LSCMiner: Efficient Low Support Closed Itemsets Mining

Yifeng Lu(✉), Florian Richter, and Thomas Seidl

Database Systems and Data Mining Group, LMU Munich, Munich, Germany
{lu,richter,seidl}@dbs.ifi.lmu.de

Abstract. Itemsets with relatively low support values are important since they usually suggest highly confident association rules, which are useful in applications such as recommendation systems and medical data analysis. However, most existing algorithms are mainly designed to mine frequent patterns and thus are time consuming in generating low support patterns. There are also a few algorithms focus on low support patterns but not efficient enough. Therefore, we propose here a low support closed pattern mining algorithm, utilizing top-down lattice traversing and novel closeness checking/pruning techniques. Extensive experiments show that our method is much more efficient to mine low support closed patterns than available alternatives.

1 Introduction

Itemset mining is an important topic in data mining for decades. Nowadays, it is still actively applied in many areas such as recommendation systems, financial data and medical data mining. Recent researches show that, compared to the advanced deep learning based recommendation systems, pattern based approaches are still competitive [8].

Most existing pattern mining algorithms are designed to mine frequent itemsets as they represent mainstream behavior. However, low support patterns or infrequent itemsets are also important since they usually imply highly confident association rules with solid support. In a large dataset, a low support pattern may actually occur hundred times. Their corresponding rules are useful in recommendation systems for better accuracy. For instance, fewer people in Europe will buy “rice” and “nori” together. Then the pattern “rice and nori” will not be included infrequent patterns, i.e., “sushi maker set” would be recommended only when low support patterns are considered. Low support patterns also play essential roles in other applications. In medicine area, they are crucial in identifying rare diseases. For domain expert, untypical responses to medications are more interesting than frequent and expected ones. In the analysis of traffic accidents, causes of accidents might hide in less frequent and abnormal behaviors. Low support patterns are also helpful in finding significant discriminative patterns [3]. Process mining approaches make use of them as well in identifying deviations in significant process [13].

© Springer Nature Switzerland AG 2019
R. Cheng et al. (Eds.): WISE 2019, LNCS 11881, pp. 293–309, 2019.
https://doi.org/10.1007/978-3-030-34223-4_19

Conventionally, low support patterns are achieved by executing frequent itemset mining algorithms with a small minimum support threshold. Thus, frequent patterns are accessed inevitably. If the huge number of frequent patterns are not of interest, such as the medication example mentioned above, this solution is time wasting. Even if in applications where both frequent and less frequent patterns are needed, the ability to mine low support patterns directly is still necessary. For instance, in dynamic environment, it is expensive for stream pattern mining approaches to track both frequent and less frequent patterns. When low support patterns are more stable over time, efficient low support pattern mining makes it possible to maintain them separately, so that we can build a more adaptive system which only tracks frequent patterns while less frequent ones are updated by user request. Therefore, a few approaches aimed at low support patterns are proposed [1,5,9,16,17]. However, they are either inefficient or with additional constraints.

In this work, we focus on mining low support itemsets. It is well known that redundancy is always a problem in itemset mining. Varies condensed representations and corresponding algorithms are proposed for frequent patterns. However, similar studies are still missing for low support patterns. Closed itemset is one of the most popular lossless condensed representations [14]. We propose a new top-down based algorithm which extracts low support closed patterns without traversing frequent ones. Our approach uses a very efficient tree-based structure. Novel closeness checking and pruning techniques are employed. We show that our approach can achieve the same level of complexity per itemset as other efficient frequent pattern mining algorithms.

2 Preliminaries

2.1 Problem Definition

Let $\mathcal{I}$ be the universe of items, a subset of $\mathcal{I}$ that contains l items is a l-itemset, denoted as $X = \{x_1, x_2, \ldots, x_l\}$. A transaction dataset $\mathcal{T}$ contains a set of transactions where each transaction $T \in \mathcal{T}$ is an itemset over $\mathcal{I}$. Let $\mathcal{T}(X) = \{T | T \in \mathcal{T}, X \subseteq T\}$ be the set of transactions in $\mathcal{T}$ that contains X, the (absolute) support of X on $\mathcal{T}$ is defined as $|\mathcal{T}(X)|$.

In this work, we tend to find less frequent or low support patterns, i.e., $|\mathcal{T}(X)| \ll |\mathcal{T}|$. Formally speaking, given two user-defined threshold: *minimum support* α and *maximum support* β, we are going to mine patterns X such that $\alpha \leq |\mathcal{T}(X)| < \beta$, where $\alpha \geq 1 \wedge \beta \ll |\mathcal{T}|$. In general, our mining task is the same as infrequent itemset mining since $\beta \ll |\mathcal{T}|$. The parameter α is introduced for more flexibility as users might consider patterns occurred less than α as noise. Conventional frequent itemset mining algorithms can also extract low support patterns by setting their minimum support threshold to α and then removing all frequent patterns with support larger than β.

An itemset X is a *closed* itemset in dataset $\mathcal{T}$ if and only if there is no other itemset Y in $\mathcal{T}$ such that $X \subset Y \wedge |\mathcal{T}(X)| = |\mathcal{T}(Y)|$. The closed itemset concept was first proposed in [14] to address the redundant problem in frequent itemset

mining problem. It is a lossless condensed representation: user can determine the support of any frequent itemsets from closed frequent itemsets. The set of closed low support patterns is $\mathcal{LP}$.

