Maximal Frequent Itemset Mining Algorithm Based on B-list (Postprint)

Authors: Zhang Chang, Wen Kai, Zheng Yunjun

Date: 2018-05-20T00:00:00+00:00

Abstract

To address the issues of excessive mining time and high memory consumption in existing maximum frequent itemset mining algorithms, this paper proposes BMFI, a maximum frequent itemset mining algorithm based on a constructed linked list B-list. The algorithm leverages the B-list data structure for frequent itemset mining, employs a total-order search tree as the search space, utilizes parent-equivalence pruning to reduce the search space, and integrates an MFI-tree-based projection strategy for superset detection to enhance efficiency. Experimental results demonstrate that BMFI outperforms both FPMAX and MFIN in terms of time and space efficiency. The algorithm exhibits favorable performance for maximum frequent itemset mining on both dense and sparse datasets.

Full Text

Title and Authors

Title: Maximal Frequent Itemset Mining Algorithm Based on B-list

Authors: Zhang Chang[1]†, Wen Kai[1],[2], Zheng Yunjun[1]

Affiliations: 1. Institute of Applied Communication Technology, Chongqing University of Posts & Telecommunications, Chongqing 400065, China 2. Chongqing Information Technology Designing Co. Ltd, Chongqing 401121, China

Abstract: To address the problems of excessive mining time and high memory consumption in existing maximal frequent itemset mining algorithms, this paper proposes a maximal frequent itemset mining algorithm called BMFI that utilizes the B-list data structure. The algorithm employs a total-order search tree as its search space, adopts parent equivalence pruning techniques to reduce the search space, and combines an MFI-tree-based projection strategy for superset

detection to improve efficiency. Experimental results demonstrate that BMFI outperforms both FPMAX and MFIN algorithms in terms of time efficiency and space efficiency, achieving good performance on both dense and sparse datasets.

Keywords: maximal frequent itemset mining; depth-first search; pruning techniques; superset detection

0 Introduction

Data mining is a computational process for discovering patterns from large datasets. Classic frequent pattern mining algorithms include the Apriori algorithm proposed by Agrawal et al. [1] in 1994 and the FP-growth algorithm proposed by Han et al. [2] in 2004. Association rules are widely applied in market basket analysis, e-commerce, and medical fields. During frequent pattern mining, several practical issues arise. For instance, when mining dense datasets with low minimum support thresholds, the number of generated frequent patterns becomes extremely large. Additionally, when frequent itemsets are long, the number of their frequent non-empty subsets also grows substantially. Under such circumstances, enumerating all frequent itemsets becomes infeasible.

Maximal frequent itemsets provide a compact representation of frequent itemsets with higher mining efficiency [3]. A frequent itemset S is called a maximal frequent itemset if none of its proper supersets are frequent. In recent years, many experts and scholars have conducted in-depth research on maximal frequent itemset mining algorithms. Bayardo et al. [4] proposed the MaxMiner algorithm, which employs a breadth-first search strategy and dynamic reordering-based superset detection. Agrawal et al. [5] proposed the DepthProject algorithm, which adopts depth-first search, dynamic reordering-based superset detection, and bucket-based itemset computation. Burdick et al. [6] proposed the MAFIA algorithm using DFS strategies and pruning techniques including PEP, FHUT, MFIHUT, and Dynamic Reordering, along with an efficient LMFI superset detection strategy. Grahne et al. [7] proposed the FPMAX algorithm, which first constructs an FP-tree from the transaction database and then implements superset detection through MFI-tree-based projection strategies to enhance performance. Shen et al. [8] introduced an N-list-based maximal frequent itemset mining algorithm that leverages N-list' s high compression ratio and efficient intersection capabilities, combined with search space pruning and superset detection methods to improve efficiency, though its performance on sparse datasets requires improvement. Lin et al. [9] proposed the MFIN algorithm based on Nodeset, which uses POC-tree structure for node encoding, total-order search space, and parent equivalence pruning techniques to improve efficiency, but still faces issues of long runtime and high memory consumption for large datasets. In 2015, Deng et al. [11] proposed the PrePost+ algorithm based on N-list, which utilizes child-parent equivalence pruning and demonstrates better performance than FIN and PrePost algorithms for frequent itemset mining.

To address the problems of excessive mining time and high memory consumption

in existing maximal frequent itemset mining algorithms, this paper proposes the BMFI algorithm based on the B-list [12] data structure. Building upon depth-first search, BMFI leverages B-list' s high compression ratio, efficient intersection operations, and fast support counting capabilities, combined with parent equivalence pruning to reduce the search space. Experimental results on multiple datasets demonstrate that BMFI achieves significant improvements in both time efficiency and space consumption compared to FPMAX and MFIN algorithms, with more pronounced optimization effects on dense datasets.

