International Journal of Engineering Applied Sciences and Technology, **2016**
Vol. 2, Issue 2, I S S N No. 2455-2143, Pages 59-69
Published Online December - January 2016 in IJEAST (http://www.ijeast.com)

NEWISTA: AN ALGORITHM FOR FINDING CLOSED FREQUENT ITEMSETS

Shalini Bhaskar Bajaj
Department of Computer Science and Engineering
Amity University Haryana
shalinivimal@gmail.com

Abstract— **Identifying frequent itemsets from a given database is a subject of research. A number of algorithms have been proposed for mining frequent itemsets from the given database. Most of these approaches enumerate candidate itemsets, determine their support and prune candidates that fail to reach the user-specified minimum support. This paper proposes the concept of intersecting transactions for finding frequent itemsets. The proposed approach of intersection makes use of closed itemsets to store the values of frequent itemsets. The proposed algorithm makes use of prefix data structure to handle the transactions. The prefix tree has certain problems associated with it i.e. large size of prefix tree. Large prefix trees not only require more memory space but at the same time handling large tree becomes difficult. Thus an attempt has been made to reduce the total number of nodes which lead to the reduced size prefix tree. The results of the proposed algorithm have been compared with the existing algorithms on both synthetic and real datasets.**

Keywords— **frequent itemset, closed frequent itemset, intersection, enumeration, prefix tree**

I. INTRODUCTION

It is hardly an exaggeration that the popular research area of data mining was initiated by the tasks of frequent itemset mining [1, 2, 3, 4] and association rule induction [12]. At least these tasks have a strong, long-standing tradition in data mining and knowledge discovery in databases and hence triggered abundance of publications in data mining conferences and journals. Most of these approaches enumerate candidate itemsets, determine their support and prune candidates that fail to reach the user-specified minimum support. This paper proposes the use of intersection approach for identifying frequent itemsets from the given database. Rest of the paper is structured as follows: Section 2 describes some of the existing algorithms for identifying frequent itemsets; section 3 describes the proposed algorithm, section 4 gives details on the performance evaluation for both proposed algorithm and existing algorithms, section 5 discusses some of the application areas and finally concluding remarks are given.

II. EXISTING ALGORITHM

A number of algorithms have been proposed for frequent itemset mining [1, 2, 3, 4]. Algorithms have also been proposed for finding maximal or closed frequent itemsets which can be used in finding frequent itemsets. Apriori algorithm [9] works by generating the candidate itemsets while FP-growth [6] works by pruning the infrequent itemsets. Eclat [10] searches in a depth first search manner to find all the frequent itemsets. Improved Carpenter algorithm [8] uses table-based implementation which is more efficient then list based implementation used by its earlier version [8]. The IsTa algorithm [5] uses intersection technique to identify frequent itemsets.

IsTa algorithm uses prefix tree data structure to maintain a repository of frequent itemsets which is updated be intersecting it with new transaction. In the intersection approach after adding the transaction to the prefix tree we need to perform the intersection of currently added transaction with all the existing transaction in the prefix tree.

For compressing the result of generated frequent itemsets, closed or maximal itemset can be used. Each frequent itemset has at least one maximal superset and also have a uniquely determined closed superset which can also preserve the knowledge of support value. Closed item set comes as a better alternate to compress the frequent itemset output.

III. PROPOSED ALGORITHM - NEWISTA

This section discusses the proposed algorithm NewIsTa that uses the concept of intersecting transactions. For processing the datasets with the help of the proposed algorithm NewIsTa, firstly items within each transaction should be sorted followed by the transaction sorting. Items are sorted on the basis of decreasing initial support values and transactions are sorted based on their size (number of items in it). Main reason behind the sorting of transaction based on their size is that if heavy transactions are processed earlier then the prefix tree becomes larger at initial level which makes it handling difficult. If smaller size transactions are processed initially then prefix tree will be small at initial level and can be handled easily.

NewIsTa algorithm tries to overcome the shortcomings of the IsTa algorithm by transposing the transaction matrix and by altering the item ordering from increasing support count to decreasing support count which helps in making the prefix tree structure more efficient. The working of proposed algorithm is discussed in Figure-3.1.

```
void new_IsTa (Transaction *T, Item *I, Pnode *root) {
*T=NULL;//initialize transaction list to null
*I=NULL;  //initialize item list to null
*root=NULL;//initialize prefix tree to null
   for (each transaction)  {
              string st=read_file(); //File read
             gettrans (st);
    }         //initialize *T and *I
copy_supp(*T and *I);  // update support for each transaction
sort I and each i in *T i.e. sort_items(*I);
sort transactions in *T by calling function sort_transactions(*T, *I);
get counter value for each t in *T by calling functional module get_counter (*T) ;
make_matix(*T,*I) //enumerate the transaction dataset
for ( each t in *T){
Pnode *p=prefix (ti); // add ti to prefix tree end
Findintersection(*p);          //intersection and prefix tree updation
}           //for loop closed
report(Pnode *root); //report and display prefix tree structure.
}           //program closed
```

Figure-3.1 NewIsTa algorithm

In NewIsTa three pointer structures are maintained, *T point towards the transaction list, *I points to the item list which contains distinct items in transaction database and *root which point towards the root of prefix tree. Initially, the input transaction file is read, each line of which represents a transaction. Transaction is taken in the form of a string and function gettrans() is called for each transaction string. Figure-3.2 describes function gettrans ().

```
void gettrans(string st){
for (each item i in st){
if(*T =NULL)
    initialize *T with the first item of st;
else{
    go to the end of *T
for(each item in t){
    go to the end of item list in last
    transaction in *T list and add I;  }// for closed
  }    //else closed
   add item to symbol list;
}
}           //program closed
```

Figure-3.2 Function gettrans

Figure-3.2 gives the functionality of gettrans(). It adds the transaction to the transaction list. It also maintains a list of distinct items for the transaction database. The given function gettrans() search the existing list of item and adds the item if it is not in the list and if it already exists in the list then increment its support value. This function also adds support value for each item in the item list.

