Arabian Journal for Science and Engineering (2022) 47:9641–9657
https://doi.org/10.1007/s13369-021-06298-9

RESEARCH ARTICLE-COMPUTER ENGINEERING AND COMPUTER SCIENCE

SR-Mine: Adaptive Transaction Compression Method for Frequent Itemsets Mining

O. Jamsheela[1] · G. Raju[2]

Received: 24 November 2019 / Accepted: 11 October 2021 / Published online: 16 November 2021

Abstract
Extraction of frequent itemsets is a key step in association rule mining. Frequent Pattern (FP) mining from a very large dataset is still a challenging research problem. The basic frequent itemset algorithms are Apriori and FP-growth. FP-growth uses Frequent Pattern Tree (FP-tree) to store the database information in a compressed form. A large number of research papers have been proposed as an improvement of the basic frequent itemset mining algorithms. Several researchers have proposed modifications to existing data structures as well as new data structures to improve the mining process. A new method, Size Reduced Mining (SR-Mine), is proposed to speed up the FP-tree creation. The proposed work is implemented with the basic FP-growth algorithm and with the other two recent algorithms based on FP-tree. The three modified algorithms have been tested with standard datasets and compared with the original algorithms. The proposed method can be applied with the frequent itemset mining algorithms which consider each transaction one by one to construct a data structure for mining. The experimental results show that the proposed method can improve the performance of the mining.

Keywords Compressed transactions · FP-tree · Size reduced mining · SR-Mine · FPclose · PrePost$^+$

1 Introduction

Discovering association rules is considered an important aspect of data mining. Frequent itemset mining is an important and time-consuming step in the task of mining association rules. It was first introduced by Agrawal et al. [1] in the context of transaction databases. The problem of frequent itemset mining can be defined as follows. A transaction database is a database containing a bag of transactions, and each transaction is associated with a unique transaction id. Let $D = \{t_1, t_2, \ldots, t_N\}$ be a transaction database and $I = \{i_1, i_2, \ldots i_n, \}$ be the set of items appearing in D, where $t_i (i \in [1 \ldots N])$ is a transaction and $t_i \subseteq I$. Each subset of I is called an itemset. If an itemset contains k items, then the itemset is called a k-itemset. The support of itemset l in database D is defined as the percentage of transactions in D containing l, that is, $\text{support}_D(l) = \mid \{t \mid t \in D$ and $l \subseteq t\} \mid / \mid D \mid$. If $\text{support}_D(l) \geq$ min _sup, where min_sup is a user-specified minimum support threshold, then l is called a frequent itemset in D. Given a transaction database and a minimum support threshold, the task of frequent itemset mining is to find all frequent itemsets in the transaction database [2].

This algorithm demands an efficient data structure to store frequent itemsets for further processing in addition to the problem of a large number of candidates. The Apriori algorithm has been proposed by Agrawal et al. [1] for mining frequent itemsets. Apriori uses a candidate generation method, such that the frequent k-itemsets in one iteration can be used to construct candidate $(k + 1)$-itemsets for the next iteration. Apriori terminates its process when no new candidate itemsets can be generated. Various methods have been proposed as the improvement of Apriori. Han et al. [3] proposed the frequent pattern growth (FP-growth) method which is one of the most efficient and feasible approaches in this area. It mines the frequent itemsets by using a frequent pattern tree (FP-tree) and avoids costly candidate generation. FP-tree is a data structure that is used for storing the frequent data of a transaction database. It achieves much better performance and efficiency than Apriori-like algorithms. FP-tree stores the transactions in the database in a compressed form. Another data structure called header table is also used with FP-tree

✉ O. Jamsheela
ojamshi@gmail.com

1 EMEA College of Arts and Science, Kondotty, India

2 Christ College, School of Engineering and Technology, Christ University, Bengaluru, India

to traverse the tree and find frequent itemsets quickly. The header table stores frequent 1-itemsets in decreasing order of their frequencies. Each item in the header table points to the first occurrence of the corresponding node in the FP-tree. All nodes with similar items in the FP-tree are connected by a link. After completing the FP-tree construction, the next step of the FP-growth algorithm is to mine frequent patterns from the FP-tree. It starts the mining from the least frequent item to the most frequent item. It traverses from the leaf nodes of the FP-tree to the root. Paths with the same prefix item in the FP-tree are used to generate the conditional pattern base, which is a sub-database. Construct a new FP-tree with each conditional pattern base which is called conditional FP-tree. Using the conditional FP-tree, the algorithm can generate frequent itemsets with the same prefix. According to the experimental results, FP-growth is faster than the Apriori algorithm and several other methods of mining frequent itemsets.

Recently many improvements have been introduced and all are faster than the basic FP-growth such as [4–8]. In this paper, a new size-reduced method (SR-Mine) is proposed. The proposed method is a novel pre-processing technique to reduce the tree construction time. The duplicate transactions are compressed to reduce the size of the database. It can be applied with FP-tree-based algorithms, i.e., any frequent itemset mining algorithms which consider each transaction one by one to construct the data structure. The proposed pre-processing method is combined with three FP-growth-based algorithms, viz. FP-growth [3], FPclose [9] and PrePost$^+$ [7]. These algorithms are already proved as efficient mining algorithms.

The new method is a pre-processing method, and it is attached to the existing algorithms before FP-tree construction. The remaining codes of the existing algorithms also have to be modified to change the support of each transaction. The three modified algorithms are named as SR-FPgrowth, SR-PrePost$^+$ and SR-FPclose. To evaluate the efficiency of the proposed method the new SR-algorithms have been tested with standard datasets and compared with the original FP-growth, PrePost$^+$ and FPclose algorithms.

The rest of this paper is organized as follows. In Sect. 2 the background and related work of frequent itemset mining are analyzed. The motivation and objectives are given in Sect. 3. In Sect. 4 a detailed description of the proposed SR-Mine is introduced. The Tree construction procedures are given in Sect. 5. Details of the algorithms are given in Sect. 6. The analysis of experimental results is shown in Sect. 7, and Sect. 8 concludes the paper.

2 Related Work

Specific patterns, with supports higher than or equal to a minimum support threshold, have been extracted by using the frequent pattern mining methods. Many mining algorithms have been introduced, but Apriori and FP-growth are still regarded as popular algorithms. Apriori is the oldest typical mining algorithm. It generates candidate patterns in advance and compares whether the candidates are frequent patterns by scanning the database. Apriori scans the database as much as the maximum length among frequent patterns. FP-growth [1] solved the above problem by introducing an efficient data structure called FP-tree, which prevents unwanted candidate pattern generation. An FP-tree consists of a tree for storing the transactions in the database in a compressed form and a header table containing item names, support counts and node links. The links are pointing to the corresponding first item in the tree. An FP-tree is composed of nodes, each of which includes an item name, a support count, a parent pointer, a child pointer and a node link. The node link is a pointer that connects all nodes with the same item name.

2.1 Improved FP-Trees

Various algorithms have been published as an improvement of the FP-growth algorithm. Gwangbum et al. [10] recommended an improved tree structure called Linear Prefix Tree (LP-Tree) to find frequent patterns. An LP tree is composed of array forms to minimize pointers between nodes. Tsay et al. [5] proposed a novel method, the Frequent Items Ultrametric trees (FIUT) for mining frequent itemsets. The FIUT-tree structure is used to enhance the efficiency in obtaining frequent itemsets. Another algorithm, i.e., FPClose, is proposed by Grahne and Zhu [9]. FPClose uses the FP-tree data structure in combination with the FP-array technique and also used various optimization methods. In the experimental results comparing our methods with existing algorithms, the results show that our methods are the fastest for many cases. The FPClose consumes more memory, but the authors state that the algorithm is still the fastest one even when the minimum support is low. Lin et al. [6] proposed an improved frequent pattern (IFP) growth method for mining frequent patterns. The authors of this paper pointed out that the proposed algorithm requires less memory and shows better performance in comparison with FP-tree-based algorithms and the authors also propose a new IFP-growth (Improved FP-growth) algorithm to improve the performance of FP-growth. IFP-growth employs an address-table structure to lower the complexity of searching in each node in an FP-tree. It also uses a hybrid FP-tree mining method to reduce the need for rebuilding conditional FP-trees. IFP-growth uses additional memory for holding the address table for each node. This results in a lack of memory to store those additional data. The address table contains the item name and pointer to its child. IFP growth does not reduce the size of trees since it still uses the original FP-tree-based structures.

Racz et al. [11] used another way to find frequent itemsets very easily without rebuilding conditional FP-trees. They used arrays to store the nodes. The authors replaced the recursive mining processes by building new tree structures, which avoids rebuilding each conditional pattern base. Deng et al. [12] propose n-lists and n-list tree (PrePost$^+$) to find frequent itemsets. The authors have used a pruning method to reduce the search space. In this method, additional information, such as pre-order and post-order which is the sequence number of the node when scanning the tree by pre-order and post-order traversal, respectively, has been stored on each node of the tree. The same authors proposed a node set, a more efficient data structure, for mining frequent itemsets (FIN) [13]. In PrePost$^+$ two properties have been used, but FIN requires only the pre-order (or post-order code) of each node. Many more algorithms have been suggested to find frequent itemsets. Borgelt et al. [14] suggested a new algorithm by stating that their algorithm is the simplest algorithm to find frequent itemsets by using a simple data structure. Tseng [15] presents an adaptive mechanism to choose and use a suitable data structure among two pattern list structures to mine frequent itemsets. The two methods are the Frequent Pattern List (FPL) for sparse databases and the Transaction Pattern List (TPL) for dense databases. The selection criteria depend on database densities, and they give a method to calculate the database density.