A frequent border set $\mathcal{FB}$ is defined as the set of longest patterns such that $|\mathcal{T}(X)| \geq \beta$, which is also known as the maximal frequent itemset [2]. $\mathcal{FB}$ is necessary to make $\mathcal{LP}$ complete. For example, given a pattern $\{ab\}$, if $\exists X \in \mathcal{LP}$ such that $\{ab\} \subseteq X$ but $\nexists X' \in \mathcal{LP}$ such that $X' \subseteq \{ab\}$, then the pattern $\{ab\}$ can be either frequent or not frequent. The border $\mathcal{FB}$ helps in this case to identify whether $\{ab\}$ is frequent or not.

2.2 Lattice Traversing and Related Works

Itemset mining is a process of itemset lattice traversing. Frequent itemset mining algorithms traverse the lattice bottom-up, i.e., starting from empty itemset. Thus, they must waste time on accessing frequent patterns before extracting less frequent itemsets. Similarly, frequent closed itemset mining algorithms also suffer the same problem when the user only want low support patterns.

Infrequent itemset mining algorithms [5,7,16] are proposed to mine patterns with support smaller than a given threshold. However, they still utilize the bottom-up traversing strategy. Rarity [17] algorithm uses the top-down traversing strategy which extracts low support patterns first. It is an apriori-like approach so that an expensive candidate generation step is necessary. These early-stage algorithms are indeed slower than well optimized frequent itemset mining approaches.

RPTree [18] suggests that there are three types of patterns: frequent patterns; infrequent patterns with infrequent items; infrequent patterns without infrequent items. It is designed only to return the second type of patterns. A negative item tree is proposed in [11] mine infrequent patterns top-down. We adopt this tree structure to mine closed patterns.

A bi-directional traversing framework is proposed in [12] to extract closed low support patterns by separating the dataset into a sparse and a dense part. Bottom-up and top-down traversing strategies are applied respectively. However, its top-down traversing part is slow due to duplication problems, which limits the performance of this framework. In this work, we make use of this framework to achieve better memory performance.

Some algorithms extracting descriptive patterns based on information theory [15] or top-k patterns of each item [10]. In theory, they could also return some low support patterns. However, the majority are left behind.

2.3 Support Counting on Negative Itemset Tree

The ni-tree [11] is initially proposed to mine all infrequent patterns. It stores support information of *negative represented (neg-rep) itemsets*. Neg-rep itemsets are itemsets represented by symbol of items that do not exist in the original itemsets. For example, given $\mathcal{I} = \{a, b, c, d, e, f\}$, an itemset $X = \{a, b, c\}$ can also be represented using the symbol of items not in X, denoted as $\overline{X} = \{d, e, f\}$.

Obviously, X and $\overline{X}$ represent the same information since $X \subseteq T \Leftrightarrow \overline{X} \supseteq \overline{T}$. Let $\overline{\mathcal{T}}$ be the negative dataset formed by neg-rep itemsets, the support of $\overline{X}$ can be defined as the number of neg-rep transactions in $\overline{\mathcal{T}}$ that covered by $\overline{X}$ such that:

$$|\overline{\mathcal{T}}(\overline{X})| := |\{\overline{T}|\overline{T} \in \overline{\mathcal{T}}, \overline{T} \subseteq \overline{X}\}| \Leftrightarrow |\mathcal{T}(X)| = |\overline{\mathcal{T}}(\overline{X})| \tag{1}$$

Tid	Itemset
1	b c
2	d e
3	a e
4	c d e
5	b d e
6	a d e

(a)

Tid	Negative Itemset
1	a d e
2	a b c
3	b c d
4	a b
5	a c
6	b c

(b)

Fig. 1. Transaction dataset and its negative dataset.

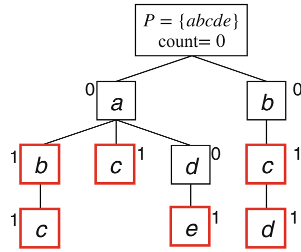


Fig. 2. The initial ni-tree of dataset in Fig. 1. Red nodes are *t-nodes*. (Color figure online)

The ni-tree is a prefix tree, as shown in Fig. 2. Items are sorted in ascending order concerning their frequency $\mathcal{T}$. Each node n is a triplet $\langle i, c, l \rangle$, where i and c are the item label and its count, l is the list of child nodes. c is initialized to 0. The root node $r = \langle P, c, l \rangle$ stores the current pattern P.

Each transaction $T \in \mathcal{T}$ is converted to $\overline{T}$ and inserted to the ni-tree. The last node, known as the *termination node* or *t-node* for short, will increase its count by 1. Thus, the count of a node n is the number of its corresponding transactions, i.e. $n.c = |\{\overline{T} \in \overline{\mathcal{T}}, \overline{T} = n.L\}|$, where $n.L$ is the set of items on the path from root to n. According to Eq. 1, $|\mathcal{T}(X)|$ can be computed by aggregating all nodes whose path from the root is fully covered by $\overline{X}$:

$$|\mathcal{T}(X)| = |\overline{\mathcal{T}}(\overline{X})| = \sum_{n.L \subseteq \overline{X}} n.c \tag{2}$$

For example, to identify the support of itemset $X = \{de\}$, the count of nodes on paths that covered by $\overline{X} = \mathcal{I} \setminus X = \{abc\}$ are aggregated, which equals to 4. Therefore, the ni-tree can be used to compute the support of a given pattern.

Moreover, given patterns X and X', $X \subset X' \Leftarrow \overline{X} \supset \overline{X'}$, the set of nodes for computing $|\mathcal{T}(X)|$ can be decomposed as: $\{n|n.L \subseteq \overline{X}\} = \{n|n.L \subseteq \overline{X'}\} \cup \{n|n.L \subseteq \overline{X}, n.L \nsubseteq \overline{X'}\}$. Thus, the aggregating process can be decomposed and computed recursively. For example, let $X = \{de\}, X' = \{bcde\}$, the support of X can be obtained by removing nodes on the path covered by $\overline{X'} = \{a\}$, which leads to a new ni-tree that represents the pattern $\{bcde\}$, as shown in Fig. 3. Then, removing nodes covered by $\overline{X}$ from the second ni-tree will generate the pattern $\{de\}$. Such process is in top-down style. In practice, we only need to create a new root node rather than a brand new ni-tree.

