A Compact FP-tree for Fast Frequent Pattern Retrieval

Tri Thanh Nguyen
Vietnam National University, Hanoi (VNUH)
University of Engineering and Technology (UET)
ntthanh@vnu.edu.vn

Abstract

Frequent patterns are useful in many data mining problems including query suggestion. Frequent patterns can be mined through frequent pattern tree (FP-tree) data structure which is used to store the compact (or compressed) representation of a transaction database (Han, et al, 2000). In this paper, we propose an algorithm to *compress* frequent pattern set into a smaller one, and store the set in a modified version of FP-tree (called compact FP-tree) as an inverted indexing of patterns for later quick retrieval (for query suggestion). With the compact FP-tree, we can also restore the original frequent pattern set. Our experiment results show that our compact FP-tree has a very good compression ratio, especially on sparse dataset which is the nature of query log.

1 Introduction

Frequent pattern mining is an important task because its results can be used in a wide range of mining tasks, such as association rule, correlation, causality, sequential pattern, etc. as reviewed by Han (2000). In some mining tasks (e.g. association rule, correlation, or causality), frequent patterns are used as intermediate data for computing final results, so there is no need to access these patterns again. However, in some other tasks, such as *query suggestion* (or *query recommendation*) (Li, 2008), when a user enter a keyword, the search engine will recommend the potential phrases (or patterns) the user may want to use, in order to: (a) save time for users, (b) make the convenience of use, and even (c) guide the user in case he/she is not sure about what to search for. In such tasks, we need to frequently search for frequent patterns containing a certain keyword (or phrase), hence, we want to have a method that supports quick retrieval of patterns. In information retrieval, one of the contemporary methods for fast retrieval of documents containing a certain word (or phrase) is *inverted indexing* (Manning, et al, 2008), which manages a mapping from a keyword to a set of documents containing it. Thus, given a keyword, we will quickly have the list of related documents.

We found that FP-tree can be used as an inverted indexing which can provide us a list of patterns containing a certain item. Thus, we propose to modify FP-tree to store the frequent patterns for later fast retrieval. The difference between our FP-tree and the original one is:

- The original FP-tree stores the compact version of a transaction database, and an algorithm (called *FP-growth*) is used to find out the frequent patterns;
- Our FP-tree stores the frequent patterns for quick access, so each path in the tree is already a pattern.

Since the number of frequent patterns generated from a transaction database can be very large, we propose an algorithm to compress them into a much smaller (compact) set and store in FP-tree data structure. We also propose to modify related algorithms to make FP-tree compatible with frequent patterns instead of transaction data. We call the tree of compact pattern set *compact FP-tree*. With the compact FP-tree, it is easily to restore the original frequent pattern set. The results of the experiments on benchmark transaction database show that our compact FP-tree has very good compression ratio.

Our paper is organized as follows: Section 2 introduces about FP-tree, and summarizes some typical literature; Section 3 introduces our compact FP-tree and the algorithms for compressing frequent patterns as well as restoring the original pattern set; Experiment and evaluation is discussed in Section 4 while conclusion and future work are provided in Section 5.

2 Background and related work

In this section, to make the paper self-containing, we will introduce frequent pattern mining problem, some detail of FP-tree, and some typical studies.

2.1 Frequent pattern mining problem

Let $I = \{a_1, a_2, ..., a_m\}$ be the **set of items** (which can be the list of goods in a supermarket), and a **transaction database** $DB=\langle T_1, T_2, ..., T_n \rangle$, where each T_i ($1 \leq i \leq n$) is a **transaction** which contains a subset of items in *I*. An example of a transaction T_i is the list of goods in a shopping basket. Define the **support count** (or the absolute support, i.e. the absolute occurrence frequency/count) of a pattern *A* (*A* is a subset of *I*) is the number of transactions in *DB* that contains *A*. Note that, in other studies, relative support is used (i.e. the percentage of a pattern in DB). It is easily to convert from absolute to relative support, and vice versa using the formula:

relative_support=absolute_support /|*DB*|

In this paper, the term *support count* is used to: (a) refer to absolute occurrence frequency, (b) distinguish from relative support, and (c) make it easier to follow the examples. Pattern *A* is called a *frequent pattern* (or *frequent itemset* – the term used in some other literature) if *A*' support count is greater than or equal a predefined minimum support count (*minsupcount*) ξ. The task of finding the *complete set of frequent patterns* in a DB with a *minsupcount* ξ is called frequent pattern mining problem. This task is claimed to be time-consuming, hence, there are many algorithms having proposed to solve the task.

One of the algorithms, that has high attention of study is frequent pattern tree (or FP-tree for short) proposed by (Han, 2000). One of the advantages of FP-tree over previous algorithms is the reduction of the number of database scans. In FP-tree construction phrase, it needs only two scans over the database. The definition of FP-tree data structure and its related algorithms are given in next subsection.