2.2 Transaction Compression

Few transaction compression approaches are suggested to reduce the size of the dataset. Ashrafi et al. presented a transaction compression method in 2003 [16]. The transactions are placed in different tables based on the first item. During the second iteration, the infrequent items are eliminated. Apriori algorithm is used with each table to generate frequent itemsets. A transaction merging method is suggested by Hung et al. [17]. The transactions are grouped into different classes. The transaction set of each class is merged into a single new transaction. Each item is represented with a frequency count. The authors used the Apriori method in the mining phase. Dai et al. [18] introduced a new transaction compression method in 2008. The authors state that the previous transaction compression approaches cannot decompress the data to its original form. A new method called Mining Merged Transactions with the Quantification Table (M2TQT) was proposed to solve these problems. M2TQT uses the relationship of transactions to merge related transactions and builds a quantification table to prune the candidate itemsets. Another interesting technique has been introduced by Nair et al. [19]. The transactions with one frequent itemset are set as null transactions by saying that they are not associated with any itemset in that particular transaction. They suggested removing all null transactions to reduce the size of the dataset.

The above-mentioned methods have introduced a variety of suggestions to compress and reduce the data size. The proposed method is another simple yet efficient method to reduce the size of the dataset. The method can be incorporated with most of the algorithms to increase efficiency by reducing the runtime of the algorithm.

3 Motivation and Challenges

Various data structures have been introduced to improve the frequent itemset mining process. The data structure, node set is introduced in PrePost [12]. The algorithm FIN [13] applied the set enumeration tree to speed up the mining. Arrays are used to represent the LP-tree in [10]. The frequent itemset mining algorithms, which construct a data structure to store the transactions of a database, require some specific pre-processing procedures before constructing the data structure. The procedures are as follows;

1. The first database scan to find frequent 1-itemsets.
2. During the second scan consider each transaction, then ;
   (a) Remove all infrequent items.
   (b) Sort the items in the transaction according to the support count.
   (c) Insert the transaction into the data structure.

The above steps have to be applied to each transaction separately. The whole data structure is constructed when the second scan finishes. The data structure creation is a time-consuming procedure. The three procedures in the second step mentioned above are applied to each transaction separately. If a set of transactions are treated as a single transaction and the processes are applied to it, the data structure construction time can be reduced. In this paper, similar transactions after removing the infrequent itemsets are represented as one transaction with a frequency count and applied the procedures.

3.1 Objectives

The main objective of this research is to develop a pre-processing step with the support of an efficient data structure to reduce the running time of the data mining task. The proposed method can be applied with any FP-tree-based data mining algorithm to improve the running time. It reduces the time of the FP-tree construction, and it also helps reduce the number of repeated transactions. To improve the overall mining process the similar transactions are compressed into a single transaction with frequency count. In data mining algorithms with FP-tree, the FP-tree is not constructed only once. The main and the full FP-tree is constructed only once, but the construction of the conditional FP-trees is recursively creating. Hence, the proposed pre-processing step will affect very efficiently during the mining process. The main aim of the proposed work is to make the data mining algorithms applicable without considering the size of the dataset. Nowadays, the need for data reduction is increased because of the ever-increasing data size. The objective of the proposed work is to incorporate the data mining algorithms in big data also.

3.2 Methodological Limitations

The literature review revealed that most of the related works are concentrated to improve the main data structure. The lack of robust methodologies used during the pre-processing steps is an important methodological gap in the data mining algorithms. Most of the transnational databases normally contain repeated transactions. After the filtering and sorting of items, the number of repeated transactions is increased substantially. Therefore, by applying transaction compression methods, the running time can be improved. The proposed pre-processing step is introduced to fill this methodological gap to improve the running time.

4 The Proposed Method: SR-Mine

In most of the recent frequent itemset mining algorithms, the procedures which are presented in Sect. 3 have been applied to the transaction database before constructing the major data structure such as trees, graphs, arrays, node sets. This paper introduces a new method to reduce the time of the second scan and also to reduce the time to construct the data structure. In the proposed method after removing the infrequent items, the transaction is stored in a HashMap. Similar transactions are represented as one single transaction with a frequency count, and the procedures are applied to it. The new method can be added with any algorithm which is going through the above-mentioned procedures. The proposed method is combined with three efficient algorithms, and the examples are illustrated with the SR-FPgrowth implementation.

5 The Tree Construction Procedure

A new approach for transaction pre-processing is presented here. In the first step, the proposed method scans the transaction database to find the frequent 1-itemsets. The frequent 1-itemsets are sorted in descending order of the frequency and stored in a header table. Scan the database for the second time to process each transaction one by one. During the second scan, take one transaction and remove the infrequent items.

The reduced transaction with frequent items has to be stored in a data structure for duplicate elimination. The data structure should be created with two fields. The first field is to store the transaction, and the second field is for storing the frequency (support) count of the transaction. Here a hash map is used to store the reduced transaction and the transaction count. The transaction is used as the key and the support count is used as the value. By setting the transactions as key the uniqueness is guaranteed. Another advantage of this method is that no extra key value is needed. Before inserting a transaction, search the HashMap to find whether the transaction is already there or not. If the transaction is already there in the hash map, increase the support count of the transaction; otherwise, insert the transaction with support count as one. Continue the process until all the transactions are inserted into the HashMap. Now the HashMap contains the compressed transactions with support count. The next step is the construction of the FP-tree. Take each transaction from the hash map and sort the items in the transaction according to the frequency descending order. Insert the sorted transaction to the FP-tree as described below.

The tree structure and tree-node structure are the same as FP-tree [3]. Each node contains the item name, support count, child-pointer, parent-pointer and link-pointer. To insert the first transaction, create a root with the label null and the first item of the transaction should be inserted as the child of the root with the support count of the transaction. The remaining items should be inserted as a branch of the child and set the support of each item with the support count of the transaction. The remaining transactions have to be inserted as follows. Take the transaction from the hash map with transaction support count and search among the children of the root for the first item of the transaction. If the item is there, increase the support count by adding the transaction count with the item support. The second item of the transaction should be searched among the children of the current node. If the item is found, repeat the above procedure.

If the item is not found among the children of the root, create a new node with the item and set the support of the item with the transaction count. Add the new node as a new child of the root. The remaining items are added as a branch from the new child and set the support of each item of the branch with the transaction support count.

The procedure is illustrated by using the transaction database shown in Table 1. The table contains 12 transactions. The items have been arranged in lexicographical order. The result of the first scan is shown in Table 2. Table 3 contains the frequent 1-itemsets with support count. The items are sorted in the descending order of the frequency. Table 4 contains the transactions after removing the infrequent items.

During the second scan take the first transaction “a b d g h” and remove the infrequent item h. The processed transaction “a b d g” should be inserted into the HashMap. In each

Table 1 The transaction database

TID	Transactions
1	a b d g h
2	c b d e
3	b d f
4	a b c d g
5	d f h
6	d h
7	a b d e g
8	b d e
9	a b d j g
10	a b d g l
11	a g l
12	a f g

Table 2 Items with frequency

	Items	Frequency
1	a	7
2	b	8
3	c	2
4	d	10
5	e	3
6	f	3
7	g	7
8	h	3
9	j	1
10	l	1

Table 3 Frequent 1-itemsets

	Items	Frequency
1	d	10
2	b	8
3	a	7
4	g	7

Table 4 Transactions after removing infrequent items

TID	Transactions
1	a b d g
2	b d
3	b d
4	a b d g
5	d
6	d
7	a b d g
8	b d
9	a b d g
10	a b d g
11	a g
12	a g

Table 5 Hashmap contents after inserting 3rd transaction

	Transaction	Frequency
1	a b d g	1
2	b d	2

Table 6 Hashmap contents after inserting all transactions

	Transactions	Frequency
1	a b d g	5
2	b d	3
3	d	2
4	a g	2

insertion search the hash map for the current transaction. If the transaction is already inserted in the HashMap, increase the support count of the transaction; otherwise, insert the transaction with support 1.

In the example the “a b d g h” is the first transaction and the HashMap is empty. Therefore it should be inserted with support as 1. The processed second transaction is “b d.” The transaction “b d” is not there in the hash map, so insert the transaction “b d” with count 1. The third transaction after removing the infrequent item is “b d.” Transaction “b d” already exists in the hash map, so increase the transaction count of “b d” by one. The contents of the hash map are displayed in Table 5 after inserting transaction 3. Continue the procedure with all the transactions. Table 6 shows the contents of the hash map after inserting all the transactions. The first and second transactions repeat 5 and 3 times, respectively, and the third and fourth transactions repeat 2 times. The original database contains 12 transactions, and the hash map contains only 4 transactions. The compression ratio is 1/4.

The items are arranged in lexicographical order as in the original database. The sorting will be done after compressing the transactions so that the time it takes for sorting each transaction is reduced drastically.

