การสืบค้นกลุ่มรายการที่เกิดบ่อยโดยใช้กลุ่มรายการขยายได้

AN EXTENDABLE ITEMSET APPROACH TO FREQUENT ITEMSETS MINING

สุภัทรา สหพงศ์

SUPATRA SAHAPHONG

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาปรัชญาดุษฎีบัณฑิต

สาขาวิชาวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2555

KMITL–2012– SC–D–002– 013

AN EXTENDABLE ITEMSET APPROACH TO FREQUENT ITEMSETS MINING

SUPATRA SAHAPHONG

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

FACULTY OF SCIENCE

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2012

KMITL-2012-SC-D-002-013

หัวข้อวิทยานิพนธ์	การสืบค้นกลุ่มรายการที่เกิดบ่อยโดยใช้กลุ่มรายการขยายได้
นักศึกษา	นางสาวสุภัทรา สหพงศ์
รหัสประจำตัว	49062906
ปริญญา	ปรัชญาดุษฎีบัณฑิต
สาขาวิชา	วิทยาการคอมพิวเตอร์
พ.ศ.	2555
อาจารย์ที่ปรึกษาวิทยานิพนธ์	รองศาสตราจารย์ ดร.วีระ บุญจริง

บทคัดย่อ

วัตถุประสงค์ของงานวิจัยนี้เพื่อพัฒนาการสืบค้นกลุ่มรายการที่เกิดบ่อยจากฐานข้อมูลรายการ งานหลักของงานวิจัยนี้ได้แก่ (1) การนำเสนอขั้นตอนวิธีใหม่ที่อ่านฐานข้อมูลเพียงหนึ่งครั้งเพื่อสร้างโครงสร้างข้อมูลที่เรียกว่าโครงสร้างดัชนีผกผัน หรือ ไอไอเอส (2) การเปลี่ยนค่าความถี่ขั้นต่ำ ไม่ส่งผลกระทบต่อโครงสร้างข้อมูลที่ใช้และไม่ต้องอ่านฐานข้อมูลซ้ำ (3) การนำเสนอขั้นตอนวิธีการสืบค้นแบบใหม่ชื่อ ไอไอเอส-ไมน์ ที่มีการใช้คุณสมบัติที่เรียกว่ากลุ่มรายการขยายได้ ซึ่งสามารถลดขั้นตอนการสืบค้นแบบเรียกซ้ำโดยปราศจากการสร้างเซตรายการแคนดิเดท จึงทำให้ค้นพบกลุ่มรายการที่เกิดบ่อยได้อย่างรวดเร็ว งานวิจัยนี้มีการนำเสนอคำจำกัดความ ตัวอย่าง ทฤษฎี การพิสูจน์ความสมบูรณ์และความถูกต้องโดยใช้คณิตศาสตร์ มีการนำเสนอผลการทดลองเชิงเปรียบเทียบกับขั้นตอนวิธีเอฟพี-โกรท ในด้านการใช้เวลาประมวลผล การใช้เนื้อที่ในหน่วยความจำและขนาดของข้อมูล เมื่อค่าความถี่ขั้นต่ำเปลี่ยนแปลงไป การประเมินค่าของขั้นตอนวิธีทั้งสองวิธีนั้นใช้ชุดข้อมูลสังเคราะห์และชุดข้อมูลจริง ผลการทดลองปรากฏว่าไอไอเอส-ไมน์มีประสิทธิภาพด้านเวลาการประมวลผลและการใช้เนื้อที่หน่วยความจำที่ดีกว่าเอฟพี-โกรท และทำงานได้ดีกับชุดข้อมูลแบบหนาแน่น

คำสำคัญ : การสืบค้นกฎความสัมพันธ์, การสืบค้นกลุ่มรายการที่เกิดบ่อย, การสืบค้นรูปแบบการเกิดบ่อย, การสืบค้นความรู้

Thesis Title	An Extendable Itemset Approach to Frequent Itemsets Mining
Student	Miss Supatra Sahaphong
Student ID	49062906
Degree	Doctor of Philosophy
Program	Computer Science
Year	2012
Thesis Advisor	Associate Professor Dr. Veera Boonjing

ABSTRACT

A new approach to mine all frequent itemsets from a transaction database is proposed. The main features of this paper are as follows: (1) the proposed algorithm performs database scanning only once to construct a data structure called an inverted index structure (IIS); (2) the change in the minimum support threshold is not affected by this structure, and as a result, a rescan of the database is not required; and (3) the proposed mining algorithm, IIS-Mine, uses an efficient property of an extendable itemset, which reduces the recursiveness of mining steps without generating candidate itemsets, allowing frequent itemsets to be found quickly. We have provided definitions, examples, and a theorem, the completeness and correctness of which is shown by mathematical proof. We present experiments in which the run time, memory consumption and scalability are tested in comparison with a frequent-pattern (FP) growth algorithm when the minimum support threshold is varied. Both algorithms are evaluated by applying them to synthetics and real-world datasets. The experimental results demonstrate that IIS-Mine provides better performance than FP-growth in terms of run time and space consumption and is effective when used on dense datasets.

