} (i.e., ignore all the "processed" itemsets) and rerun the FPS algorithm again with the new constraint C_{new}. Obviously, this approach can be costly because it does not reuse any "processed" frequent itemsets satisfying C_{old}.

Is there any better approach? When computing frequent itemsets satisfying C_{new}, can we reuse some of those "processed" frequent itemsets satisfying C_{old}? The answer to these questions is yes. In the remainder of this section, we show how our dyFPS algorithm pushes the constraints deep inside the mining process for effective pruning. The algorithm reuses the "processed" itemsets (i.e., valid frequent itemsets that have been produced with respect to C_{old}) as much as possible when mining valid frequent itemsets satisfying C_{new}.

3.1 Handling Dynamic Changes to a SAM Constraint

By definition, a **tightening change** from C_{old} to C_{new} corresponds to a restriction of the old solution space V_{old}. In other words, the new solution space V_{new} is a *subset* of V_{old}. To accommodate C_{new} dynamically, the dyFPS algorithm carries out two main operations as follows:

- For processed frequent itemsets satisfying C_{old}, check if they still satisfy C_{new}.
- For unprocessed itemsets, dyFPS only generates those satisfying C_{new}.

Recall that any frequent itemset ν satisfying a SAM constraint is composed of only valid items (i.e., items satisfying the constraint individually): $\nu \subseteq G^V$ (where G^V denotes a set of valid items). For a tightening change, any frequent itemset ν satisfying C_{new} is a subset of G^V_{new} (where G^V_{new} denotes the set of items satisfying C_{new}), which is a *subset* of G^V_{old} (where G^V_{old} denotes the set of items satisfying C_{old}):

$$\nu \subseteq G^V_{new} \subseteq G^V_{old}. \tag{1}$$

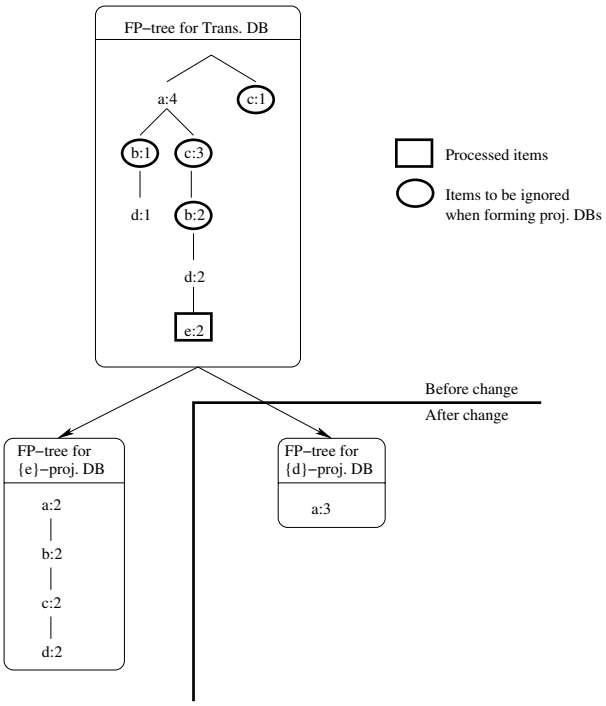


Fig. 3. Handling a dynamic tightening change to a SAM constraint using dyFPSam

Hence, after a SAM constraint is tightened, dyFPSam uses the existing FP-tree to generate the unprocessed itemsets satisfying C_{new}. Specifically, the algorithm forms projected databases *only* for unprocessed items satisfying C_{new}. When forming an $\{X\}$-projected database for an item $X \in G^V_{new}$, the dyFPSam algorithm *excludes* all **negative delta items** (i.e., the items that satisfy C_{old} but not C_{new}), which can be efficiently enumerated due to succinctness. As a result, dyFPSam can compute all the valid frequent itemsets (that have not been processed before the constraint change). To complete the mining process, dyFPSam needs to check the validity of all the frequent itemsets that were processed before the change (i.e., to check whether these itemsets still satisfy C_{new}). Example 2 shows how dyFPSam handles a tightening change to a SAM constraint.