After processing the input file, transaction list contain transactions and item list contain the list of distinct item with their support value. To add support for each item in the transaction list function copy_supp() is executed, details of which are given in Figure 3.3.

```
void copy_cupp(*T and *I){
for (each transaction in *T) {
t=current transaction;
for (each item in t){
search the item list and add support for the item; }// for closed
}//for closed
} //program closed
```

Figure-3.3 Function copy_supp

Figure-3.3 gives function copy_supp() to add the support value of the items in transaction list from item list. This support value of items will be used further to sort the items within each transaction.

After getting the transaction and item list, sorting is done for both the lists. Bubble sort is used to sort the items according to their initial support value. We are using the decreasing orientation for the item based on their initial support value. Similarly transactions are sorted on the basis their size (total number of items present in the transaction). Function sort_transcation() is used for this purpose and this function also make use of bubble sort. Figure-3.4 describe the working of sort_transactions().

```
void sort_transactions(*T and *I){
for (each t in *T) {
for (each item i in t) {
for (i+1 to the end of item list for t)
if( supp( i)<supp(i+1))
     change their respective positions;
  }//inner for loop closed
}// outer for loop closed
for (each t in *T) {
for (t+1 to the end of *T) {
if (size ( t)>size (t+1))
      change their respective positions; }//inner for loop closed
}// outer for loop closed

}//program closed
```

Figure-3.4 Function sort_transactions

After passing the transaction list to the above function, the sorted list of transactional database is obtained.

For making the prefix tree out from this database we use prefix() which takes the sorted *T and *I as input and make a prefix tree from the transaction and intersection data. Its structure and functionality is explained in Figure-3.5

```
void prefix (*T, *I){
for (each transaction t in *T) {
if (*root is not initialized) {
initialize the *root of prefix tree; }
else {
go to the end of prefix tree and add t;}
call findintersection( t, *root);//pass the current prefix tree
}// for closed
}//program closed
```

Figure-3.5 Function prefix

```
void findintersection (transaction t, Pnode *root) {
for (each transaction tj in *T) {
for (each item i in t) {
for (each item j in tj) {
if (i==j){
    add to the intersection list;}// if closed
}//innermost for loop closed
}// middle for loop closed
}// outer for loop closed
for (each item li in intersection list){
   for (each item i in tj) {
if (li is equal to i) Count++; //count is the measure of matches of
intersection list items with prefix tree transaction items
if (intersection list is empty) {
do nothing; } // first case closed
else if (count is equal to size of tj) {
increase support of all items in tj; }// second case closed
elseif (count<size(tj) && (count is equal to intersection list
size))
{ for (i<=count)
increase support of item in tj;
}// third case closed
else if (count<size(tj) &&(! (count is equal to intersection
list size)))
{ for (i<=count)
increase support of item in tj;
add rest of items in intersection list as branch in the path of tj;
}// fourth case closed
else (count==0) {
add intersection list items as separate branch in prefix tree;
} // fifth case closed
}// for loop closed (i.e. tj)s
}//program closed
```

Figure-3.6 Function findintersection

Function prefix() add the transaction to the prefix tree and then pass the prefix tree pointer to the findintersection() along with the currently added transaction.

Figure-3.6 describes the function Findintersection() which process the prefix tree and find the intersection of currently added transaction with all the existing transactions in the prefix tree. This function generates an intersection list which contains the list of intersecting items and then the desired updating in the prefix tree is applied.

3.1 Inserting a new transaction to the prefix tree

Let the prefix tree contains (n-1) transactions which ranges from t_1 to t_{n-1}. For adding the nth transaction to the prefix tree, we need to carry out the intersection of nth transaction with all the existing transaction in the prefix tree. Let the nth transaction is denoted by t_n, then for intersecting t_n with t_k where *k* ranges from 1 to (n-1) following set of rules are used.

For identifying common items between transaction t_n and t_k and storing them in set S following set of rules are used. If there is no item in common in t_n and t_k (S is empty) then do nothing. If there exists a complete branch in the prefix tree which contain all the intersected nodes (all the items of S) then update the status of each node in that branch. If a branch contains all items of intersection and still have child node that are not in intersection then update the support of intersecting nodes in that branch and do not perform the updating of support value to the child node those are not in the intersection.

If there exists only initial i items of the set S in continuity then

(a) Update the support of those nodes and assume the last node as *k*.

(b) Make a new branch out of the remaining item from intersection with support value of one.

(c) Add them as the sibling of node *k*

If no branch contains the intersected items then make a new branch out of the items in S and add that as the completely new branch in the prefix tree at level one.

For illustrating the proposed algorithm discussed in Figure-3.1, dataset given in Table-3.1 is used that consists of 5 transactions.

