International Journal of Recent Trends in Engineering, Vol. 1, No. 1, May 2009

Memory Efficient Mining of Maximal Itemsets using Order Preserving Generators

R V NATARAJ[1] and S SELVAN[2]
[1] Department of Information Technology, PSG College of Technology, Coimbatore, India.
Email: rvnataraj@mail.psgtech.ac.in
[2] Department of Computer Science, St. Peter's Engg. College, Chennai, India
Email: drselvan@ieee.org

***Abstract*— In this paper, we propose a memory efficient algorithm for maximal frequent itemset mining from transactional datasets. We propose OP-MAX* (Order Preserving – MAXimal itemset mining) algorithm, which mines all the maximal itemsets from transactional datasets with less space and time. Our methodology uses a memory efficient maximality checking technique to generate frequent maximal itemsets. We have also incorporated several optimization techniques to improve the mining efficiency. Experiments involving publicly available datasets show that our algorithm takes less memory and less computation time than other algorithms in most cases.**

***Index Terms*— Data Mining – Closed Itemsets – Maximal Itemsets - Algorithms**

I. INTRODUCTION

Frequent itemset mining is fundamental to several data mining tasks including Association Rule Mining [5][11][12], life science data analysis[1] and social network analysis[9]. The problem is stated as follows. Given a collection of transactions with each transaction consisting of set of items, a frequent itemset is a subset of a set of items that occur in at least a user specified percentage (support) of the transactions. Frequent itemset mining is a computationally demanding task and has been an active area of research in the field of data mining. Several algorithms have been proposed to mine frequent itemsets including Apriori, F-Apriori, FP-growth[15], FP-growth*[6] and Transaction Mapping Algorithm[5]. But the main drawback is "too many frequent itemsets". For example, if the itemset length is x, then 2^x frequent itemsets would be generated. To overcome the "too many itemsets" disadvantage, closed itemset concept was proposed [11]. The set of closed itemsets of the given transactional database is the condensed representation of the set of all frequent itemsets without any loss of information. A frequent itemset is said to be closed if it has no superset with the same frequency (support). Several algorithms have been proposed in the literature including A-Close, FP-Close[6], AFOPT-Close [16], B-Miner & C-Miner [1], and DCI-Close [2]. However, when there are very long itemsets present in the dataset, the generation of all the frequent closed itemsets is not trivial and it suffers from the same problem as frequent itemsets. To further condense the set of frequent closed itemsets, Maximal itemset concept was proposed. A frequent itemset is said to be maximal if none of its supersets are frequent. Several algorithms have been proposed including AFOPT-Max[16], FP-Max[6], MAFIA[7] and Eclat-Max. However, mining all the maximal itemsets from the given dataset is computationally more expensive than frequent closed itemset and frequent itemset mining. This is because there is no relationship between any two maximal itemsets and all the maximal itemsets are unique to each other, whereas all the frequent closed itemsets can be related either by subset or superset relationship with respect to each other. Moreover, it has been proven that the complexity class of maximal itemset mining is NP-hard [8]. Several algorithms for mining maximal itemsets are based on the enumeration of frequent itemsets and it outputs maximal itemsets among them. Unlike frequent itemsets, this requires the itemsets to be stored in the main memory for maximality checking and consumes much main memory.

This paper proposes OP-MAX* (Order Preserving-MAXimal Itemset Mining) algorithm to enumerate maximal frequent itemset using order preserving generators [2] with fast and memory efficient maximality checking. We have already proposed two algorithms in this area: OP-MAX [3] and OP-MAX+ [4]. OP-MAX stores all the maximal closed itemsets in main memory and generates maximal itemsets among them whereas OP-MAX+ uses closed transaction sets to enumerate maximal itemsets. OP-MAX and OP-MAX+ stores intermediate itemsets in main memory whereas OP-MAX* computes all maximal itemsets without storing intermediate results in the main memory. The rest of the paper is organized as follows. Section II gives the preliminaries, section III presents the proposed algorithm, section IV gives the experimental results and section V concludes the paper.