Example 2. Consider the same transaction database & minimum support threshold as in Example 1. Suppose a SAM constraint $C_{old} \equiv min(S.Qty) \geq 300$ is tightened to $C_{new} \equiv min(S.Qty) \geq 500$ after the $\{e\}$-projected database has been processed (i.e., after dyFPSam computed all 15 itemsets containing e, such as $\{e, a\}, \{e, b\}, ..., \{e, a, b, c, d\}$). Then, due to succinctness, the negative delta items can be efficiently enumerated: items b and c. Therefore, as shown in Fig. 3, when forming the $\{d\}$-projected database, dyFPSam excludes items b and c from the extracted paths $\langle a, b\rangle$:1 and $\langle a, c, b\rangle$:2, and builds an FP-tree for

the $\{d\}$-projected database (which contains a:3). Consequently, a valid frequent itemset $\{d, a\}$ is computed. Finally, to complete the mining process, dyFPSam checks all the 15 itemsets containing e (i.e., the "processed" itemsets satisfying C_{old}), and finds that all of them satisfy C_{new}. □

We have discussed how dyFPSam handles a dynamic tightening change to a SAM constraint. In the remainder of this section, let us consider how dyFPSam handles a dynamic relaxing change to a SAM constraint. In general, a **relaxing change** from C_{old} to C_{new} has different and tougher computational requirements than a tightening change. The reason is that a tightening change leads to the situation where $V_{new} \subseteq V_{old}$; so, all that is needed is to verify whether every itemset ν satisfying C_{old} also satisfies C_{new}. In contrast, for a relaxing change, this verification is unnecessary because $V_{new} \supseteq V_{old}$. What is needed, however, is to compute all the "missing" itemsets (i.e., the itemsets that satisfy C_{new} but were not generated before the constraint was relaxed). Therefore, dyFPS needs to carry out the following operations:

- For processed itemsets satisfying C_{old}, no further constraint check is required because they also satisfy C_{new}.
- For unprocessed itemsets, dyFPS generates (i) the remaining itemsets satisfying C_{old} and (ii) those satisfying C_{new} but not C_{old}.

For a relaxing change, any frequent itemset ν satisfying C_{new} is a subset of G^V_{new}, which is a *superset* of G^V_{old}:

$$\nu \subseteq G^V_{new}, \quad \text{but } G^V_{new} \supseteq G^V_{old}. \tag{2}$$

Hence, after a SAM constraint is relaxed, dyFPSam adds **positive delta items** (i.e., items satisfying C_{new} but not C_{old}) to appropriate branches of the tree[2]. Then, the algorithm forms projected databases for all unprocessed items satisfying C_{new}. Again, due to succinctness, these items – as well as positive delta items – can be efficiently enumerated. The example below shows how dyFPSam handles a relaxing change to a SAM constraint.

Example 3. Consider the same transaction database & minimum support threshold as in Example 1. Suppose a SAM constraint $C_{old} \equiv min(S.Qty) \geq 500$ is relaxed to $C_{new} \equiv min(S.Qty) \geq 300$ after the $\{e\}$-projected database has been processed (i.e., after obtaining itemsets $\{e, a\}, \{e, d\}$ and $\{e, a, d\}$). Then, due to succinctness, the positive delta items can be efficiently enumerated: items b and c. They are added to the appropriate branches of the tree. Afterwards, as shown in Fig. 4, projected databases are formed for all unprocessed items satisfying C_{new} (e.g., for items d, b and c). Valid frequent itemsets can then be computed from these projected databases. □

[2] If I/O is a concern, one can keep both valid and invalid items in the tree. By so doing, positive delta items are already in the tree. This would save a database scan.

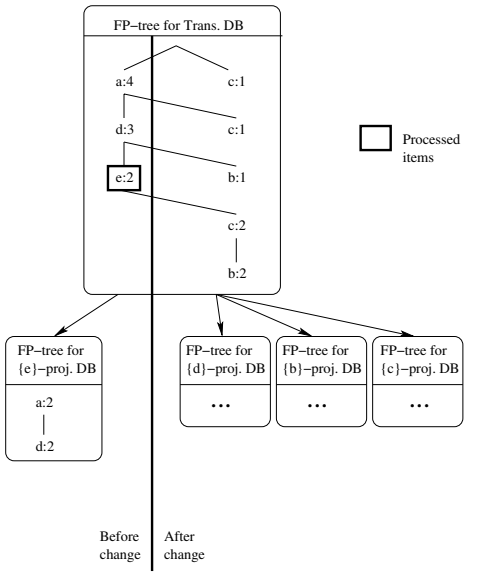


Fig. 4. Handling a dynamic relaxing change to a SAM constraint using dyFPSam

3.2 Handling Dynamic Changes to a SUC Constraint

One can observe from the previous section that dynamic changes to a SAM constraint cause the inclusion, or the exclusion, of items from the FP-tree. However, for dynamic changes to a SUC constraint, the situation is different. A reason is that, for a SUC constraint, its initial FP-tree contains not only mandatory items (i.e., items satisfying the constraint C) but also optional items (i.e., items not satisfying C). Changes to a SUC constraint only cause *a change of membership* (e.g., from the mandatory group G^M to the optional group G^O for the tightening change, and the opposite for the relaxing change).