The next step is the construction of the FP-tree with the compressed transaction. The procedure is illustrated by using the same example. Only 4 transactions need to be sorted and inserted into the tree instead of 12 because the hash map contains only 4 transactions. Create a root with a null value. Take the first transaction “a b d g ” and sort according to the frequency count. The sorted transaction is “d b a g.” Insert the transaction into the tree. Item “d” will be the first child of root, and the remaining items will be formed as a branch from the child “d.” The support of each item should be set as 5 which is the transaction count of “a b d g” in the hash map. The FP-tree after inserting the first transaction is displayed in Fig. 1a. The remaining transactions are inserted as follows. The second transaction is “b d” with transaction count 3. Sort the transaction as “d b.” Search item “d” among the children

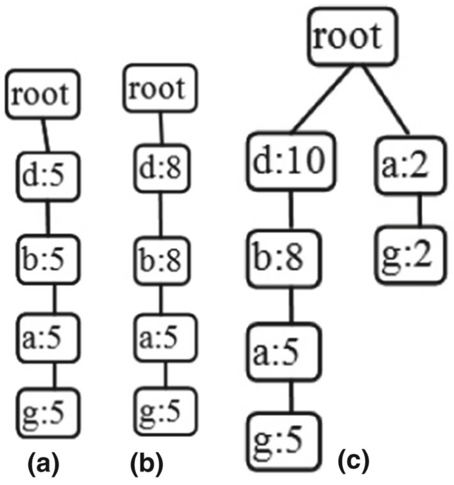
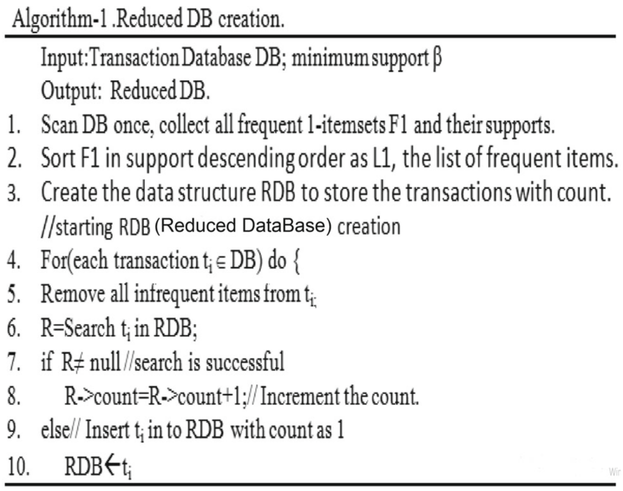


Fig. 1 **a** FP-tree after inserting transaction-1. **b** FP-tree after inserting transaction-2. **c** FP-tree after inserting all transactions



```
Algorithm-1 .Reduced DB creation.
    Input:Transaction Database DB; minimum support β
    Output: Reduced DB.
1.  Scan DB once, collect all frequent 1-itemsets F1 and their supports.
2.  Sort F1 in support descending order as L1, the list of frequent items.
3.  Create the data structure RDB to store the transactions with count.
    //starting RDB (Reduced DataBase) creation
4.  For(each transaction $t_i \in$ DB) do {
5.  Remove all infrequent items from $t_i$;
6.  R=Search $t_i$ in RDB;
7.  if R≠ null //search is successful
8.      R->count=R->count+1;// Increment the count.
9.  else// Insert $t_i$ in to RDB with count as 1
10.     RDB←$t_i$
```

Fig. 2 Algorithm-1 stores transactions in the database to the size reduced database (hash map)

of the root. item "d" is the first child of the root, so increase the support count by 3. The support of node "d" becomes 8(5+3). Search "b" among the children of "d." "b" is the only child of "d." Increase the support count of "b" with 3. Figure 1b is the tree after inserting transaction-2. Insert all the remaining transactions. Figure 1c shows the FP-tree after inserting all the transactions.

In the basic FP-tree construction and its variants, each transaction is processed separately to construct the tree during the second scan. If the dataset contains 100,000 transactions, the prepossessing should be done 100,000 times. If the average number of frequent items in a transaction is 25, in the worst-case scenario the sorting complexity will be $100{,}000 * (25^2)$.

The insertion process also should be done 2,500,000 (100,000 * 25) times. In the proposed method, if the average compression ratio is 1/4, the insertion will be reduced to 625,000.

In the proposed method the sorting and insertion need not be done for each and every transaction one by one. The transactions in the database have been arranged in some specific order. The order may be lexicographical if the items are strings or ascending or descending order if the items are numbers or in any other order. The original order is maintained while inserting the transactions into the hash map. The sorting is done with the compressed transactions before inserting them into the tree.

6 The Modified Algorithms

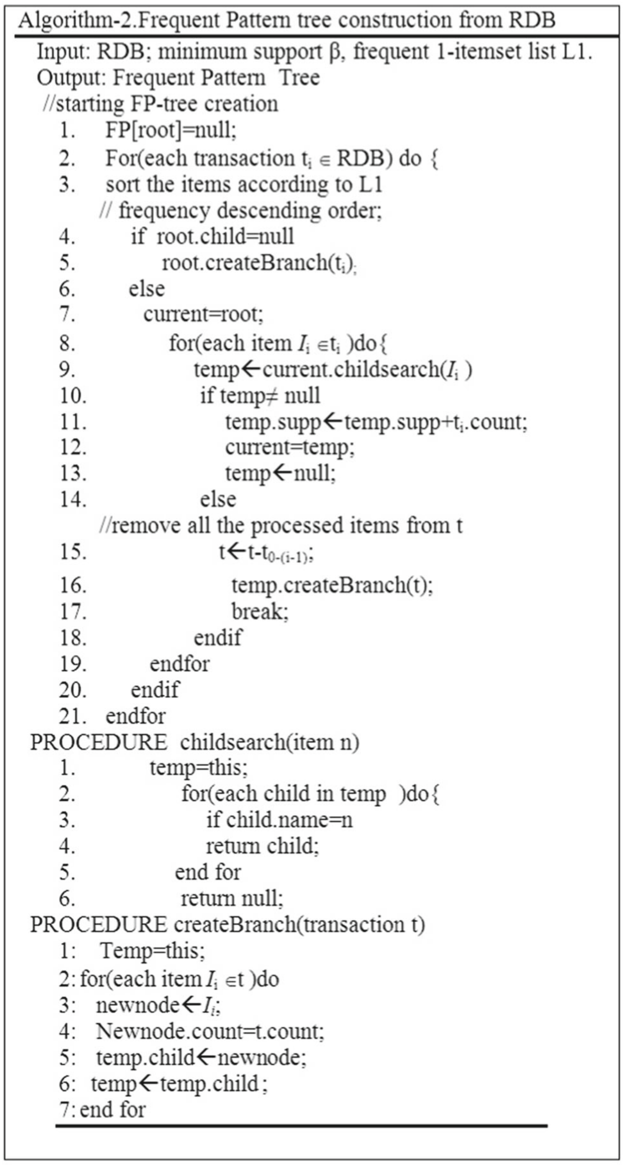
Three FP-tree-based algorithms are selected to attach the proposed pre-processing method. The SR-FPgrowth, SR-FPclose and SR-PrePost$^+$ use the prefix tree structure to store the transactions. The proposed method is a pre-processing method and is applied before the tree construction. The first algorithm "algorithm-1" in Fig. 2 is used to find frequent 1-itemsets and used to store the transactions to the hash map after removing the infrequent items. The transaction database DB and minimum support value β are the inputs to the algorithm-1. The output of the algorithm is the HashMap with compressed transactions.

The second algorithm "algorithm-2" in Fig. 3 is used to construct the tree with the contents in the hash map. The output of algorithm-2 is the prefix tree. The mining algorithm is not included because the selected three algorithms have been used different mining methods and in this implementation, the mining phase is not changed. Only two algorithms are included here because after creating the data structure(tree) the mining procedure is different for each algorithm.

The block diagram in Fig. 4 shows the procedure of the above- mentioned three algorithms. The block diagram in Fig. 5 represents the procedure after modifying the algorithms with the new method.

7 Experimental Evaluation and Performance Analysis

In this section, the performance of the proposed method is evaluated. The method is a pre-processing step. The proposed method can be applied with the frequent itemset mining algorithms which consider each transaction one by one to construct a data structure for mining. The performance of the proposed method is evaluated by comparing it with three algorithms. The three algorithms are FP-growth [3], PrePost$^+$ [7] and FPclose [9]. The same three algorithms are used to combine with the proposed method.



```
Algorithm-2.Frequent Pattern tree construction from RDB
  Input: RDB; minimum support β, frequent 1-itemset list L1.
  Output: Frequent Pattern  Tree
   //starting FP-tree creation
     1.    FP[root]=null;
     2.    For(each transaction $t_i \in$ RDB) do {
     3.    sort the items according to L1
           // frequency descending order;
     4.       if  root.child=null
     5.            root.createBranch($t_i$);
     6.       else
     7.          current=root;
     8.             for(each item $I_i \in t_i$ )do{
     9.                temp←current.childsearch($I_i$ )
     10.               if temp≠ null
     11.                  temp.supp←temp.supp+$t_i$.count;
     12.                  current=temp;
     13.                  temp←null;
     14.               else
           //remove all the processed items from t
     15.                  t←t-$t_{0-(i-1)}$;
     16.                  temp.createBranch(t);
     17.                  break;
     18.              endif
     19.         endfor
     20.      endif
     21.   endfor
 PROCEDURE  childsearch(item n)
     1.         temp=this;
     2.             for(each child in temp  )do{
     3.                if child.name=n
     4.                return child;
     5.           end for
     6.            return null;
 PROCEDURE createBranch(transaction t)
     1:   Temp=this;
     2: for(each item $I_i \in$t )do
     3:   newnode←$I_i$;
     4:   Newnode.count=t.count;
     5:   temp.child←newnode;
     6:  temp←temp.child;
     7: end for
```

Fig. 3 Algorithm-2 used to create the FP-tree with the transactions in the hash-map

Implementations of the original algorithms have been taken from the SPMF website (http://www.philippe-fournier-viger.com/spmf/index.php?link=license.php) [20]. FP-growth algorithm is chosen as the baseline algorithm. PrePost$^+$ has been proven as the best algorithm among all node-based methods. FPclose is included to compare the new method with a closed frequent itemset mining algorithm. An itemset I is closed in a transaction dataset TD if there exists no proper super itemset K such that K has the same support count as I in TD. An itemset I is a closed frequent itemset in dataset TD if I is both closed and frequent in TD.

In the experimental evaluation, 10 datasets have been used. These datasets are publicly available datasets downloaded from the FIMI repository (http://fimi.ua.ac.be) [21]. Retail and T10I4D100K are sparse datasets. The Mushroom, Connect, Pumsb, Accidents, etc., are dense datasets. The details of the datasets are shown in Table 7. The datasets Accidentsdouble and Pumsbdouble are the double-sized datasets of original Accidents and Pumsb data. Other datasets are used in their original form without losing any data. The datasets are real datasets except for T10I4D100K and T40I10D100K. Experimented with the three selected algorithms and the three modified algorithms with 10 datasets. Here only relevant results are included.