2.2 FP-tree introduction

Han, et al, proposed the FP-tree data structure that can store the *complete set of frequent patterns* using only *two scans* over the *DB*. The biggest contribution to speed up the frequent pattern mining task is reduction of the number of scans over the DB down to only 2, since the speed of reading data in the secondary storage is slow. FP-tree is a tree structure as defined below:

1. It consists of one root labeled as “null", a set of **item prefix subtrees** as the children of the root, and a **frequent-item header table**.
2. Each node in the **item prefix subtree** consists of 4 fields: *item-name*, (support) *count*, *parent-link*, and *node-link*, where item-name registers which item this node represents, count registers the number of transactions represented by the portion of the path reaching this node, and node-link links to the next node in the FP-tree carrying the same item-name, or null if there is none, the parent-link links to the parent node[1].
3. Each entry in the **frequent-item header table** consists of three fields: (1) *item-name,* (2) the (support) *count*[2], and *head of node-link* which points to the first node in the FP-tree carrying the *item-name*.

An example of a FP-tree is given in Figure 1, now we will study how to construct the FP-tree in this figure. With the *minsupcount* $\xi=3$, based on the *DB* listed in Table 1. This table shows a simple database of transactions of a supermarket, where the first column is the transaction identification, each row in the second column is the list of items that were bought by a customer. The FP-tree construction is described briefly as follows:

TID	Items Bought	(Ordered) Frequent Items
100	*f, a, c, d, g, i, m, p*	*f, c, a, m, p*
200	*a, b, c, f, l, m, o*	*f, c, a, b, m*
300	*b, f, h, j, o*	*f, b*
400	*b, c, k, s, p*	*c, b, p*
500	*a, f, c, e, l, p, m, n*	*f, c, a, m, p*
600	*f, c, g, s*	*f, c*

Table 1. Transactions in DB and their frequent items

FP-tree construction starts with the first scan over the *DB* to find the list of frequent items (i.e. frequent itemsets with the cardinality of 1 having the support no less than ξ). The result of the scan over the DB in Table 1 is the list *h* of $\langle$(*f*: 5), (*c*: 5), (*a*: 3), (*b*: 3), (*m*: 3), (*p*: 3)$\rangle$,

[1] Though this field is not clearly mentioned in Han’s paper, it is important in forming the tree, so we list it here for the sake of completeness.

[2] This field is not clearly mentioned in Han’s paper neither, we list it here for later use.

where the number after the colon ":" is the support count of items, and the *h* is sorted in support count descending order denoted as *L*. *h* is used to build the *frequent-item header table* (or header table for short), where each entry in the table consists of the *item-name* and a pointer (called *head of node-links*) to the first (appeared) node having the same item-name in the FP-tree as depicted in Figure 1. The second scan will get the list of frequent items of each transaction, sort it according to *L*, and insert it into the FP-tree. To make it easier to observe, this list of each transaction is showed in the third column of Table 1. The tree construction algorithm is listed in Algorithm 1.

Algorithm 1: FP-tree_construction

```
  Input: A transaction database DB and a minsupcount $\xi$.
  Output: The frequent pattern tree F
1. Scan the DB to get the list L of frequent items,
     and sort it in support descending order.
   Create a FP-tree F by:
       Create the header table, and set all the head-of-node-links to null.
       Create the root node T of the tree having the item-name of null.
       Set the parent-link and node-link of T to null.
2. Scan the DB again
   For each transaction Tran in DB do
       Get the list of frequent items.
       Sort it according to the order L.
       Let this list be [p|P], where p is the first item
         and P is the remaining items.
       Call insert_tree([p|P], T).
```

where the insert_tree(.) procedure is defined in Algorithm 2.

Algorithm 2: insert_tree

```
  Input: the ordered list [p|P] of frequent items, and
         a node T of a FP-tree.
  Output: the updated FP-tree.
if T has a child node N such that the item name of
      N and p is the same then
    Increase the count of N by 1
else
    Create a new node N.
    Set the item_name of N to p.item_name.
    Set the count of N to 1.
    Link the parent-link of N to T.
    Set the node-link of N to null.
    if the head-of-node-links of the item h in the
      header table having the same name as p is
      null then
        Set head-of-node-links of h to p;
    else
        Traverse through the head-of-node-links
          of h to the end of the list, and link the
          node-link of the end-node to p.
if P is not empty then
    Let P=[$p_1$|$P_1$]
    Call insert_tree([$p_1$|$P_1$], N).
```

The key idea why the algorithm needs to sort the items in a transaction in the order *L* is: the more frequent an item is, the more common it is in transactions, hence the transactions will share the items as a prefix. And such transactions will be "compressed" in the prefix, and make the tree compact.

