Knowledge-Based Systems 55 (2014) 125–139

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Efficient frequent pattern mining based on Linear Prefix tree

Gwangbum Pyun [a], Unil Yun [a,*], Keun Ho Ryu [b]

[a] Department of Computer Engineering, Sejong University, Seoul, Republic of Korea
[b] Department of Computer Science, Chungbuk National University, Cheongju, Republic of Korea

ARTICLE INFO

Article history:
Received 24 April 2013
Received in revised form 11 October 2013
Accepted 12 October 2013
Available online 24 October 2013

Keywords:
Data mining
Frequent pattern mining
Linear tree
Pattern growth
Knowledge discovery

ABSTRACT

Outstanding frequent pattern mining guarantees both fast runtime and low memory usage with respect to various data with different types and sizes. However, it is hard to improve the two elements since runtime is inversely proportional to memory usage in general. Researchers have made efforts to overcome the problem and have proposed mining methods which can improve both through various approaches. Many of state-of-the-art mining algorithms use tree structures, and they create nodes independently and connect them as pointers when constructing their own trees. Accordingly, the methods have pointers for each node in the trees, which is an inefficient way since they should manage and maintain numerous pointers. In this paper, we propose a novel tree structure to solve the limitation. Our new structure, LP-tree (Linear Prefix – Tree) is composed of array forms and minimizes pointers between nodes. In addition, LP-tree uses minimum information required in mining process and linearly accesses corresponding nodes. We also suggest an algorithm applying LP-tree to the mining process. The algorithm is evaluated through various experiments, and the experimental results show that our approach outperforms previous algorithms in term of the runtime, memory, and scalability.

1. Introduction

As a part of the association rule mining, frequent pattern mining is a method for finding frequent patterns in large data [15]. The patterns obtained from mining operations are usefully utilized to analyze data characteristics or gain information needed for decision-making. In addition, it can be applied in a variety of real data analyses such as web data [20], customer data in finance, correlation of product data, vehicle and communication data [9], bio data [13], hardware monitoring of computer system [45], and regular pattern mining [28]. In pattern mining, a pattern is a set of items in a certain database, and a support of the pattern is defined as the number of transactions containing the pattern, where we regard patterns satisfying a given minimum support threshold as frequent ones. Apriori [1] and FP-growth [14] are fundamental algorithms in frequent pattern mining, and current studies are proceeding based on the two algorithms. Moreover, other numerous methods have been suggested. First, there are methods using closed patterns such as BMCIF [8], and CEMiner [9] and those for maximal patterns such as MAFIA [5], FP-MAX [12], LFIMiner [16], MCWP [41], and MWS [42]. Furthermore, there exist other approaches for stream environments such as WMFP-SW [19], BSM [30], CPS-tree [31], and RPS-tree [32], and for utility patterns such as HUIPM [2], HUPMS [3], and UP-growth [34]. The following techniques apply item weights into the mining process. WARM [33], WAS [39], and MWFIM [40] are weight-based algorithms, and TIWS [7] adds weights with times. In addition, there is an approach which finds frequent patterns from the top support to *k*th support without any given minimum support threshold. The method is called Top-k pattern mining, and typical studies are MinSummary [18], PND [24], Chenoff [35], Topk-PU [43], SpiderMine [44], etc. In the sequential pattern mining considering item sequence, there are SeqStream [6], StreamCloseq [10], ApproxMAP [17], TD-seq [22], CSP [27], WSpan [38], and so forth. U2P-Miner [23] mines uncertain data, and GAMiner [36] gives meaning to interesting patterns and then extracts patterns. Developing an improved algorithm for the frequent pattern mining can contribute to advancing mining performance in various mining fields. FP-growth-based frequent pattern mining, such as FP-growth* [12], patricia-tree [26], and IFP-growth [21], has the following characteristics. FP-growth has connection information among all nodes in FP-tree in order to search the nodes. Therefore, it has many pointers for connecting nodes, thereby using a lot of runtime and memory resources. In this paper, we, therefore, propose a novel tree structure, LP-tree (Linear prefix–tree) and an algorithm using the tree, called LP-growth which can conduct mining operations more quickly and efficiently than previous algorithms. Our LP-tree can solve the above limitation due to its special structure based on the linear form. We can obtain advantages by converting tree's nodes as array forms. It can increase memory efficiency through arrayed nodes since they can reduce connection

* Corresponding author. Tel.: +82 234082902.
E-mail addresses: gbpyun@sju.ac.kr (G. Pyun), yunei@sejong.ac.kr (U. Yun), khryu@cbnu.ac.kr (K.H. Ryu).

0950-7051/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.knosys.2013.10.013

information. We can also speed up item traversal times since LP-tree does not use pointers in most cases and generates a large number of nodes at once due to its linear structure. By applying the features of LP-tree to mining process, we can obtain the following benefits: (1) Tree generation rate of our approach becomes faster than that of FP-growth since ours can create multiple nodes at once by a series of array operations. Meanwhile, FP-growth makes nodes one by one. (2) We can access parent or child nodes without corresponding pointers when searching trees since the nodes are stored as an array form. (3) Memory usage for each node becomes relatively small since LP-tree does not require internal node pointers. (4) It is possible to traverse trees more quickly compared to searching for them with pointers since our approach directly accesses corresponding memories due to the feature of the array structure. This paper is organized as follows. In Section 2, we introduce related work with respect to LP-tree and LP-growth, and describe details for our techniques and algorithm in Section 3. Next, we compare performance of our algorithm with those of previous algorithms through various experiments in Section 4, and we finally conclude this paper in the last section.