1.1 Basic Concepts

Let I = $\{i_1, i_2, \cdots, i\}$ be a set of n distinct items. A set containing m ($0 \leq m \leq n$) items is called an itemset.

Definition 1: An itemset X is a frequent itemset if its support is greater than or equal to the user-specified minimum support threshold (minSup).

Definition 2: A frequent itemset X is a maximal frequent itemset if none of its supersets Y are frequent.

shows a sample transaction database DB. [Figure 1: see original paper] illustrates the corresponding TB-tree for DB with minSup=0.4.

Algorithm 1: TB-tree Construction Input: Transaction database DB **Output:** TB-tree, L1

The algorithm scans transaction database DB to obtain frequent 1-itemsets L_1 and sorts items in DB by descending support. It creates a root node and initializes global variables start = 0 and finish = 0. For each distinct first item p in DB, it calls BuildTree(p, Node).

```
function buildTree(p, parent)
    Let TP be a list of transactions in DB which contain prefix p
    Create node N:
        N.item-name = item-name of the last item in p
        N.count = count of transactions in TP
        N.parent = parent
        N.start-build = ++start

    for each different first item q in TP do
        Call buildTree(p   q, N)
    end for

    N.finish-build = ++finish
end function
```

1.2 TB-tree

Existing frequent itemset mining algorithms suffer from complex tree construction and low efficiency issues with structures like PPC-tree [13] and POC-tree [10]. This paper introduces the TB-tree structure [12], which consists of a root node and item prefix subtrees. Each node in the prefix subtree comprises five components: item-name, count, parent-pointer, start-build, and finish-build.

While TB-tree' s structure is similar to PPC-tree, PPC-tree requires both pre-order and post-order traversals to obtain each node' s pre-order and post-order information. POC-tree, though more concise than PPC-tree as it needs only one type of encoding (pre-order or post-order), still requires a traversal to acquire node information. In contrast, TB-tree obtains each node' s start-build and finish-build values during the tree construction process itself—node information is available immediately upon tree completion, making TB-tree construction more time-efficient than PPC-tree and POC-tree.

As shown in , the BMFI algorithm first scans the transaction dataset, sorts transactions by descending support, and removes transactions with support below minSup. It then constructs the TB-tree from the sorted transaction set. The advantage of this construction method is that node information is obtained during the building process without requiring additional traversals. Detailed construction procedures are available in reference [12]. The TB-tree construction process is presented in Algorithm 1, and [Figure 1: see original paper] shows the completed TB-tree for database DB.

1.3 B-list Structure

Definition 3: B-info-code. In the TB-tree, each node N has a B-info-code represented as [(N.start-build, N.finish-build), sup(N)], where start-build and finish-build denote the node' s construction start and end positions.

Property 1: For two nodes N_1 and N_2, N_1 is an ancestor of N_2 if and only if N_1.start-build $<$ N_2.start-build and N_1.finish-build $>$ N_2.finish-build both hold.

Example 1: In the TB-tree shown in [Figure 1: see original paper], nodes B and A have B-info-codes B_b = {[(4,11),4]} and B_c = {[(7,10),3]}. Since B_b.start-build $<$ B_c.start-build ($4 < 7$) and B_b.finish-build $>$ B_c.finish-build ($11 > 10$), b is an ancestor of a. Any two nodes can determine their ancestor-descendant relationship using this property based on start-build and finish-build information.

Definition 4: B-list for 1-itemsets. For a constructed TB-tree and node N, the B-list of node N, denoted as BL_N, is the set of all B-info-codes for nodes named N.

Example 2: In the TB-tree shown in [Figure 1: see original paper], the B-list of node B is BL_B = {[(2,1),1], [(7,4),2]}.

Definition 5: B-list for 2-itemsets. For a 2-itemset $\{i_1,i_2\}$ where i_1 precedes i_2, let the B-list of i_1 be $\{[(s_{11},f_{11}),c_{11}], [(s_{12},f_{12}),c_{12}], \cdots, [(s_1 ,f_1),c_1]\}$ and the B-list of i_2 be $\{[(s_{21},f_{21}),c_{21}], [(s_{22},f_{22}),c_{22}], \cdots, [(s_2 ,f_2),c_2]\}$. The B-list for $\{i_1,i_2\}$ is generated following these rules: a) If $[(s_1 ,f_1),c_1]$ $(1 \leq i \leq n)$ is an ancestor of $[(s_2 ,f_2),c_2]$ $(1 \leq j \leq m)$, add $[(s_1 ,f_1),c_2]$ to the B-list of $\{i_1,i_2\}$. b) For nodes in $\{i_1,i_2\}$' s B-list with identical (s,f) pairs, their supports can be merged by summation: $[(s,f), c_1+c_2+\cdots+c]$.