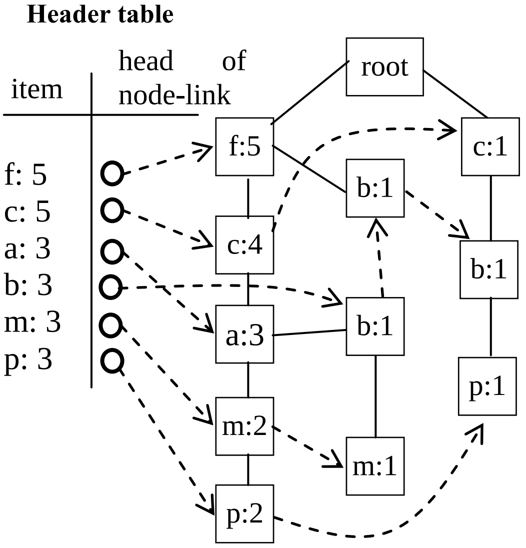

Figure 1. FPTree corresponding to the DB in Table 1

Han (2000) proved that FP-tree has both compactness (i.e. it has a compact representation) and completeness (i.e. it stores the complete information of the database in relevant to frequent pattern mining).

After having built the FP-tree, frequent patterns can be extracted from the item at the bottom of the header table (i.e. the item having the smallest count) (Han, 2000). For example, from the FP-tree in Fig. 1, we start with item *p* in the header table, and have one frequent pattern (*p*: 3) and two paths: 〈(*f*: 5), (*c*: 5), (*a*: 3), (*b*: 3), (*m*: 2), (*p*: 2)〉 and 〈(*c*: 1), (*b*: 1), (*p*: 1)〉. The items (called *p*'s prefix) that appear *together* with *p* in the two paths are 〈(*f*: 2), (*c*: 2), (*a*: 2), (*b*: 2), (*m*: 2) 〉 and 〈(*c*: 1), (*b*: 1)〉, correspondingly. The prefix set of *p* {〈(*f*: 2), (*c*: 2), (*a*: 2), (*b*: 2), (*m*: 2)〉, 〈(*c*: 1), (*b*: 1)〉} is called *p*'s conditional pattern base (i.e. the sub-pattern base under the condition of *p*'s existence). This set is used as a set of transactions to construct another FP-tree called *p*'s conditional FP-tree with respect to minimum support count: ξ. The only frequent item in *p*'s conditional pattern base is (*c*: 3), hence we can produce one frequent pattern (*pc*:3). Table 2 lists the conditional pattern bases, conditional FP-trees, and frequent patterns of other items of the FP-tree in Fig. 1.

Item	Conditional pattern base	Cond' FP-tree	Frequent pattern
p	{(*f*:2, *c*:2, *a*:2, *m*:2), (*c*:1, b:1)}	{(*c*:3)}|*p*	{(*p*:3), (*pc*:3)}
m	{(*f*: 2, *c*:2, *a*:2), (*f*:1, *c*:1, *a*:1, b:1)}	{(*f*:3, *c*:3, *a*:3)}|*m*	{(*m*:3), (*macf*:3, (*mac*:3), (*maf*:3), (*ma*:3), (*mcf*:3), (*mc*:3), (*mf*:3)}
b	{(*f*:1, *c*:1, *a*:1, *m*:1), (*f*:1, *b*:1), (*c*:1)}	{}	{(*b*:3)}
a	{(*f*:3, *c*:3)}	{(*f*:3, *c*:3)}|*a*	{(*a*:3), (*acf*:3), (*ac*:3), (*af*:3)}
c	{(*f*:4)}	{(*f*:4)}|*c*	{(*c*:5), (*cf*:4)}
f	{}	{}	{*f*:5}

Table 2. Frequent pattern generation

When a single path in one item's conditional FP-tree having the length greater than 1 as in case of *m* and *a* (see Table 2), a set of combinations of items (i.e. the non-empty subsets) in the path is generated as frequent patterns with the same support as the considered item's. For example, the item *a* has the conditional FP-tree consisting of only one path (*fc*:3), and the combinations of the items in this path are {(*fc*:3), (*c*:3), (*f*:3)}. These combinations are used as a prefix of *a* to produce frequent patterns: (*acf*:3), (*ac*:3), (*af*:3). The algorithm to extract frequent patterns from a FP-tree is given in Algorithm 3.

