A New Association Rule Mining Algorithm

B. Chandra and Gaurav

Indian Institute of Technology, Delhi
Hauz Khas, New Delhi, India 110 016
bchandra104@yahoo.co.in

Abstract. A new algorithm called STAG (Stacked Graph) for association rule mining has been proposed in this paper using graph theoretic approach. A structure is built by scanning the database only once or at most twice that can be queried for varying levels of minimum support to find frequent item sets. Incremental growth is possible as and when new transactions are added to the database making it suitable for mining data streams. Transaction scanning is independent of the order of items in a transaction. Performance of this algorithm has been compared with other existing algorithms using popular datasets like the mushroom dataset, chess and connect dataset of the UCI data repository. The algorithm excels in performance when the dataset is dense.

Keywords: Association rule mining, minimum support, frequent item set, undirected graph.

1 Introduction

The problem of association rule mining introduced by Agrawal et al. [2] aims at finding frequent item sets according to a user specified minimum support and the association rules according to a user specified minimum confidence. Finding frequent item sets is computationally more expensive than finding association rules. An efficient association rule mining algorithm is highly desired for finding frequent item sets. Apriori, AprioriTID and AprioriHybrid algorithms for association rule mining were developed by Agrawal et al. [3]. All these algorithms find frequent sets in a bottom-up fashion. A combinatorial explosion of item sets occurs when the minimum support is set low amounting to a high execution time. Pincer search algorithm developed by Lin et al. [4] is a 2-way algorithm that conducts a search in both bottom-up and top-down manner. An additional overhead of maintaining the maximal frequent candidate set and maximal frequent set is involved.

FP-Tree growth algorithm developed by J.Han et al. [5] compresses the database into a conditional pattern tree and mines frequent item sets separately. This algorithm incurs an additional cost by processing items in each transaction in the order of increasing support count and heavily uses memory when the dataset is large. Charu Agrawal et al. [1] gave a method for online mining by storing item sets satisfying a minimum support threshold in the form of a directed graph. The approach does not work if the user specified minimum support

M. Ishikawa et al. (Eds.): ICONIP 2007, Part II, LNCS 4985, pp. 366–375, 2008.

is less than the minimum support threshold. Zaki et al. [6] proposed an approach for finding frequent item sets using equivalence classes and hyper graph clique clustering. Hyper graph clique clustering produces more refined candidate sets as compared to equivalence class approach but identifying cliques in a graph is an NP-Complete problem.

The work presented in this paper is a new graph based approach (using one scan and two scans of the database) for finding frequent item sets in Market Basket Data (MBD). The best feature of one scan algorithm is that it requires only one scan of the database. The one scan algorithm aims at reducing the I/O drastically whereas the two scan algorithm reduces the computational time, run time storage and I/O at the same time. The efficiency of these algorithms have been compared with other existing association rule mining algorithms using popular datasets viz. mushroom, chess and connect datasets from the UCI data repository. It has been observed that this algorithm outperforms existing algorithms in dense datasets for lower minimum support.

2 ALGORITHM STAG

A new association rule mining algorithm, STAG (Stacked Graph) has been proposed, based on graph theoretic approach. Two issues have been addressed: the first one aiming at reducing the I/O drastically and the second one to bring a reduction in computational time, run time storage and I/O at the same time. This is achieved by one-scan STAG and two-scan STAG algorithm.

STAG overcomes the difficulty of answering a very low support online query by the user, if used for OLAP purposes. In comparison to disk based algorithms like Apriori, Pincer Search algorithm; it minimizes input-output operations by scanning a database only once or at most twice and the addition of new transactions does not require re-scanning of existing transactions. Some association rule mining algorithms require the items in a transaction to be lexicographically sorted or incorporate an additional step of sorting the items according to support value but there is no such imposition on items in STAG. The order of scanning of transactions is immaterial and the items need not be sorted (using support or lexicographically). The algorithm utilizes a depth first strategy to expand the search space of potential frequent item sets. The experiments with real life data show that it performs especially well in dense datasets i.e. datasets, which have a high average number of items per transaction. The transactions in a market basket data are scanned in their natural order but in the unlikely event of this order being disrupted, a sorting procedure on the numeric transaction identifiers can be incorporated. The algorithm consists of two steps: Building a graph structure (undirected weighted acyclic or cyclic graph with or without self loops) by scanning the transactions in the database and utilizing this structure in the second step to find frequent item sets, without scanning the database again.