Table-3.1 Sample dataset

Transaction id	Items
t1	a, b, c, d, e
t2	a, b, c, e
t3	a, b, g
t4	c, d, f
t5	a, d

According to the proposed algorithm, function gettrans() takes Table-3.1 as input and generate Transcaton List and Item List. Item list stores all the distinct items in the dataset along with their support value and transaction list contains all the transactions. From the dataset given in Table-3.1, initial support value stored for each item in the item list is given by Table 3.2.

Table-3.2 Initial support of items

Item	Initial support
a	4
b	3
c	3
d	3
e	2
f	1
g	1

Function sort_transactions() given in Figure-3.4 takes transaction list as input, sort the items within each transaction and produce output which is given in Table-3.3. Function sort_transactions() uses decreasing ordering for the items. Finally, the items in the Table-3.3 are arranged according to the decreasing initial support value.

Table-3.3 Decreasing ordering of items

Items Tids	a	b	c	d	e	F	g
t1	4	3	3	3	2	0	0
t2	3	2	2	0	1	0	0
t3	2	1	0	0	0	0	1
t4	0	0	1	2	0	1	0
t5	1	0	0	1	0	0	0

After sorting the transactions function sort_transactions() arrange the transactions in the increasing size in terms of number of item in the transaction. Final output of function sort_transactions() is given by Table-3.4.

Table-3.4 Transaction ordering according to size

Items Tids	a	b	c	d	E	f	g
t1	**4**	**0**	**0**	**3**	**0**	**0**	**0**
t2	**0**	**0**	**3**	**2**	**0**	**1**	**0**
t3	**3**	**3**	**0**	**0**	**0**	**0**	**1**
t4	**2**	**2**	**2**	**0**	**2**	**0**	**0**
t5	**1**	**1**	**1**	**1**	**1**	**0**	**0**

For making a prefix tree from the data given in Table-3.1 function prefix() is used (refer Figure-3.5). This function makes use of the intersection rule and generates the prefix tree.

IV. PERFORMANCE EVALUATION

A series of executions were made to show the performance of the proposed algorithm with the existing algorithm. All experiments were performed on an Intel I3 processor running at 2.20 GHz speed, supporting Dev-C++. Section-4.1 shows the experiments that are performed on synthetic dataset and section-4.2 shows the experiments that were performed on real datasets.

4.1 Discussion of results on Synthetic Datasets

Synthetic datasets were generated using IBM data generator to compare the performance of proposed algorithm with the existing algorithm. IBM data generator takes three parameters, total number of transactions to be generated (represented by T), average length of transaction (represented by L) and number of distinct items (represented by I). Table-4.1 shows the synthetic datasets generated along-with their nomenclature.

Table-4.1 Different parameters used for generating synthetic datasets

Synthetic Datasets	Total number of transaction(T)	Average length of Transaction dataset(L)	Total number of distinct items(N)
T100k_L10_N10k	100	10	10
T100k_L10_N20k	100	10	20
T100k_L10_N30k	100	10	30
T200k_L10_N10k	200	10	10
T200k_L10_N20k	200	10	20
T200k_L10_N30k	200	10	30
T300k_L10_N10k	300	10	10
T300k_L10_N20k	300	10	20
T300k_L10_N30k	300	10	30
T400k_L10_N10k	400	10	10
T400k_L10_N20k	400	10	20
T400k_L10_N30k	400	10	30
T500k_L10_N10k	500	10	10
T500k_L10_N20k	500	10	20
T500k_L10_N30k	500	10	30

Another set of synthetic datasets were generated containing number of transactions ranging from 1 lakh to 5 lakh, number of distinct items varying from 10k to 30k and average transaction length of 10. The datasets generated are divided into 3 categories based on the number of distinct items. Datasets having number of distinct items 10k, 20k and 30k are grouped into Type-I, Type-II and Type-III respectively.

Table-4.2, Table-4.3 and Table-4.4 shows the comparison for the proposed algorithm and existing algorithm on datasets belonging to Type-I, Type-II and Type-III respectively. Total execution time consists the input file reading time, time used for filtering, sorting and recording of items, time used for sorting of transactions, intersecting time and output file writing time.

Table-4.2 Comparison of number of intersecting nodes and execution time for the IsTa and NewIsTa algorithm on Type-I datasets

Type-I Datasets	IsTa algorithm			NewIs Ta algorithm		
	Number of intersecting nodes	Intersecting time (sec.)	Total execution time (sec.)	Number of intersecting nodes	Intersecting time (sec.)	Total execution time (sec.)
T100k_L10_N20k	23327	9.11	10.15	7984	1.84	2.03
T200k_L10_N20k	42714	32.65	33.06	13360	5.87	6.29
T300k_L10_N20k	61357	65.05	65.67	17998	12.29	12.82
T400k_L10_N20k	76557	106.88	107.61	21369	20.56	21.3
T500k_L10_N20k	92543	161.6	164.02	24923	31.06	31.96