Recall that any frequent itemset ν satisfying a SUC constraint is composed of at least one mandatory item and possibly some optional items:

$$\nu = \{X\} \cup \beta \cup \gamma \tag{3}$$

where (i) $X \in G^M$, (ii) $\beta \subseteq G^M$, and (iii) $\gamma \subseteq G^O$. So, after a SUC constraint is tightened, some of the mandatory items become optional (i.e., change their membership from mandatory to optional). These items are the ones that satisfy C_{old} but not C_{new}; we denote the group containing these items as $\boldsymbol{G^{M \to O}}$, which is a subgroup of G^M. Similarly, we denote the subgroup of G^M that contains items satisfying both C_{new} and C_{old} (i.e., the unchanged group) as $\boldsymbol{G^{M \to M}}$.

A complication of handling dynamic changes to a SUC constraint is that effective computation of itemsets satisfying a SUC constraint relies on the item ordering in the FP-tree. So far, we have only imposed the "inter-group" item ordering. All items from the mandatory group must come before any items from

the optional group (i.e., all mandatory items appear below any optional items in the FP-tree). However, there is no restriction imposed on the "intra-group" ordering (e.g., no restriction on the ordering of items within G^M). An item belonging to $G^{M \to M}$ may consistently appear above or below an item belonging to $G^{M \to O}$ in the FP-tree. If dyFPSuc were to apply the usual projection technique, the algorithm could be incomplete because it could possibly miss some itemsets containing an item in $G^{M \to O}$.

To ensure completeness, one could reorder the items so that items belonging to $G^{M \to O}$ appear after/above all the items belonging to $G^{M \to M}$. However, this approach may incur a costly overhead due to node split and/or node merge (i.e., the tree reorganization cost). The dyFPSuc algorithm avoids constructing a new FP-tree by using the existing FP-tree as follows. It forms projected databases *only* for items satisfying C_{new}. When dyFPSuc forms an $\{X\}$-projected database for an item X, in addition to *including* all items that are *above* X (as usual), dyFPSuc also *includes* all "unprocessed" items belonging to $G^{M \to O}$ that are *below* X. Due to succinctness, the items belonging to $G^{M \to O}$ can be efficiently enumerated. Note that this approach gives the same results as those produced by reordering items, but it avoids the costly overhead. As usual, for a tightening change, the algorithm needs to check the validity of all the frequent itemsets that were processed before the change (i.e., to check whether these "processed" itemsets still satisfy C_{new}). To gain a better understanding on how dyFPSuc handles a tightening change to a SUC constraint, let us consider the following example.

Example 4. Consider the same transaction database & minimum support threshold as in Example 1. Suppose a SUC constraint $C_{old} \equiv max(S.Price) \geq 20$ is tightened to $C_{new} \equiv max(S.Price) \geq 28$ after the $\{e\}$-projected database has been processed (i.e., after obtaining all 31 itemsets containing the mandatory item e, such as $\{e, a\}, \{e, b\}, ..., \{e, a, b, c, d, g\}$). Then, due to succinctness, items belonging to $G^{M \to O}$ can be efficiently enumerated: items c and e. Therefore, as shown in Fig. 5, when forming the $\{d\}$-projected database, dyFPSuc includes (i) all the items that are above d and (ii) all the "unprocessed" items belonging to $G^{M \to O}$ that are below d. Since item e has already been processed, it is not included. Then, dyFPSuc forms the $\{a\}$-projected database. Here, in addition to including all the items that are above a (i.e., items b and g), dyFPSuc also includes all the "unprocessed" items belonging to $G^{M \to O}$ that are below a (i.e., item c). Valid frequent itemsets can then be computed using these projected databases. Finally, to complete the mining process, dyFPSuc checks all the 31 itemsets containing e (i.e., the "processed" itemsets satisfying C_{old}), and finds that 24 of them satisfy C_{new}. □

A tightening change to a SUC constraint causes some items to change their membership from mandatory to optional. In contrast, a relaxing change to a SUC constraint leads to the opposite situation – that is, some items to change their membership from optional to mandatory.