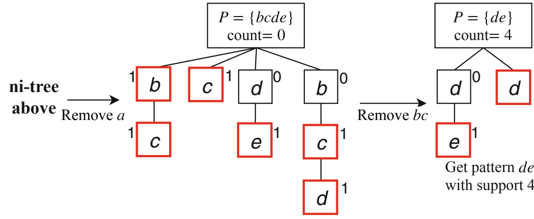


Fig. 3. Counting support on ni-tree.

3 Closed Itemset on Negative Itemset Tree

Though the ni-tree is not initially designed for closed pattern mining, we found that the closeness can be determined readily by using t-nodes.

3.1 Closed Itemset Determination

According to the definition, an essential property of a closed pattern X is that its support must be different from its supersets. In ni-tree, the count value of any node, except t-nodes, is 0. Thus, if the count of all removed nodes is 0, the generated pattern is not closed. A closed pattern can only be achieved if at least one t-node is involved in the aggregating process. Formally speaking:

Theorem 1. *Given $\mathcal{I}$ and the initial ni-tree, let N_X be the set of nodes been removed from the initial ni-tree to achieve the pattern X. Let $N_X^t \subseteq N_X$ be the set of t-nodes been removed. Then pattern X is closed if and only if the set of items been removed ($\overline{X}$) equals to the set of items on paths to t-nodes:*

$$\mathcal{I} \setminus X = \overline{X} = \bigcup_{n \in N_X^t} n.L \tag{3}$$

Proof. Obviously, $N_X = \{n | n.L \subseteq \overline{X}\}, N_X \supseteq N_X^t$. Thus,

$$\overline{X} = \bigcup_{n \in N_X} n.L \supseteq \bigcup_{n \in N_X^t} n.L \tag{4}$$

As non-terminated nodes are counted at 0 in the ni-tree, the support of pattern X is the sum of all t-nodes:

$$|\mathcal{T}(X)| = |\overline{\mathcal{T}}(\overline{X})| = \sum_{n.L \subseteq \overline{X}} n.c = \sum_{n \in N_X^t} n.c \tag{5}$$

Let $M = \overline{X} \setminus (\bigcup_{n \in N_X^t} n.L)$. Thus, any node $n' \in N_X$ with item $n'.i \in M$ is not on the path to a t-node in N_X^t. Removing such nodes or not won't affect the support value, i.e. $|\mathcal{T}(X)| = |\mathcal{T}(X \cup M)|$. By closeness definition, $M = \emptyset \Leftrightarrow X$ is closed. ◻

In short, a pattern X is closed if all removed items can be found on paths towards removed t-nodes. For example, in the ni-tree of Fig. 2, pattern $X = \{be\}$ is not closed since item d is removed but its corresponding nodes are not on a path towards t-nodes covered by $\overline{X} = \{acd\}$. On the other hand, itemset $X = \{de\}$ is closed.

3.2 Naïve Method

According to Theorem 1, top-down closed pattern mining can be realized by simply enumerating and removing all combinations of paths towards t-nodes. The ni-tree is slightly adapted. The root node and each t-node stores a list of pointers (l_t) linked to their child t-nodes, as shown in Fig. 4. In each step, nodes on the path from one t-node (excluding) to its child t-node (including) are removed together, which guarantees that only closed patterns are generated. Figure 4 illustrates an example. By removing all nodes on the path to t-node 1, a new ni-tree is generated, and the corresponding closed pattern $\{de\}$ is returned. Then removing t-node 2 in the new ni-tree lead to another closed pattern $\{e\}$. Enumerating all removing combinations generate all closed patterns.

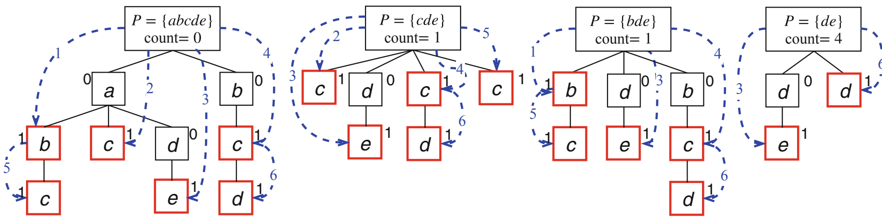


Fig. 4. The adapted initial ni-tree with t-node links (blue) and corresponding ni-tree by removing t-node 1, 2 or 1, 2 together. Each link is marked with the t-node id. In each step, only child t-nodes of root are considered (e.g. node 6 can only be removed after node 4). (Color figure online)

The main disadvantage of this naive approach is the duplicate accessing problem. For example, given $\mathcal{I} = \{abcde\}$, the pattern $X = \{ab\}$ can be achieved by either removing $\{cd\}$ and $\{ce\}$ or removing $\{cd\}$ and $\{de\}$. A pattern X might be accessed repeatedly up to $O(2^{|\mathcal{T}(X)|})$ times. Extra duplicate checking and pruning step are necessary. By examining patterns discovered so far, we can avoid the majority of duplicates. However, the overhead of the pruning step plus remaining duplicates are still time-consuming. Indeed, this naïve method is the top-down part used in the bi-directional traversing framework [12].