II. PRELIMINARIES

In the context of association rule mining, a database D is a triple $D=\{T,I,R\}$, where T and I are finite set of transactions and items respectively. $R \subseteq T \times I$ is a binary relation between transactions and items. Each pair $(t,i) \in R$ denotes the fact that the transaction $t \in T$ is related to the item $i \in I$. An itemset $X \subseteq I$ is frequent if support count of X exceeds user defined minimum support. An itemset $X \subseteq I$ is a maximal itemset if there exists no other itemset X' such that X' is frequent and $X \subseteq X'$. For more details on order preserving generators, readers may refer [2] [3] [4].

III. MAXIMAL ITEMSET MINING

OP_MAX* computes the set of maximal frequent itemsets from the set of frequent closed maximal itemsets using the *pre_set* [2][3][4] elements. It should be noted that the algorithm OP-MAX* does not store any closed or maximal itemset in the main memory and hence the algorithm is highly memory efficient. The space complexity of our algorithm is *O(D)*, where *D* is the input dataset.

A. OP-MAX* Algorithm

Let *D* denote the given dataset, Ξ denote the set of frequent closed maximal itemsets of *D* and Ψ denote the set of maximal itemsets of *D*. Then, $\Psi \subseteq \Xi$ and all closed itemsets $X \in \Xi$ are leaf nodes in the tree. It should be noted that the closed itemset mining using order preserving generators explore the closed itemset lattice in depth first manner with all of its leaf nodes as maximal closed itemsets. All the elements from the *post_set* that successfully contributed a new valid generator are stored in *pre_set* for duplicate checking. The OP-MAX* algorithm exploits this concept for fast and memory efficient maximality checking. Let I be the set of items and *X* be a frequent itemset. In maximality checking, we check whether $X \cup i$ is frequent where $i \in I \setminus X$. if $\exists i$ such that $X \cup i$ is frequent then X is not a maximal itemset and if $\forall$ i such that $X \cup i$ is frequent then *X* is a maximal itemset. In OP-MAX* algorithm, we check whether $X' \cup i$ is frequent where X' is a leaf node and $i \in pre\_set(X')$. Since we accomplish maximality checking with less number of items from the *pre_set*, our algorithm takes less running time than other algorithms. It should be noted that the number of elements present in *pre_set(X')* is usually lesser than the number of elements in $X' \setminus I$. We have also incorporated several optimization techniques and one such technique is itemset ordering which is explained in the next section.

B. Itemset Ordering

The order in which the items are processed play a vital role in reducing the amount of computation in the depth first exploration of the search space. The aim is to reduce, as soon as possible, the elements in the *post_set* which limits the depth of the search space tree and also to reduce the number of elements scanned in *pre_set* for duplicate detection. To accomplish this, we sort all the *post_set* elements in their support ascending order and the processing is carried out in the same order i.e. items with small support are processed first. Also, all the items are sorted with respect to its support and are remapped to continuous integer space starting from zero. While processing item *i*, all the elements that succeed *i* are placed in *pre_set* in their ascending order and all the elements that precede *i* are placed in the *post_set* in descending order. The advantage of ordering all the elements in the *pre_set* in their support descending order and all the elements in the *post_set* in their support ascending order is two fold. Firstly, it reduces the computations needed for duplicate detection and secondly, it reduces the computations in computing closures as explained below. Order preserving algorithm builds the generator by adding various elements from *post_set* to *closed_set* and the validity of the generator is tested using elements in the *pre_set.* If the generator satisfies minimum support and is not a subset of any of the elements in the *pre_set* then the generator will lead to a closed itemset. By ordering the *post_set* elements in the support ascending order, we reduce the *post_set* items to the maximum extent in every closure computation and this limits the depth of the recursive tree. Also, since the items in the *post_set* are reduced to minimum, it saves the computation in every level of the search space tree. Note that, all the elements in the *post_set* are processed for subset checking with *g(closed_set)* for computing the closure. By ordering the elements of *pre_set* in their support descending order, we considerably reduce the total number of elements processed for duplicate detection. The concept is based on the following fact: "a generator of the closed itemset is more likely to be the subset of items with high support than the items with low support". The ordering of elements in the *post_set* in its support ascending order not only save the amount of computation involved in computing the closure but also reduces the items processed from *pre_set* in detecting duplicate generators.