Algorithm 3: FP_growth

```
Input: FP-tree constructed based on Algorithm 1,
       using DB and a minsupcount $\xi$, and a pattern
       prefix $\alpha$
Output: The complete set of frequent patterns.
Procedure FP-growth (Tree, $\alpha$)
if Tree contains a single path P then
    for each combination (denoted as $\beta$) of the
      nodes in the path P do
        Generate pattern $\beta \cup \alpha$ with support
          =minimum support of nodes in $\beta$;
else for each $a_i$ in the header of Tree do
    Generate pattern $\beta = a_i \cup \alpha$ with support =
      $a_i$.support;
    Construct $\beta$'s conditional pattern base and then
      $\beta$'s conditional FP-tree $Tree_\beta$ with respect to
      minsupcount $\xi$;
    If $Tree_\beta \neq \emptyset$ then call FP-growth ($Tree_\beta$, $\beta$)
```

To extract patterns from a FP-tree *F*, we call FP-growth(*F*, null).

FP-tree data structure has attracted many studies in both application and modification (improvement) aspects. Li, et al (2008) parallelized FP-tree and pattern generation to detect relation among tags and and webpages for query recommendation. Kumar and Rukmani, (2010) used both FP-tree and Apriori for web usage mining problem. Xu, et al (2011) mined associated factors about emotional disease based on FP-tree growing algorithm. Patro, et al, (2012) proposed to use Huffman coding to compress FP-tree. Yen, et al (2012) proposed Search Space Reduced (SSR) algorithm to speed up the pattern extraction from FP-tree. Bernecker, et al (2010) added probability to FP-tree to mine uncertain databases. Concretely, the authors proposed the first probabilistic FP-Growth (ProFP-Growth) and associated probabilistic FP-Tree (ProFP-Tree), which we use to mine all probabilistic frequent patterns in uncertain transaction databases without candidate generation. Shrivastava, et al (2010) mined multiple level association rules based on FP-tree and co-occurrence frequent item tree (CFI). Lin, et al (2010) added constraints on FP-tree for multi-constraint pattern discovery.

2.3 Inverted indexing

In document representation, a document can be represented as a vector of which each element is a feature (e.g. a binary value indicating whether a word appears in the document or not). Vector representation of document is suitable for computation, such as classification. However, it is not good for searching (i.e. given a keyword, find all the documents containing it). Inverted indexing is a data structure that stores a mapping from content (e.g. keywords) to documents. All the distinct keywords in the universal document set are used to form a dictionary. Each keyword in the dictionary is attached a list of document identifiers (doc_id) called *posting list*. For example, given a set of 3 documents:

{ T[0]= "What is inverted indexing?"
T[1]= "Inverted indexing is a data structure"
T[2]= "Inverted indexing is used in search engine"}

The distinct keywords in the document set is {"a", "data", "engine", "in", "indexing", "inverted", "is", "search", "structure", "used", "what"}, and the inverted indexing of the document set is[3]:

"a"	=> {1}
"data"	=> {1}
"engine"	=> {2}
"in"	=> {2}
"indexing"	=> {0, 1, 2}
"inverted"	=> {0, 1, 2}
"is"	=> {0, 1, 2}
"search"	=> {2}
"structure"	=> {1}
"used"	=> {2}
"what"	=> {0}

With this data structure, given a keyword we will quickly have the list of documents containing it. When we want to find documents that contain some keywords all together, we simply find out the document list of each keyword and calculate the intersection. Inverted indexing is usually employed in search engines (Manning, et al, 2008).

3 Compact FP-tree

When the frequent pattern set generated from a transaction database is large, while we need to access the frequent patterns regularly, is it possible to: (a) compress the set into a smaller one, and (b) facilitate the access to frequent patterns? In this section, we will address the two questions through: (a) frequent pattern compression method, and (b) compact FP-tree.

3.1 Frequent pattern set compression

Given a set of frequent patterns $FP=\{fp_1{:}s_1, fp_2{:}s_2, ..., fp_n{:}s_n\}$, where fp_i is a frequent pattern and s_i is its support. The frequent pattern set compression is defined as:

Find another frequent pattern set *FP'* such that $|FP'| < |FP|$, and it is possible to restore the *FP* from *FP'*. Formally, we need to find to procedure compress(.) and uncompress(.) such that: if *FP'*=compress(*FP*), then $|FP'| < |FP|$ and uncompress(*FP'*)=*FP*.

Our compression idea is based on the fact that if there are two frequent patterns $fp_i{:}s_i$ and $fp_j{:}s_j$ (where s_i is the support of the pattern) such that s_i=s_j and $fp_i \subset fp_j$ then we can remove the pattern fp_i.

For example, from the set of frequent patterns generated from the item *m* in Table 2 {(*m*:3), (*a*:3), (*macf*:3), (*mac*:3), (*maf*:3), (*ma*:3), (*mcf*:3), (*mc*:3), (*mf*:3)}, since the pattern *mc* is a subset of pattern *mcf* with the same support of 3, we can remove *mc* from the set. Similarly, *mf* is a subset of *mcf*, we remove *mf*. Repeating this process exhaustively, the above set is reduced to the set {(*macf*:3)}.