4 Algorithm: LSCMiner

4.1 Divide-and-Conquer Paradigm

The naive approach described above is a top-down based algorithm. However, it is not efficient due to the expensive duplicate accessing problem. To take

advantage of top-down traversing, we propose our **L**ow **S**upport **C**losed Miner (LSCMiner), which employs the depth-first traversing strategy with novel closeness checking and pruning steps. The general mining process employed a divide-and-conquer paradigm, which is commonly used in bottom-up based algorithms. The main difference is that we remove items recursively, rather than grow patterns.

First of all, let operators $\prec$ and $\succ$ denote the concept of *"before (smaller)"* and *"after (larger)"* with respect to the ascending frequency order used by the ni-tree. Given $\mathcal{I} = \{a \prec b \prec c \prec \dots\}$, the top-down mining process removes items recursively, which can be represented as a tree as shown in Fig. 5. We call the tree above as the *deletion tree*. Each node in the tree is the set of items to be removed, known as the *deletion set*. Given a node in the deletion tree, we say that deletion sets in its sub-tree and right to it are *under* or *after* the deletion set in the node, as shown in Fig. 5.

The first challenge is to combine the closeness checking process with the divide-and-conquer paradigm. According to Theorem 1, we need to check if every removed item can be found on paths to removed t-nodes. To solve the problem, we let each t-node n^t contains a list $n^t.L$, which stores items on the path from itself (including) to its proceeding t-node (excluding), as shown in Fig. 6. During the removing process, a set U is maintained to track items that are not covered by paths towards removed t-nodes yet. In one recursive step, we first add the current item to U. If a t-node n^t is removed, all items exist in $n^t.L$ are removed from U. When $U = \emptyset$, we knew that the current pattern is closed.

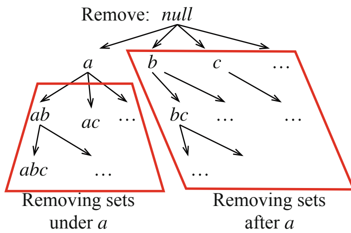


Fig. 5. We solve the mining task by removing items in a recursive way. Such process can be represented as a tree.

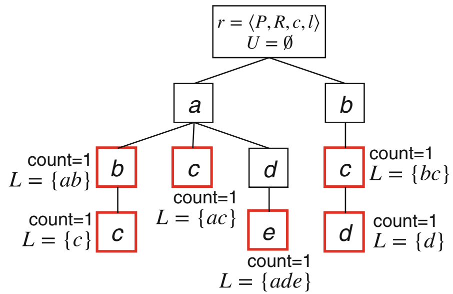


Fig. 6. The adapted ni-tree used in LSCMiner. R is the set of items been removed so far.

Figure 7 gives an example of the closed pattern mining process. We first remove item a, no t-node is removed right now. Thus, $U = \{a\}$ and the pattern $\{bcde\}$ is not closed. Then, we recursively remove b from the current ni-tree. There is a t-node of b is removed and $U = \{ab\} \setminus \{ab\} = \emptyset$. Pattern $\{cde\}$ is closed and should be added to the result set.

Algorithm 1 illustrates the pseudo code of the LSCMiner. Each iteration step removes one item (Line 5). If there are t-nodes in removed nodes list l_i, we remove

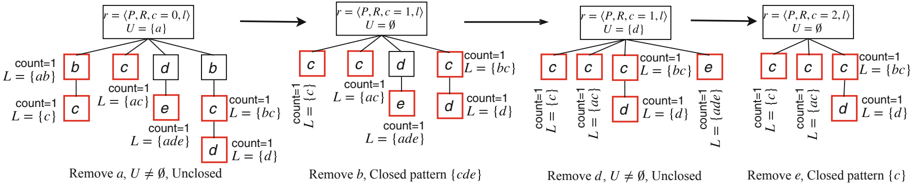


Fig. 7. Recursive steps of the removing process $a \rightarrow ab \rightarrow abd \rightarrow abde$.

items from U and aggregate counts (Line 10–13). If the current candidate pattern is not frequent, we attach nodes with larger item to the new ni-tree root for the next recursive call (Line 18). The recursive mining process is continued until the aggregated count is larger than the given maximum threshold β. Variables iM_1 and iM_2 are pruning thresholds as described later.

4.2 Pruning

An efficient algorithm should be able to prune unclosed itemsets as early as possible, known as the *"look ahead"* ability [14,19]. In our LSCMiner, we fully utilized the closeness property of the ni-tree. Two types of pruning methods are utilized.

Trial-and-Error Pruning. Our first pruning method (Line 27, Algorithm 1) is based on the following observation:

Theorem 2. *Given the current ni-tree root r and the current unclosed items set $U \neq \emptyset$. Let R be the set of items been removed so far. If $\exists i \in r.l$ such that no closed pattern in deletion sets under $\{R \cup i\}$, then there is also no closed pattern in deletion sets after $\{R \cup i\}$.*

Proof. The sub-ni-tree under item i must contain at least one t-node. Let l be the set of items from i (including) to a t-node n^t (including) in its sub-ni-tree. Obviously, we have $n^t.L \supseteq l$ and $i \in n^t.L$.