C. OP-MAX* Pseudocode

This section presents OP-MAX* pseudocode. The algorithm takes two parameters as input: transactional dataset, and the minimum support value and it output all the maximal itemsets that satisfy *min-support* constraint.

```
INPUT: transactional dataset, min-support
OUTPUT: all maximal itemsets satisfying support constraints
1.  Compute F1 (frequent-1 items}, sort F1 items based
    on their support and map to continuous integer space
2.  ∀ i ∈ F1
3.     pre_set = { i' ∈ F1 | i' ≠ i }
4.     closed_set = null
5.     post_set =i ∪ { i' ∈ F1 | i' ≥ i }
6.     reorder the vertical bit-vector space such that
          supporting transactions of i are consecutive
          in its bit-vector space.
7.     tid_set_c = { set of transactions supporting
    closed_set }
8.    OP-MAX*(post_set, closed_set, pre_set, tid_set_c )

9.  OP-MAX*( post_set, closed_set, pre_set, tid_set_c )
10. {
11.    while (post_set!=null)
12. z:   i''=min(post_set)
13.      tid_set_g = tid_set_c ∩ g(i'')
14.      if | tid_set_g |>min_support &&
                    (∀ j ∈ pre_set,  tid_set_g ∉ g(j))
15.         write closed_set, post_set,
                 pre_set ∪ i'', tid_set_c to stack
16.         closed_set=closed_set ∪ i''
17.         ∀ k ∈ post_set
18.            if tid_set_g ⊆ g(k)
19.               closed_set=closed_set ∪ k
20.               post_set=post_set \k
21.            endif
```

```
22.          tid_set_c = tid_set_g
23.          write closed_set to disk
24.       else
25.          if  (post_set!=null)  goto z:  endif
26.       endif
27.       if  (post_set==null && stack is not empty)
28.             if  $ r Ì pre_set  and
                    support(closed_set ∪ r) > minsupp
29.                   discard the closed_set
30.             else
31.                   output the closed_set as maximal
    itemset
32.             endif
33.             pop from stack to closed_set, preset,
                              post_set and tid_set_c
34.       endif
35.       if (closed_set==null)
36.             return
37.       endif
38.    endwhile
39. }
```

D. Pseudocode Description

The algorithm uses three sets (*pre_set, post_set and closed_set*) as used in [2]. Besides, it uses three sets *F1 and tid_set*. *F1* is used to store the set of frequent-1 items whereas *tid_set* is used to store the set of supporting transactions of the current closed itemset The algorithm uses a stack to store the necessary information for backtracking and the stack structure contains four sets (*pre_set, post_set, closed_set and tid_set)*. Whenever, an extension of closed itemset is found to be closed, the previous itemset is pushed to the stack along with its *pre_set, post_set and tid_set (*line no 15). For example, let *X* and *Y* be a closed itemset and *Y* Ê *X*. Whenever *Y* is generated, *X* will be pushed to stack. When X is popped, all other itemsets *Y′* will be generated where *Y′* ≠ *Y* and *Y′* Ê *X*. After popping from the stack, if the *closed_set* is *null*, the procedure returns. The condition indicates that all closed itemsets that start with the item *i* are generated. It should be noted that the associated *post_set* of all the leaf nodes of the depth first search tree is empty and all leaf nodes form a set of closed maximal itemsets. Since the subset of leaf nodes form a set of maximal itemsets, we incorporate maximality checking (line no 28-32) whenever we pop from the stack. Before we pop from the stack, we check whether *closed_set* ∪ *i* is frequent where *i* Ì *pre_set*. If Ø$ *i* such that closed_set ∪ *i* is frequent, we output the closed itemset as maximal itemset. The existence of *i,* such that *closed_set* ∪ *i* is frequent, indicates that there exists another closed itemset *X* and *X* is a superset to current *closed_set.* Hence the current *closed_set* is not maximal and is discarded.