The proposed algorithm is implemented by using Java language and run in 3.3 GHz Intel processor, 4 G byte memory and Windows 7 32-bit OS.

A huge number of frequent itemsets are produced during the experiments with all the datasets. These outputs are too large to compare manually. So to evaluate the accuracy of the proposed method, a subset of data from each dataset is used and compared the output of every datasets with the output of other algorithms. It is observed that the frequent itemsets generated from the proposed algorithm are the same as the frequent itemsets generated by other algorithms. Figures 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26 and 27 show all the experimental results.

7.1 Performance Evaluation of SR-FPgrowth

SR-FPgrowth is the improved version of the FP-growth algorithm. The performance of the SR-FPgrowth is compared with the FP-growth algorithm, and Figs. 7, 8, 9, 10, 11, 12 and 13 show the results.

7.1.1 Runtime Analysis

Figures 7, 8, 9 and 10 show the runtime of the proposed SR-FPgrowth compared with the FP-growth algorithm. In the Retail dataset, a consistent improvement in runtime can be noticed. The performance of SR-FPgrowth on dataset Connect is magnificent. With minimum support of 55, the runtime improvement is 26 s. The performance with dataset Accidentsdouble is better than the performance with Accidents. Minimum support 20% is not included in the figure of Accidents because a big variation is there from 30 to 20% in runtime. If the result is added with 20%, it will hide all other results. The results of SR-FPgrowth with other datasets are also better than the performance of the FP-growth algorithm.

7.1.2 Memory Usage Evaluation

Figures 11, 12 and 13 show the memory consumption of the algorithm SR-FPgrowth and FP-growth. The efficient memory usage of FP-growth is always appreciated when it is compared with all other recent algorithms. Here also

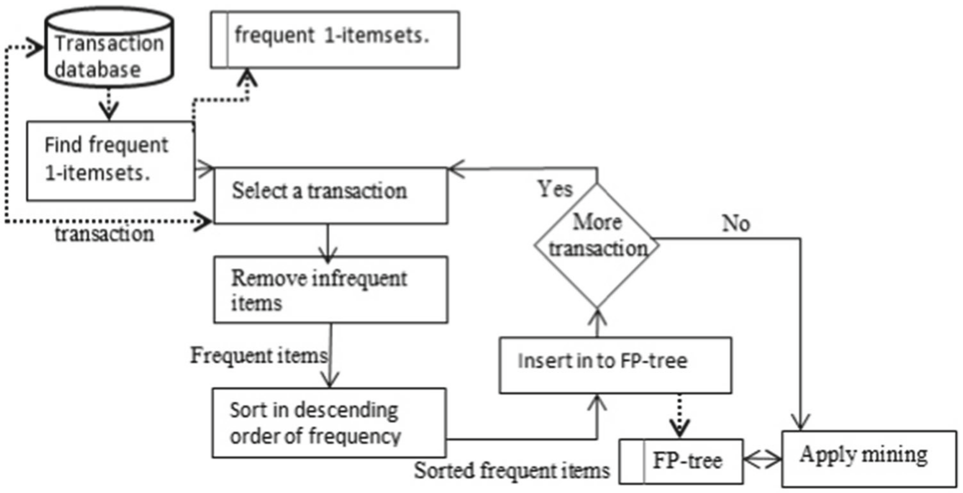


Fig. 4 Block diagram-1 representing the working of FP-tree- based algorithms

Table 7 Datasets

	Datasets	Transactions	Items
1	Accidents	340,183	468
2	Accidentsdouble	680,366	468
3	Chess	3196	75
4	Connect	67,557	129
5	Mushroom	8124	119
6	Pumsb	49,046	2113
7	pumsbdouble	98,092	2113
8	Retail	88,162	16,470
9	T10I4D100K	100,000	870
10	T40I10D100K	100,000	942

SR-FPgrowth consumes more memory than the FP-growth with most of the datasets. But with dataset Mushroom, the SR-FPgrowth consumes less memory than the FP-growth. The memory consumption of SR-FPgrowth in Retail and Pumsb is not good, but a noticeable runtime improvement in Pumsb can be seen. The difference in memory usage with other datasets is negligible.

7.2 Performance Evaluation of SR-PrePost^{+}

Figures 14, 15 and 16 display the experimental results of algorithms SR-PrePost^{+} and PrePost^{+}.

7.2.1 Runtime Analysis

The PrePost^{+} performs better than SR-PrePost^{+} with dataset Accidents. But the performance of SR-PrePost^{+} is far better than the performance of PrePost^{+} with the dataset Accidents-double. It reveals that the efficiency of the proposed method is increased when the size of the data increases. The minimum support is varied from 60 to 20%, but the time difference is almost the same with all minimum support values except 65%. All other results show that the proposed method is better than the original PrePost^{+} except the Mushroom dataset in which the performance of the two algorithms is the same.

7.2.2 Memory Usage Evaluation

Figures 17, 18 and 19 present the memory usage of the algorithms SR-PrePost^{+} and PrePost^{+}. The memory usage of SR-PrePost^{+} is less in dataset Accidents. In all other results, a negligible increase in memory usage can be seen when compared with the original PrePost^{+}.

7.3 Performance Evaluation of SR-FPclose

The performance of the proposed SR-FPclose and FPclose algorithms with different datasets is shown in Figs. 20, 21, 22 and 23.

7.3.1 Runtime Analysis

The proposed pre-processing method is not well suited with FPclose with all the datasets. When the size of the data is increased, an increase in the performance is visible in Pumsb and Pumsbdouble. SR-FPclose performs well with dataset chess, connect and Pumsbdouble when minimum supports become low as shown in Figs. 21a,b and 22b. With other data sets, SR-FPclose keeps a consistent increase in the performance while comparing with FPclose.

7.3.2 Memory Usage Evaluation

Figures 24, 25, 26 and 27 display the memory consumption of the algorithms SR-FPclose and FPclose. SR-FPclose

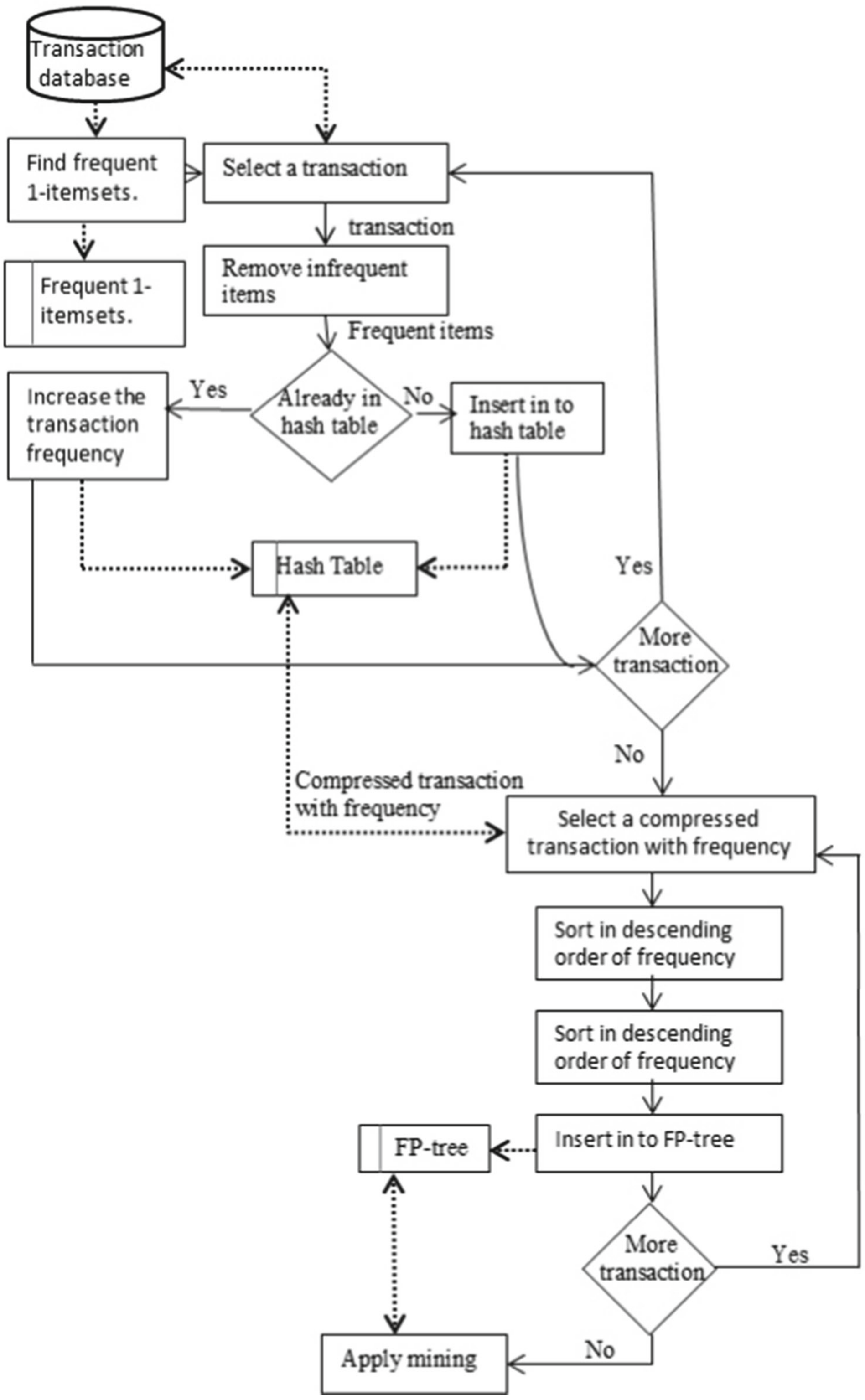
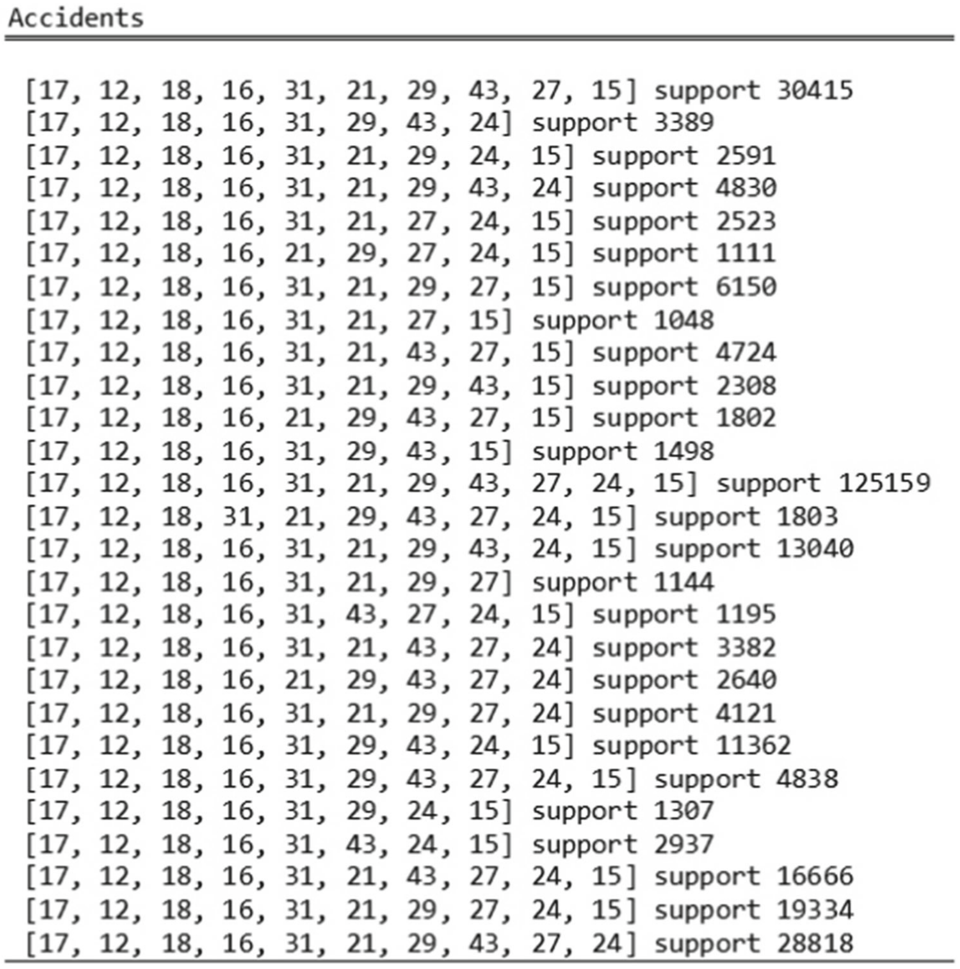