Keywords : association rule mining, data mining, frequent itemsets mining, frequent patterns mining, knowledge discovering

ACKNOWLEDGEMENTS

I would like to express my grateful thanks to all of those people and institutes who have provided me various supports in this thesis. First, this work was supported by the National Centre of Excellence in Mathematics, PERDO, Office of the Higher Education Commission, Thailand. Second, Ramkhamhaeng University (RU) which provided partial scholarship support for this study. Third, my mother, Mrs. Jintana Sriaroon who has given my live with inspiration and motivation to complete the highest education. Fourth, all teachers at Faculty of Science, King Mongkut's Institute of Technology Ladkrabang (KMITL), all examiner committees and my advisor, Associate Professor Dr. Veera Boonjing, for all of their help and valuable advice. Last, Professor Dr. Apichart Suksamrarn from Faculty of Science, RU who have helped and provided so many valuable advices and supports. Moreover, I would like to express my sincerely thanks to Associate Professor Dr. Tawesak Tanwandee, M.D., Faculty of Medicine, Siriraj Hospital, Mahidol University (MU) for his language advice and Assistance Professor Dr. Gumpon Sritanratana, Faculty of Science, MU who for his mathematical advice and to all Ph.D. students at KMITL who have helped and supported me throughout the studying period.

Supatra Sahaphong

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT)

LIST OF FIGURES

Figures **Page**

LIST OF TABLES

Tables **Page**

CHAPTER 1

INTRODUCTION

1.1 Statements of Problem

Nowadays there are tremendous amounts of data which have been collected and stored in large databases, and which cannot be analyzed using manual systems. For example, how can one find the items which occur together from large amounts of data, how do the items in a massive data set relate to the others, what items in the data set frequently appear, etc.? Therefore, powerful data analysis is necessary to solve these requirements. One solution is to use frequent itemsets mining. Frequent itemset mining is a popular problem in frequent pattern mining. It leads to the discovery of associations and correlations among items in a large transaction data set.

The objective of frequent itemset mining is to identify all frequently occurring itemsets using a support threshold. Decision-makers are interested in all itemsets associated with high frequencies. Association rule mining algorithms can be broken down into two major phases. The first phase finds all of the itemsets that satisfy the minimum support threshold, which are the frequent itemsets. The second phase is rule generation, in which all the high confidence rules from the frequent itemsets found in the previous phase are extracted [1]. Many previous investigations focused on the first phase. Early algorithms based on generated and tested candidate itemsets have two major defects. First, the database must be scanned multiple times to generate candidate itemsets, which increases the I/O load and is time-consuming. The search space of itemsets that must be explored grows exponentially. Second, enormous candidate itemsets are generated and calculated from their supports, which consumes a large amount of CPU time [2]. To overcome the above-mentioned problems, a next generation of algorithms using a compact tree structure was proposed, called a frequent-pattern (FP) tree [3], which finds frequent itemsets directly from the data structure. However, most of the FP-tree-based algorithms have the following weaknesses. First, the mining of frequent itemsets from the FP-tree to generate a huge conditional FP-tree requires a large amount of run time and space. The best case is when a database has the same set of transactions; an FP-tree then contains only a single branch of nodes. The worst case is when a database has a unique set of transactions. Second, when the users change to a new minimum support threshold for their new decision, the algorithm restarts the

whole operation and scans the database twice. Many researchers have tried to solve the above problems using a vertical data layout. However, most of the algorithms have the drawback of increasing the run time and space consumption due to the following reasons. First, when the users change to a new minimum support threshold for their new decision, the algorithm restarts the whole operation more than one time to scan a database and construct their data structure. Rescanning the database for a new minimum support threshold wastes both run time and space. Second, all of the FP-tree-based algorithms generate a huge conditional FP-tree, which has a large number of recursive processing steps and requires a large amount of run time and space consumption.

From the above-mentioned, the motivations of this thesis are proposed a new, efficient method to improve both a data structure and a mining algorithm for decreasing the consumption of run time and space.

1.2 Research Objectives

This thesis is proposed to study theories and algorithms for development of a new approach to mine all frequent itemsets from a transaction database which can improve run time and space consumption. The proposed method performs database scanning only once to construct a data structure called an inverted index structure (IIS), the change in minimum support threshold is not affected by this structure, the proposed algorithm called IIS-Mine which used an efficient property of an extendable itemset to reduce the recursion of mining steps without generating candidate itemsets. This thesis has provided definitions, examples, and a theorem, the completeness and correctness of which is shown by mathematical proof. The experiments, i.e. run time, memory consumption and scalability, are tested in comparison with a frequent-pattern growth (FP-growth) algorithm when the minimum support threshold is varied by using synthetics and real-world datasets.

1.3 Research Methodology

The operation research of the proposed was started from the followings. First, the definitions, examples and algorithms are created to demonstrate how to construct the IIS and IIS-Mine algorithm. Second, a theorem and proof are giving to demonstrate that the proposed IIS-

Mine algorithm can mine frequent itemsets completely and correctly. Last, the experiments are presented to test the run time, memory consumption and scalability of the IIS-Mine and FP-growth algorithm with different datasets and varying minimum support thresholds. All algorithms were code using C languages. Two groups of benchmark datasets, i.e. two synthetic datasets which are generated by the IBM Almaden Quest research group and real datasets from the UCI Machine Learning Repository, were used.