IV. Experimental results and discussions

We have implemented our algorithm using C language and the code was compiled using 32-bit Microsoft Visual C++ compiler. All the experiments were conducted on Pentium 4 machine with 1 GB of main memory loaded with windows XP operating system. The executables of other algorithms were obtained from the respective authors. The description of the datasets used in our experiments is given in Table 1. Among the datasets, gazelle (also known as BMS Webview) is a real dataset derived from click-stream data and accidents dataset contains (anonymized) traffic accident data. Connect and chess datasets are mathematically structured datasets derived from their respective game types. T25I20D10k and T10I4D10k datasets are synthetic datasets generated from IBM synthetic dataset generator. The dataset generator is downloaded from Illimine project's website. For TxIyDz, x indicates the average transaction length, y indicates the average itemset length and z indicates the total number of transaction instances.

Table 1. Datasets Used

Sl.No	Dataset Name	Total Items	Total Transactions
1	gazelle	497	59601
2	accidents	468	340183
3	connect	42	8196
4	pumsb-star	7117	49046
5	chess	75	3196
6	retail	16470	88162
7	T25I20D10k	1000	10000
8	T10I4D10k	1000	10000

To obtain the accurate peak memory usage of executions, we have written our own stub code using windows process library API that will fetch the main memory usage statistics whenever a process is terminated. Since we extract the needed information from the windows kernel itself, the load made by this program on the memory and the processor is completely negligible. We have checked the running times while running this piece of code in background and while this software was not running in background. The observed differences are only in microseconds and in most cases we didn't observe any difference. The experimental results given in Table 2 show that OP-MAX* algorithm takes less running time than other algorithms in most cases. Similarly, Table 3 and Table 7 show the total running time taken for accident and pumsb-star dataset respectively.

Table 2: Total Running time taken for gazelle dataset

Supp-ort	Running time in Seconds (gazelle)			
	FP-MAX	AFOPT	OP-MAX*	MAFIA
59	0.796	2.312	0.609	4.453
54	0.813	2.375	0.75	4.625
48	0.828	2.469	1.031	4.859
42	0.906	2.672	1.781	5.328
36	1.312	3.5	4.14	5.828
30	2.61	6.656	5.719	10.312
24	1.297	13.641	6.907	14.985
18	2.39	37.812	12.016	33.922
12	3.141	43.484	32.422	52.250
6	33.609	102.609	59.843	65.734
1	573.031	1109.094	29.407	507.48

Table 3: Total Running time for accidents dataset

Supp-ort	Running time in Seconds (accidents)			
	FP-MAX	AFOPT	OP-MAX*	ECLAT-MAX
75	32.906	49.313	5.172	22.891
70	32.953	49.484	6.766	22.937
65	33.031	49.703	12.14	23.078
60	33.172	50.047	24.188	23.234
55	33.234	50.297	30.109	23.5
50	33.375	50.906	53.156	23.875

The data given in table 4 shows the peak main memory usage of different algorithms and the experimental data clearly shows that OP-MAX* takes less amount of main memory for its execution. Table 5 shows the peak page file usage data and OP-MAX* clearly outperforms its competitors. The dataset connect is highly dense and from the results given in Table 6, our algorithm enumerates all maximal itemsets with very less memory. The other algorithms FP-MAX, LCM-MAX and MAFIA takes 2 times, 3 times, and 18 times more memory than taken by OP-MAX* algorithm respectively for a support value of 70. Table 7 and Table 8 show the results for pumsb_star dataset while Table 9 presents the results of chess dataset. Similaly, Table 10, Table 12 and Table 13 show the results obtained from synthetic datasets.

Table 4: Peak Main memory usage for accidents dataset.

Supp-ort	Peak Main-Memory usage in Bytes			
	FP-MAX	AFOPT	OP-MAX*	Eclat
59	2109440	2109440	1597440	53608448
54	2183168	2260992	1724416	53604352
48	2355200	2703360	1851392	53624832
42	3084288	4222976	1978368	53751808
36	3665920	5140480	2060288	53792768
30	5218304	8466432	2150400	53878784

Table 5: peak page file usage for accidents dataset

Supp-ort	Peak Page File usage in Bytes			
	FP-MAX	AFOPT	OP-MAX*	Eclat
59	2453504	1634304	1253376	53825536
54	2453504	1830912	1388544	53825536
48	2453504	2224128	1523712	53825536
42	3043328	3776512	1658880	53891072
36	4157440	4694016	1748992	53956608
30	6389760	8036352	1839104	54022144