Fig. 5 Block diagram-2 representing the working of FP-tree- based algorithms after incorporating with the proposed method

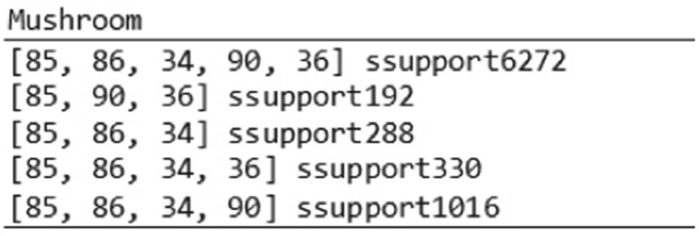
Fig. 6 **a** A portion of Hash map contents with dataset Accidents. **b** A portion of Hash map contents with dataset Mushroom



```
Accidents

[17, 12, 18, 16, 31, 21, 29, 43, 27, 15] support 30415
[17, 12, 18, 16, 31, 29, 43, 24] support 3389
[17, 12, 18, 16, 31, 21, 29, 24, 15] support 2591
[17, 12, 18, 16, 31, 21, 29, 43, 24] support 4830
[17, 12, 18, 16, 31, 21, 27, 24, 15] support 2523
[17, 12, 18, 16, 21, 29, 27, 24, 15] support 1111
[17, 12, 18, 16, 31, 21, 29, 27, 15] support 6150
[17, 12, 18, 16, 31, 21, 27, 15] support 1048
[17, 12, 18, 16, 31, 21, 43, 27, 15] support 4724
[17, 12, 18, 16, 31, 21, 29, 43, 15] support 2308
[17, 12, 18, 16, 21, 29, 43, 27, 15] support 1802
[17, 12, 18, 16, 31, 29, 43, 15] support 1498
[17, 12, 18, 16, 31, 21, 29, 43, 27, 24, 15] support 125159
[17, 12, 18, 31, 21, 29, 43, 27, 24, 15] support 1803
[17, 12, 18, 16, 31, 21, 29, 43, 24, 15] support 13040
[17, 12, 18, 16, 31, 21, 29, 27] support 1144
[17, 12, 18, 16, 31, 43, 27, 24, 15] support 1195
[17, 12, 18, 16, 31, 21, 43, 27, 24] support 3382
[17, 12, 18, 16, 21, 29, 43, 27, 24] support 2640
[17, 12, 18, 16, 31, 21, 29, 27, 24] support 4121
[17, 12, 18, 16, 31, 29, 43, 24, 15] support 11362
[17, 12, 18, 16, 31, 29, 43, 27, 24, 15] support 4838
[17, 12, 18, 16, 31, 29, 24, 15] support 1307
[17, 12, 18, 16, 31, 43, 24, 15] support 2937
[17, 12, 18, 16, 31, 21, 43, 27, 24, 15] support 16666
[17, 12, 18, 16, 31, 21, 29, 27, 24, 15] support 19334
[17, 12, 18, 16, 31, 21, 29, 43, 27, 24] support 28818
```

(a) accidents



```
Mushroom
[85, 86, 34, 90, 36] ssupport6272
[85, 90, 36] ssupport192
[85, 86, 34] ssupport288
[85, 86, 34, 36] ssupport330
[85, 86, 34, 90] ssupport1016
```

(b) mushrooms

consumes more memory than FPclose, but the memory consumption is varying with different minimum supports.

The Accidentsdouble and Pumsbdouble datasets are the same as Accidents and Pumsb but doubled the full data to make a large-sized data.

By analyzing the above results it is evident that the form of the proposed method is better than the existing ones and can increase the efficiency of the frequent itemset mining algorithms which are using a prefix tree structure. Further, the efficiency is improved when larger datasets are used and hence suitable in the context of big data. The memory usage is increased while applying the proposed method because a HashMap is used to store the compressed transactions. While comparing the run time with memory usage, the increase in memory is negligible and sometimes the memory usage is the same or lesser than the compared algorithms. From the experimental study, it is proved that the proposed pre-

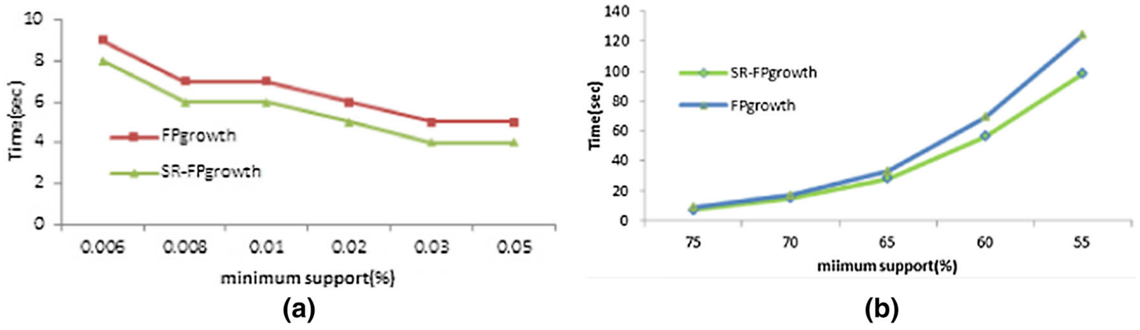


Fig. 7 **a** Runtime of Retail dataset. **b** Runtime of Connect dataset

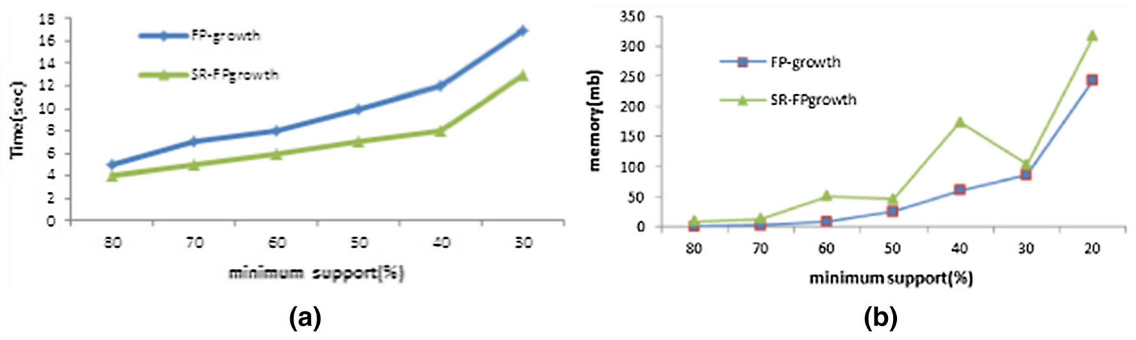


Fig. 8 **a** Runtime of Accidents dataset. **b** Runtime of Accidentsdouble dataset