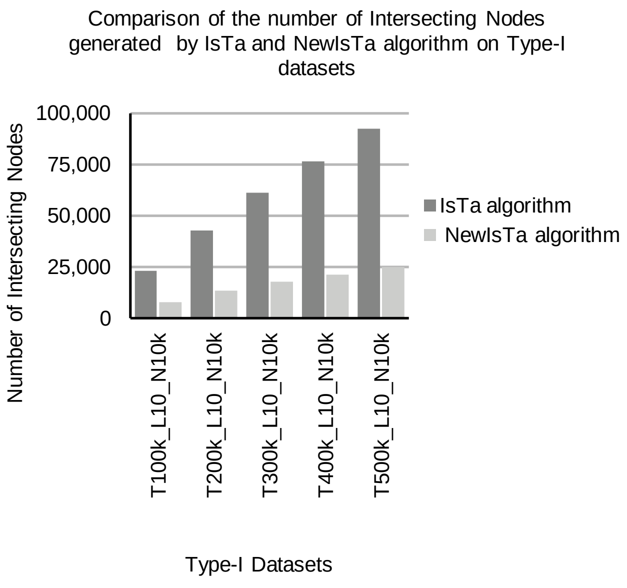
Table-4.3 Comparison of number of intersecting nodes and execution time for the IsTa and NewIsTa algorithm on Type-II datasets

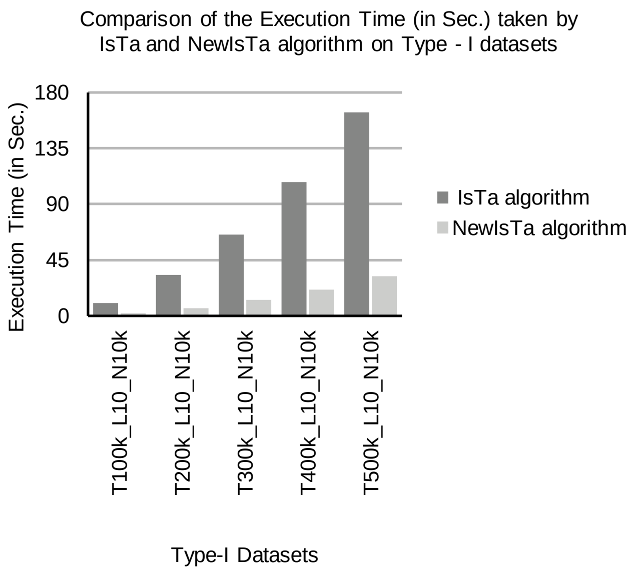
Type-II Datasets	IsTa algorithm			NewIs Ta algorithm		
	Number of intersecting nodes	Intersecting time (sec.)	Total execution time (sec.)	Number of intersecting nodes	Intersecting time (sec.)	Total execution time (sec.)
T100k_L10_N20k	1240	0.12	0.77	448	0.06	0.25
T200k_L10_N20k	1661	0.28	0.89	470	0.11	0.44
T300k_L10_N20k	2356	0.39	1.28	477	0.14	0.69
T400k_L10_N20k	2675	0.58	1.58	486	0.2	0.94
T500k_L10_N20k	2710	0.67	1.7	505	0.23	1.1

Table-4.4 Comparison of number of intersecting nodes and execution time for the IsTa and NewIsTa algorithm on Type-III datasets

Type-III Datasets	IsTa algorithm			NewIs Ta algorithm		
	Number of intersecting nodes	Intersecting time (sec.)	Total execution time (sec.)	Number of intersecting nodes	Intersecting time (sec.)	Total execution time (sec.)
T100k_L10_N30k	316	0.02	0.35	233	0.01	0.2
T200k_L10_N30k	370	0.028	0.41	235	0.015	0.35
T300k_L10_N30k	440	0.03	0.65	233	0.02	0.57
T400k_L10_N30k	459	0.036	0.77	238	0.025	0.69
T500k_L10_N30k	490	0.04	1.02	238	0.03	0.87

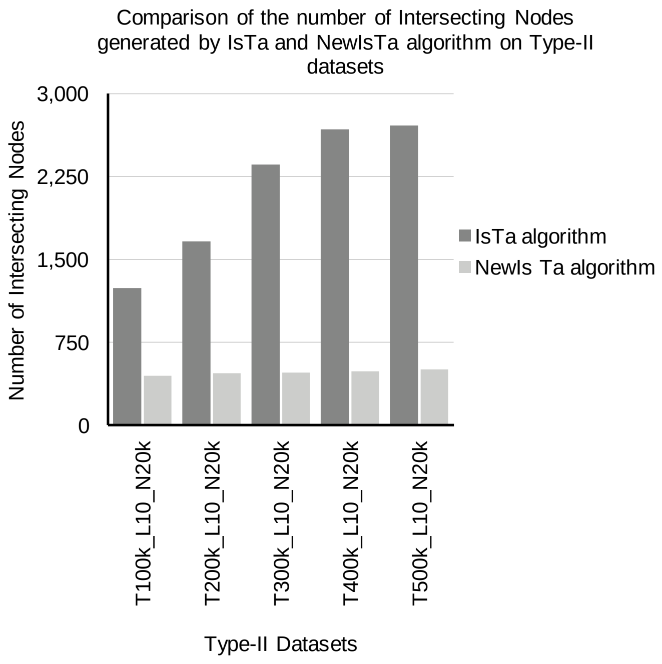
Figure-4.1(a) and Figure-4.1(b) corresponds to Table-4.2 presents the comparison for total intersecting nodes and total execution time for IsTa and NewIsTa algorithm respectively. From Figure-4.1(a) and Figure-4.1(b), it can be inferred that proposed algorithm has lesser number of intersecting nodes that leads to reduced intersecting time and finally total execution time for the proposed algorithm is reduced. Another observation that can be made from the Figure-4.1(a) and Figure-4.1(b) is that with the increase in the number of transactions the time taken by the proposed algorithm reduces as compared to the existing algorithm.