Since both tightening and relaxing changes cause changes of membership, their treatment is quite similar, except the following. For a relaxing change, some optional items become mandatory; thus, dyFPSuc forms projected databases

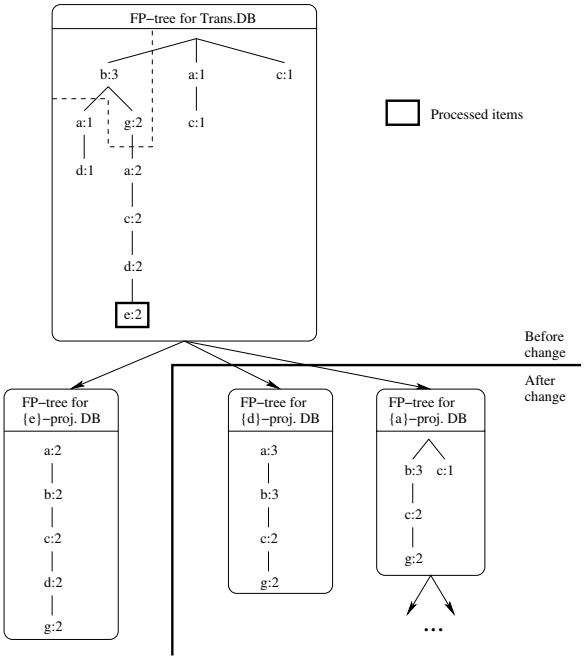


Fig. 5. Handling a dynamic tightening change to a SUC constraint using dyFPSuc

in such a way that it gives the same results as those produced by reordering items (i.e., reordering the items in such a way that mandatory items appear before/below all the optional items). For lack of space, we do not describe it further. We use Example 5 to illustrate how dyFPSuc handles a relaxing change to a SUC constraint.

Example 5. Consider the same transaction database & minimum support threshold as in Example 1. Suppose a SUC constraint $C_{old} \equiv max(S.Price) \geq 28$ is relaxed to $C_{new} \equiv max(S.Price) \geq 20$ after the $\{d\}$-projection has been processed (i.e., after obtaining all 31 itemsets containing d, such as $\{d, a\}, \{d, b\}, \{d, c\}, ...,$ $\{d, a, b, c, e, g\}$). Then, due to succinctness, the optional items (w.r.t. both C_{old} and C_{new}) can be efficiently enumerated: items b and g. Therefore, as shown in Fig. 6, dyFPSuc forms the $\{a\}$-projected database as usual. When forming the $\{e\}$-projected database, dyFPSuc includes all the items that are above e (i.e., items b and c) and all the "unprocessed" optional items that are below e (i.e., item g). Similarly, when dyFPSuc forms the $\{c\}$-projected database, dyFPSuc includes all the items that are above c (which is none) and all the "unprocessed" optional items that are below c (i.e., items b and g). Consequently, valid frequent itemsets can be computed by using these projected databases. □

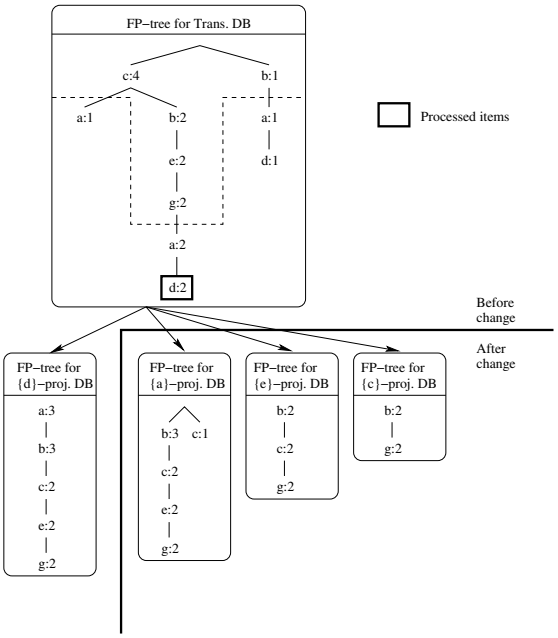
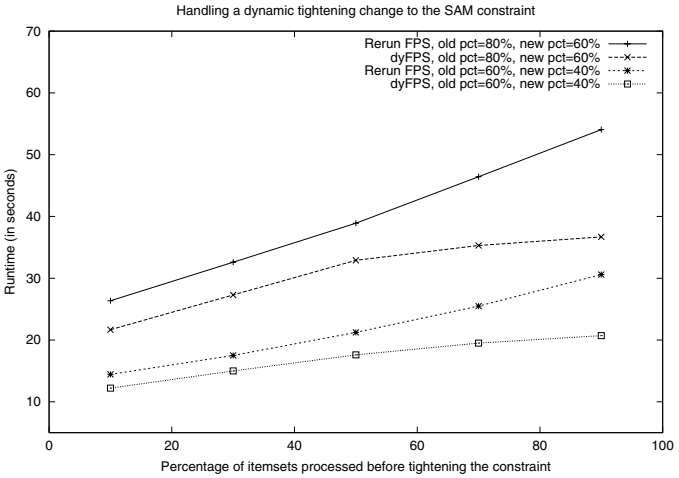


