A FP-Tree-Based Method for Inverse Frequent Set Mining

Yuhong Guo, Yuhai Tong, Shiwei Tang, and Dongqing Yang

Department of Computer Science
Peking University, Beijing 100871, China
{yhguo, yhtong, tsw, dqyang}@pku.edu.cn

Abstract. Recently, the inverse frequent set mining problem has received more attention because of its important applications in different privacy preserving data sharing contexts. Several studies were reported to probe the NP-complete problem of inverse frequent set mining. However, it is still an open problem that whether there are reasonably efficient search strategies to find a compatible data set in practice. In this paper, we propose a FP-tree-based method for the inverse problem. Compared with previous "generation-and-test" methods, our method is a zero trace back algorithm, which saves huge computational costs. Furthermore, our algorithm provides a good heuristic search strategy to rapidly find a FP-tree, leading to rapidly finding the compatible databases. More importantly, our method can find a *set* of compatible databases instead of finding only *one* compatible database in previous methods.

1 Introduction

As the frequent itemsets can be considered as a kind of summary of the original data set, recently the inverse frequent set mining problem inferring the original data set from given frequent itemsets with the supports has received more attention because of its potential threat to privacy [1], its use in synthetic benchmark data set generation [2], and its potential application in sensitive association rule hiding in database [3].

Inverse frequent set (or "itemset") mining can be described as follows: "Given a collection of frequent itemsets and their supports, find a transactional data set (or "database") such that the new dataset precisely agrees with the supports of the given frequent itemset collection while the supports of other itemsets would be less than the pre-determined threshold" [4]. This inverse data mining problem is related to the questions of how well privacy is preserved in the frequent itemsets and how well the frequent itemsets characterize the original data set. It has very practical applications in different privacy preserving data sharing contexts from privacy preserving data mining (PPDM) to knowledge hiding in database (KHD), as the problem roots in people's increasing attention to information protection either to individual private data preserving or to business confidential knowledge hiding.

Mielikainen first proposed this inverse mining problem in [1]. He showed finding a dataset compatible with a given collection of frequent itemsets or deciding whether there is a dataset compatible with a given collection of frequent sets is NP-complete.

1.1 Related Work

Towards the NP-complete problem, several methods were proposed. The authors of [2, 3] designed a linear program based algorithm for *approximate* inverse frequent itemset mining, aiming to construct a transaction database that *approximately* satisfies the given frequent itemsets constraints. As for the *exact* inverse frequent itemset mining, Calder in his paper [5] gave a naive "generate-and-test" method to "guess" and build a database horizontally(transaction by transaction) from given frequent itemsets. On the contrary, the authors of [4] proposed a vertical database generation algorithm to "guess" a database vertically---column by column when looking the transaction database as a two-dimensional matrix. Unfortunately, under the generation-test framework, neither of the algorithms works well in terms of effectiveness and efficiency as they belong to simple enumerative search approaches essentially, which blindly try all possible value assignments, even those devoid of solutions. This means, once the "test" processes fail, the algorithms must rollback some costly "generate-and-test" operations, leading to huge computational cost.

Thus, a feasible and sufficient solution to the *exact* inverse frequent itemset mining is still expected. Obviously, if the given frequent sets collection comes from a real database, at least this original database must be one which exactly agrees with the given even though it is computationally hard to "render" it. The questions and challenges are: Is it unique? Can we find an efficient algorithm without trace back to find one? Can we find a good heuristic search strategy to reach one quickly?

This paper describes our effort towards finding an efficient method to find a set of databases that *exactly* agree with the given frequent itemsets and their supports discovered from a real database. Compared with previous "generation-and-test" methods, our proposed FP-tree-based method is a zero trace back algorithm, which saves huge computational costs. Furthermore, our algorithm provides a good heuristic search strategy to rapidly find a FP-tree, leading to rapidly finding the compatible databases. More importantly, our method can find a *set* of compatible databases instead of finding only *one* compatible database in previous methods.

1.2 Paper Layout

In section 2 we define the inverse frequent set mining problem that we focus on. In section 3 we review the FP-tree structure, and in section 4 we present our proposed algorithm. We analyze correctness and efficiency of our algorithm and discuss the number of databases generated in section 5. Section 6 summarizes our study.

2 Problem Description

Let $I = \{I_1, I_2, ..., I_m\}$ be a set of items, and a transaction database $D = \{T_1, T_2, ..., T_n\}$ where $T_i (i \in [1..n])$ is a transaction which contains a set of items in I. The support of an itemset $A \subseteq I$ in a transaction database D over I, denoted *support(A)*, is defined as the number of transactions containing A in D. A is a frequent itemset if A's support is no less than a predefined minimum support threshold "σ". If A is frequent and there exists no superset of A such that every transaction containing A also contains the superset, we say that A is a frequent *closed* itemset.