Table 6: Peak Memory usage for connect dataset

Supp-ort	Peak memory usage in Bytes			
	FP-MAX	LCM	OP-MAX*	Mafia
70	2150400	12226560	933888	17510400
65	2203648	12730368	950272	17551360
60	2293760	13512704	978944	17608704
55	2375680	13783040	995328	17625088
50	2506752	14176256	1003520	17645568

Table 7: Total Running time for pumsb-star dataset

Supp-ort	Running time in Seconds			
	FP-MAX	AFOPT	OP-MAX*	Mafia
55	9.687	12.219	1.39	34.781
50	9.688	12.266	1.703	34.844
45	9.828	12.39	2.813	34.937
40	9.859	12.547	4.828	35.031
35	9.938	12.797	9.25	35.156

Table 8: Peak Memory usage for pumsb_star dataset

Supp-ort	Memory usage in Bytes			
	FP-MAX	AFOPT	OP-MAX*	Mafia
55	2347008	2138112	843776	17186816
50	2670592	2293760	892928	17297408
45	3403776	2781184	970752	17448960
40	4149248	3379200	1028096	17559552
35	5173248	4612096	1089536	17686528

Table 9: Peak Memory usage for Chess dataset

Supp-ort	Memory Usage in Bytes			
	OP-MAX*	LCM-MAX	FP-MAX	Mafia
959	602112	4091904	10166272	21082112
799	602112	4104192	17608704	38432768
639	602112	4173824	33026048	74416128
479	602112	4308992	71979008	1.59E+08
319	618496	4304896	175685632	3.42E+08

Table 10: Total running time for T25I20D10K dataset

Supp-ort	Running Time in Seconds			
	OP-MAX*	ECLAT MAX	AFOPT -MAX	FP-MAX
50	5.594	6.578	12.641	11.141
40	7.156	9.453	13.094	11.25
30	10.828	10.485	15.14	16.485
20	23.094	11.765	12.828	9.015
10	155.062	435.688	26.719	16.5

Table 11: Peak Memory Usage for T25I20D10k dataset

Supp-ort	Memory Usage T25I20D10k			
	OP-MAX*	ECLAT MAX	AFOPT -MAX	FP-MAX
50	1568768	4739072	3960832	8486912
40	1605632	4857856	4370432	8622080
30	1650688	4935680	5210112	8815592
20	1695744	5570560	6959104	9043968
10	1748992	9912320	19210240	10031104

Table 12: Total running time for T10I4D10k dataset

Supp-ort	Running Time in Seconds			
	OP-MAX*	ECLAT MAX	AFOPT -MAX	Mafia
5	3.468	1.344	1.906	5.203
4	4.422	1.453	2.188	9.101
3	6.765	1.765	2.703	23.115
2	15.047	3.516	4.188	29.26
1	7.953	2249.188	13.203	404.985

Table 13: Peak main memory usage for retail dataset

Supp-ort	Memory Usage in Bytes			
	OP-MAX*	ECLAT MAX	AFOPT -MAX	FP-MAX
793	1835008	9728000	3969024	3436544
705	1998848	9834496	4063232	3629056
617	2289664	9900032	4288512	3969024
528	2719744	10100736	4526080	4493312

440	3346432	10006528	4780032	5079040
352	4448256	10321920	5103616	6082560

Table 14: Total running time taken for retail dataset

Supp-ort	Time RETAIL			
	OP-MAX*	ECLAT MAX	AFOPT -MAX	FP-MAX
793	0.953	2.766	6.969	3.906
705	1.078	2.781	6.984	3.844
617	1.297	2.859	7.032	3.953
528	1.656	2.969	7.078	4
440	2.438	3.156	7.109	3.938
352	3.844	3.485	7.141	3.922

V. CONCLUSION

We have proposed OP-MAX*, a fast and memory efficient algorithm for mining maximal frequent itemsets. Though our algorithm takes less memory than other algorithms, the problem that OP-MAX* takes more running time for some cases, is yet to be solved. Currently, we are investigating other optimization techniques to further reduce the running time of the algorithm.