Definition 6: B-list for k-itemsets (k $\geq$ 3). Given two (k-1)-itemsets N = $i_1i_2\cdots$i_{k-1} and M = $i_1i_2\cdots$i_{k-1}, with B-lists $\{[(s_{11},f_{11}),c_{11}], \cdots, [(s_1 ,f_1),c_1]\}$ and $\{[(s_{21},f_{21}),c_{21}], \cdots, [(s_2 ,f_2),c_2]\}$ respectively, the k-itemset B-list is generated through intersection and merging following rules similar to the 2-itemset generation process (detailed in reference [12]). The intersection algorithm is shown in Algorithm 2.

Algorithm 2: Intersection Input: B-lists of itemsets X (BL_{cur}) and Y (BL_{HUT}), minimum support threshold **Output:** Intersection R of the two B-lists and its support count RSupport

```
Initialize i = 0, j = 0, RSupport = 0
Let C1 be the support count of BL_{cur} and C2 be the support count of BL_{HUT}
Let s = 1

While (i < BL_{cur}.size and j < BL_{HUT}.size) do
    if (BL_{cur}[i].start-build > BL_{HUT}[j].start-build) then
        if (BL_{cur}[i].finish-build < BL_{HUT}[j].finish-build) then
            if (R.size > 0 and R[R.size-1].start-build = BL_{HUT}[j].start-build) then
                R[R.size-1].support += BL_{cur}[i].support
                C2 = C2 - s * BL_{HUT}[j++].support
            else
                R $\leftarrow$ B-info-code{BL_{HUT}[j].start-build, BL_{HUT}[j].finish-build
                C2 = C2 - s * BL_{HUT}[j++].support
            end if
        else
            C1 = C1 - BL_{cur}[i++].support
        end if
    else
        C1 = C1 - BL_{cur}[i++].support
    end if

    if (C1 < threshold or C2 < threshold) then
        return NULL
    end if
end while

return R and RSupport
```

1.4 BMFI Algorithm

This section presents the BMFI (B-list-based Maximal Frequent Itemset) algorithm. BMFI first constructs the TB-tree from transaction data while simultaneously generating B-lists. It then connects two frequent (k-1)-itemsets according to B-list intersection principles to generate frequent k-itemsets and obtain their support counts. The algorithm employs a total-order search tree [6] as its search space, uses depth-first search (DFS) to traverse the entire tree, applies parent equivalence pruning to reduce the search space, and finally utilizes an MFI-tree-based projection strategy for superset detection to improve efficiency.

1.4.1 Depth-First Search Strategy Based on B-list

Assume a complete transaction database where items are arranged in lexicographic order, denoted as i ≤_L j when item i appears before item j. [Figure 2: see original paper] shows a lexicographic search tree structure constructed from four items (a ≤_L b ≤_L c ≤_L d). In this structure, each node must be unioned with every node to its right to generate the node' s subsets. BMFI generates k-itemset B-lists by intersecting two (k-1)-itemset B-lists.

Definition 7 [6]: In a total-order search tree, for node C, let C.head denote its itemset and C.tail denote the set of items that can extend it. Each item in C.tail is a 1-extension itemset of node C. The set of sibling nodes to the right of C under the same parent is denoted C.sibling$^+$.

Example 3: In [Figure 2: see original paper], assuming {b, c} represents node C, then C.head = {b, c}, C.tail = {d}, C.HUT = {b, c, d}, and C.sibling$^+$ = {{b, d}}. Here, d is called the 1-extension information of node C.

Property 2: Let X C.head and y C.tail. If the transaction set of X is identical to that of X {y}, i.e., t(X) = t(X {y}), then for all S C.tail, the support of X S equals the support of X {y} S.

Proof: Since t(X) = t(X {y}), we have Sup(X) = Sup(X {y}). Therefore, any transaction containing X must also contain y. Consequently, any itemset containing S must also contain y, so Sup(X S) = Sup(X {y} S).