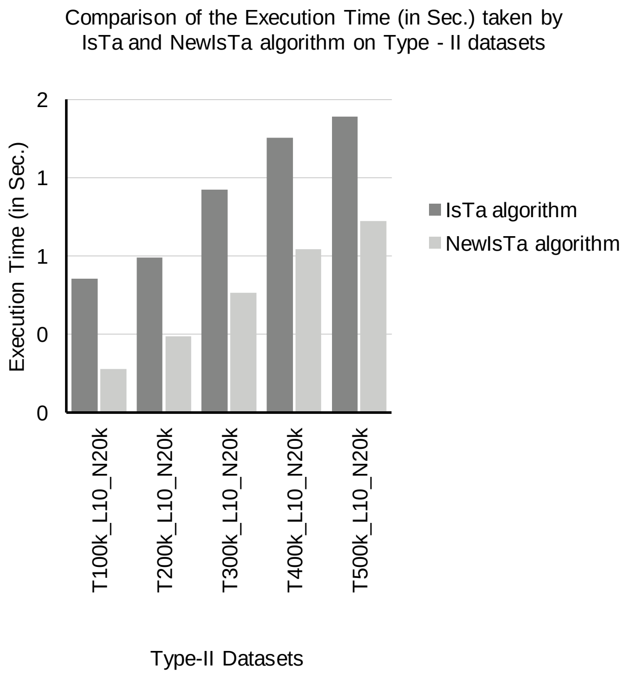




(a) (b)

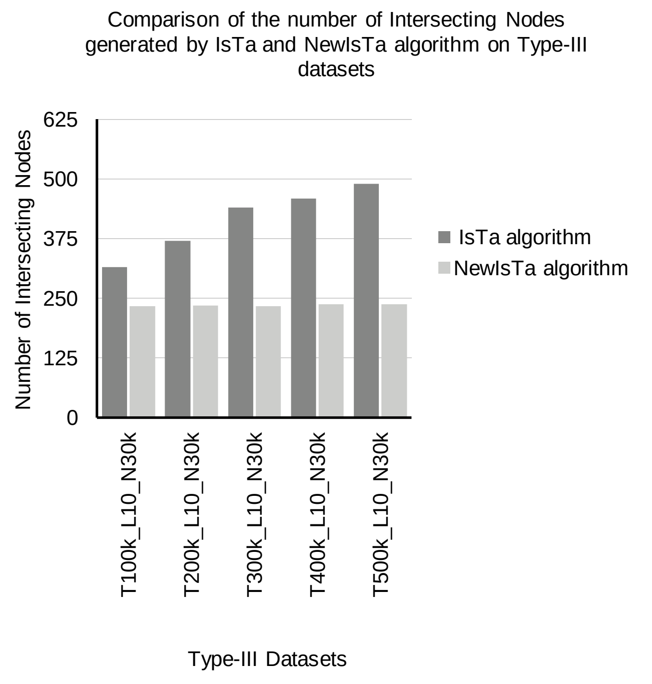
Figure 4.1Comparison of number of intersecting nodes generated and execution time taken by IsTa and NewIsTa algorithm on Type-I datasets

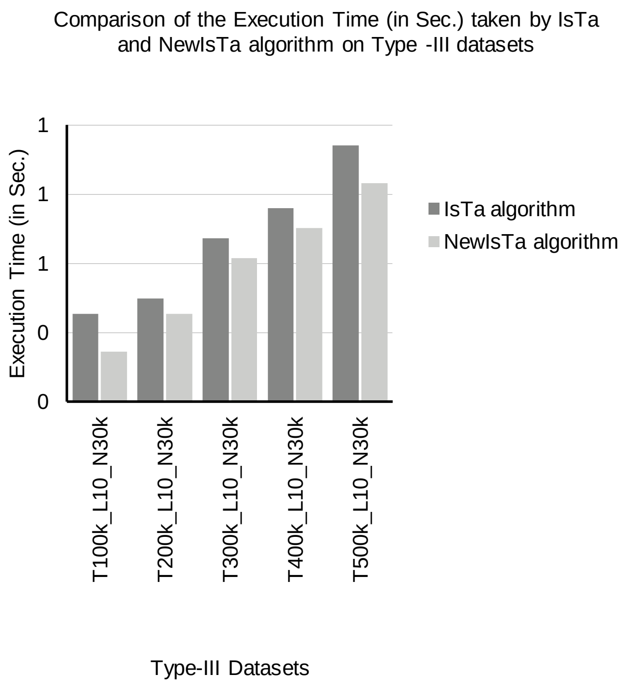




(a) (b)

Figure 4.2 Comparison of number of intersecting nodes generated and execution time taken by IsTa and NewIsTa algorithm on Type-II dataset





(a) (b)

Figure 4.3 Comparison of number of intersecting nodes generated and execution time taken by IsTa and NewIsTa algorithm on Type-III dataset

In the similar way, Figure-4.2(a) and Figure-4.2(b) represents the details given in Table-4.3 and Figure-4.3(a) and Figure-4.3(b) gives graphical representation of Table-4.4. From the table 4.2, 4.3 and 4.4 and figure 4.1, 4.2 and 4.3, it can be inferred that proposed algorithm gives better result for all three type of synthetic datasets. It can be concluded from Figure-4.1, 4.2 and 4.3 that the proposed algorithm takes less intersecting time hence leads to reduced total execution time.