Let deletion sets after i be $R_{\succ i}$. Assuming the deletion set of i and deletion sets under i are not closed. If $\exists p \in R_{\succ i}$ which will lead to a closed pattern, then removing all items in $\{p \cup l\}$ will also lead to a closed pattern (by further removing the node n^t mentioned above). Obviously, $\{p \cup l\}$ is a deletion set of i or under i, which is contradict to our assumption (Fig. 8). $\square$

In short, if removing i does not generate a closed pattern, the recursion call will be executed (Line 21, Algorithm 1). This recursive call will try all possible combinations of items with respect to i. If no closed pattern is generated, iterations on items (Line 5, Algorithm 1) after i can be canceled.

```
   Input: Ni-tree root r, Minimum support α, Maximum support β
   Output: Infrequent Itemset List LP, Frequent Border List FB
 1 LP ← ∅, FB ← ∅;
 2 LSCMiner(r,∅,+∞, −∞) ;
 3 return LP, FB ;
 4 Function LSCMiner(r, U, iM1, iM2)
 5     foreach Item i ∈ r.l ∧ i ⪯ iM1 do
 6         l_i ←List of nodes in r.l with label i ;
 7         U' ← U ∪ {i}, P' ← r.P \ {i}, c' ← r.c;
           /* Closeness checking                                  */
 8         foreach Termination node n ∈ l_i do
 9             U' ← U' \ n.is, c' ← c' + n.c ;
10         end
11         if c' < β then
12             if c' ≥ α ∧ U' = ∅ then
13                 Add P' to LP
14             end
15             l' ← {n' ∈ r.l|n'.i ≻ i} ∪ {⋃_{n∈l_i} n.l}, r' ← {P', c', l'};
               /* Initial new end index                           */
16             if P' is closed then
17                 iM'1, iM'2 ← +∞, −∞;
18             else
19                 iM'1, iM'2 ←UpperBound(l, iM2);
20             end
21             LCSMiner(r', U', iM'1, iM'2) ;
22             if No closed pattern generated in the recursive call above then
23                 Break;                         // Trial-and-Error Pruning
24             end
25         else
26             if U' = ∅ then
27                 Add P' to FB;
28             end
29         end
30     end
31 end
```

Algorithm 1: LSCMiner

Upper-Bound Pruning. The second pruning technique computes the largest possible item as an upper bound for the next recursion step. Given the current removed item i and the list of nodes to be removed l_i, assuming all nodes in l_i are not terminated, then the largest possible item iM_1' that can be removed in the next recursion step is the largest item among all children of nodes in l_i.

The reason is straightforward: the item i will be covered by a t-node if and only if at least one of its child is removed. If we remove an item $i' \succ iM_1'$ in the next recursion step, all items to be removed in the future are also larger than iM_1'. Thus, it is impossible to reach a t-node that covers i. For example, given the left ni-tree in Fig. 9, assuming now we are removing item a, which results

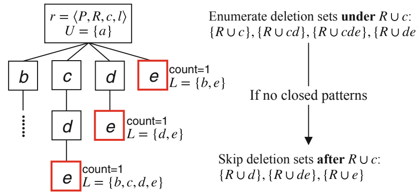


Fig. 8. Given the current unclosed ni-tree, Trial-and-Error pruning will try all deletion sets under $R \cup c$. If no closed patterns exists, then later deletion sets can be skipped.

in the right ni-tree in Fig. 9. However, t-nodes that cover a only exist in the sub-ni-tree of a. Thus, the upper bound for item removing on the second ni-tree is b. Further removing process on items after b is pruned since the item a will never be covered.

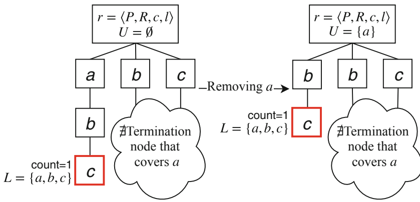


Fig. 9. The maximum item under nodes of a is $iM'_1 = b$. Later removing process on the right ni-tree must removing b first since otherwise, t-nodes that cover a will not be removed.

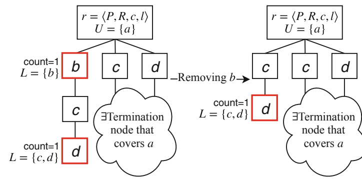


Fig. 10. There is a t-node that covers b. Then the maximum upper bound up to now (iM'_2), which equals to the upper bound when removing a, is used as the upper bound for further removing process on the right ni-tree.

The above upper bound assumes that $\nexists n \in l_i$, i.e., item i can only be covered by children of nodes in l_i. However, if one node of item i is terminated, then i is covered by a node of itself. Removing items larger than the upper bound iM'_1 can still lead to closed patterns. Another weaker upper bound iM'_2 is introduced for this case, which is defined as the largest upper bound, except for infinity, among all previous recursion steps. For example, assuming the left ni-tree in Fig. 10 is achieved by removing item a, and the right ni-tree is achieved by further removing item b. Since node b is terminated, removing items larger than its children is valid. However, the previously removed item a needs to be covered so that the upper bound iM'_2 is the upper bound when removing a. Algorithm 2 computes both upper bounds described here.

4.3 Complexity

Pattern mining is an NP-hard problem. The overall runtime is highly dependent on the number of desired patterns. For instance, one of the most efficient frequent

```
1  Function UpperBound(l_i, iM_2)
2      iM'_1 ← −∞
3      foreach n ∈ l_i ∧ n is not t-node do
4          x_last ←Last item in n.l
5          if iM'_1 ≺ x_last then
6              iM'_1 ← x_last
7          end
8      end
9      if iM'_1 ≠ −∞ then
10         iM'_2 ← max(iM'_1, iM_2)
11     end
12     if ∃n ∈ l_i, n is terminated then
13         iM'_1 ← iM'_2
14     end
15     return iM'_1, iM'_2
16 end
```

Algorithm 2: Compute the new upper bound.

closed pattern mining algorithms, LCM [19], declares that it extracts each closed pattern in polynomial time: $O(P(|\mathcal{T}|))$. Let $\mathcal{U}$ and $\mathcal{UC}$ be the set of desired and undesired patterns, the time complexity per itemset of the LCM algorithm can be written as: $O(\frac{|\mathcal{U}|+|\mathcal{UC}|}{|\mathcal{U}|}P(|\mathcal{T}|))$, where $\mathcal{UC}$ contains frequent patterns in the low support closed pattern mining scenario.