A heuristic method to reduce the search space to find out the frequent patterns of which one can be a subset of another is to sort the patterns according to the support, and then the frequent patterns. After sorting, patterns having the same support and prefix will be grouped into a segment. The search performed per segment will be faster, since it has a smaller search space.

To uncompress the pattern set, we reverse the compress process. For a pattern $fp_i{:}s_i$ whose cardinality is greater than 1, we generate a set of all the combinations (i.e. the non-empty subsets) of its items: $\{fp_{i1,}, fp_{i2}, ..., fp_{in}\}$, each combination fp_{ik} is assigned the support s_i to form a frequent pattern. Added the pattern $fp_i{:}s_i$ to the generated set we produce the output set $\{fp_{i1}{:}\ s_{i,}, fp_{i2}{:}\ s_i, ..., fp_{in}{:}\ s_i, fp_i{:}s_i\}$.

The above output set sometimes is not original set due to the fact that some combination in the original set can have a bigger support count. For example, the uncompressed set of the pattern (*macf*:3) is {(*macf*:3), (*mac*:3), (*maf*:3), (*ma*:3), (*mcf*:3), (*mc*:3), (*mf*:3), (*m*:3), (*a*:3), (*c*:3), (*f*:3)}. However, in the compressed set we have the frequent pattern (*f*:5), so the pattern (*f*:3) is redundant and need to remove. We call this phenomenon *redundant pattern* for later reference.

Based on the above discussion, we first introduce the compress(.), uncompress(.) procedures, and prove the correctness later.

[3] We ignore additional techniques while building the dictionary as well as the inverted indexing for simplicity. Interested readers can refer to (Manning, et al, 2008)

The compress(.) procedure is defined as Algorithm 4.

Algorithm 4: compress(.)

```
    Input: a set of frequent patterns $FP$
    Output: a compact set of frequent patterns
(1) while exist two patterns $fp_i$:$s_i$ and $fp_j$:$s_j$ in $FP$ such
        that $s_i$=$s_j$ and $fp_i$ is a subset of $fp_j$ do
(2)     $FP = FP \setminus \{fp_i : s_i\}$
(3) return $FP$
```

And the uncompress(.) procedure is defined as Algorithm 5.

Algorithm 5: uncompress(.)

```
    Input: a set of compact frequent patterns $FP$
    Output: the original set of frequent patterns
(1) $FP'$={}
(2) while exist a pattern $fp_i$:$s_i$ in $FP$ such that $s_i$ >1 do
(3)     Let $F_i$ be the set of all frequent patterns that
        are combinations of items in $fp_i$ with the
        support $s_i$
(4)     $FP = FP \setminus \{fp_i : s_i\}$
(5)     $FP' = FP' \cup F_i \cup \{fp_i : s_i\}$
(6) $FP' = FP' \cup FP$
(7) while exist two pattern $fp_i$:$s_i$ and $fp_j$:$s_j$ in $FP'$ such
        that $s_i$ < $s_j$ and $fp_i$ = $fp_j$ do $FP' = FP' \setminus \{fp_i : s_i\}$
(8) return $FP'$
```

Lemma 3.1: Given a set of frequent patterns generated from a transaction database, after having compressed by the above compress(.) procedure, the uncompress(.) procedure will produce the original frequent pattern set.

Proof: According to *Apriori* property (Agrawal, et al 1993): if a pattern *p* is frequent, then all of its subsets are frequent, thus, if there is a pattern fp_j:s_j is compressed by compress(.), then all the combinations of the items in fp_j {fp_{i1}: $s_{i,}$, fp_{i2}: s_i, ..., fp_{in}: $s_{i,}$} are frequent. The only phenomenon is the *redundant pattern* discussed earlier. However, this phenomenon can be solved by simply removing the pattern with lower support count (in line 7 of Algorithm 5). □

Applying the compress(.) procedure to the frequent pattern set generated from the FP-tree in Fig. 1, we have the compressed set {(*f*:5), (*c*:5), (*cf*:4), (*macf*:3)}. We can see that the compressed set is much smaller than the original one. Our compression algorithm produces the *closed and maximal item set* as (Grahne and Zhu, 2003). Grahne and Zhu proposed an algorithm based on FP-tree called FPClose to mine closed and maximal item set. The compressed set will be stored in the compact FP-tree, which *again* helps to reduce the storage as discussed in Section 3.2.