The well-known frequent itemset mining problem aims to find all frequent itemsets from a transaction database. And the objective of frequent closed itemset mining is to find all frequent *closed* itemsets. Conversely, inverse frequent itemset mining is to find the databases that satisfy the given frequent itemsets and their supports. If the given frequent itemsets are frequent *closed* itemsets, we call the inverse mining process *inverse frequent closed itemset mining*. Furthermore, if the frequent closed itemsets and their supports are discovered from a real database, we call the inverse mining process *inverse real frequent closed itemset mining*. Here, "real" only represents "existent", which means the real database can be an artificial data set.

In this paper, we focus on the *inverse real frequent closed itemset mining* defined as: Given a set of items $I = \{I_1, I_2, ..., I_m\}$, minimum support threshold "σ", and a set of all frequent *closed* itemsets $F = \{f_1, f_2, ...f_n\}$ with fix supports $S = \{support(f_1), support(f_2), ..., support(f_n)\}$ discovered from a real database D, find a set of databases *DBs* in which each database D' satisfies the following constraints:

(1) D' is over the same set of items I;

(2) From D', we can discover exactly same set of frequent closed itemsets"F"with the same support"S"under the same minimum support threshold"σ".

3 Frequent Pattern Tree

Frequent pattern tree (or FP-tree in short) proposed by Jiawei Han and efficiently used in frequent set mining, is an extended prefix-tree structure for storing compressed, crucial information about frequent patterns.

TID	Items	Ordered Frequent Item Sets
1	A, B, E	B, A, E
2	B, D	B, D
3	B, C	B, C
4	A, B, D	B, A, D
5	A, C	A, C
6	B, C	B, C
7	A, C	A, C
8	A, B, C, E	B, A, C, E
9	A, B, C	B, A, C

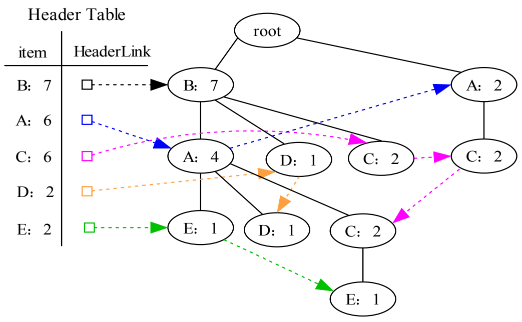


Fig. 1. A transaction database and its frequent pattern tree (FP-tree)

Fig. 1 gives an example of a transaction database and its FP-tree, which will be used in the next section. The database includes nine transactions comprising the items in the set *{A, B, C, D, E}*, which are shown in the mid column of the table. The FP-tree is constructed by two scans of the database. First scan of the database derives a *list* of frequent items 〈*B:7, A:6, C:6, D:2, E:2*〉 (the number after ":" indicates the support), in which items ordered in support descending order. The frequent items in each transaction are listed in this ordering in the rightmost column of the table. The

FP-tree forms in the second scan of the nine transactions in the rightmost column, with each transaction 'climbing' the FP-tree one by one. An item header table is built to facilitate tree traversal. Details of FP-tree construction process can be found in [6].

4 Proposed Method

4.1 Basic Idea

Our method to generate a database *D* from given frequent itemsets uses FP-tree as a transition "bridge" and can be seen as the reverse process of the FP-tree-based frequent itemsets mining method proposed in [6]. The idea comes from the fact that FP-tree is a highly compact structure which stores the complete information of a transaction database *D* in relevance to frequent itemsets mining. Thus we can look upon FP-tree as a medium production between an original database and its corresponding frequent itemsets. Intuitively, FP-tree reduces the gap between a database and its frequent itemsets, which makes the transformation from given frequent itemsets to database more smoothly, more naturally and more easily.

Our method works as follows. First, we try to "guess" a FP-tree that satisfies all the frequent itemsets and their supports. We call such a FP-tree a compatible FP-tree. Second, generate a corresponding database *TempD* directly from the compatible FP-tree by outspreading all the paths of the tree. Third, generate expected databases based on *TempD* by scattering some infrequent itemsets into the transactions in *TempD*, with the "new" itemsets brought below the given minimum support threshold.