Our approach can also achieve the same level of complexity. Given the current ni-tree root r and the current unclosed items set U, removing item i from the child list $r.l$ involves the following steps:

1. aggregate counts in removed nodes, which requires $O(|l_i|)$ time, where l_i is the list of nodes in $r.l$ labeled with i.
2. closeness checking if t-nodes exist, which requires $O(|U|\log(|n^t.L|))$ time, where $n^t.L$ is the set of items in a t-node and binary search is employed
3. add children of nodes in l_i to the new root node r', which takes $O(\sum_{n\in l_i}|n.l|)$ time.
4. add all nodes in $r.l$ with label larger than i to the new root node r', which requires $O(|l_{\succ i}|)$ time, where $l_{\succ i}$ is the list of nodes.

The total complexity is $O(|l_i|+|U|\log|n^t.L|+\sum_{n\in l_i}|n.l|+|l_{\succ i}|)$. $|U|$ and $|n^t.L|$ are limited to the size of a single transaction so that the second term can be seen as a constant. The length of $l_i, l_{\succ i}$ and $n.l$ are limited to the size of the dataset. Thus, the complexity of removing item i is polynomial. A closed itemset X is achieved by removing items in $\overline{X}$. The complexity to extract X is $O(\sum_{i\in\overline{X}}P(|\mathcal{T}|)) \in O(P(|\mathcal{T}|))$ since $|\overline{X}|$ is small compared to $|\mathcal{T}|$. Considering that our approach also accesses some unclosed patterns, its complexity per itemset is also $O(\frac{|\mathcal{U}|+|\mathcal{UC}|}{|\mathcal{U}|}P(|\mathcal{T}|))$, where $\mathcal{UC}$ are those unclosed patterns.

In terms of memory complexity, it is obvious that our approach is limited by the size of the dataset, similar to algorithms such as FPGrowth [6].

However, LSCMiner has to store the negative dataset such that a scale factor $s = \frac{|\mathcal{I}|}{\text{avg. transaction length}}$ exists, known as the sparsity of the dataset. s can be huge on sparse dataset. In this case, the bi-directional traversing framework proposed in [12] can be used so that our LSCMiner only need to handle the densest part of a dataset.

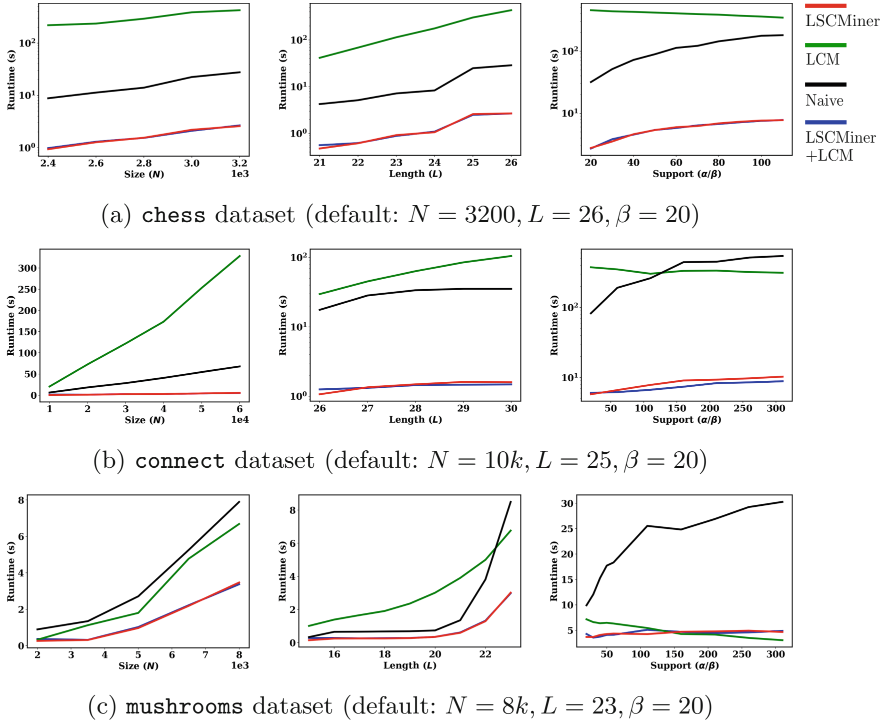
5 Experiments

We first conduct the runtime performance of our LSCMiner. In experiments, the naive approach described in Sect. 3.2 represents the performance of a simple top-down based algorithm. The LCM [19] algorithm represents the most efficient bottom-up based algorithm in solving low support pattern mining problem. We also conduct the bi-directional traversing framework [12] by combining the LCM algorithm with our LSCMiner. Other infrequent pattern mining algorithms are not included since they are either represented by LCM (bottom-up based) or too memory expensive to finish (apriori alike).

Database	Size (N)	Items ($\mathcal{I}$)	Avg. length (L)
mushrooms	8k	119	23
chess	3k	75	37
connect	67k	129	43
accident	340k	468	33.8
kddcup99	1000k	135	16
BMS1	59k	497	2.5

Fig. 11. Real-life datasets in our experiments.

Algorithms are implemented using Java. The LCM implementation comes from [4]. 6 real-world datasets obtained from the fimi repository (http://fimi.ua.ac.be/data/) are used as our test datasets. Necessary information of those datasets are listed in Fig. 11. There are three small dense datasets, two large dense datasets, and one sparse dataset. First N transactions and first L items in each transaction are used in our experiments. We are interested in the time difference in accessing patterns on a certain level of support from different directions. Thus, we set $\alpha = \beta - 10$. The default value of N, L and β are provided in each experiment. The splitting threshold for the bi-directional framework is set to $\delta = 1\%$.