3.2 Compact FP-tree

From the definition of FP-tree data structure, it has a header table containing all the frequent items. Each item has a pointer that links all its occurrences in the patterns of the tree. This header table is similar to inverted indexing mechanism discussed in Section 2.3. The only difference is: each item in header table maintains a list of patterns (not a list of *pattern_id*s as in inverted indexing). Therefore, FP-tree data structure has the same characteristics of inverted indexing, i.e. it facilitates the fast retrieval of patterns containing a certain item, and we propose to use FP-tree to store the compressed frequent pattern set. Since, we can not use the original FP-tree as well as its related algorithms, we define another version of FP-tree called *compact FP-tree* (with the differences from the original of FP-tree definition are in bold):

1. It consists of one root labeled as “null", a set of item prefix subtrees as the children of the root, and a frequent-item header table.
2. Each node in the item prefix subtree consists of 4 fields: *item-name*, ***support***[4], *parent-link*, and *node-link*, where *item-name* registers which item this node represents, ***support* is used to calculate the support of the pattern containing this item**, and node-link links to the next node in the compact FP-tree carrying the same item-name, or null if there is none, the parent-link links to the parent node.
3. Each entry in the frequent-item header table consists of three fields: (1) *item-name*, (2) the ***support***, and *head of node-link* which points to the first node in the compact FP-tree carrying the *item-name*.

Lemma 3.2: The order of frequent items in FP-tree storing the compressed frequent pattern set is the same as that of the original FP-tree (i.e. the FP-tree corresponding to the uncompressed pattern set).

Proof: the items in header table of the FP-tree constructed from a transaction database are themselves frequent, hence, their order remains the same if we copy them to another FP-tree. □

[4] This can be relative or absolute support, in this paper we use absolute support for consistency

The algorithm to construct a compact FP-tree is defined as:

Algorithm 6: FP-tree_construction

```
Input: A compressed pattern set $S$.
Output: The compact FP-tree $F$
Generate the list $L$ of frequent items from $S$, and
    sort L in descending order of support.
Create a FP-tree $F$ by:
    Create the header table based on $L$, and set
        all the head-of-node-links to null.
    Create the root node $T$ of the tree having the
        item-name of null.
    Set the parent-link and node-link of $T$ to null.
Remove all frequent items from $S$.
for each pattern $fp$:s in $S$ do
    Sort the items in $fp$ according to the order $L$.
    Let this list be $[p|P]$, where $p$ is the first item
        and $P$ is the remaining items.
    Call insert_pattern($[p|P]$, $T,s$).
```

where insert_pattern(.) is defined as:

Algorithm 7: insert_pattern

```
Input: the ordered list $[p|P]$ of frequent items, a
    node $T$ of a compact FP-tree, and the
    support $s$.
Output: the updated compact FP-tree.
if $T$ has a child node $N$ such that the item name of
        $N$ and $p$ is the same then
    if p.support>N. support then N. support =p.
        support
else
    Create a new node $N$.
    Set the item_name of $N$ to $p$.item_name.
    $N$. support =$p$. support
    Link the parent-link of $N$ to $T$.
    Set the node-link of $N$ to null.
    if the head-of-node-links of the item $h$ in the
            header table having the same name as $p$ is
            null then
        Set head-of-node-links of $h$ to $p$;
    else
        Traverse through the head-of-node-links
            of $h$ to the end of the list, and link the
            node-link of the end-node to $p$.
if $P$ is not empty then
    Let $P=[p_1|P_1]$
    Call insert_pattern($[p_1|P_1]$, $N$, $s$)
```

The compressed pattern set of the FP-tree in Fig. 1 as discussed in Section 3.1 has the compact FP-tree as Fig. 2.

3.3 Searching in compact FP-tree

Given an item, we follow the node-link pointer from the header table to get all the patterns containing it. If we want to search for patterns containing more than one item (e.g. this case is frequently occurs in web search, where users can search for a phrase instead of a keyword), then we search for patterns containing the lowest support item, then filter out the patterns containing all the given items. For example, if we want to search for patterns containing {a, f, m}, we just search for patterns containing m, then filter out the patterns containing both a and f.

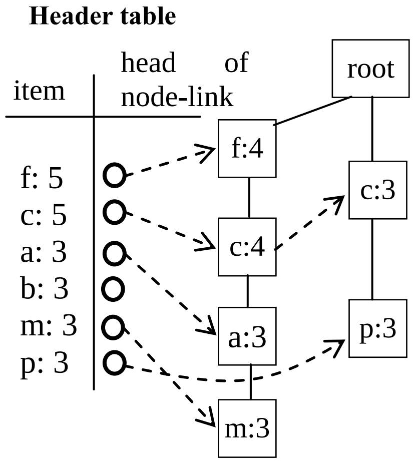

Figure 2: A compact FP-tree

However, there is a limitation of current compact FP-tree, i.e. if we search for patterns containing an item that is not a leaf node, then we can not extract the whole pattern (from the root to the leaf). For example, in the compact FP-tree in Fig. 2, if we search the tree based on item a, then we can only extract a pattern (acf:3), m is absent in the pattern. Fortunately, we can overcome this limitation by adding pointers from a parent node to its children nodes, so we can traverse in both directions from a leaf to the root and vice versa.