2.1 Structure Building

Market basket data (MBD) is represented in the form of a graph denoted by $G(V, E)$ where V = vertex set and E = edge set. The vertex set V is defined as the set of all items occurring in the database i.e. If $I = \{i_1, i_2, \ldots i_n\}$ is the universe of items in a database where i_j is the j^{th} item then $V = I$ and number of vertices, $|V| = n$. The structure building starts by creating a node labeled i, for all $i \in V$. An edge $X \rightarrow Y$, marked with the TID t is added to E if two items X and Y co-occur in transaction t. Such edges are called marked edges. Each transaction is scanned starting with the first item present in it and its occurrence with other items in the same transaction is considered to generate marked edges between the corresponding nodes. The structure building has been illustrated using market basket data [4] in horizontal format as shown in Table 1. The Boolean format is given in Table 2. The MBD consists of four transactions i.e. the set of transaction identifiers $T = \{1, 2, 3, 4\}$ and the universe of items, $I = \{1, 2, 3, 4, 5\}$. Considering $V = I$ and $|V| = 5$. For each item $i \in V$, the first step is to create a node labeled i. Figure 1 gives the algorithm for building

Table 1. Market basket data (MBD)

TID	Items
1	1 2 3 4 5
2	1 3
3	1 2
4	1 2 3 4

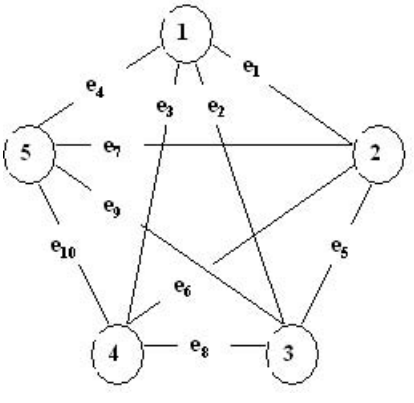
Table 2. Equivalent Boolean format of MBD

TID	Item1	Item2	Item3	Item4	Item5
1	1	1	1	1	1
2	1	0	1	0	0
3	1	1	0	0	0
4	1	1	1	1	0

the STAG structure. It assumes a Boolean horizontal format of the MBD. Function **BuildNode( )** creates the nodes of the structure by taking the vertex set as its argument. Function **MainFillStructure( )** creates the edges by considering the combinations of items present in the transaction and by passing them and the transaction identifier as parameters to function **AddToTransactionList( )**. $t(i)$ is a Boolean value indicating whether item i is present in transaction t or not. **Counter** $[i][j]$ gives the support of item i with item j and **TransactionList**$[i][j]$ gives the transaction identifiers common to items i and j. The structure is obtained as shown in Fig. 2.

```
BuildNode (VertexSet V)
  {
        for each item i ∈ V, create a node labeled i.
  }
MainFillStructure (Database D)
  {
   for each transaction t ∈ T
     for each item i in t such that t(i) = 1 and i = n-1
       for each item j such that t(j) = 1 and i = j = n
        AddToTransactionList(i, j, t)
  }
AddToTransactionList (item i, item j, TID t);
  {
     if item i = item j
        Counter[i] [i] ++
     else
        Counter[i] [j] ++ and Counter[j] [i] ++
        TransactionList[i] [j] = (TransactionList[i] [j]) ∪ t
  }
```

Fig. 1. STAG structure building algorithm



e_1	1, 3, 4
e_2	1, 2, 4
e_3	1, 4
e_4	1
e_5	1, 4
e_6	1, 4
e_7	1
e_8	1, 4
e_9	1
e_{10}	1

Fig. 2. Complete STAG structure

A transaction containing a single item does not contribute to the edge set. Such transactions produce self-loops in the graph structure and contribute only towards increasing the support count of an item.

The support count of an item can be found by taking the union of the list of transactions on the edges touching it and adding the transaction numbers contained in the self loop. For example the support count of item 1 is the union of $\{1, 3, 4\}$, $\{1, 2, 4\}$ $\{1, 4\}$ *and* $\{1\}$. The resulting set obtained has four transaction identifiers $\{1, 2, 3, 4\}$. Since item 1 has no self-loops, the final set has four TIDs,

which is the support count of item 1. After building the complete structure we proceed to find the frequent item sets.