Based on Property 2, BMFI employs parent equivalence pruning to reduce the search space. When searching node C, if the condition in Property 2 is encountered, y can be removed from C.tail and added to C.head, improving mining efficiency without affecting the accuracy of MFI mining.

Property 3: If node C in the lexicographic search tree has no right neighbor and C.head is not a subset of any already mined maximal frequent itemset, then the itemset is a maximal frequent itemset.

Proof: If node C has a right sibling, the itemset represented by C could be a subset of the itemset represented by that sibling, so it may not be a subset of an already discovered maximal frequent itemset. Similarly, if a node represents an

itemset that is a subset of a previously mined maximal frequent itemset, then that node does not represent a maximal frequent itemset.

Based on Property 3, this paper adopts the MFI-tree projection strategy proposed by Grahne et al. [7] for superset detection. During bottom-up depth-first traversal of FP-tree space, for single-path FP-tree header table itemsets and itemsets composed of items with maximum support in the header table, superset existence can be determined. If the header table does not contain the current item, no superset can exist. Only when overlapping portions exist on the current node path can the MFI-tree contain a superset of that itemset, thus achieving superset detection. If no superset exists in the MFI for an itemset, that itemset is maximal and is added to the MFI. Algorithm 4 details this process.

Algorithm 3: DFS Algorithm Based on B-list Input: Transaction database DB and minSup **Output:** All MFIs in DB

```
MFI $\leftarrow$
root.head $\leftarrow$
root.tail $\leftarrow$ Set of frequent items in DB (sorted by ascending support)

After TB-tree construction, obtain B-lists for all frequent 1-itemsets, denoted BL$_{1}$

Call DFS_{{{based}}_{{BList}}}(root, NULL, NULL)

Function DFS_{{{based}}_{{BList}}}(Current node C, C's BList BL_{cur}, C.sibling$^{+}$'s BLi
    k $\leftarrow$ item with maximum support in C.head

    for each item i in C.tail
        if (BL_{cur} == NULL)
            BL_{child}[i] = BL$_{1}$[i]

        C_{HUT} $\leftarrow$ C.head  {i} - {k}
        BL_{HUT} $\leftarrow$ element in BLS recording C_{HUT}
        BL_{child}[i] = BL_{intersection}(BL_{cur}, BL_{HUT})
    end for

    for each item i in C.tail
        Cn.head $\leftarrow$ C.head  {i}
        Cn.tail $\leftarrow$ {j  C.tail | i $\leq$_L j}
        C_{HUT} $\leftarrow$ C.head  {i} - {k}
        BLS_{child} $\leftarrow$ BL_{child}

        if (BL_{child}[i].support $\geq$ minSup)
            DFS_{{{based}}_{{BList}}}(Cn.head, BL_{child}[i], BLS_{child})
        end if
    end for
```

```
    if (C is a leaf and C.head is not in MFI)
        MFI $\leftarrow$ MFI   C.head
    end if
End Function
```

1.4.2 Optimization Strategies

The core idea of depth-first search is to examine each 1-extension item i in node C' s C.tail. The algorithm computes the support of C.head i. If all supports are less than minSup, then C is a leaf node in the frequent pattern tree. It then checks whether C.head is a subset of any set in the maximal frequent itemsets MFI. If not, C is added to MFI. If all C.head i supports are greater than minSup, the algorithm continues recursion. Drawing from the combination of DFS with pruning strategies in reference [6] and N-list with DFS in reference [8], this paper introduces a B-list-based DFS algorithm. Due to B-list' s high compression ratio and TB-tree' s efficient construction, the algorithm achieves higher efficiency in mining maximal frequent itemsets.

Algorithm 4: Superset Detection Based on MFI-tree Projection Strategy Input: Pruned itemset M sorted by descending support **Output:** Boolean indicating superset existence

```
int len = M.length;

if (!headerTable.contains(M[len-1]))
    return false;

BMFINode node = headerTable.get(M[M.len-1]);

while (node != null) {
    BMFINode high = node;
    while (high != null) {
        if (high.label == M[len-1]) {
            len--;
            high = high.parent;

            if (len == 0)  // Superset exists
                return true;
        }
        high = high.parent;
    }
    node = node.sibling;
    len = M.length;
}

return false;
```

2 Experimental Results and Analysis

Since FPMAX maintains good performance across different datasets, we validate BMFI' s effectiveness by comparing its runtime for mining maximal frequent itemsets against FPMAX and MFIN algorithms across various datasets.