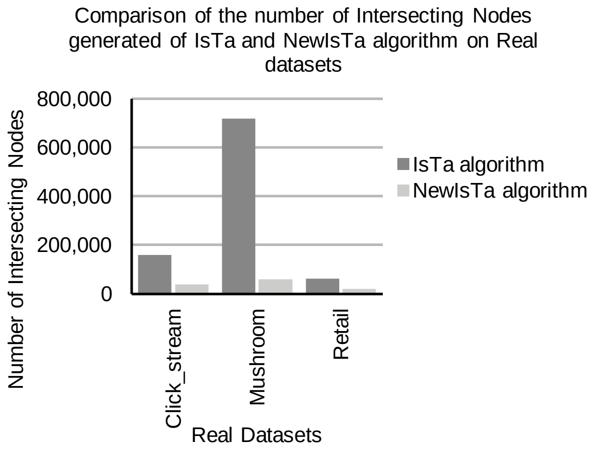
4.2 Discussion of results on Real Datasets

This section discusses the performance of the proposed algorithm in comparison to IsTa algorithm on real datasets such as click_stream, mushroom, retail in terms of number of intersecting nodes generated and execution time. Details are given in Table 4.5 and Figure 4.4(a) and 4.4(b).

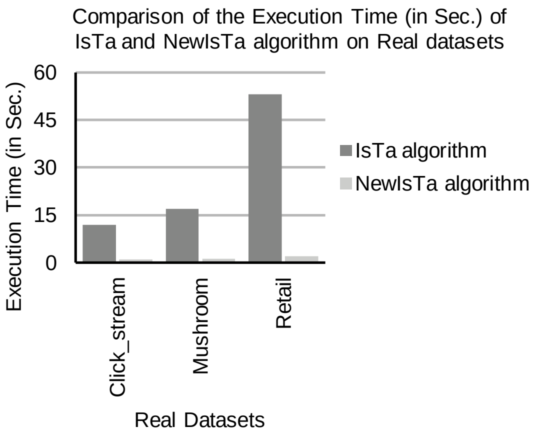
Click_stream dataset contains data related to real time browsing pattern. It contains 17431 total instances and it is collected over 74919 web hits. Mushroom dataset describe mushroom's physical characteristics such as classification, poisonous or edible. It contains 2726 number of instances with 22 attribute values. Retail dataset contains data from a retail shop and contains 21292 transactions over 10348 items.

Table-4.5 Comparison of number of intersecting nodes and execution time for the IsTa and NewIsTa algorithm on real datasets

Real Datasets	IsTa algorithm			NewIsTa algorithm		
	Number of intersecting nodes	Intersecting time (sec.)	Total execution time (sec.)	Number of intersecting nodes	Intersecting time (sec.)	Total execution time (sec.)
Click_stream	159656	11.73	11.89	37745	1.00	1.08
Mushroom	718182	16.79	16.89	58794	1.06	1.14
Retail	62203	52.98	53.12	19211	2.01	2.08



(a)



(b)

Figure 4.4 Comparison of number of intersecting nodes generated and execution time taken by IsTa and NewIsTa algorithm on Real datasets

From Figure-4.4(a) and Figure-4.4(b), it can be inferred that the proposed algorithm proves better for all the real datasets but for mushroom dataset it gives best results. For mushroom dataset the proposed algorithm has approximately one lakh lesser number of intersecting nodes. From the above comparison table and corresponding graphical representation, it can be concluded that the proposed algorithm gives better results for both synthetic and real datasets.

V. APPLICATION AREAS

Frequent itemset mining has a very wide area of application such as market basket analysis, intrusion detection system, heterogeneous genome data, remotely sensed images/data mining and product assortment decisions. Subsection 5.1 discusses maximum profit item selection and susection 5.2 discusses intrusion detection system in details.

5.1 Maximum Profit Item Selection (MPIS)

There are certain applications where the generated data is very large such as transaction data of Wal-Mart in Hedberg. In Wal-Mart 200 lakhs sales transactions get generated within a single day. For such type of data an efficient mining solution is required that can generate reports on profit generated items and also have the capability to discard the losing items. It may be simple enough to sort items by their profit and make the selection. However, this ignores a very important aspect in market analysis 'the cross-selling effect'. Cross-selling items are those which do not generate much profit by themselves but they are the catalysts for the sales of other profitable items.

For better understand the cross-selling effect three items-monitors, keyboards and telephones are taken with profits of 1000, 100 and 300 respectively. The problem here is to select a subset from the given set of items so that the estimated profit of the resulting selection is maximal among all choices. This subset would be {monitors, telephones} here. However, there is strong cross-selling effect between monitor and keyboard. If the shop stops carrying keyboard, the customers of monitor may choose to shop elsewhere to get both items. The profit from monitor may drop greatly, and we may be left with profit of 300 from telephones only. If both monitors and keyboards are chosen, then the profit can be expected to be 1100 which is higher.