(a) chess dataset (default: $N = 3200, L = 26, \beta = 20$)

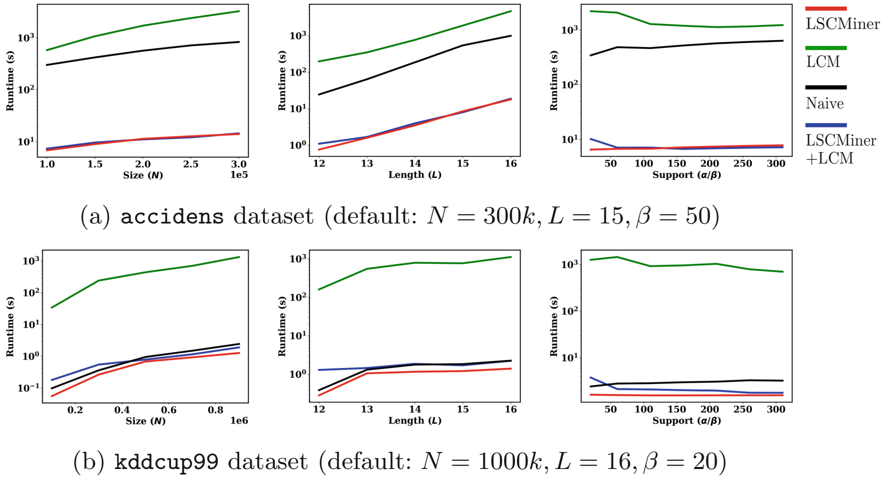
(b) connect dataset (default: $N = 10k, L = 25, \beta = 20$)

(c) mushrooms dataset (default: $N = 8k, L = 23, \beta = 20$)

Fig. 12. Runtime on small dense dataset.

Dense Data. Figure 12 illustrates the performance on small dense dataset. On these datasets, the bottom-up algorithm, LCM, is up to two order of magnitude slower than our top-down LSCMiner on the first two datasets. It is even slower than the naive approach under some settings. This is mainly because the bottom-up LCM algorithm has to traverse all frequent patterns. When β increased, i.e., we become more interested in frequent patterns, the runtime of LCM is reduced and may surpass top-down approaches since it needs to traverse less frequent patterns. On the `mushrooms` dataset, top-down approach is slower with $\beta > 150$. This is mainly because that the `mushrooms` dataset has less number of patterns. Our LSCMiner is very efficient. Its runtime grows similar to the LCM approach with increasing dataset size, which indicates that the time complexity of both approaches is on the same level. The combined approach is also efficient under most settings. Our LSCMiner under the bi-directional framework only need to handle the densest part of the dataset, which reduces the memory consumption, as discussed in Sect. 4.3. However, the slowness of the bottom-up part under some settings drag down its performance. The performance gap between top-down and bottom-up approaches is further enlarged on large dense datasets, as shown in Fig. 13. Both `accidents` and `kddcup99` datasets have larger size and longer transactions. The LCM algorithm is up to 3 order of magnitude slower

than our LSCMiner. Even the naive approach is better under most cases. Though increasing β slows down our LSCMiner, it is still hard for LCM algorithm to overtake in the low support pattern mining scenario.



(a) `accidens` dataset (default: $N = 300k, L = 15, \beta = 50$)

(b) `kddcup99` dataset (default: $N = 1000k, L = 16, \beta = 20$)

Fig. 13. Runtime on large dense datasets.

Sparse Data. In theory, bottom-up algorithms should perform better than our top-down approach on a sparse dataset. According to our analysis above, both LCM and our LSCMiner have a complexity of $O(\frac{|\mathcal{C}|+|\mathcal{UC}|}{|\mathcal{C}|}P(|\mathcal{T}|))$. In the case of sparse datasets, $|\mathcal{UC}|$ in LCM is much smaller since the number of frequent patterns is tiny. On the other hand, a sparse dataset leads to a huge ni-tree, which is a substantial overhead for LSCMiner. Figure 14 illustrates the runtime performance on a sparse dataset `BMS2`. The results fulfill our expectations. The combined approach performs the best since we can take advantage of both bottom-up and top-down algorithms.

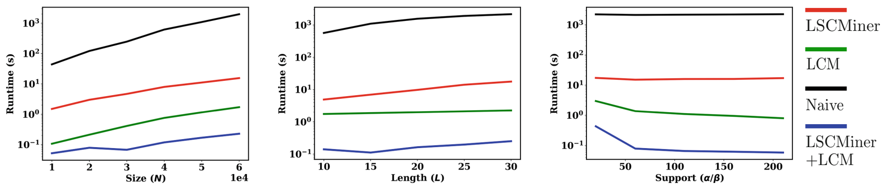
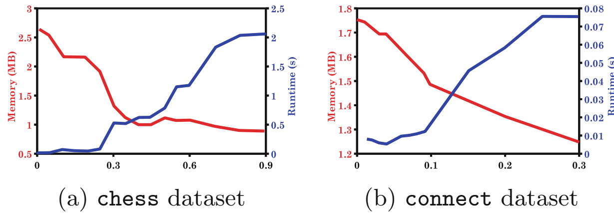


Fig. 14. Runtime on `BMS1` dataset (default: $N = 60k, L = 30, \beta = 20$)

Memory-Performance Trade-Off. As discussed above, the performance of our LSCMiner is limited by the dataset size. An extra constant scale factor s exists since the ni-tree stores the negative dataset. Sparsity is a crucial factor that affects the memory consumption of LSCMiner. By applying the bi-directional traversing framework and adjusting the value of splitting threshold δ, the LSCMiner only need to traverse the densest part of the dataset, which costs much fewer memories. We can still benefit from the efficient top-down traversing since top-down traversing is powerful on dense dataset while bottom-up traversing is good at the sparse dataset, as shown in experiments above. In this section, we study the relation between memory consumption and runtime performance by investigating how the trade-off behaviors with respect to different (relative) dividing threshold δ.