2. Related work

Frequent pattern mining extracts specific patterns with supports higher than or equal to a minimum support threshold, and many of mining methods have been researched as mentioned above, but Apriori [1] and FP-growth [14] are still regarded as underlying algorithms. Apriori is the oldest conventional mining algorithm, and it performs mining operations by extending pattern lengths. The algorithm generates candidate patterns through the pattern extension in advance, and then confirms whether the candidates are actually frequent patterns by scanning a database. Consequently, Apriori has no choice but to scan the database as much as the maximum length among frequent patterns. UT-Miner [37] is an improved Apriori algorithm specialized in sparse data, where sparse data indicate that most transactions are different from each other. The algorithm uses an array structure, *unit triple* storing relations between items and transactions in a database to improve mining performance. However, UT-Miner does not guarantee fine performance in terms of runtime and memory usage since the algorithm is based on Apriori method. On the other hand, FP-growth [14] solved the above problem by scanning a database only twice. It uses a tree structure, called FP-tree, which can prevent the algorithm from generating candidate patterns. FP-tree consists of a tree for storing database information and a header table containing item names, supports, and node links. A tree is composed of nodes, where each of them includes an item name, a support, a parent pointer, a child pointer, and a node link. The node link is a pointer that connects all nodes with the same item to each other. Since the FP-growth algorithm was proposed, various algorithms have been published on the basis of the algorithm. FP-growth-goethals [11] is a FP-growth implementation which is optimized by Bart-Goethals. To increase efficiency of search space in FP-growth, FP-growth-tiny [25] generates conditional FP-trees using conditional patterns without creating any conditional database. In CT-PRO [29], the authors suggested Compressed FP-tree adding a count array into the nodes of the FP-tree, where each entry of the array corresponded to the number of itemset's occurrences. The algorithm mines frequent patterns using the information added in the tree without recursive calls. IFP-growth [21] enhanced pruning effect with a new tree structure, FP-tree+, where the tree adds an address table to the FP-tree. Therefore, the algorithm decreases the number of conditional FP-trees, thereby improving mining speed. Meanwhile, it needs more information than the original FP-tree. In addition, IFP-growth does not upgrade memory efficiency although this contributes to reducing runtime. MAFIA-FI [5] saves data information into a bitmap form so as to reduce the number of tree searches. The bitmap is made up of two dimensions, where x-axis means items and *y*-axis is transactions. For example, a point (2,4) of a certain bitmap means that the second item exists in the fourth transaction. Thus, MAFIA-FI can compute patterns or items' supports through "AND" operations of the bitmap without tree traversals. In addition, the algorithm can prevent creating needless trees with infrequent patterns and maximize pruning efficiency. However, MAFIA-FI requires more memory although its runtime is faster than the original method. Patricia-tree [26] also uses an array structure to a part of the FP-tree, where the algorithm generates paths with the same support as an array. Meanwhile, the LP-tree proposed in this paper constructs all paths as arrays regardless of items' supports, where the shapes of the arrays vary depending on each transaction's form. FP-growth* [12] proposed FP-array with pattern information and increased pruning efficiency with FP-array. The approach calculates supports of patterns to be expanded in advance, and eliminates infrequent patterns effectively through the proposed FP-array. However, FP-growth* also does not reduce the size of trees since it still uses the original FP-tree-based structures. As a result, we need to develop a new tree structure to improve fundamental performance of the mining algorithm. Consequently, we propose a novel tree and algorithm for satisfying both runtime and memory efficiency. In our LP-tree, its runtime and memory performances are more outstanding than those of FP-growth* due to its special tree structure based on the array.

3. Frequent pattern mining based on Linear Prefix-tree

In this section, we present details of LP-growth algorithm and related techniques. The algorithm conducts mining operations with LP-tree and corresponding growth methods.

3.1. Preliminaries

Given a transaction database, D, $I = \{i_1, i_2, \ldots, i_n\}$ is a set of items composing D, and D consists of multiple transactions. All transactions have each a unique set of items. D includes unique *ID*s, called *TID*s, with respect to each transaction. A pattern is defined as a sub or whole set of I. Assuming that any pattern P has several items and its first and last ones are i_b and i_e respectively, P is denoted as follows.

$$P = \{i_b, \ldots, i_e\}, \quad 1 \leqslant b < e \leqslant n.$$

P's support means the number of transactions containing in D. In other words, this indicates how much P occurs in D. Let $|P|$ be the number of transactions including P and $|D|$ be the number of all transactions in D. Then, we can calculate P's support rate, $sup(P)$ as follows.

$$sup(P) = |P|/|D|,$$

where $0 \leqslant sup(P) \leqslant 1$. P is regarded as a frequent pattern if $sup(P)$ is not smaller than a given minimum support (or *minsup*). Denoting the frequent P as L, it is also included in I and satisfies $sup(L) \geqslant minsup$, where $0 \leqslant minsup \leqslant 1$.

$$L = \{P \subseteq I | sup(P) \geqslant minsup\}.$$

For instance, given a database, {{*TID*1: a, b, c}, {*TID*2: a, b}, and {*TID*3: b, c, d, e}}, I becomes I = {a, b, c, d, e}. If a minimum support threshold is 60%, a pattern, {a, b} is frequent since it appears in *TID*1 and 2; thus, its support is higher than the threshold. Meanwhile, another