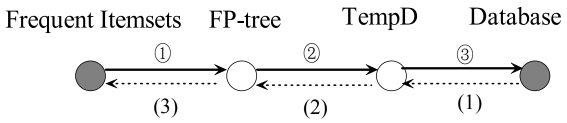


Fig. 2. Basic process of FP-tree-based inverse frequent set mining vs. frequent set mining

Fig. 2 shows the basic process of our proposed method for inverse frequent set mining, which is marked as ①, ② and ③. It corresponds the three steps described above. The process of the FP-tree-based frequent itemsets mining is also shown in the Fig. 2, which is composed of the three steps: (1), (2) and (3). Detailed information about the FP-tree-based frequent itemset mining process can be found in [6].

4.2 Algorithm

The sketch of our proposed inverse frequent closed set mining algorithm, which is composed of three procedures, is given as follows.

```
Gen_DB(F, I,σ)
Begin
1.  DBs← ∅, $I_{\overline{F}}$ ← I - { all items in F };
2.  FP←Gen_FPtree(F,σ);
3.  TempD←Outspread(FP);
```

```
4.  DBs←DBs ∪ {TempD};
5.  NewTempD←insert "some" items in $\overline{IF}$ into "some" transactions in TempD, be
    sure that each item in $\overline{IF}$ can only be inserted less than "$\sigma$" transactions;
6.  DBs←DBs ∪ {NewTempD};
7.  **Goto** 5 **until** no different NewTempD generated;
8.  **Return** DBs;
**End**

**Gen_FPtree**(F, $\sigma$)
**Begin**
1.  Create the root of an FP-tree, FP, and label it as "null";
2.  F' ←Sort(F);
3.  **While** (F' != ∅) **DO**
      a)  Select the first itemset f1:s1 , where f1 is the itemset and s1 is its support;
      b)  Let f1 be [p|P], where p is the first element and P is the remaining list of
          f1;
      c)  Insert_tree([p|P]:s1, FP);
      d)  Update F':
            **For** all f∈F' and f⊆f1,
              i.  support(f)←support(f) - s1;
              ii. **if** (support(f)=0) **then** F'←F'-{f};
      e)  F' ←Sort(F');
4.  **Return** FP;
**End**

**Outspread**(FP)
**Begin**
1.  TempD ← ∅
2.  **if** (FP=null) **then return** TempD
    **else**
      (a) search the tree by in-depth order to find a leaf node ln:s, where ln is its item
          and s is its count;
      (b) t←all items in the path from the root FP to the leaf node ln;
      (c) **For** i=1 to s, TempD ← TempD ∪ { t};//TempD can include duplicates for t
      (d) Update FP:
          **For** each node n in the path from the root FP to the leaf node ln,
            i.  n.count←n.count - s;
            ii. **if** (n.count=0) **then** delete n from the tree FP;
      (e) Outspread(FP);
**End**
```

The input of the algorithm is a set of items *I*, minimum support threshold "σ"and frequent closed itemsets collection *F* with the support *S*. The output is a set of databases *DBs*. Each element of the *DBs* is a transaction database that agrees with *F*.

In the main procedure "*Gen_DB()*", we use *FP* to represent the tree obtained from the frequent closed itemsets collection *F* by calling the sub-procedure "*Gen_FPtree()*". We use *TempD* to represent the result of outputting *FP* by calling the sub-procedure "*Outspread()*". Notice that *FP* and *TempD* include only the items occurring in *F*. $I\overline{F}$ represents infrequent items included in *I* but not occurring in *F*. *NewTempD* is used to record the new generated database based on *TempD*.

In the sub-procedure "*Gen_FPtree()*", the function *Sort(F)* sort the itemsets in *F* by the number of items and support in descending order. Moreover, the items in each itemset are sorted in 1-itemset's support descending order. The function *Insert_tree([p|P]:s1, FP)*is performed as follows: If *FP* has a child *N* such that *N.item-name=p.item-name*, then increment *N*'s count by *s1*; else create a new node *N*, and let its count be *s1*, its parent link be linked to *FP*. If *P* is nonempty, call *Insert_tree(P,N)* recursively.

We use *F'* to store the sorted frequent itemsets so far by the number of items and support in descending order. First, an itemset *f1:s1* in the forefront of *F'* "climbs" the FP-tree. Then, the supports of all frequent itemsets in *F'* that are subset of *f1* subtract *s1* and *F'* is updated. The two steps repeat until all supports of the frequent itemsets in *F'* are equal to "0", and *F'* is equal to $\emptyset$. The sort routine of *F'* insures that each time the longest itemset with highest support is submitted first to "climb" the FP-tree. That is, the longer itemsets with higher supports are always satisfied prior to the shorter itemsets with lower supports during the FP-tree generation. This heuristic idea leads that once the longest itemset with highest support in the forefront of *F'* "climbs" the tree, the remaining tasks decrease sharply because more supports will probably be subtracted and more itemsets will probably be wiped off in the updating process of *F'*.