Two dense datasets, `chess` and `connect`, are selected as representatives since they have both sparse and dense parts. We measure the memory consumption using the first 1k transactions in each dataset. The runtime value for `connect` dataset is measured with the first 10k transactions instead. When the value of δ is close to 0, all patterns are extracted by the LSCMiner. When the value of δ is close to the relative support of the most frequent item, the bottom-up approach extracts all patterns. Thus, by increasing δ, the bi-directional traversing is moving from purely LSCMiner to purely LCM algorithm (Fig. 15).



(a) `chess` dataset (b) `connect` dataset

Fig. 15. Memory consumption and runtime under different δ values. δ is set up to 0.4 on `connect` dataset since almost all items occurred less than 40%.

On the `chess` dataset, the runtime of the bi-directional framework increased about 20 times while the memory consumption decreased about 2.5 times when moving from LSCMiner to LCM approach. On the `connect` dataset, the runtime increased about 7 times while the memory consumption decreased about 30%. LSCMiner is beneficial on both datasets: we spend some memory but get much better performance.

6 Conclusion

We present a very efficient low support closed pattern mining algorithm, LSCMiner, which avoids traversing undesired frequent patterns. It is particularly effective on datasets with huge amounts of frequent patterns. Though it

is memory expensive to store the ni-tree, much better runtime performance is achieved in return. Furthermore, we can balance the memory consumption and runtime performance by using the bi-directional traversing framework. If only those dense datasets are considered, our LSCMiner guarantees to provide the best performance in time complexity.

References

1. Adda, M., Wu, L., Feng, Y.: Rare itemset mining. In: Sixth International Conference on Machine Learning and Applications, ICMLA 2007, pp. 73–80. IEEE (2007)
2. Burdick, D., Calimlim, M., Gehrke, J.: Mafia: a maximal frequent itemset algorithm for transactional databases. In: Proceedings of the 17th International Conference on Data Engineering, pp. 443–452. IEEE (2001)
3. Fang, G., Pandey, G., Wang, W., Gupta, M., Steinbach, M., Kumar, V.: Mining low-support discriminative patterns from dense and high-dimensional data. IEEE Trans. Knowl. Data Eng. **24**(2), 279–294 (2012)
4. Fournier-Viger, P., et al.: The SPMF open-source data mining library version 2. In: Berendt, B., et al. (eds.) ECML PKDD 2016. LNCS (LNAI), vol. 9853, pp. 36–40. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46131-1_8
5. Gupta, A., Mittal, A., Bhattacharya, A.: Minimally infrequent itemset mining using pattern-growth paradigm and residual trees. In: Proceedings of the 17th International Conference on Management of Data, p. 13 (2011)
6. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: ACM Sigmod Record, vol. 29, pp. 1–12. ACM (2000)
7. Hoque, N., Nath, B., Bhattacharyya, D.: An efficient approach on rare association rule mining. In: Bansal, J., Singh, P., Deep, K., Pant, M., Nagar, A. (eds.) BIC-TA 2012, vol. 201, pp. 193–203. Springer, India (2013). https://doi.org/10.1007/978-81-322-1038-2_17
8. Kamehkhosh, I., Jannach, D., Ludewig, M.: A comparison of frequent pattern techniques and a deep learning method for session-based recommendation. In: RecTemp@ RecSys, pp. 50–56 (2017)
9. Koh, Y.S., Ravana, S.D.: Unsupervised rare pattern mining: a survey. ACM Trans. Knowl. Discov. Data (TKDD) **10**(4), 45 (2016)
10. Leroy, V., Kirchgessner, M., Termier, A., Amer-Yahia, S.: TopPi: an efficient algorithm for item-centric mining. Inf. Syst. **64**, 104–118 (2017)
11. Lu, Y., Richter, F., Seidl, T.: Efficient infrequent itemset mining using depth-first and top-down lattice traversal. In: Pei, J., Manolopoulos, Y., Sadiq, S., Li, J. (eds.) DASFAA 2018. LNCS, vol. 10827, pp. 908–915. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91452-7_58
12. Lu, Y., Seidl, T.: Towards efficient closed infrequent itemset mining using bi-directional traversing. In: DSAA 2018, pp. 140–149. IEEE (2018)
13. Mannhardt, F., De Leoni, M., Reijers, H.A., Van Der Aalst, W.M.: Balanced multi-perspective checking of process conformance. Computing **98**(4), 407–437 (2016)
14. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Efficient mining of association rules using closed itemset lattices. Inf. Syst. **24**(1), 25–46 (1999)
15. Smets, K., Vreeken, J.: Slim: directly mining descriptive patterns. In: Proceedings of SIAM International Conference on Data Mining, pp. 236–247. SIAM (2012)

16. Szathmary, L., Napoli, A., Valtchev, P.: Towards rare itemset mining. In: 19th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2007, vol. 1, pp. 305–312. IEEE (2007)
17. Troiano, L., Scibelli, G.: A time-efficient breadth-first level-wise lattice-traversal algorithm to discover rare itemsets. Data Min. Knowl. Disc. **28**(3), 773–807 (2014)
18. Tsang, S., Koh, Y.S., Dobbie, G.: RP-tree: rare pattern tree mining. In: Cuzzocrea, A., Dayal, U. (eds.) DaWaK 2011. LNCS, vol. 6862, pp. 277–288. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23544-3_21
19. Uno, T., Kiyomi, M., Arimura, H.: LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In: Fimi, vol. 126 (2004)