All programs were implemented in C/C++ on an Intel(R) Core(TM) i5 CPU M330 @ 3.1GHz with 4 GB RAM, running 64-bit Windows 10. We implemented FPMAX, BMFI, and MFIN algorithms in this environment. Experiments used the Pumsb and Retail datasets, along with the synthetic T10I4D100K dataset generated by IBM' s data generator (dataset parameters are shown in). We varied minimum support thresholds to mine frequent patterns and compared algorithm runtime and memory consumption. Runtime comparisons are shown in [Figure 3: see original paper], and memory consumption comparisons in [Figure 4: see original paper].

Dataset Parameters - Number of items, average length, and number of transactions for Pumsb, Retail, and T10I4D100K

Our analysis compared BMFI against FPMAX and MFIN across different datasets. First, all three algorithms extracted identical content and quantities of maximal frequent itemsets under the same dataset and support threshold, confirming BMFI' s correctness. The Pumsb, Retail, and T10I4D100K datasets represent dense, sparse, and synthetic datasets respectively. As shown in [Figure 3: see original paper], all three algorithms' runtimes decrease as support increases. On the Retail dataset, while runtime differences are minor across various minimum support thresholds, BMFI shows significant advantages at lower support thresholds. On T10I4D100K, when minSup = 0.1, BMFI' s runtime is nearly twice as fast as FPMAX, though this advantage diminishes as minSup increases. As shown in [Figure 4: see original paper], memory consumption for all three algorithms decreases with increasing support thresholds across different datasets. At lower support thresholds, BMFI' s memory consumption is significantly lower than both FPMAX and MFIN. These results demonstrate BMFI' s high time and space efficiency across different dataset types.

3 Conclusion

This paper proposes a novel maximal frequent pattern mining algorithm called BMFI. The algorithm leverages B-list's high compression ratio and efficient intersection operations for rapid support counting, employs parent equivalence pruning to reduce search space, and combines MFI-tree-based projection strategies for superset detection to improve efficiency. Results demonstrate that BMFI offers significant performance advantages over FPMAX and MFIN algorithms. However, with the advent of the big data era, further optimization of BMFI is needed, particularly in pruning optimization strategies, which will be the focus of future work.

References

[1] Agrawal R, Srikant R. Fast algorithms for mining association rules [C]// Proc of the 20th International Conference on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers, 1994: 487-499

[2] Han J, Pei J, Yin Y, et al. Mining frequent patterns without candidate generation: a frequent-pattern tree approach [J]. Data Mining & Knowledge Discovery, 2004, 8 (1): 53-87.

[3] Aggarwal C C, Han J. Frequent pattern mining [M]. [S. l]: Springer International Publishing, 2014

[4] Jr Bayardo R J. Efficiently mining long patterns from databases [J]. ACM Sigmod Record, 1998, 27 (2): 85-93.

[5] Agarwal R C, Aggarwal C C, Prasad V V V. Depth first generation of long patterns [C]// Proc of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. NewYork: ACM Press, 2000: 108-118.

[6] Burdick D, Calimlim M, Flannick J, et al. MAFIA: a maximal frequent itemset algorithm [J]. IEEE Trans on Knowledge and Data Engineering, 2005, 17 (11): 1490-1504.

[7] Grahne G, Zhu J. High performance mining of maximal frequent itemsets [C]// Proc of the 6th International Workshop on High Performance Data Mining. 2003.

[8] 沈戈晖, 刘沛东, 邓志鸿. NB-MAFIA: 基于 N-List 的最长频繁项集挖掘算法 [J]. 北京大学学报: 自然科学版, 2016, 52 (2): 199-209.

[9] 林晨, 顾君忠. 基于 Nodeset 的最大频繁项集挖掘算法 [J]. 计算机工程, 2016, 42 (12): 204-207, 216.

[10] Deng Z H, Lv S L. Fast mining frequent itemsets using Nodesets [J]. Expert Systems with Applications, 2014, 41 (10): 4505-4512.

[11] Deng Z H, Lv S L. PrePost+: an efficient N-lists-based algorithm for mining frequent itemsets via children–parent equivalence pruning [J]. Expert Systems with Applications, 2015, 42 (13): 5424-5432.

[12] Dam T L, Li K, Fournier-Viger P, et al. An efficient algorithm for mining top-rank-k frequent patterns [J]. Applied Intelligence, 2016, 45 (1): 9-23.

[13] Deng Z H, Wang Z H, Jiang J J. A new algorithm for fast mining frequent itemsets using N-lists [J]. Science China Information Sciences, 2012, 55 (9): 1838-1850.

Note: Figure translations are in progress. See original paper for figures.