2.2 Finding Frequent Item Sets

The algorithm for finding frequent item sets is shown in Fig. 3. It uses a depth first traversal as opposed to a breadth first traversal (used by Apriori, Pincer search algorithm) to find frequent item sets. Stacks facilitate the depth first traversal by storing intermediate particulars like generating item, intersection lists and the large item sets. The following notation is being used in the algorithm. **item-set[item_num][gen_item][]** gives the frequent item set being generated by the item **item_num** using the generating item **gen_item**. IntersectList holds the transaction identifiers resulting from the intersection of transaction lists. The notation $n(x)$ where x is a set gives the number of elements in the set x e.g. **n(IntersectList)** gives the elements in the current intersection list. **minsupp** is the user defined minimum support. The three stacks **S1, S2 and S3** are used for storing the generating item, intersection list and frequent item set respectively. The function **ItemsetGeneration( )** starts by searching for an item i such that **counter**$[i][i]$ is greater than or equal to minsupp . The large item set being generated by item_num with item j is denoted by **item-set [item_num]** $[j]$ []. While locating i and j the intersection list remains null (does not contain any transaction identifiers). Next search for an item $k > j$ such that k is not visited from j and n (TransactionList $[j]$ $[k] \cap$ IntersectList) $\geq$ minsupp. If item k is added to the list of large item set it is termed as a "successful traversal". On a successful traversal (except to the n^{th} item) it is required to store item j in a stack since there might be some item $l > k$ such that $\{i, j, l \ldots\}$ is also a large item set but $\{i, j, k, l \ldots\}$ is not. After scanning the last item , pop the particulars from the three stacks into the appropriate data structures, if the stacks are non-empty. The process is repeated with the popped items and stops when there is no item left to pop. After emptying the stacks, the item next to 'i' is considered. i.e. The algorithm finds an item p such that **counter** $[p]$ $[p]$ is greater than or equal to minsupp and sets i equal to p.

Working of the proposed algorithm has been illustrated on item1 using Fig. 2 and minsupp equal to two in Tables 3 to 6. The following notation is used: $X \rightarrow Y$ denotes an edge from item X to item Y, TL (Transaction List), IL (Intersection List), R (Result = TL $\cap$ IL), LI (Large item set), S1 (Stack for the generating item), S2 (Stack for the Intersection List) and S3 (Stack for the large item set). Start with item 1 which has a support of four.

Table 3.

$X \rightarrow Y$	TL	IL	R	L1	S1	S2	S3
$1 \rightarrow 2$	{1, 3, 4}	NULL	{1, 3, 4}	{1, 2}	NULL	NULL	NULL
$2 \rightarrow 3$	{1, 4}	{1, 3, 4}	{1, 4}	{1,2,3}	2	{1, 3, 4}	{1, 2}
$3 \rightarrow 4$	{1, 4}	{1, 4}	{1, 4}	{1, 2, 3, 4}	3	{1, 4}	{1,2,3}
$4 \rightarrow 5$	{1}	{1, 4}	{1}	{1, 2, 3, 4}	-same-	-same-	-same-

```
ItemsetGeneration (Counter, TransactionList, minsupp)
 {
    item_num = 1
    While (item_num ≤n-1)
      {
        If counter [item_num] [item_num] = minsupp)
         {
              Find min j > item_num such that counter [item_num] [j] = minsupp && j
              not visited from item_num

              item-set [item_num] [j] [ ] = item_num ∪ j;
              IntersectList = TransactionList [item_num] [j]
              gen_item = j
Label 1:  While (j < n)
            Find min k > j such that
counter[j] [k] = minsupp && n (IntersectList ∩TransactionList[j] [k]) = minsupp && k ≤n && k is not
visited from j
              If (k<n)
                Push (S1, gen_item)
                     Push (S2, IntersectList)
                Push (S3, itemset [item_num] [gen_item] [ ])
                IntersectList = IntersectList ∩ TransactionList [j] [k]
            item-set [item_num] [gen_item] [] = Itemset [item_num] [gen_item] [ ]∪k
              j = k
            }
          If (j = n or no k found)
           {
             If ( stack is nonempty)
              {
                  j = Pop (S1)
                  IntersectList = Pop (S2)
                  Itemset [item_num] [gen_item] [ ] = Pop (S3)
               Goto Label 1;
             }
                  }
        Item_num++
        }
              large_itemset [item_num] [gen_item] [] = max (n (itemset [item_num] [gen_item] [ ] ))
  }
```

Fig. 3. Algorithm for finding frequent item sets

Since 5 is the last item, the process of popping the stacks begins.