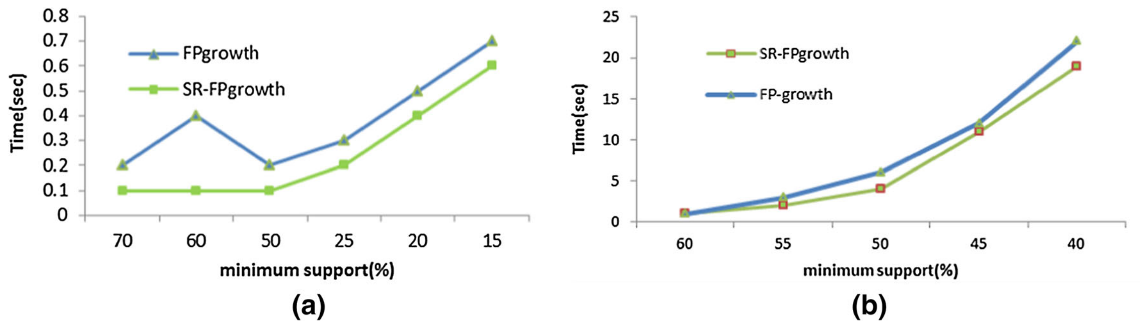
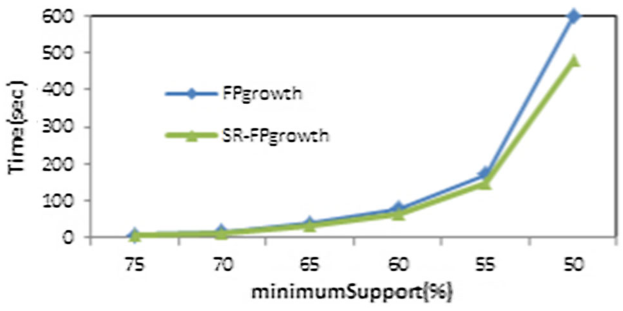


Fig. 9 **a** Runtime of Mushroom dataset. **b** Runtime of Chess dataset

Fig. 10 Runtime of Pumsb dataset



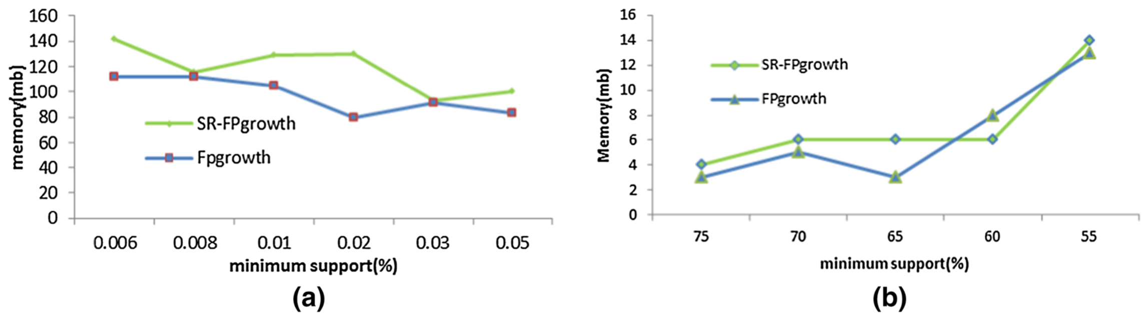


Fig. 11 **a** Memory usage of Retail dataset. **b** Memory usage of Connect dataset

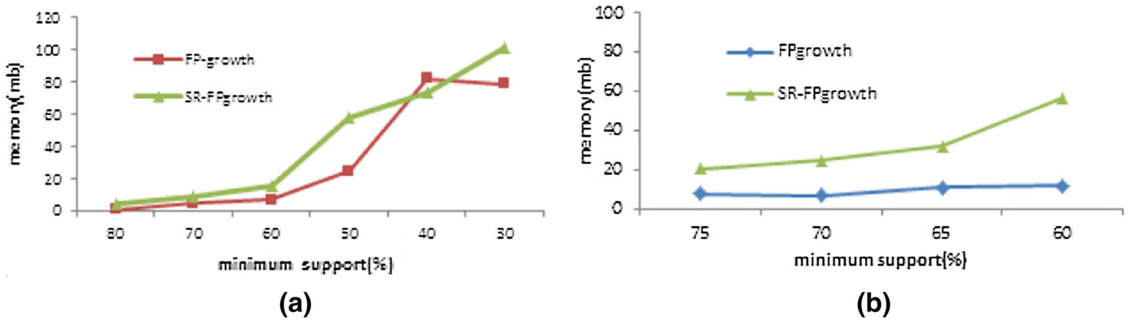


Fig. 12 **a** Memory usage of Accidents dataset. **b** Memory usage of Pumsb dataset

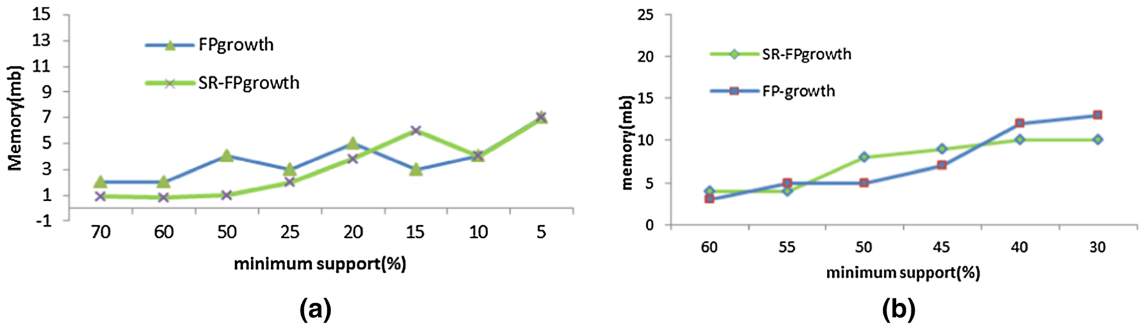


Fig. 13 **a** Memory usage of Mushroom dataset. **b** Memory usage of Chess dataset

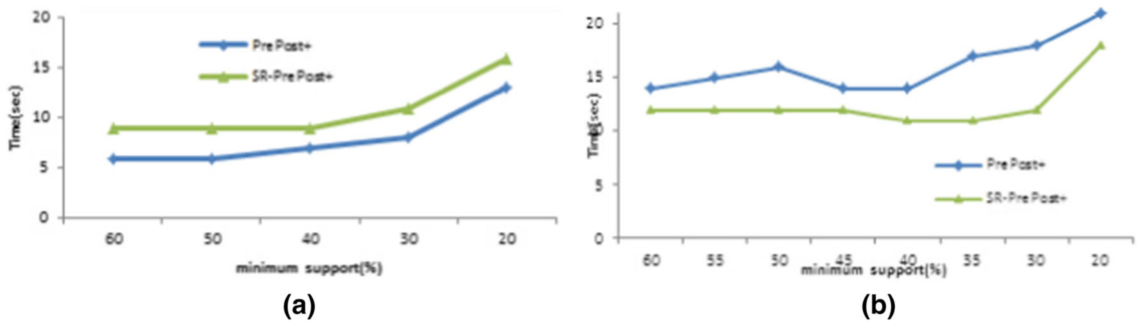


Fig. 14 **a** Runtime of Accidents dataset. **b** Runtime of Accidentsdouble dataset

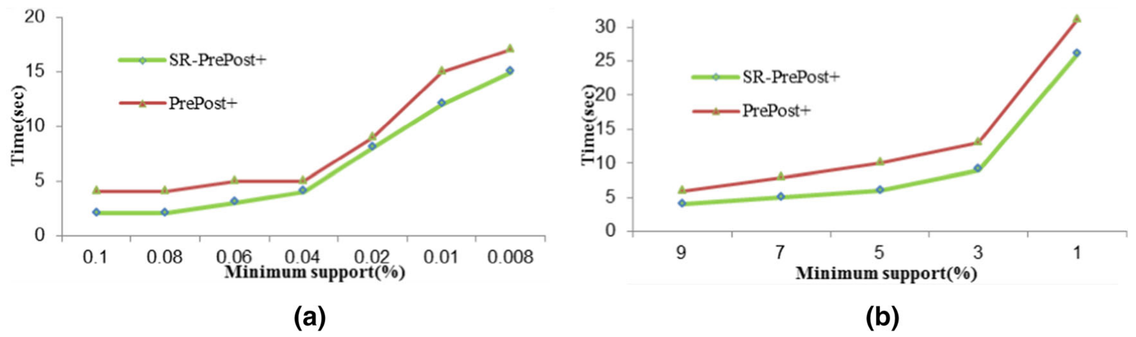


Fig. 15 **a** Runtime of T10I4D100K dataset. **b** Runtime of T40I10D100K dataset