In some situations, we want to get the patterns with higher support first, i.e. in query suggestion, we want to suggest users with more frequent keywords first. To support this situation, we simply reorder the node-link of nodes so that the highest support pattern is pointed by the head-of-node-link (i.e. the pointer of the header table).

3.4 Original frequent pattern set recovery

In case we want to restore the original frequent pattern set (the uncompressed one) we can easily do through the compact FP-tree. For each path in the compact FP-tree, we generate the combinations of its items with the lowest support of the item in the combination. After generation, we also face *redundant pattern* problem, which can be solved by a removal step. The recovery algorithm is described in Algorithm 8.

Algorithm Pattern_extraction

```
Input: a compact FP-tree Tree
Output: The complete set S of frequent patterns.
S={all items in the header table}
for each path in Tree do
    Generate the set P of all combinations with the
     length>1, each of which has the support of the
     lowest support item.
    $S = S \cup P$
while exist two pattern $fp_i$:$s_i$ and $fp_j$:$s_j$ in S such
     that $s_i < s_j$ and $fp_i = fp_j$ do $S = S \setminus \{ fp_i : s_i \}$
```

4 Experiments

We used the open source FP-growth package[5] developed by (Borgelt, 2005). The patterns having the same support are generated in next to each other as a group as listed in Table 2, where the number in the parenthesis is the (relative) support (in percent) of the pattern.

Frequent Pattern
m (60.0)
m a (60.0)
m a c (60.0)
m a c f (60.0)
m a f (60.0)
m c (60.0)
m c f (60.0)
m f (60.0)

Table 2. Frequent patterns generated from FP-growth

By exploiting this characteristic, we simply find the longest pattern in the group which will be the compressed pattern of the group. Hence, we wrote another algorithm as Algorithm 8 which has the complexity of $O(N)$.

Algorithm 8: Pattern_compression

```
Input: a list P of frequent patterns
Output: The set S of compressed frequent patterns
     having the length>1, and a list L of frequent
     items for building header table
L={} ;
S={}
iterate through the list P
    if the current pattern p has the length=1 then
        L=L + p
    else this is the starting of a group with the same
        support, so find the longest pattern p in this
        group
    $S = S \cup p$
```

We used two types of transaction databases that are published as benchmark datasets[6]: sparse (i.e. T10I4D100 and T20I6D100) and dense (i.e. mushroom and C20D10). For evaluation, we compare the size (in term of the total of nodes) of the compact FP-tree and its original FP-tree called *compression ratio* (in %) as follows:

$$ratio = \frac{\sum nodes\_in\_compact\_FP-tree}{\sum nodes\_in\_original\_FP-tree} * 100\%$$

The smaller the ratio is, the better compression is.

With sparse databases, we had to use very small relative minimum support thresholds ranging from 0.1% to 1%. The compression ratio of sparse databases is given in Fig. 3 where we can see that the compact FP-tree is drastically reduced. With the relative support of 0.1%, the size of compact FP-tree is reduced to 8% and 2% on T20I6D100 and T10I4D100, correspondingly. With the relative support of 1%, the compression ratio is extremely good: 0.05% on both transaction databases.

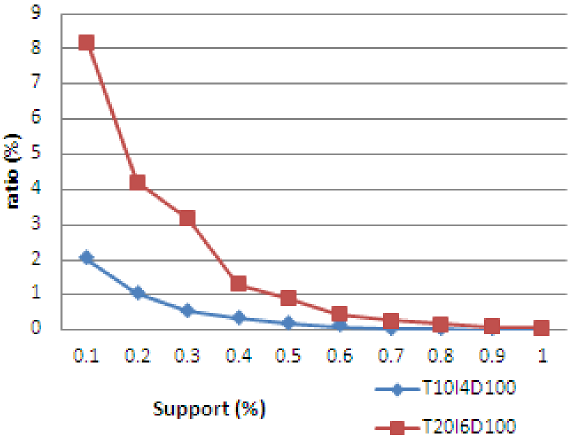

Figure 3: The compression ratio on sparse databases

With dense databases, we used big relative minimum support thresholds ranging from 10% to 40%. The results of the experiments are given in Fig. 4. Compact FP-tree does not have much

[5] http://www.borgelt.net/fpgrowth.html

[6] http://keia.i3s.unice.fr/?Jeux_de_Données___Benchmark_Datasets

power with dense databases, since the compression ratio is not good.