Table 4.

$X \rightarrow Y$	TL	IL	R	L1	S1	S2	S3
$3 \rightarrow 5$	{1}	{1, 4}	{1}	{1,2,3}	2	{1, 3, 4}	{1, 2}

No other distinct frequent item set is found with item 3. Since item 5 is the last item the stacks are popped.

Table 5.

$X \rightarrow Y$	TL	IL	R	L1	S1	S2	S3
$2 \rightarrow 4$	{1, 4}	{1, 3, 4}	{1, 4}	{1, 2, 4}	2	{1, 3, 4}	{1, 2}
$4 \rightarrow 5$	{1}	{1, 4}	{1}	{1, 2, 4}	-same-	-same-	-same-

Since traversal from item 4 to 5 is successful, we again push item 2 in the stack S1.

Table 6.

$X \rightarrow Y$	TL	IL	R	L1	S1	S2	S3
$2 \rightarrow 5$	{1}	{1, 3, 4}	{1}	{1, 2}	NULL	NULL	NULL

The search tree induced by the above example is shown in the Fig. 4: The large item sets found till this point are $\{1, 2\}$, $\{1, 2, 4\}$ *and* $\{1, 2, 3, 4\}$. The algorithm further continues by considering the edge from item $1 \rightarrow 3$, $1 \rightarrow 4$ and $1 \rightarrow 5$. After fully inspecting item 1 the algorithm starts with edges starting with item 2.The largest frequent item set found with item 1 is a 4-item set viz. $\{1, 2, 3, 4\}$.

2.3 One-Scan and Two-Scan Strategies

The algorithm described is called one-scan algorithm since it makes only one pass over the database. One-scan does not take into account the *minsupp* for building the structure. It builds the structure first and then utilizes the *minsupp* for finding the frequent item sets. The elements of vertex set in one-scan is the same as the universe of items i.e. $V = I$ and $|V| = n$. In order to reduce the space and execution time further, we introduce a two-scan algorithm which makes two-passes over the database. The two-scan algorithm first identifies the items that satisfy the *minsupp* by counting the support of 1-item sets from the database (the first pass) and then uses only those items in the vertex set to build the structure. For the two-scan algorithm the vertex set $V \subseteq I$ and $|V| \leq n$. The

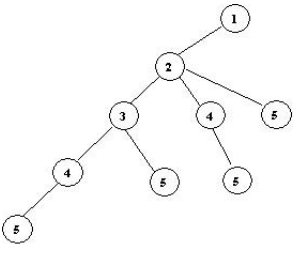
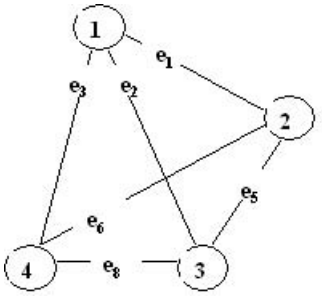


Fig. 4. Search Tree

second pass over the database is used to create the structure using the nodes obtained in the first pass. Hence the two-scan algorithm utilizes the *minsupp* to create the structure and builds a new structure for each different minimum support. Due to reduction in the number of nodes and associated overhead, it performs better than the one-scan algorithm in terms of computational time and run-time storage requirement. Figure 5 shows the structure of STAG using the two-scan strategy with minimum support equal to two.



e_1	1, 3, 4
e_2	1, 2, 4
e_3	1, 4
e_5	1, 4
e_6	1, 4
e_8	1, 4

Fig. 5. STAG structure using two-scan strategy

2.4 Early Stopping Criterion for Intersection of Transaction Lists

This section deals with the early stopping criterion for intersection of transaction lists. Let $\{a_1, a_2, \ldots, a_M\}$ denote the current intersection list IL and $\{b_1, b_2, \ldots, b_N\}$ denote the transaction list of item j with item k TL$[j][k]$,β be the minimum support for $a_i < a_j$ *and* $b_i < b_j \forall\ i < j$.