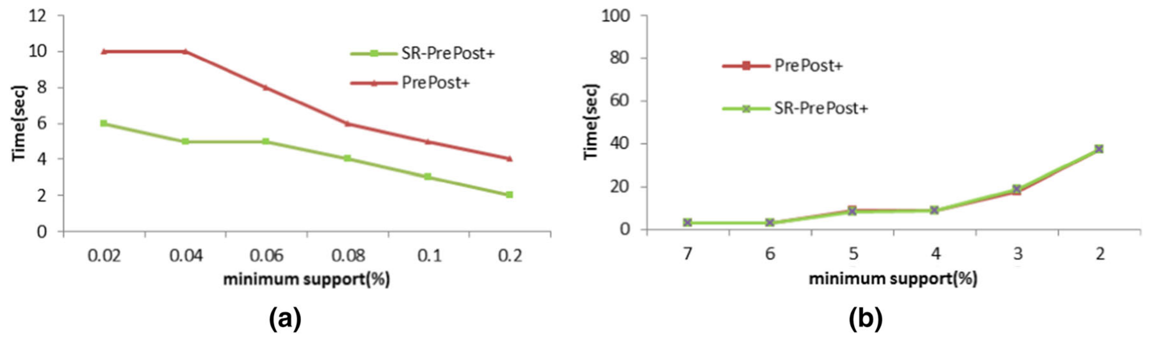


Fig. 16 **a** Runtime of Retail dataset. **b** Runtime of Mushroom dataset

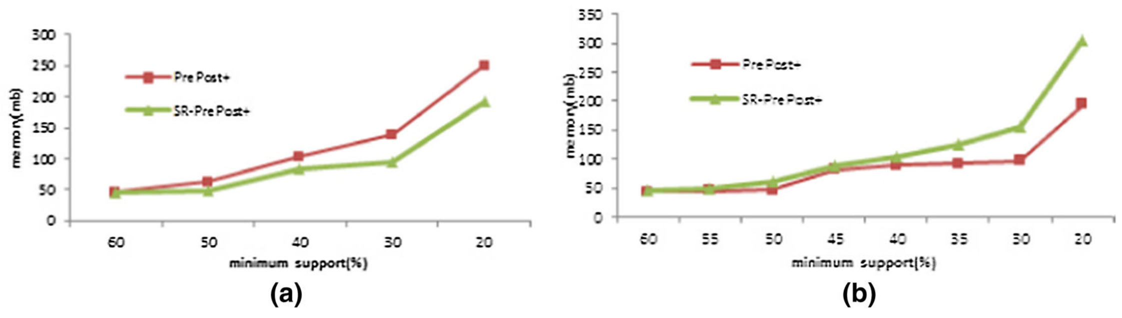


Fig. 17 **a** Memory usage of Accidents dataset. **b** Memory usage of Accidentsdouble dataset

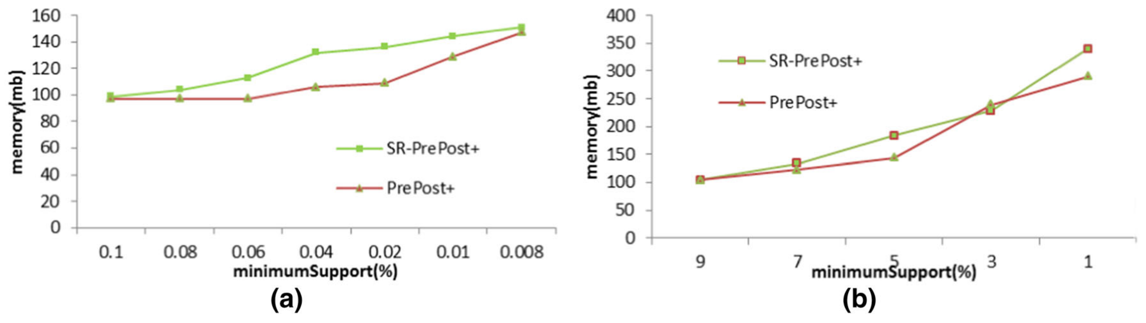


Fig. 18 **a** Memory usage of T10I4D100K dataset. **b** Memory usage of T40I10D100K dataset

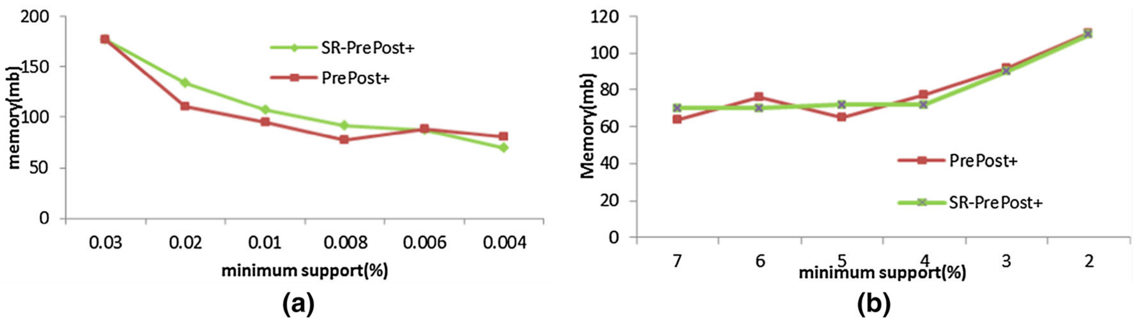


Fig. 19 **a** Memory usage of Retail dataset. **b** Memory usage of Mushroom dataset

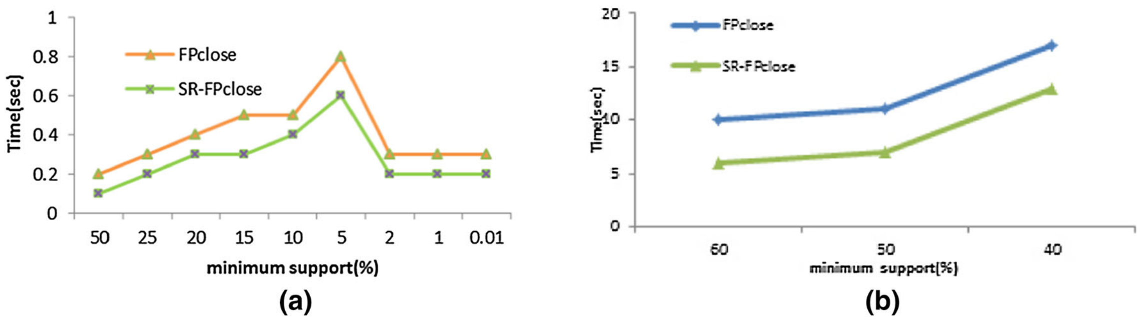


Fig. 20 **a** Runtime of Mushroom dataset. **b** Runtime of Accidents dataset

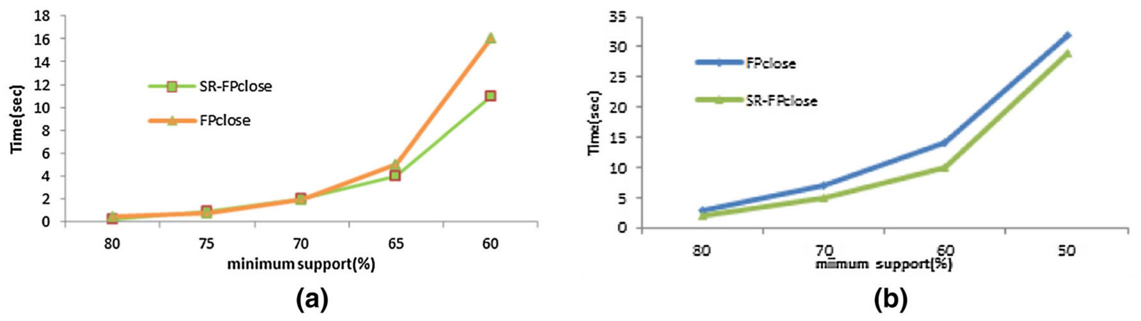


Fig. 21 **a** Runtime of Chess dataset. **b** Runtime of Connect dataset

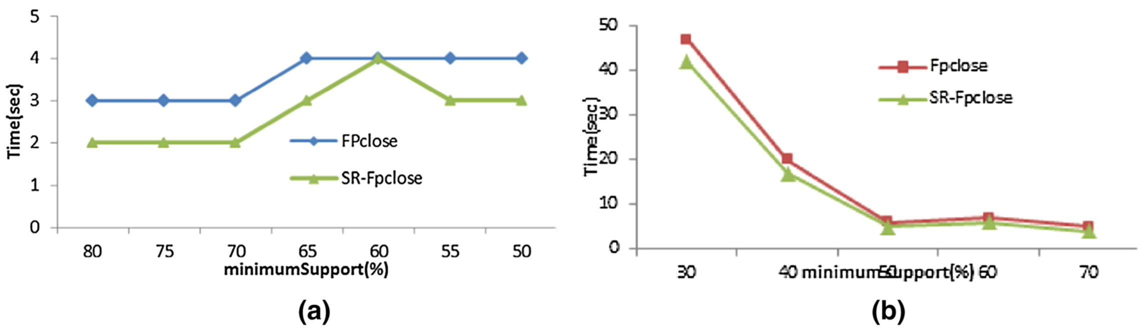


Fig. 22 **a** Runtime of Pumsb dataset. **b** Runtime of Pumsbdouble dataset

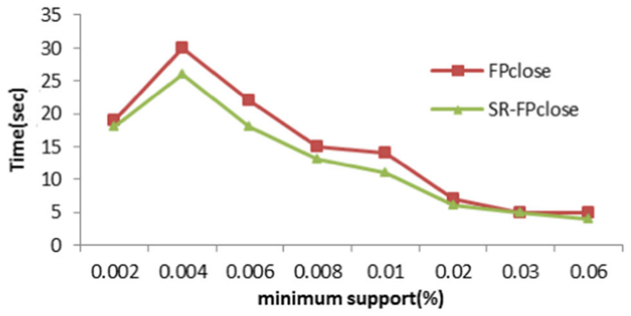


Fig. 23 Runtime of Retail dataset

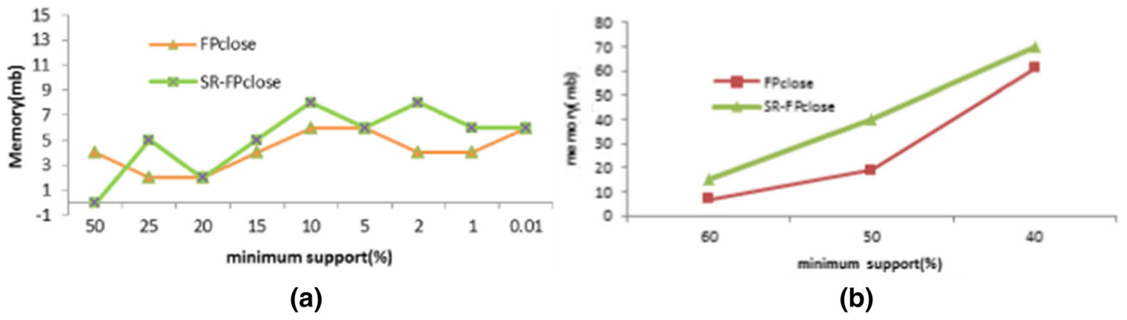


Fig. 24 **a** Memory usage of Mushroom dataset. **b** Memory usage of Accidents dataset