In the sub-procedure "*Outspread()*", the itemsets on each path of the FP-tree "come down" from the tree and form transactions of *TempD* one by one until the tree is equal to null. The result of this sub-procedure *TempD* can be seen as the status of an "Ordered Frequent Items" transaction database in [6] deleting infrequent items in each transaction (like the database in the rightmost column of the table in Fig. 1).

Lines 4-7 of the procedure "*Gen_DB()*" generate a set of databases *DBs* by scattering some infrequent items (elements of $I\overline{F}$) into *TempD,* just be sure that each infrequent item can only be scattered less than "σ"(minimum support threshold) transactions of *TempD*. Concretely speaking, suppose the number of infrequent items equals to *n* ($|I\overline{F}|=n$), $I\overline{F}=\{item_1, ..., item_i, ..., item_n\}$, $|TempD|=m$, then $DBs=\{TempD\} \cup NewTempDSet_1 \cup \ldots \cup NewTempDSet_i \cup \ldots \cup NewTempDSet_n$, where $NewTempDSet_1$ is a set of all the new generated databases by scattering $item_1$ into *TempD*, and $NewTempDSet_i$ is a set of all the new generated databases by scattering $item_i$ into all the previous generated databases in $\{TempD\} \cup \ldots \cup NewTempDSet_{i-1}$.

4.3 Example

Let's illustrate our algorithm with an example: Given *I={A, B, C, D, E}*, minimum support threshold "$\sigma=1$", and frequent closed itemsets collection $F=\{EABC_1, EAB_2, DBA_1, DB_2, C_6, BC_4, BCA_2, AC_4, A_6, AB_4, B_7\}$ (the subscripts represent supports) discovered from the transaction database in Fig. 1 of section 3.

① Generate FP-tree

We first sort itemsets in F by the number of items and support in descending order. We get $F'=\{EABC_1, BCA_2, EAB_2, DBA_1, AB_4, BC_4, AC_4, DB_2, B_7, A_6, C_6\}$. Then the items in each itemset are sorted in 1-itemset's support descending order. Now $F'=\{BACE_1, BAC_2, BAE_2, BAD_1, BA_4, BC_4, AC_4, BD_2, B_7, A_6, C_6\}$.