In the process of finding the intersection of IL and TL$[j][k]$, let the number of common transactions found till the current point be C and the number of transaction identifiers in TL be N. Assume that C common transactions have been found after examining bt (the tth element in TL). The intersection process is stopped if $C + (N - t) < \beta$. This essentially means that if the sum of number of common transactions (C) found till the t^{th} transaction and the transaction identifiers remaining in TL i.e. $(N-t)$) is less than minsupp there is no possibility of item k being added to the large item set generated by item i with j.

3 Results

The performance of STAG (one-scan and two-scan) was compared with Apriori, Pincer search and FP-Tree growth algorithm. Comparison of performance was made by finding frequent item sets on three popular datasets taken from the UCI data repository. All experiments were performed on a system having the following specifications: Speed: 2.66GHz, Pentium 4 Memory: 512MB RAM Operating system: Mandrake Linux 9.2

Table 7, 8 and 9 give the execution time for finding frequent item sets using various algorithms for Chess, Musroom and Connect dataset. In Table 7 and 9 * signifies that the execution time is more than one hour.
Chess Dataset: Total Transactions = 3196 Total Items = 75 All the other

Table 7.

Minsupp	1- Scan	2- Scan	FP-Growth	Apriori	Pincer Search
3000	1	0	0.24	*	*
2000	11	8	1.58	*	*
1000	26	26	136.17	*	*

algorithms except STAG and FP-Tree growth perform considerably slower on this dense dataset. The observations show the effectiveness of STAG in dense databases as the minimum support decreases.
Mushroom Dataset: Total Transactions = 8124 Total Items = 119 Mushroom

Table 8.

Minsupp	1- Scan	2- Scan	Apriori	Pincer Search	FP-Growth
7000	1	1	4	5	0.34
6000	0	0	5	5	0.35
5000	0	0	6	6	0.36
4000	1	1	13	14	0.43
3000	4	2	56	61	0.49
2000	9	6	361	376	0.59

data set is a sparse dataset with few items per transaction. The execution time of one-scan and two-scan show that they are faster than Apriori and Pincer search algorithm but not with respect to FP-tree growth. However in the case of dense datasets like Connect which is shown below, the one-scan and two scan algorithms outperform.
Connect Dataset: Total Transactions = 5000 Total Items = 127

Table 9.

Minsupp	1- Scan	2- Scan	FP-Growth	Apriori	Pincer Search
4000	43	38	10.88	*	*
3000	68	61	143.35	*	*

Connect dataset is more dense than the chess dataset and it is seen from Table one scan and two scan algorithms outperform Apriori and Pincer search algorithms and performs better than FP-Tree growth algorithm for lower minimum support.

4 Conclusion

A new algorithm STAG, for finding frequent item sets in market basket data has been proposed in this paper. The most redeeming feature of this algorithm is that it outperforms all other existing algorithms when the dataset is highly dense. The one-scan strategy scans the database only once but requires a greater amount of memory compared to two-scan strategy. The two-scan strategy performs better than one-scan with respect to computational time and memory. Both the strategies have no imposition on the order of scanning items within transactions or transactions in a database and require very low I/O. The execution time is low in dense datasets that makes them suitable for data mining applications in a memory constrained environment.

References

1. Aggarwal, C.C., Yu, P.S.: Online Generation of Association Rules. In: ICDE Conference (1998)
2. Agrawal, R., Imielinski, T., Srikant, R.: Mining associaton rules between sets of items in large databases. In: SIGMOD (May 1993)
3. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In: Proc. of the 20th Int'l Conf. on Very Large Databases (VLDB 1994), Santiago, Chile (June 1994)
4. Lin, D., Kedem, Z.M.: Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Set. In: Proc. of the Sixth European Conf. on Extending Database Technology (September 1997)
5. Han, J., Pei, J., Yin, Y.: Mining frequent Patterns without Candidate Generation. In: ACM-SIGMOD, Dallas (2000)
6. Zaki, M.J., Parthasarthy, S., Ogihara, M., Li, W.: New Algorithms for Fast Discovery of Association Rules. In: Proc. of the 3rd Int'l Conf. on KDD and Data Mining (KDD 1997), Newport Beach California (August 1997), `http://kdd.ics.uci.edu/`