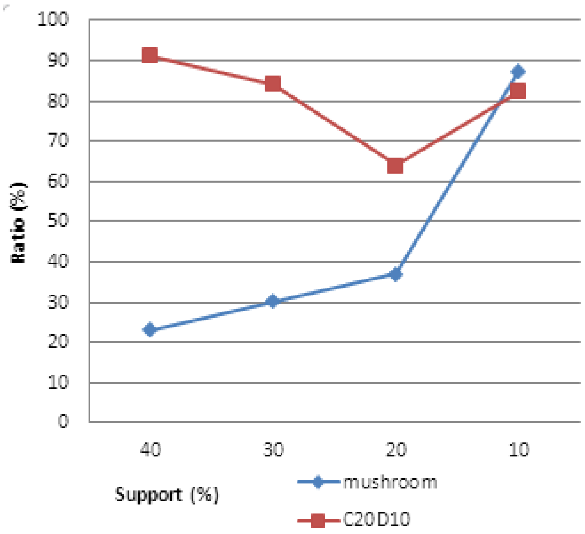

Figure 4: The compression ratio on dense databases

From experimental results, we can see that, on sparse databases, the compact FP-tree has very good compression ratio, whereas, it does not expose its power in dense databases. There are many domains, where the data is sparse, such as text document collections of which the number of dimension is so big that the data is very sparse. Another example of domain is query logs, where the queries are diverse, especially on multi-language search engines (e.g. Google). Thus, the application of compact FP-tree is very promising.

5 Conclusion and future direction

In this paper, we proposed to compress the frequent pattern set mined from a transaction database to a compact set. The compact set is useful in application where the longest pattern is usually used, such as query suggestion (i.e. we prefer to suggest the longest frequent pattern containing a certain word/phrase to users). The practical compression algorithm is very effective with the low complexity of $O(N)$. In order to speed up the retrieval of frequent patterns, we proposed to modify the FP-tree into compact FP-tree which stores the compressed pattern set as an inverted indexing data structure.

Our experimental results on benchmark databases show that the proposed method is very useful in sparse databases.

In the future direction, we will study the method to construct the compact FP-tree directly form its FP-tree.

References

A. P. Xu, et al, 2011. *Mining Associated Factors about Emotional Disease Bases on FP-Tree Growing Algorithm*, International Journal of Engineering and Manufacturing, vol. 4, pp. 25-31.

B. S. Kumar and K. V. Rukmani, 2010. *Implementation of Web Usage Mining Using APRIORI and FP-Growth Algorithms*, Int. J. of Advanced Networking and Applications, Vol 01, Issue: 06, pp 400-404.

C. Borgelt, 2005. *An Implementation of the FP-growth Algorithm*. Workshop Open Source Data Mining Software (OSDM'05, Chicago, IL), pp. 1-5, ACM Press.

C. D. Manning, et al, 2008. *Introduction to Information Retrieval*, Cambridge University Press.

G. Grahne and J. Zhu, 2003. *Efficiently Using Prefix-trees in Mining Frequent Itemsets*, Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, vol. 17, no. 10, pp. 1347-1362.

H. Li, et al, 2008. *PFP: Parallel FP-Growth for Query Recommendation*, Proceedings of the 2008 ACM conference on Recommender systems, pp 107-114.

J. Han, et al, 2000. *Mining Frequent Patterns without Candidate Generation*, Proc. Conf. on the Management of Data (SIGMOD’00, Dallas, TX.

M. Singh, et al, 2010. *FP-Tree Improve Efficiency & Increase Scalability by Applying Parallel Projected*, Binary Journal of Data Mining & Networking, Vol 1, No 1. pp 14-16.

R. Agrawal, et al, 1993. *Mining association rules between sets of items in large databases*, Proceedings of the 1993 ACM SIGMOD international conference on Management of data pp. 207-216.

S. J. Yen, et al, 2009. *The Studies of Mining Frequent Patterns Based on Frequent Pattern Tree*, Lecture Notes in Computer Science Volume 5476, pp. 232-241.

S. J. Yen, et al, 2012. *A Search Space Reduced Algorithm for Mining Frequent Patterns*, Journal Of Information Science And Engineering, Vol 28, pp 177-191.

S. N. Patro, et al, 2012. *Construction of FP Tree using Huffman Coding*, International Journal of Computer Science Issues, Vol 9, Issue 3, pp 446-469.

T. Bernecker, et al, 2010. *Probabilistic Frequent Pattern Growth for Itemset Mining in Uncertain* Databases, Cornell University Technical Report.

V. K. Shrivastava, et al, 2010. *FP-tree and COFI Based Approach for Mining of Multiple Level Association* Rules *in Large Databases*, International Journal of Computer Science and Information Security, Vol. 7 No. 2, pp. 273-279.

W. Y. Lin, et al, 2010. *MCFPTree: An FP-tree-based algorithm for multi-constraint patterns discovery*, Int. J. of Business Intelligence and Data Mining, Vol. 5, No 3, pp. 231 – 246.