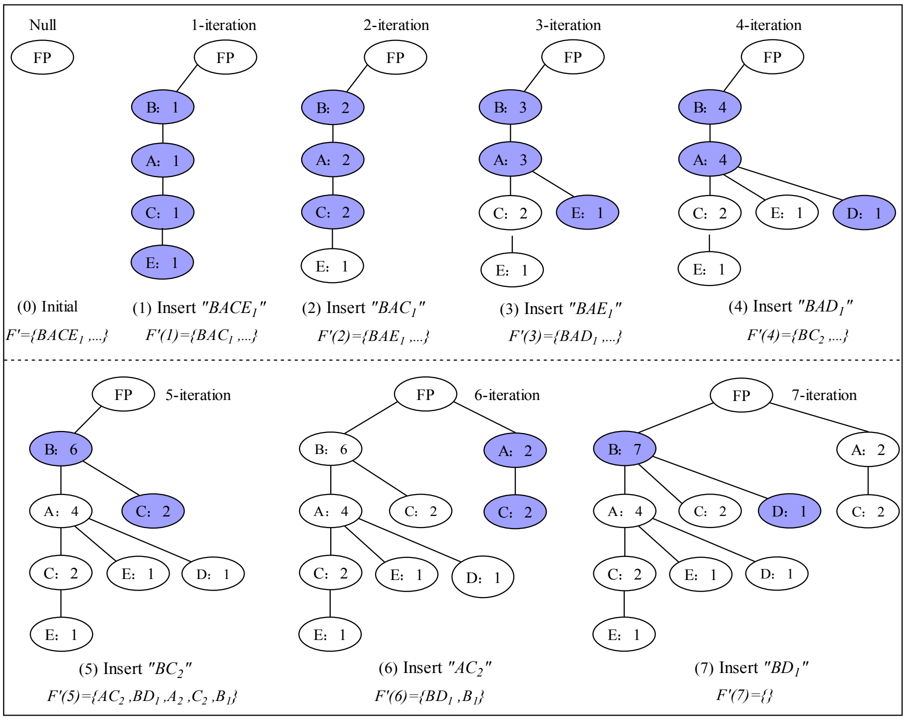


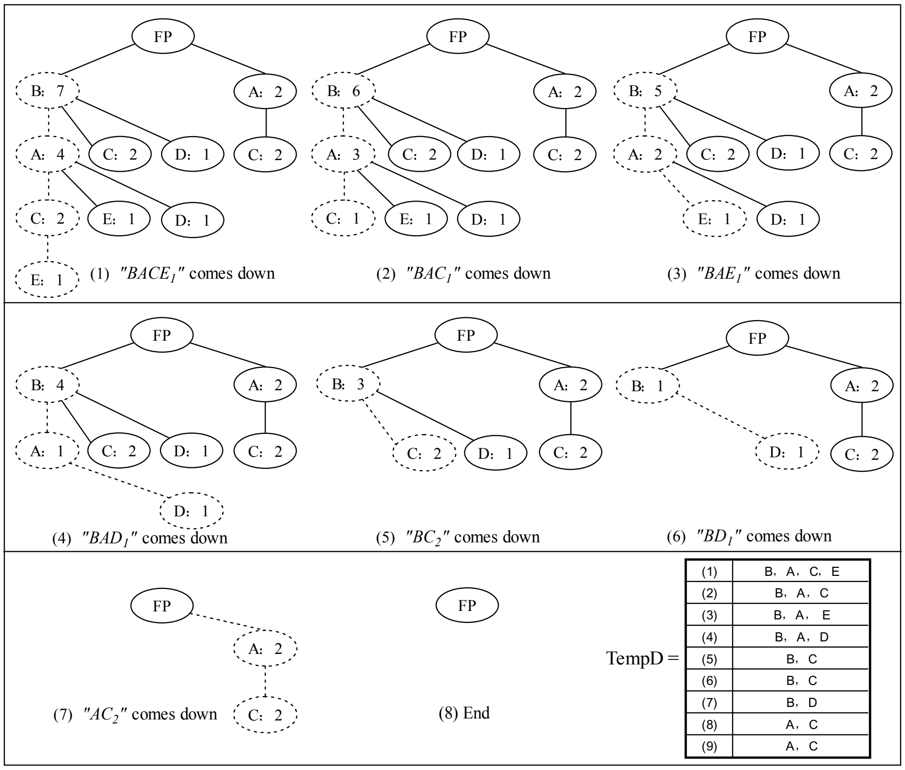
Fig. 3. The change of the FP-tree during the FP-tree generation

During the first iteration, the first itemset "$BACE_1$" is selected and inserted to the tree rooted as "*FP*". We get the tree like Fig.3-(1). Then *F'* is updated by subtracting "1" from the supports of all subsets of "*BACE*" occurring in *F'*. Itemsets with support equal to "0" are wiped off during updating. Now we get $F'(1)= F'-\{BACE, BAC, BAE, BA, BC, AC, B, A, C\}_1=\{BAC_1, BAE_1, BAD_1, BA_3, BC_3, AC_3, BD_2, B_6, A_5, C_5\}$. The itemsets in *F'(1)* has already been sorted by the number of items and support in descending order, so the result of function *Sort(F')* performing on *F'(1)* is still *F'(1)*. We can see that after the first iteration, *F'* becomes *F'(1)* whose itemsets and related supports decrease much, which means the remaining itemsets and their related supports needed to satisfy in the followed iteration decrease much. This dramatically reduces the cost effects on the FP-tree generation.

During the second iteration, "BAC_1" is inserted to the tree and we get $F'(2)=\{BAE_1, BAD_1, BA_2, BC_2, AC_2, BD_2, B_5, A_4, C_4\}$. Then "$BAE_1$", "$BAD_1$", "$BC_2$", "$AC_2$" and "$BD_1$" are inserted to the tree (or we say "climb" the tree) one after the other and we get the following update sequence of F' after each iteration: ⟦ $F'(3)=\{BAD_1, BC_2, AC_2, BD_2, BA_1, B_4, C_4, A_3\}$; $F'(4)=\{BC_2, AC_2, BD_1, C_4, B_3, A_2\}$; $F'(5)=\{AC_2, BD_1, A_2, C_2, B_1\}$; $F'(6)=\{BD_1, B_1\}$; $F'(7)=\varnothing$⟧ . The "Gen_FPtree" process terminates after seven iterations when $F'=\varnothing$. Fig. 3 shows the change of the FP-tree during the whole process of the FP-tree generation.