1.4 Organization of Thesis

This thesis is organized as the followings.

Chapter 2 provides the literature review which includes basic knowledge of frequent itemsets mining.

Chapter 3 presents the proposed method which includes design and construction of an inverted index structure, and algorithm named IIS-Mine.

Chapter 4 describes the experiment result to demonstrate the efficiency of our new method including run time, memory consumption, and scalability.

Chapter 5 presents conclusion and recommendation.

CHAPTER 2

LITERATURE REVIEWS

This chapter describes the literature review which is related to the subjects in this dissertation. In section 2.1, we follow the mining of frequent itemsets using three methods of database scanning: multiple times, two times, and one time. We presented the weaknesses at the end of each category. The basic concept and FP-Growth algorithm are presented in section 2.2.

2.1 Frequent Itemsets Mining

Frequent itemset is the one type of frequent patterns. Frequent patterns are patterns (such as itemsets, subsequences, or substructures) that appear in a data set frequently [1]. For example, a set of items, such as milk and bread that appear frequently together in a transaction data set is a frequent itemsets. Frequent itemset mining is a useful step for discovering the interesting associations between items which are hidden in large database. The uncovered associations between items can be represented in the form of association rule or sets of frequent items. The classic application is market basket analysis which is the analysis how the items purchased by customers are associated. The association rule mining is wildly applied in many research fields such as decision support, financial forecasts, bioinformatics, web mining, and scientific data analysis. Moreover, it helps in data classification, clustering and others data mining task as well. Association rule mining algorithms can be broken down into two major phases. The first phase finds all of the itemsets that satisfy the minimum support threshold, which are the frequent itemsets. The second phase is rule generation, in which all the high confidence rules from the frequent itemsets found in the previous phase are extracted.

The computational requirements for frequent itemsets generation are more expensive than rule generation. In decades, many researchers have been trying to extend and improve both data structure and mining algorithm to mine frequent itemsets in large database. We followed mining of frequent itemsets by using the revolution of scanning of database which divided into three categories of database scanning such as the scanning of database multi time, two times, and one time.

2.1.1 Frequent Itemset Mining By Scanning a Database Multiple Times

The algorithms in this group use level-wise algorithms. They traverse the itemset lattice one level at a time, from the level of 1-frequent itemset to the maximum size of frequent itemsets. These algorithms employ a generate-and-test strategy for finding frequent itemsets. At each level, the algorithm must scan the database once, so the total number of database scannings needed is determined by the maximum size of frequent itemsets.

The first algorithm to generate all frequent itemsets is the AIS algorithm, which was first introduced by Agrawal et al [4]. However, this algorithm constructs a list of all of the possible itemsets at each level of traversal, so infrequent itemsets that are not needed are also generated. Later, the algorithm was improved upon and renamed the Apriori algorithm by Agrawal et al [5]. The Apriori algorithm uses a level-wise and breadth-first search approach for generating association rules. Many efficient association mining techniques have been developed based on the Apriori algorithm. Vu et al. [6] proposed a rule-based location prediction technique to predict the user's featured location, but this proposal generates more candidate itemsets than are required. These algorithms are also expensive in terms of I/O load and run time when the database must be scanned multiple times to generate candidate itemsets.

2.1.2 Frequent Itemset Mining By Scanning a Database Twice

The algorithms in this group do not subscribe to the generate-and-test paradigm of the Apriori algorithm. They use a compact tree structure and find frequent itemsets directly from the data structure. The algorithms scan a database twice. The first scan of the database is to discard infrequent itemsets; the second scan is to construct a tree. Many algorithms have been proposed to improve the problems mentioned in the previous section.

The FP-growth algorithm, developed by Han et al. [7], is the most popular method. It performs a depth-first search approach in a search space. It encodes a dataset using a compact data structure called an FP-tree or prefix tree and extracts frequent patterns directly from the FP-tree. Many approaches have been proposed to extend and improve upon this algorithm. Pei et al. [8] developed the H-mine algorithm using array- and tree-based data structures to improve the main memory cost. The PatriciaMine algorithm [9] compressed Patricia tries to store datasets, which is space efficient for both dense and sparse datasets. The

FP-growth algorithm [10] reduces the FP-tree traversal time using an array technique. Zhu [11] proposed a new method to compress a large database into an FP-tree with a children table but not a header table, and applied a depth-first search with this tree for the mining step, which reduces both the run time and the space consumption. Sahaphong and Boonjing [12] proposed a new algorithm which constructs a pattern base using a new method that is different from the pattern base in the FP-growth and mined frequent itemsets using a new combination method without the recursive construction of a conditional FP-tree. An approach based on the FP-tree and co-occurrence frequent items (COFI) was proposed to find frequent items in multilevel concept hierarchy by using a non-recursive mining process [13]. A new data structure called improved FP-tree was proposed, which can reduce space consumption and enhance the efficiency of an attribute reduction algorithm [14]. To maintain the anti-monotone property of approximate weighted frequent patterns, a robust concept was proposed to relax the requirement for exact equality between the weighted supports of patterns and a minimum threshold [15]. However, most of the FP-tree-based algorithms require a large amount of run time and space to generate the huge conditional FP-trees. Moreover, the algorithm restarts the whole operation and requires that a database be scanned twice when the minimum support threshold is changed.