Fig. 6. Handling a dynamic relaxing change to a SUC constraint using dyFPSuc

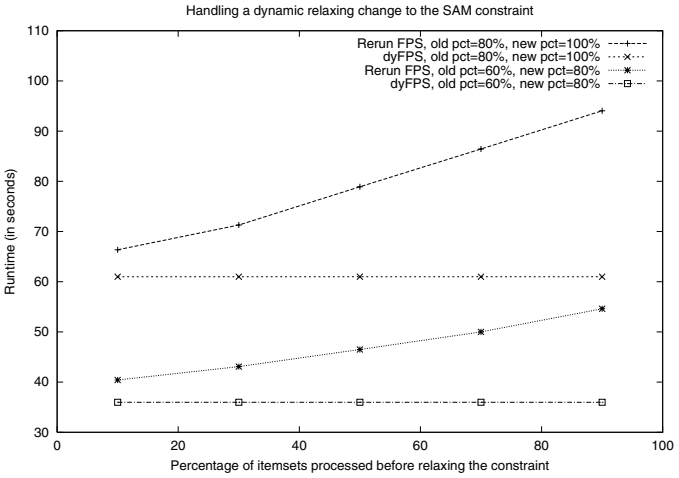
4 Experimental Results

The experimental results cited below are based on a transaction database of 100k records with an average transaction length of 10 items, and a domain of 1000 items. The database was generated by the program developed at IBM Almaden Research Center [3]. Unless otherwise specified, we used a minimum support threshold of 0.01%. All experiments were run in a time-sharing environment in a 700 MHz machine. The reported figures are based on the average of multiple runs. In the experiment, we mainly compared two algorithms that were implemented in C: (i) **dyFPS** vs. (ii) **Rerun FPS**.

Recall that a naïve approach of handling dynamic changes to succinct constraints is to simply ignore all valid frequent itemsets that have been produced so far with respect to the old constraint C_{old} (i.e., ignore all "processed" itemsets) and to rerun the FPS algorithm again using the new constraint C_{new}. A better approach is to use our dyFPS algorithm, which reuses all "processed" itemsets. To evaluate the effectiveness of our algorithm, in this experiment, a SAM constraint $C_{old} \equiv max(S.Price) \leq 10$ is tightened to $C_{new} \equiv max(S.Price) \leq 8$. The percentage *old pct* of items having $Price \leq 10$ and the percentage *new pct* of items having $Price \leq 8$ are set in such a way that *new pct* = *old pct* − 20%. We varied *old pct* from 60% to 80%. The x-axis in Fig. 7(a) shows the percentage x of itemsets processed before tightening the constraint, and x varied from 10% to 90%. The y-axis shows the total runtime (in seconds) of both algorithms (dyFPS



(a) Tightening change



(b) Relaxing change

Fig. 7. Effectiveness of handling dynamic constraint changes: dyFPS vs. Rerun

and the Rerun FPS). From the graph, it is clear that our dyFPS algorithm always beats the rerun of FPS, but the extent varies under different situations. When x is high (i.e., more itemsets are processed), the relative speedup is high. The reason is that the Rerun FPS ignores all the itemsets produced w.r.t. C_{old} and (re-)computes itemsets satisfying C_{new}. Hence, when x is high, more itemsets are generated and ignored by the Rerun FPS. Out of these itemsets (produced using C_{old}), many of them satisfy C_{new}. The higher the percentage x, the higher is the number of processed itemsets being ignored! Our dyFPS algorithm, on the other hand, reuses the processed itemsets. Hence, when x increases, the number

of processed itemsets needed to be check at the final step of the mining process increases. However, the time required for checking the validity of these itemsets is much less than that required for (re-)computing itemsets. This explains why it is more beneficial to use dyFPS than to rerun FPS – especially when x is high.