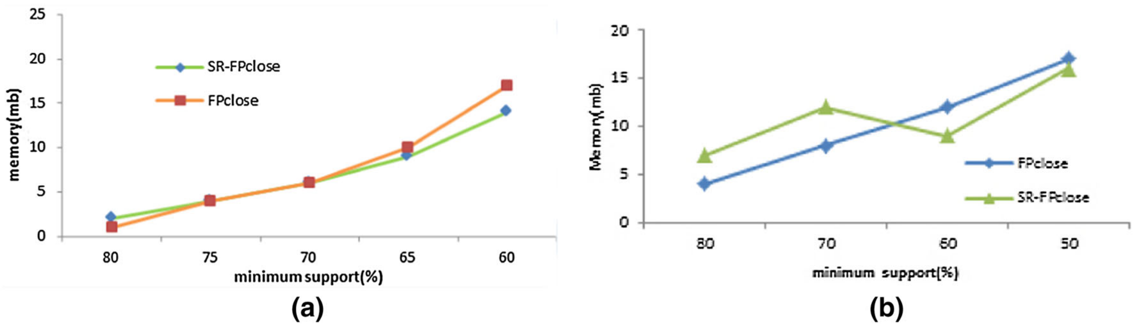


Fig. 25 **a** Memory usage of Chess dataset. **b** Memory usage of Connect dataset

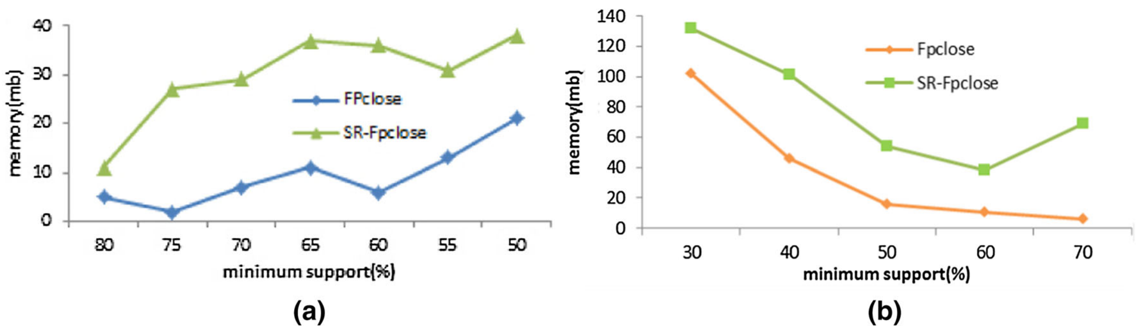


Fig. 26 **a** Memory usage of Pumsb dataset. **b** Memory usage of Pumsbdouble dataset

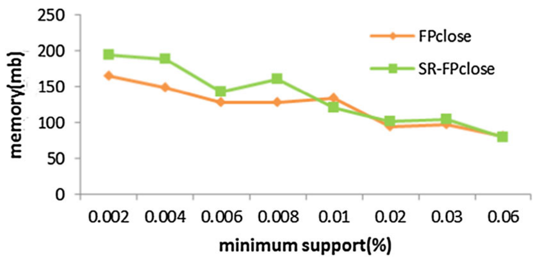


Fig. 27 Memory usage of Retail dataset

processing steps are adaptable with those mining algorithms which are going through the procedures mentioned in Sect. 3.

Figure 6a, b is the contents of the hash map during executing the proposed method SR-FPgrowth with datasets Mushroom and Accidents. The support is the frequency of each transaction. A set of selected transactions are included in the table. The full hash map cannot be presented because the datasets are very huge in size.

The implementation of the proposed method with all three algorithms (SR-FPgrowth, SR-PrePost$^+$,SR-FPclose) is performed well and far better than the original algorithms. The increase in memory consumption of SR-FPgrowth is negligible, but SR-FPclose consumes more memory. In the experimental results the proposed algorithm outperforms all the other algorithms except two results. But with most of the datasets, its memory consumption is almost near to the memory consumption of the original algorithms except a few ones.

The algorithm performs better with the dense dataset. Dataset Retail is a sparse dataset and also with the highest number of items. Hence the possibility of similar transactions is very less. The proposed method performs well with duplicate transactions. It is proved when the datasets are doubled. The worst performance among the results shown is the performance of SR-FPclose with dataset Accidents, but with the dataset Accidentsdouble, the same algorithm performs better. It proved that the algorithm is efficient with the dataset which contains duplicate transactions. The three modified algorithms performed better than their original form with all datasets except the above-mentioned one. The most important and noticeable result from our analysis is the one with dataset Connect. Both algorithms perform well with Connect dataset with normal memory consumption. Dataset Connect is a dense dataset with a lesser number of items. Other datasets are there with a lesser number of items, but the number of transactions is also less than the dataset Connect. In Connect dataset, each item has a chance to appear in more transactions. Hence transaction similarity is increased. This particular result indicates that the proposed method is suitable for a Dense dataset with similar transactions.

8 Conclusion

A novel pre-processing method to speed up the construction of FP-tree is proposed, implemented and tested with standard datasets. The major contribution of the proposal is the incorporation of a hash table which reduces efforts required for sorting and insertion. The experiments carried out with the standard databases proved that the proposed approach is efficient than the conventional FP-tree-based algorithms. An interesting outcome of the study is that the effectiveness of the proposed approach increases with an increase in the size of the data. The merit of the proposed algorithm is more pronounced when huge datasets are used compared to smaller ones.

The proposed approach can be implemented in association with most of the FP-tree-based algorithms. A theoretical and empirical study is required to establish the effectiveness of the proposed method, and also an extensive comparative analysis is required to be carried out.

References

1. Agrawal, R.; Imieliński, T.; Swami, A.: Mining association rules between sets of items in large databases. ACM SIGMOD Rec. **22**(2), 207–216 (1993)
2. Hu, J.; Yang-Li, X.: A new fast frequent itemsets mining algorithm based on forest. In: Fifth International Conference on Fuzzy Systems and Knowledge Discovery, 2008. FSKD’08, Vol. 2. IEEE (2008)
3. Han, J.; Pei, J.; Yin, Y.: Mining frequent patterns without Candidate Generation. In: SIGMOD ’00 Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 1–12
4. Narvekar, M.; Shafaque, F.S.: An optimized algorithm for association rule mining using FP tree. Procedia Comput. Sci. **45**, 101–110 (2015)
5. Tsay, Y.-J.; Hsu, T.-J.; Jing-Rung, Yu.: FIUT: a new method for mining frequent itemsets. Inf. Sci. **179**(11), 1724–1737 (2009)
6. Lin, K.-C.; Liao, I.-E.; Chen, Z.-S.: An improved frequent pattern growth method for mining association rules. Expert Syst. Appl. **38**(5), 5154–5161 (2011)
7. Deng, Z.-H.; Lv, S.-L.: PrePost+: an efficient N-lists-based algorithm for mining frequent itemsets via Children–Parent Equivalence pruning. Expert Syst. Appl. **42**(13), 5424–5432 (2015)
8. Agarwal, V.; Kushal, M.; Kumar, D.: An improvised frequent pattern tree Based association rule mining Technique with mining frequent itemsets algorithm and a modified header Table. arXiv preprint arXiv:1504.07018 (2015)
9. Grahne, G.; Zhu, J.: Fast algorithms for frequent itemset mining using FP-trees. IEEE Trans. Knowl. Data Eng. **17**(10), 1347–1362 (2005)
10. Pyun, G.; Yun, U.; Ryu, K.H.: Efficient frequent pattern mining based on linear prefix tree. Knowl.-Based Syst. **55**, 125–139 (2014)
11. Rácz, B.: nonordfp: an FP-growth variation without rebuilding the FP-tree. In: FIMI (2004)
12. Deng, Z.H.; Wang, Z.H.; Jiang, J.J.: A new algorithm for fast mining frequent itemsets using N-lists. Sci. China Inf. Sci. **55**(9), 2008–2030 (2012)

13. Deng, Z.-H.; Lv, S.-L.: Fast mining frequent itemsets using Nodesets. Expert Syst. Appl. **41**(10), 4505–4512 (2014)
14. Borgelt, C.: Simple Algorithms for Frequent Item Set Mining. Advances in Machine Learning II, pp. 351–369. Springer, Berlin (2010)
15. Tseng, F.-C.: An adaptive approach to mining frequent itemsets efficiently. Expert Syst. Appl. **39**(18), 13166–13172 (2012)
16. Ashrafi, M.Z.; Taniar, D.; Smith, K.: A compress–based association mining algorithm for large dataset. In: International Conference on Computational Science. Springer, Berlin (2003)
17. Hung, M.-C.; et al.: Efficient mining of association rules using merged transactions approach. WSEAS Trans. Comput. **5**(5), 916–923 (2006)
18. Dai, J.-Y.; et al.: An efficient data mining approach on compressed transactions. Int. J. Electr. Comput. Eng. **3**(2), 76–83 (2008)
19. Nair, B.; Tripathy, A.K.: Accelerating closed frequent itemset mining by elimination of null transactions. J. Emerg. Trends Comput. Inf. Sci. **2**(7), 317–324 (2011)
20. Fournier-Viger, P.; Gomariz, A.; Gueniche, T.; Soltani, A.; Wu, C.; Tseng, V.S.: SPMF: a java open-source pattern mining library. J. Mach. Learn. Res. (JMLR) **15**, 3389–3393 (2014)
21. http://fimi.ua.ac.be/src