2.1.3 Frequent Itemset Mining By Scanning a Database Once

In sections 2.1.1 and 2.1.2, most of the algorithms that mine frequent itemsets use a horizontal data layout. However, many researchers use a vertical data layout. The Eclat algorithm was proposed [16] to generate all frequent itemsets in a breadth-first search using the joining step from the Apriori property when no candidate items can be found. The Eclat algorithm is very efficient for large itemsets but is less efficient for small ones. The diffset technique [17] was introduced to improve the memory requirement. Chai et al. [18] detailed a data structure called large-item bipartite graph to accommodate the data when a database is scanned. Similar to the FP-growth algorithm, this method mines frequent patterns using the recursive conditional FP-tree. The BitTableFI algorithm [19] uses horizontal and vertical data layouts to compress a database. Yen [20] presented an algorithm based on an undirected itemset graph that finds frequent itemsets by searching undirected graphs. When the database and minimum support change, this algorithm requires that the graph structure be re-searched to generate new frequent itemsets. The Index-

BitTableFI [21] was developed to reduce the cost of candidate generation and to support counting. Sahaphong and Boonjing [22] proposed a new algorithm that reduces the run time. The drawback of this algorithm is its large memory consumption from generation of many repeated nodes. The JoinFI-Mine algorithm [23] uses a sorted-list structure constructed from the vertical data layout and finds all frequent itemsets using a depth-first search for joining frequent itemsets. Therefore, this algorithm consumes time and space in its joining step.

2.2 Prior Knowledge

2.2.1 Basic Concept

This subsection introduces the basic concepts of mining frequent itemsets. All terminologies in this section have been proposed by Han *et al.* [2].

Let $I = \{x_1, x_2, ..., x_m\}$ be a set of items and $DB = \{T_1, T_2, ..., T_n\}$ be a transaction database, where $T_1, T_2, ..., T_n$ are transactions that contain items in I. The support, or *supp* (occurrence frequency), of a pattern A, where A is a set of items, is the number of transactions containing A in DB. A pattern A is frequent if A's support is no less than a predefined minimum support threshold, *minsup*.

Given a DB and a minimum support threshold *minsup*, the problem of finding the complete set of frequent itemsets is called the frequent itemsets mining problem.

For greater understanding, we provide an example to illustrate the above definitions.

Example 2.1. The example of the database by Han *et al.* [2] has been used by our group. Here, Table 2.1 is a DB. It consists of 5 transactions (T_1, T_2, T_3, T_4, and T_5), labeled as *Transactions* in the table, and 17 items (*a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p*, and *s*), labeled as *Items* in the table. For example, the first transaction is T_1, which contains *f, a, c, d, g, i, m*, and *p*.

Table 2.1. A transaction database

Transactions	Items
T_1	*f, a, c, d, g, i, m, p*
T_2	*a, b, c, f, l, m, o*
T_3	*b, f, h, j, o*
T_4	*b, c, k, s, p*
T_5	*a, f, c, e, l, p, m, n*

2.2.2 FP-growth Algorithm

This section introduces the concepts for mining of frequent itemset by using FP-growth. A FP-growth algorithm is introduced by Han J. et al [16], [17]. The algorithm does not subscribe to the generation of candidate itemsets paradigm of Apriori. It encodes the data set using a compact data structure called FP-tree and extracts frequent pattern directly from this prefix tree. This method consists of four main steps.

1. Scanning the DB once to find 1-frequent itemsets.
2. Construction the header table from 1-frequent itemsets by descending support sorted. Therefore, it consists of frequent itemsets and infrequent itemsets. The frequent itemsets are sorted by descending support and infrequent itemsets are discarded.
3. Scanning DB for the second times to construct FP-tree.
4. Exploring frequent pattern from FP-tree by using FP-growth algorithm.

The research paper [17] gives the definitions and examples that present as below.

Definition 2.1 (Frequent pattern tree). A frequent pattern tree (FP-tree) is a tree structure defined below.

(i) It consists of a root labeled as "null", a set of items-prefix subtrees which are the children of root and a frequent-header table.

(ii) Each node in the item-prefix subtree consists of three fields: *item-name*, *support* and *node-link*, where item-name is the name of node represents, support is the number of transactions represented by the portion of the part reaching this node, and node-link links to the next node in FP-tree carrying the same item-name or null.

(iii) The frequent-header table consists of two fields: *item-name* and *head of node-link* that point to the first node in FP-tree.

The construction of FP-tree algorithm presents as below:

Algorithm 2.1: FP-tree Construction.

Input: DB, *minsup*.

Output: FP-tree.

Procedure FP-treeConstruction

```
1 Begin
2  Scan the DB once to collect the frequent itemsets and their  support then sort in support
3          descending and create in header table
4  FP-tree = null
5  For each transaction t_i in DB do
6   Select and sort the frequent items in t_i by order in header table
7    Call InsertTree(FP-tree, t_i )
8  End //For
9 End //Begin
Procedure Insert_tree(root, DB)
1 Begin
2  For each item x_i in DB do
3      If root has a child N then N.item-name = x_i
4       Increment N++
5       root = N
6    Else
7       Create the new node x_i is the child of root
8       Link header table to node
9    End //If
10  End //For
11 End //Begin
```

Example 2.2. The DB is scanned for second times to create FP-tree. We first create the root of the tree.