While Figure 7(a) shows the results for a tightening change, Figure 7(b) shows the results for a relaxing change. Specifically, we relaxed a SAM constraint $C_{old} \equiv max(S.Price) \leq 10$ to $C_{new} \equiv max(S.Price) \leq 12$. The percentage *old pct* of items having $Price \leq 10$ and the percentage *new pct* of items having $Price \leq 12$ are set in such a way that *new pct* = *old pct* + 20%. We varied *old pct* from 60% to 80%. Again, it is clear from the graph that our dyFPS algorithm always beats the rerun of FPS. Similar to the tightening case, the Rerun FPS ignores all the itemsets produced w.r.t. C_{old} and (re-)computes itemsets satisfying C_{new}. Hence, the higher the percentage x, the higher is the number of itemsets that are generated and ignored by the Rerun FPS. Here, *all* of these itemsets (produced using C_{old}) satisfy C_{new} because $V_{new} \supseteq V_{old}$ for the relaxing change. Thus, using the Rerun FPS is really a waste of computation! On the other hand, our dyFPS algorithm reuses all these processed itemsets. After the constraint C_{old} is relaxed to become C_{new}, our dyFPS algorithm generates (i) the remaining itemsets satisfying C_{old} and (ii) those satisfying C_{new} but not C_{old}. This explains why the runtime of dyFPS is quite steady. Therefore, it is more beneficial to use dyFPS than to rerun FPS – especially when x is high.

We have also experimented with the following cases: (i) tightened a SUC constraint $C_{old} \equiv min(S.Price) \leq 10$ to $C_{new} \equiv min(S.Price) \leq 8$, and (ii) relaxed a SUC constraint $C_{old} \equiv min(S.Price) \leq 10$ to $C_{new} \equiv min(S.Price) \leq 12$. The results produced are consistent with those for the SAM constraint. Our proposed dyFPS algorithm outperforms the rerun of FPS. In summary, these experimental results show the effectiveness of exploiting succinctness properties of the constraints (e.g., reusing the "processed" itemsets).

5 Conclusions

A key contribution of this paper is to optimize the performance of, and to increase functionality of, a dynamic FP-tree based constrained mining algorithm. To this end, we proposed (and studied) the novel algorithm of dyFPS. The algorithm efficiently handles dynamic changes to succinct constraints, and effectively exploits succinctness properties. As a result, the constraints are pushed deep inside the mining process, and thereby avoiding algorithm rerun and leading to effective pruning.

In ongoing work, we are interested in exploring improvements to the dyFPS algorithm. For example, we are interested in investigating effective technique for handling dynamic changes to some more sophisticated Boolean combinations of succinct constraints. Along this direction, an interesting question to explore is how to extend our dyFPS algorithm to mine frequent patterns satisfying non-succinct constraints.

Acknowledgement

This project is partially sponsored by The University of Manitoba in the form of research grants.

References

1. Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: Proc. SIGMOD 1993. 207–216
2. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast discovery of association rules. In: Advances in Knowledge Discovery and Data Mining. AAAI/MIT Press (1996) chapter 12
3. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proc. VLDB 1994. 487–499
4. Bayardo, R.J., Agrawal, R., Gunopulos, D.: Constraint-based rule mining in large, dense databases. In: Proc. ICDE 1999. 188–197
5. Garofalakis, M.N., Rastogi, R., Shim, K.: SPIRIT: sequential pattern mining with regular expression constraints. In: Proc. VLDB 1999. 223–234
6. Grahne, G., Lakshmanan, L.V.S., Wang, X.: Efficient mining of constrained correlated sets. In: Proc. ICDE 2000. 512–521
7. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proc. SIGMOD 2000. 1–12
8. Lakshmanan, L.V.S., Ng, R., Han, J., Pang, A.: Optimization of constrained frequent set queries with 2-variable constraints. In: Proc. SIGMOD 1999. 157–168
9. Leung, C.K.-S., Lakshmanan, L.V.S., Ng, R.T.: Exploiting succinct constraints using FP-trees. SIGKDD Explorations **4**(1) (June 2002) 40–49
10. Ng, R.T., Lakshmanan, L.V.S., Han, J., Pang, A.: Exploratory mining and pruning optimizations of constrained associations rules. In: Proc. SIGMOD 1998. 13–24
11. Pei, J., Han, J., Lakshmanan, L.V.S.: Mining frequent itemsets with convertible constraints. In: Proc. ICDE 2001. 433–442
12. Srikant, R., Vu, Q., Agrawal, R.: Mining association rules with item constraints. In: Proc. KDD 1997. 67–73