It has been suggested that it is possible to make a decision for Maximal-Profit Item Selection (MPIS) with Cross-selling considerations in [7]. MPIS approach can be proved beneficial here. MPIS utilises the concept of the relationship between selected items and unselected items. The cross-selling factor has been proposed in the form of loss-rule. A loss-rule has a form $I \rightarrow \Diamond d$, where $\Diamond d$ means the purchase of any item d. From the history, whenever a customer buys the item I, he/she also buy at least one of the items in d. This rule can be used to predict the customer behaviour. This rule holds when we have d in the stock but if there is no item d in the stock, then customer will not purchase I. This is because if the customer still purchases I, without purchasing any items in d, then the pattern would be changed. Therefore, the higher the confidence of $I \rightarrow \Diamond d$, the more likely the profit of I should not be counted.

NewIsTa can be used for such kind of applications. For using NewIsTa the transactions dataset should be in the acceptable format. Most of the retail shops have a format similar to the format used in our experimental results but some shops use the following format:-

Tids < item list >

Here, Tids means the transaction identification number which uniquely identifies each transaction in the given database. The <item list> contains a list of items that are purchased. Item code is used in place of the item name. Item code is assigned to each and every item presented in the store, so that handling of transaction data becomes easy. This can be explained with the help of an example discussed below.

Table-5.1 Transaction database

Transactions	Items
t1	(1, 2, 5, 7, 8)
t2	(1, 11, 7)
t3	(8, 3, 6, 2)
t4	(9, 2, 5, 3, 6, 8, 7, 10)
t5	(18, 17, 13, 3, 6)

For converting such kind of format to the desired format Tids are removed, items for a single transaction are presented in the same line with single space as the separator and next line implies the next transaction.

5.2 Intrusion Detection

An intrusion detection system (IDS) is a device or software application that monitors network or system activities for malicious activities or policy violations and produces reports to a Management Station. We can apply data mining on the network data to learn rules that accurately capture the behaviour of intrusions and normal activities [11]. These rules can then be used for misuse detection and anomaly detection.

Here the question arises that how can we apply data mining on the network data. Actually the program and user activity shows certain kind of patterns. Each user has a privilege to use the system such as the privilege of opening the files and editing the files. These normal user patterns generate a user profile which contains certain aspects related to user system usage such as the kind of operations carried out by user and its peak hours of usage. The data which are getting generated from usage can be viewed with the help of exact time and command. Likewise, in network connection data, the combination of the features timestamp, source and destination hosts, source port, and service (destination port) uniquely identifies a connection record. Thus, they can be the essential features. We can apply mining of this data; however each of the inputs can vary depends upon the need of miner. For example, in the network data we can take a host as the mining parameters and can generate association rule related to the host. Similarly, to a single computer we can analyse the user pattern by taking commands as the parameter. These generated association rules form the basis of user profile generation. Any deviation from the normal user behaviour gives a sight of the attack. To better understand it let us take an example of a bank network dataset. We have profiles of usage such as enquiry counter staff generally uses the network highest at afternoon and has the access of only reading the account values. If we come across the writing and higher access statements at the evening time then it shows deviation from the normal profile behaviour. Hence it shows some kind of abnormal behaviour or malicious kind of activity. Below in Table-5.2 we have presented the network database for a bank. It has a sequence of attribute access, here each attribute value represent a unique operation (as shown in Table-5.2) and the subscript here denotes the read and write property for that attribute.

Table-5.2 Bank network data

Txn ID	Attribute Access Sequence
1	11r, 13w, 4r, 8r, 2r, 6r, 1r, 3r
2	7r, 2r, 7r, 2r, 3r, 9w
3	6r, 1r, 3r, 3r, 9w, 1w, 2r, 7w
4	11r, 12w, 2r, 4w, 6r, 1r, 3r
5	2r, 4w, 2r, 7w, 7r, 8r, 2r
6	11r, 13w, 4r, 8r, 2r, 2r, 4w
7	3r, 9w, 4r, 8r, 2r, 8r, 2r
8	7r, 8r, 2r, 2r, 2r, 8w, 5w, 2r, 4w
9	8r, 2r, 3r, 9w, 7r, 2r
10	3r, 9w, 6r, 1r, 3r, 3r, 9w, 1w

In intrusion detection system we need to pay special attention to the minimum support value. Because each sequence have different operation and based on the type of operation each sequence need different pruning threshold. Below we discuss the minimum support calculation for each network sequence to be processed. Table-5.3 shows the enumeration of different access patterns. The Table-5.2 shows the simple access pattern, but access is categorized based on the person which is using the services such as:

- High Sensitivity (HS)
- Medium Sensitivity (MS)
- Low Sensitivity (LS)

The sensitivity of an attribute depends on the particular database application where it is used. Such as higher authorities have the power to view the data, read data, write data, new account entry, personal information access etc. They are categories as high sensitivity data statements. The lower workers have the permission to only view and edit account statements. They are not capable of viewing and editing the personal details of the account holder. They are considered with lower sensitivity. Based on the application we can specify the sensitivity levels. For the above sensitivity level we can assign weight to different sensitivity level such as w1, w2, w3 (refer to Table-5.4). Weight should be assign in the order w1≤w2≤w3 and w1, w2, w3 ∈ R. Here R is the set of real numbers. Let d1, d2, d3 ∈ R be the additional weights of the write operations for each category such that d3 ≤ d2 ≤ d1. Let x ∈ HS be an attribute that is accessed in a read operation. Then the weight given to x is w1. If it is accessed in the write operation, then the weight given to x is w1 + d1.