The first transaction is leaded to the construction of the first branch of FP-tree: (<*f*:1>,<*c*:1>,<*a*:1>,<*m*:1>,<*p*:1 >). The items in the transaction will be inserted into the tree ordering by descending support value.

The second transaction (<*f*>,<*c*>,<*a*>,<*b*>,<*m*>) shares the same prefix (<*f*>,<*c*>,<*a*>) with the existing part (<*f*>,<*c*>,<*a*> ,<*m*>,<*p*>). Thus, the support of each node along the prefix is increasing by 1 and one new node <*b*:1> is created and link as a child of (*a*:2) and another new node <*m*:1> is created and linked as a child of <*b*:1>.

The third transaction (<*f*>,<b>), it shares only the node <*f*> and *f*'s support is increased by 1. The remained item <*b*:1> will be created as the new node and link as a child of <*f*:3>.

The fourth transaction is scanned and constructed to second branch of tree,(<*c*:1>,<*b*:1>,<*p*:1>).

The last transaction is (<*f*>,<*c*>,<*a*>,<*m*>,<*p*>) is identical to the first one, the path is shared with the count of each node along the path incremented by 1.

After all transactions are scanned, FP-tree is presented in figure 2.1.

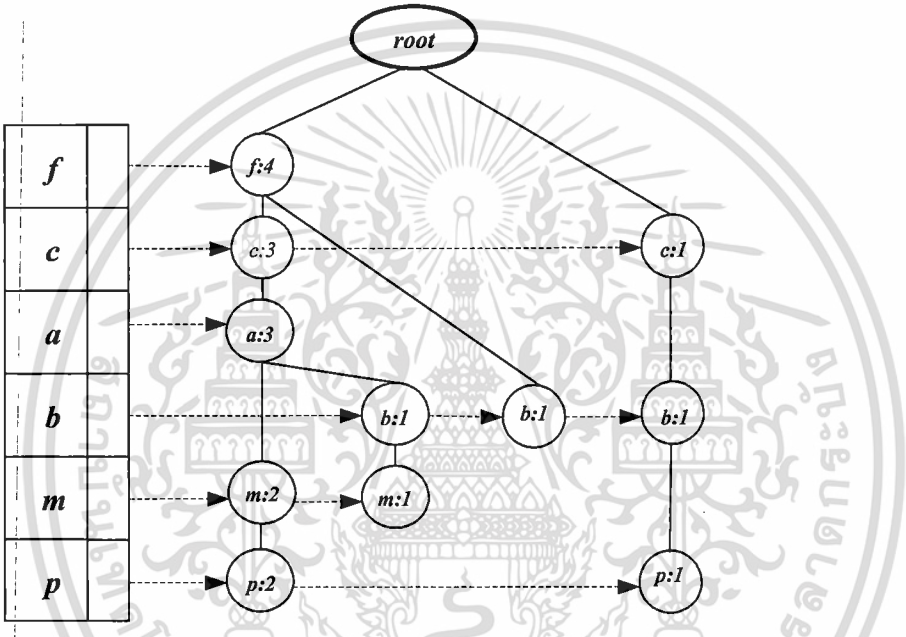

Figure 2.1. The FP-tree

Definition 2.2 (Conditional pattern-base). The conditional pattern-base is a sub database which consists of a set of prefix paths in FP-tree which co-occurrence with a suffix pattern.

Example 2.3. Let us examine the mining process for construction of pattern-base by starting from the bottom of the node-link of the header table, according to Algorithm 2.1.

For node *p*, there are two paths in FP-tree: (<*f*:4>,<*c*:3>,<*a*:3>,<*m*:2>,<*p*:2>), and (<*c*:1>,<*b*:1>,<*p*:1>). All nodes in the first path appear twice together with <*p*>, so we get only <*p*>'s prefix path (<*f*:2>,<*c*:2>,>,<*a*:2>,<*m*:2>) or <*fcam*:2> . Similarly, all nodes in the second path appear once together with <*p*>, <*p*>'s prefix path is (<*c*:1>,<*b*:1>) or <*cb*:1>. This two prefix paths of <*p*>, "{<*fcam*:2> , <*cb*:1>.}", from <*p*>'s subpattern-base, which is called <*p*>'s conditional pattern base.

For node *m*, it has two paths, (<*f*:4>,<*c*:3>,<*a*:3>,<*m*:2>) and (<*f*:4>,<*c*:3>,<*a*:3>,<*b*:1>, <*m*:1>). Similar to the above, <*m*>'s conditional pattern-base is {<*fca*:2>,<*fcab*:1>}.

The remaining nodes in the header table such as <*b*>, <*a*>, <*c*> and <*f*> in sequence are constructed in the same way. We summarize all conditional pattern-base in table 2.2.