② Generate temporary transaction database TempD by outspreading FP-tree

Fig. 4 shows the change of the FP-tree during the whole process of the *TempD* generation. First, "$BACE_1$" which is the leftmost branch of the original FP-tree "comes down" from the tree. At the same time, all the items in this branch form into the first transaction of *TempD* : *TempD(1)=(B, A, C, E)* . After deleting the "$BACE_1$" branch from the tree, the "*FP*" tree changes into the form shown as Fig.4-(2). By calling the *Outspread(FP)* recursively, we perform the similar operations on the remaining six "*FP*" trees (Fig.4-(2) to Fig.4-(7)) in turn. "BAC_1", "BAE_1", "BAD_1",



(1)	B, A, C, E
(2)	B, A, C
(3)	B, A, E
(4)	B, A, D
(5)	B, C
(6)	B, C
(7)	B, D
(8)	A, C
(9)	A, C

Fig. 4. The change of the FP-tree during the *TempD* generation

"BC_2", "BD_1" and "AC_2" comes down from the tree one by one and we get the remaining eight transactions of *TempD*: *TempD(2)=(B, A, C)*; *TempD(3)=(B, A, E)*; *TempD(4)=(B, A, D)*; *TempD(5)=(B, C)*; *TempD(6)=(B, C)*; *TempD(7)=(B, D)*; *TempD(8)=(A, C)*; *TempD(9)=(A, C)*. Notice that when "BC_2" and "AC_2" come down, our algorithm generates two same transactions respectively. The whole algorithm ends when *FP=null* (Fig.4-(8)) and we get a temporary transaction database *TempD* shown as the table in Fig. 4.

③ Generate a set of databases DBs by scattering infrequent items into TempD
In last process ②, we have generated a transaction database *TempD* from the FP-tree generated in ①. In fact, *TempD*, which involves only frequent items, keeps all information about frequent itemsets and constitutes skeleton of the compatible databases we are to find. By scattering infrequent items into *TempD*, we can get more than one database, exactly a set of databases satisfying the given constraints.

In our example, the set of infrequent items $I\overline{F}=\varnothing$, as all the items in the set of *I={A, B, C, D, E}* occur in the set of frequent itemsets $F=\{EABC_1,EAB_2,DBA_1,DB_2,C_6,BC_4,BCA_2,AC_4,A_6,AB_4,B_7\}$. So according to lines 4-7 of the procedure "*Gen_DB()*" in our whole algorithm, no *NewTempD* is generated and the eventual *DBs={TempD}*. This means we find only one transaction database *TempD* satisfying *F* in this example. Interestingly, *TempD* (see Fig.4) happens to be the transaction database shown in Fig.1 of section 3, without regard to the order of transactions and the items order in each transaction. Another interesting thing is the FP-tree generated in this example by our inverse mining algorithm (see Fig.3-(7)) happens to be the tree shown in Fig.1 of section 3 generated from the transaction database in Fig. 1 by the FP-growth algorithm in [6], without regard to the order of children of each node. What do the interesting results indicate? At least we can get the following three valuable hints from the interesting results.

First, it validates the correctness of the result, as the input frequent sets *F* is discovered from the database in Fig. 1. So from *TempD* we must be able to discover exactly the same *F*, and *TempD* really satisfies *F*. Second, it indicates the feasibility and effectiveness of our method, as we really find a database satisfying *F* only based on the inputs (I, F,σ) and our algorithm, without knowing any other things about the original database. Third, it induces us to think: It is what factors that lead to the interesting results? How many compatible databases can be found by the proposed algorithm in usual cases? These questions will be probed in the next section 5.

5 Analysis

In this section, we analyze the correctness, efficiency of our algorithm. Then we focus on discussing the number of compatible databases that our algorithm can generate.

(1) Correctness
The correctness of our algorithm can be ensured by the three steps during our algorithm performing. The first step "Generate FP-tree" insures the generated FP-tree is compatible with the given frequent sets constraints, because all the given frequent sets "climb" the FP-tree and FP-tree can store the complete information in relevance

to frequent itemsets mining. The second step "Generate *TempD* by outspreading FP-tree" ensures the generated *TempD* is also compatible with the given frequent sets constraints, because all frequent sets "come down" from the FP-tree and from *TempD* we can construct a same FP-tree. The third step "Generate a set of databases *DBs* by scattering infrequent items into *TempD*" guarantees all the frequent sets and their supports related information keeps down exactly, no changes happen on any frequent sets' supports, and no new frequent sets are brought. So that all the databases in *DBs* preserve the complete and exact information of the given frequent sets constraints and are correct compatible databases we are to find.