Table-5.3 Bank database schema

Table Name	Column Name (Integer Encoding of Attributes)
Customer	Name (9), Customer_id (10), Address (4), Phone_no (1)
Account	Account_id (2), Customer_id (3), Status (7), Open_dt (5), Close_dt (6), Balance (8)
Account_type	Account_type (11), Max_tran_per_month, (13), Description (12)

Table-5.4 Weight table for the attributes used in the bank database

Sensitivity Group	Attributes	Weights
HS	7, 8, 13	3
MS	5, 6	2
LS	1, 2, 3, 4, 9, 10, 11, 12	1

Let 3, 2 and 1 be the weights of HS, MS and LS, respectively and 0.75, 0.50, 0.25 be the additional weights of write operation for HS, MS and LS. Let us say that there are attributes a1, a2, a3, a4, a5, where a1, a3 ∈ HS, a2 ∈ MS and a4, a5 ∈ LS, and we have following four sequences

(i) <a1r ,a3r ,a2w ,a4w> (ii) <a1r ,a3w ,a4w>
(iii) <a5r ,a2w ,a4r> (iv) <a4r ,a5w>

In sequence (i) the most sensitive attributes are a1, a3, which are in HS. Hence, the weight of this sequence is 3. Sequence (ii) contains the same set of most sensitive attributes as (i) but since in this sequence a3 has been present with writing operation, it is assigned a weight of 3 + 0.75. In the third sequence, the most sensitive attribute is a2, which is in MS and it is with write operation. So, the sequence (iii) gets a weight of 2 + 0.50. The last sequence contains sensitive attributes a4, a5, which are in the LS and a5 is in writing operation. Hence, it gets a weight of 1 + 0.25. The weights are normalized so that they add up to unity. The weights assigned to the sequences, used to calculate the support of each sequence in the transaction, are required in the second pruning step. Let there be a sequence s with weight Ws. Let N be the total number transactions. If s is present in n out of N transactions, then the support of sequence s is: Support(s) = (n * Ws) / N.

VI. CONCLUSION

Arranging of items within the transaction has an effect on the size of prefix tree. Hence we conclude that arranging the items according to the deceasing initial support value reduces the size of prefix tree. It considerably reduces the total number of branches and nodes in the prefix tree and the effect of this reduction reflect in the total memory utilisation of the prefix tree. The smaller prefix tree takes less time in searching for any particular node and also makes it's handling easier. The main fact behind the reduction in size is that items with higher initial support value have more probability of occurring in the intersection. Arranging the items in decreasing order of support value allow the higher support item to exist at the first level in prefix tree. These higher support items exist more often in intersection while we search for intersecting items in the tree they can be found at the first level of the prefix tree thus it prevents extra branches to be added in the tree and thus the number of branches in the prefix tree reduces.

REFERENCES

[1]R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo, “Fast discovery of association rules”, In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, Advances in Knowledge Discovery and DataMining, pages 307{328. AAAI Press / MIT Press, Cambridge, CA, USA, 1996

[2]R. Agrawal and R. Srikant, “Fast algorithms for mining association rules”, In Proc.20th Int. Conf. on Very Large Databases (VLDB 1994, Santiago de Chile), pages 487-499, San Mateo, CA, USA, 1994. Morgan Kaufmann.

[3]B. Chandra, S. Bhaskar, “A Novel Approach for Finding Frequent Itemsets in Data Stream”, Int. J. Intell. Sys., 28(3), pp. 217-241, 2013

[4]B. Chandra, S. Bhaskar, “A novel approach for finding frequent itemmsets in high speed data streams”, FSKD 2011, pp. 40-44

[5]C. Borgelt, X. Yang, R. Nogales-Cadenas, P. Carmona-Saez, A. Pascual-Montano, “Finding Closed Frequent Itemsets by Intersecting Transactions”, EDBT/ICDT ’11 proceedings of the 14thInternational Conference on Extending Database Technology, Pages 367-376.

[6]C. Borgelt, “An Implementation of the FP-growth Algorithm”, OSDM ’05 Proceedings of the 1st international workshop on open source datamining: Frequent pattern mining implementation, pages 1-5.

[7]R. Chi-Wing Wong, A. Wai-Chee Fu, “Association Rule Mining and its Application to MPIS”, Encyclopedia of Data Warehousing and Mining, Information Science Publishing, John Wang. United States of America and United Kingdom: Idea Group Inc, Spring 2005, pp. 65-69,Book chapter, 2005

[8]F. Pan, G. Cong, A. Tung, J. Yang, and M. Zaki. Carpenter, "Finding closed patterns in long biological datasets", In Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2003, Washington, DC), pages -642, New York, NY, USA, 2003. ACM Press

[9]E .Ramaraj, N. Venjatesan, "An efficient pattern mining analysis in health care database", Journal of Theoretical & Applied Information Technology . 5/1/2009, Vol. 5 Issue 5, p543-549

[10]C. Pacheco and M. D. Ernst, "ECLAT: Automatic generation and classification of test inputs", A.P.Black (Ed.): ECOOP 2005, LNCS 3586, pp. 504-527, 2005

[11]A. Srivastava, S. Sural1 and A. K. Majumdar," Database Intrusion Detection using Weighted Sequence Mining", Journal of computers,Vol. 1, 4 July 2006

[12]S. B. Bajaj, "ARAS: Efficient generation of Association Rules Using Antecedent Support, FSKD 2014, pp. 289-294