Table 2.2. All conditional pattern-base

Items	Conditional pattern base
p	{<*fcam*:2>,<*cb*:1>}
m	{<*fca*:2>,<*fcab*:1>}
b	{<*fca*:1>,<*f*:1>,<*c*:1>}
a	{<*fc*:3>}
c	{<*f*:3>}
f	{}

Definition 2.3 (Conditional FP-tree). The conditional FP-tree is the sub tree which is constructed from a conditional pattern-base as mining pattern.

Example 2.4. From previous example, construction of an FP-tree on <*p*>'s conditional pattern-base is called <*p*>'s conditional FP-tree. The construction of <*p*>'s conditional FP-tree by doing the following steps. The <*p*>'s header table is constructed from conditional pattern-base by examining the support of each item that is not less than *minsup*.

Item	*f*	*c*	*a*	*b*	*m*
Support	2	3	2	1	2

Figure 2.2. The items and supports of the conditional pattern base of <*p*>

From the above table, there are only one item that has support is not less than *minsup*, <*c*:3>. The conditional FP-tree is constructed by examining the first path in conditional pattern-base, <*fcam*:2> which exclusion of infrequent items (or node which not appear in <*p*>'s header table) follow by insertion of frequent item <*cb*:1> to <*p*>'s conditional FP-tree, as seen in figure 2.3.

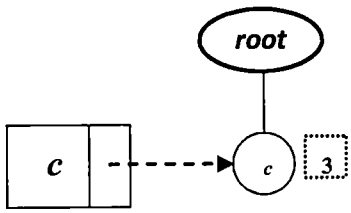

Figure 2.3. A conditional FP-tree for item <*p*>

Similar to the above, we derive <*m*>'s conditional FP-tree is (<f:3>,<c:3>,<a:3>), as shown in figure 2.4.

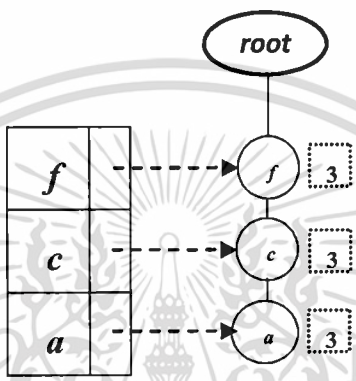

Figure 2.4. A conditional FP-tree for item <*m*>

Algorithm 2.2: FP-growth.

Input: FP-tree and *minsup*.

Output: The set of all frequent patterns.

Method: call FP-growth(FP-tree, *null*).

Procedure FP-growth(*Tree*, α)

1 Begin

2 if tree contains a single prefix path // Mining single path

3 let *P* be the single prefix-path part of tree;

4 let *Q* be the multiple path with the top branching node replaced by a *null* root;

5 for each combination (denote as β) of node in *P* do

6 generate itemset $\beta \cup \alpha$ with support=*minsup* of node in β ;

7 let freq_pattern_set(*P*) be the set of patterns so generated;

8 else let *Q* be Tree;

9 for each α_i in *Q* // Mining multiplepath

10 generate pattern $\beta = \alpha_i \cup \alpha$ with support= α_i .*suppot*;

11 construct β 's conditional pattern-base and β 's conditional FP-tree β ;

12 if $Tree_\beta \neq \{\}$ then call FP-growth($Tree, \beta$)

13 let freq_pattern_set(Q) be the set of patterns so generated;

14 return(freq_pattern_set(P) $\cup$ freq_pattern_set(Q) $\cup$

15 (freq_pattern_set(P) $\times$ freq_pattern_set(Q))

16 End //For

17 End //Begin

Example 2.5. This example presents the mining of frequent patterns. The mining process illustrate in figure 2.5 is based on construction FP-tree shown in figure 2.1. First, starting mine at node *p*.

For node *p*, the conditional pattern is {<*fcam*:2>,<*cb*:1>}. Constructing conditional FP-tree on branch <*c*:3>. Then we mine by calling "*mine* (<*c*:3>|*p*)". The items that involve mining is <*c*>. Hence, only one frequent item <*cp*:3> is derived.

For node *m*, the conditional FP-tree is then mined recursively by calling "mine (<*f*:3>,<*c*:3>,<*a*:3>)|*m*" involves minining three items <*a*>,<*c*>,<*f*> in sequence. The first derives a frequent item <*am*:3>, a conditional pattern-base {<*fc*:3>}, and then a call "*mine* (<*f*:3>,<*c*:3>)|*am*"; the second derives a frequent itemset <*cm*:3>, a conditional pattern-base <*f*:3>, and then call "mine (*f*:3)|*cm*"; and the third derives only a frequent itemset <*fm*:3>. Further recursive call of "mine(<*f*:3>,<*c*:3>)|*am*" derives two itemsets <*cam*:3> and <*fam*:3>, and conditional pattern-base {<*f*:3>}, call "mine (<*f*:3>)|*cam*" derives one pattern <*fcm*:3>. Therefore, the set of frequent itemsets involving <*m*> is {<*m*:3>, <*am*:3>, <*cm*:3>, <*fm*:3>, <*cam*:3>, <*fam*:3>, <*fcam*:3>,<*fcm*:3>}.