(2) Effectiveness

The effectiveness of our algorithm lies in the two facts. One fact is our algorithm is a zero trace back algorithm with no rollback operations, since during constructing a compatible FP-tree process each itemset "climbs" the FP-tree following the prescribed order. And the remaining two transformations "from FP-tree to *TempD*" and "from *TempD* to the set of compatible databases" are natural and direct, with no rollback too. The other fact is, with the longest itemset with highest support "climbing" the FP-tree first during each iteration, our algorithm provides a good heuristic search strategy to rapidly find a compatible FP-tree.

Suppose the number of the given frequent closed sets in collection *F* is *k* and the number of transactions generated in *TempD* is *m*, i.e. $|F|=k$, $|TempD|=m$. Then the FP-tree construction can be accomplished in $O(klogk+(k-1)log(k-1)+...+1)$ time, in which *klogk* represents the time to sort the *k* frequent closed sets in *F* in the frequent sets length and support descending order. The number of elements in *F* decrease one each time the first frequent set climbs the FP-tree. The time consumed in *TempD* generation is determined by the number of branches in the FP-tree and approximates to $O(m)$. Hence the first two processes in our algorithm can both be accomplished in polynomial time. The most time-consuming process in our algorithm may be the third process to generate a set of compatible databases *DBs*. This is because our algorithm may generate an exponential number of compatible databases (see the number of compatible databases analysis in part (3) of this section). But it does not show our algorithm is inefficient. On the contrary, it shows the effectiveness of our algorithm because we can generate so many compatible databases. In fact, in our algorithm the generation of new databases is very easy and quick just scattering a new infrequent item into all the previous databases in prescribed principle. It may be time-consuming only because there are so many answers to be output.

All in all, with no trace back and with the good search strategy, our algorithm can work very effectively generating lots of compatible databases.

(3) The number of compatible databases

Fig. 5 illustrates mapping relation among compatible database space, compatible FP-tree space and given frequent closed sets collection, which helps to probe the number of compatible databases that our algorithm can output. In Fig. 5, *FCS* is a frequent closed sets collection discovered from one of the databases in DBs_i undergoing $TempD_i$ and $FP\text{-}tree_i$ by FP-growth method in [6]. DBs_j is the output set of compatible databases generated from *FCS* undergoing $FP\text{-}tree_j$ and $TempD_j$ by our algorithm. All

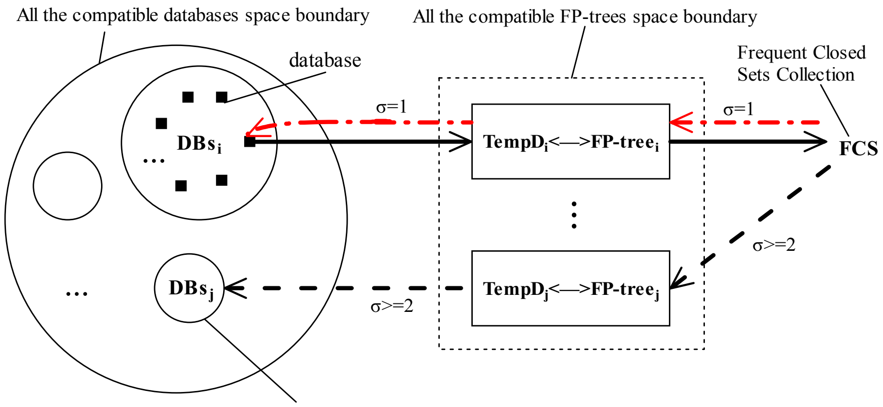

All the compatible databases space boundary
All the compatible FP-trees space boundary
database
Frequent Closed Sets Collection
σ=1
σ=1
DBs_i
TempD_i<—>FP-tree_i
FCS
σ>=2
DBs_j
TempD_j<—>FP-tree_j
σ>=2

Fig. 5. Mapping relation among database space, FP-tree space and FCS

the databases in DBs_i map into the same $FP\text{-}tree_i$ and have the same number of transactions as $TempD_i$, so do the DBs_j, $FP\text{-}tree_j$ and $TempD_j$. The figure shows that what our algorithm outputs is only a small part (a class having the same number of transactions and corresponding to the same FP-tree) of the whole compatible database space. Then how many databases our algorithm can output?