ACKNOWLEDGMENT

We wish to thank C. Luchesse and S. Orlando for responding to our queries. We also thank Prof. Bart Goethls for providing us the datasets and for supporting the FIMI website. We thank the authors of MAFIA for responding to our queries.

REFERENCES

[1] Liping Ji, Kian-Lee Tan,K H. Tung, "Compressed Hierarchical Mining of Frequent Closed Patterns from Dense Data Sets," IEEE Trans. on Knowledge and Data Engineering, Vol 19, No.9, Sept 2007.

[2] C. Lucchese,S. Orlando and R. Perego, "Fast and Memory Efficient Mining of Frequent Closed Itemsets", IEEE Transactions on Knowledge and Data Engineering, VOL 18, No 1, pages 21-36, January 2006.

[3] S. Selvan and R V Nataraj, "Efficient Mining of Maximal Patterns using Order Preserving Generators", in proc. 16th Intl. Conf. on Advanced Computing and Communications, Chennai, India, Dec. 2008

[4] S. Selvan and R V Nataraj, "Maximal Itemsets Mining using Closed Transaction Sets", Under Review for publication.

[5] Mingjun Song, Sanguthevar Rajasekaran, "A Transaction Mapping Algorithm for Frequent Itemsets Mining", IEEE Transactions on Knowledge and Data Engineering,, VOL 18, No 4, pages 472-481, April 2006.

[6] G. Grahne, J. Zhu, "Fast Algorithms for Frequent Itemset Mining Using FP-Trees", IEEE Transactions on Knowledge and Data Engineering, Vol 17, No 10, pages 1347-1362, October 2005.

[7] D. Burdick, M.Calimlim ,J.Flannick, J.Gehrke,Y.Yiu, "MAFIA: A Maximal Frequent Itemset Algorithm", IEEE Transactions on Knowledge and Data Engineering, VOL 17, No 11,Pages 1490 - 1504, November 2005.

[8] Guizhen Yang, "The complexity of Mining Maximal Frequent Itemsets and Maximal Frequent Patterns", KDD'04, Seattle, Washington, August 2004.

[9] Jinyan Li, Guimei Liu, Haiquan Li, Limsoon Wong, "Maximal Biclique Subgraphs and Closed Pattern Pairs of the Adjacency Matrix: A One-to-One Correspondence and Mining Algorithms," *IEEE Trans. Knowledge and Data Engineering*, vol. 19, No. 12, pp. 1625-1637, Dec. 2007.

[10] Dao-l Lin and Zvi M. Kedem, "PINCER-SEARCH: An Efficient Algorithm for Discovering the Maximum Frequent Set", IEEE Trans. on Knowledge and data Engineering, VOL 14, No. 3, June 2002.

[11] N. Pasquier, Y. Bastide, R. Taouil, and L.Lakhal, "Discovering Frequent Closed Itemsets for Association Rules", Proc. 7th Int. Conf. Database Theory (ICDT'99), pages 398-416, January 1999.

[12] R. Agrawal, T. Imielinski, and A. Swami. "Mining association rules between sets of items in large databases", In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pages 207-216, Washington, DC, May 1993.

[13] T. Uno, M. Kiyomi, and H. Arimura, "LCM ver.2: Efficient mining algorithms for Frequent/closed/maximal itemsets," In Proc. IEEE ICDM'04 Workshop FIMI'04, 2004.

[14] K. Gouda, M.J.Zaki, "GenMax: An Efficient Algorithm for Mining Maximal Frequent Itemsets", Journal of Data Mining and Knowledge Discovery, pages 1-20, 2005

[15] Jiawei Han, Jian Pei, Yiwen Yin, Runying Mao, "Mining Frequent Pattern without candidate Generation : A Frequent Pattern Approach" Journal of Data Mining and Knowledge Discovery , Springer, pages 53-87, 2004.

[16] G.Liu, "Supporting Efficient and Scalable Frequent Pattern Min-ing," PhD dissertation, Dept. of Computer Science., Hong Kong University., May 2005.