The remaining nodes in the header table such as <*b*>, <*a*>, <*c*> and <*f*> in sequence are constructed in the same way. The conditional FP-trees and the sets of frequent pattern are summarized in table 2.3. All frequent itemsets which separate by sequential of the *k*-level are summarized in table 2.4.

Table 2.3. All frequent patterns by item

Items	Conditional FP-tree	Frequent itemsets
p	{<*c*:3>}|*p*	{<*p*:3>,<*cp*:3>}
m	{<*f*:3>,<*c*:3>,<*a*:3>}|*m*	{<*m*:3>, <*am*:3>, <*cm*:3>, <*fm*:3>, <*cam*:3>, <*fam*:3>, <*fcam*:3>,<*fcm*:3>}.
b	{}	{}
a	{<*f*:3>,<*c*:3>}|*a*	{<*a*:3>, <*fa*:3>, <*ca*:3>, <*fca*:3>}
c	{<*f*:3>}|*c*	{<*c*:4>, <*fc*:3>}
f	{}	{<*f*:4>}

Table 2.4. All frequent itemsets by length

k-frequent itemset	Frequent itemset
1	<*a*>, <*c*>, <*f*>, <*m*>, <*p*>
2	<*ac*>, <*af*>,<*am*>, <*cf*>, <*cm*>, <*cp*>, <*fm*>
3	<*acf*>, <*acm*>, <*amf*>,<*cfm*>
4	<*acfm*>

CHAPTER 3

AN EXTENDABLE ITEMSET APPROACH TO FREQUENT ITEMSETS MINING

This chapter is organized as follows. Section 3.1 presented how to design and construct an inverted index structure. Section 3.2 presented how to mine frequent itemset using IIS-Mine algorithm. In section 3.3, a theorem and proof are given to demonstrate that the proposed IIS-Mine algorithm can mine frequent itemsets completely and correctly. Last, time and space complexity are shown in section 3.4.

3.1 Inverted Index Structure: Design and Construction

In this section, a data structure that contains transaction data, called an Inverted Index Structure (IIS) is presented. The IIS is a structure that holds a relationship between items and the transactions included within. The IIS is constructed from one scan of the DB. This original IIS can support every *minsup* threshold; therefore, it does not need to rescan the DB when the *minsup* threshold is changed. According to the definitions in the previous section, we present a new definition with an example and an algorithm to demonstrate how to construct this IIS.

Definition 1 (Inverted Index Structure). Let DB be a transaction database and I be a non-empty finite set of all items in each transaction in the DB, where each transaction is a set of items in I associated with an identifier, and let S be a set of all non-empty subsets of DB. An IIS is the function $f : I \rightarrow S$ defined by $f(a) = S_1$ if T contains a for each $T \in S_1$. This function can be identified as a table consisting of the attributes of the items in I and the corresponding transactions in the DB. That is, each row in the IIS contains an item in I as well as the transactions in the DB that contain that item. The set of transactions are written in the order of their ascending identification numbers.

With the above definition, the IIS represents the relationship between each item in I and its corresponding transactions; therefore, the IIS can apply to all minimum support thresholds, and a rescan of the database is not required. We demonstrate the steps to construct the IIS through the following example.

Example 3.1. We use the example of a DB in Table 2.1. The DB is scanned once to create the IIS. The scan of the first transaction is T_1, which consists of the items *f, a, c, d, g, i, m,* and *p*. The transaction T_1 will be inserted for each corresponding item sorted in ascending order (*a, c, d, f, g, i, m, p*). T_1 will be the first transaction inserted in the transactions of item *a*. The second examined item is *c*, so we insert T_1 at item *c*. Next, we examine item *d*; we then subsequently insert T_1 at item *d*. The remaining items (*f, g, i, m,* and *p*) in T_1 can be inserted the same way. The remaining transactions (T_2, T_3, T_4, and T_5) in the DB are performed in a similar manner.

Algorithm 3.1 shows how to construct the IIS. Figure 3.1 shows all of the items of the IIS after scanning the DB once, and the bold items are all the frequent items that have a support greater than or equal to the *minsup*, which is assumed to be 3.

Item	Transaction			
a	T_1	T_2	T_5	
b	T_2	T_3	T_4	
c	T_1	T_2	T_4	T_5
d	T_1			
e	T_5			
f	T_1	T_2	T_3	T_5
g	T_1			
h	T_3			
i	T_1			
j	T_3			
k	T_4			
l	T_2	T_5		
m	T_1	T_2	T_5	
n	T_5			
o	T_2	T_3		
p	T_1	T_4	T_5	
s	T_4			

Figure 3.1. An example of the IIS

Algorithm 3.1 (Inverted-Index-Structure Construction)

Input: DB.

Output: IIS.

Method: The IIS is constructed as follows.

```
1 Begin
2   Create header that contains all items.
3   For each transaction T in DB do  // scanning DB once
4     Sort items in T                // ascending order
5     Create transaction to each corresponding item
6   End //For
7 End //Begin
```

3.2 IIS-Mine Algorithm

In this section, a new algorithm called IIS-Mine is presented. This algorithm uses a new tree structure, called the IIS_{item}Tree, to mine frequent itemsets. The main features of this algorithm are as follows: (1) every frequent itemset is found without generating candidate itemsets; (2) the algorithm reduces the recursion of mining steps using the property of extendable-itemset; and (3) the algorithm supports the mining of frequent itemsets with any value of the *minsup* threshold without needing to rescan the database. From the above features, we can quickly find the frequent itemsets and completely and correctly obtain them. We now introduce the terminologies of the IIS_{item}Tree, its construct, the theorem, the examples and the algorithms to describe how to mine frequent itemsets.