The number of compatible databases that our algorithm can output (indicated as $|DBs|$) is related to the three parameters: (1) the number of transactions in *TempD*, i.e. $|TempD|$; (2) the number of infrequent items in $\overline{IF}$, i.e. $|\overline{IF}|$; (3) the minimum support threshold"σ". Suppose $|TempD| = m$, $|\overline{IF}| = n$ and $f(n)$ represents the number of generated databases after the *n*-th infrequent item in $\overline{IF}$ has been scattered into all of the previous generated databases fully, we have the recurrence equation: $f(n+1)= f(n)+ f(n)(C_m^1 + C_m^2 + \cdots + C_m^{\sigma-1})$ $(n\geq 0,\ 2\leq\sigma\leq m,\ m\geq 1,\sigma \in N)$ (C_m^1 means the number of selecting one transaction from *m* transactions of a generated database that has not included the $(n+1)$-th infrequent item); and $f(0)=1$ which means when there is no infrequent items in $\overline{IF}$, there is only one compatible database (*TempD*) our algorithm finds. By solving the recurrence equation, we get $|DBs| = f(n) = (1 + C_m^1 + C_m^2 + \cdots + C_m^{\sigma-1})^n$ $(n\geq 0,\ 2\leq\sigma\leq m,\ m\geq 1,\sigma \in N)$. When$\sigma=1$, $|DBs| = 1$; and when$\sigma<<m$, $|DBs|$ is in direct proportion to $(\sigma C_m^{\sigma})^n$. In practice we can limit the number of compatible databases to be generated when $|DBs|$ is astronomical or when we are trying to find fixed number of compatible databases.

Notice that our other examples show usually(when$\sigma\geq 2$) $FP\text{-}tree_j$ is different from $FP\text{-}tree_i$, DBs_i and DBs_j are disjoint, and the database set that our algorithm outputs does not include the original database. However, when$\sigma=1$, $FP\text{-}tree_j$ happens to be the same with $FP\text{-}tree_i$, leading $TempD_j$ is just the same with $TempD_i$, and the only compatible database we are to find is just the original database because there exists no infrequent items can be scattered under "$\sigma=1$". This explains the two interesting facts in the example in section 4: The *TempD* in Fig.4 happens to be the original database in Fig.1; and the FP-tree in Fig.3-(7) happens to be the same FP-tree in Fig.1. The two dashed lines with arrowheads in Fig.5 illustrate the different execution paths of our algorithm in usual case ($\sigma>=2$) and special case ($\sigma=1$).

6 Conclusions

We have presented a feasible and efficient algorithm for the NP-complete problem of inverse frequent set mining. The algorithm can effectively generate a set of databases that *exactly* agree with the given frequent closed itemsets and their supports discovered from a real database. Compared with previous "generation-and-test" methods, our method is a zero trace back algorithm, without rollback operations during the databases' generation, which saves huge computational costs. Furthermore, our algorithm provides a good heuristic search strategy to rapidly find a FP-tree satisfying the given frequent sets constraints, leading to rapidly finding the compatible databases. More importantly, our algorithm can find a *set* of compatible databases (usually a lot of databases) instead of finding only *one* compatible database in previous methods. We also probe the number of databases found by our algorithm.

This study is just our first step towards solving this inverse mining problem. More work will be done in the near future, such as refinement of the algorithm, and empirical experiments on real databases. However, for this NP-complete inverse mining problem, our study has shown that there do exist reasonably efficient search strategies and solutions to find *some* (at least one, not all, but usually a lot of) data sets compatible with a given data set. This study can be used to deal with privacy preserving data sharing, in which data owners will have a choice in releasing different versions of the original data for different sharing (benchmark, mining, etc.).

References

1. Mielikainen, T.: On Inverse Frequent Set Mining. In: IEEE ICDM Workshop on Privacy Preserving Data Mining, IEEE Computer Society (2003) 18–23
2. Wu, X., Wu, Y., Wang, Y., Li, Y.: Privacy-Aware Market Basket Data Set Generation: A Feasible Approach for Inverse Frequent Set Mining. In: Proc. 5th SIAM International Conference on Data Mining (2005)
3. Wang, Y., Wu, X.: Approximate Inverse Frequent Itemset Mining: Privacy, Complexity, and Approximation. In: Proc. 5th International Conference on Data Mining (2005) 482–489
4. Chen, X., Orlowska, M.: A Further Study on Inverse Frequent Set Mining. In: Proc. 1st International Conference on Advanced Data Mining and Applications (ADMA), Lecture Notes in Computer Science, Vol. 3584. Springer-Verlag (2005) 753–760
5. Calders, T.: Computational Complexity of Itemset Frequency Satisfiability. In: Proc. 23rd ACM PODS 04, ACM Press (2004) 143–154
6. Han, J., Pei, J., Yin, Y.: Mining Frequent Patterns without Candidate Generation. In: Proc. of the ACM SIGMOD International Conference on Management of Database (2000) 1–12