Definition 2 (Itemset-tree Structure). An *itemset-tree structure* is a tree structure constructed from the IIS. It is a finite set of one or more nodes with the following structure:

(i) It consists of the root, which contains an item, a set of item subtrees as the children of the root, and a set of header tables.

(ii) Each node in this tree comprises five fields: *item-name*, which registers which item this node represents; *support,* which registers the number of transactions represented by the portion of the path reaching this node; *same-item,* which represents a pointer that points to the node in the itemset-tree structure that carries the same item-name; *parent*, which represents a

pointer that points to the previous node in the same path; and *child*, which represents a pointer that points to the child node.

(iii) Each member of the *header table* consists of two fields, *item-name* and *head of node link*, where *head of node link* represents a pointer that points to the first node in the itemset-tree structure carrying the *item-name*.

Definition 3 (IIS_{item}Tree). Let x_0 be an arbitrary frequent item in a given transaction database and the IIS be the inverted index structure of the transaction database. A tree T constructed from the IIS is called an *inverted index structure-*x_0 tree, denoted by IIS_{x_0}*tree*, if it satisfies the following:

(i) Each node of T is of the form $(A:s)$, where A is a frequent itemset and s is its support. If $(A:s)$ is a node of T and $A=\{a\}$, where a is a frequent item, then $(A:s)$ is simply written by $(a:s)$.

(ii) Let $(x_0:s_0)$ be its root, where s_0 is the support of x_0.

(iii) Let $x_0, x_1, ..., x_k$ be frequent items in the IIS, and $s_i = supp\{x_0, x_1, ..., x_i\}$ for all $i = 0,1,...,k$. In this case, $P=((x_0:s_0),(x_1:s_1),...,(x_k:s_k))$ is a path from the root $(x_0:s_0)$ to a leaf $(x_k:s_k)$ of the tree T if and only if $s_0 \geq s_1 ... \geq s_k > 0$, $x_0 <_l x_1 <_l ... <_l x_k$, where $<_l$ is the lexicographic order; if a is a frequent item in the IIS and if $(a:s)$ is a node of T such that $x_k < a$ or $x_i <_l a <_l x_{i+1}$ for all $i = 0,1,...,k$, then $supp\{x_0, x_1, ..., x_i, a\} = 0$ and $(a:s)$ is not a node of P.

The *header table* of IIS_{x_0}*tree* is the set of all frequent items a of a node $(a:s)$ of this tree.

Based on the above definition, we have the IIS_{item}Tree construction algorithm, shown in Algorithm 3.2. It is evident that if *minsup*>0 is a minimum support threshold, then every frequent itemset can be derived from an IIS_{item}Tree.

Algorithm 3.2 (IIS_{item} Tree Construction)

Input: IIS.

Output: IIS_{item} Tree.

Method: An IIS_{item} Tree is constructed as follows.

1Begin

2 Create *header table*

3 Read frequent item x in IIS

4 Create root R and initial $supp(R)$ to 1

5 Link R to *header table*

6 For each transaction T of root R where $T = T_1$ to T_n do

7 While next frequent item≠(last frequent item)+1 do

8 Read next frequent item (N) that has same T with R

9 Call InsertTree (N,R)

10 End//While

11 End//For

12End //Begin

Procedure InsertTree (N,R)

1Begin

2 If $IIS_R Tree$ has a node C such that *C.item-name* = *N.item- name* then

3 Increment *supp*(N) by 1

4 Else

5 Create new node N and initial *supp*(N) to 1

6 Link N to N's *parent*

7 Link N to N's *header table*

8 Link N to *same-item*

9 End //If

10End //Begin

If no confusion arises, then $\{x_1, x_2, ..., x_n\}$ and $(\{x_1, x_2, ..., x_n\} : s)$ are replaced by $x_1 x_2 ... x_n$, and $< x_1 x_2 ... x_n : s >$, respectively, where $x_1, x_2, ..., x_n$ are items.

Example 3.2. In this example, we describe the steps to construct all of the IIS_{item}Trees except the last frequent item p using the IIS in Figure 3.1 and *minsup* = 3. The first frequent item in the IIS is a; therefore, we first construct the $IIS_a Tree$, and item a is a root. We obtain two paths: (<a:2>, <c:2>, <f:2>, <m:2>, <p:2>) and (<a:1>, <b:1>, <c:1>, <f:1>, <m:1>). The first path consists of frequent items (a,c,f,m,p) that appear twice in the DB. Similarly, the second path indicates the frequent items (a,b,c,f,m) that are contained in only one transaction in the DB. These two paths share the frequent item a; thus, a appears three times in the DB. Similarly, other IIS_{item}Trees, except the $IIS_p Tree$, can be constructed in the same way. In Figure 3.2, all of the IIS_{item}Trees, except the last $IIS_p Tree$, are illustrated.

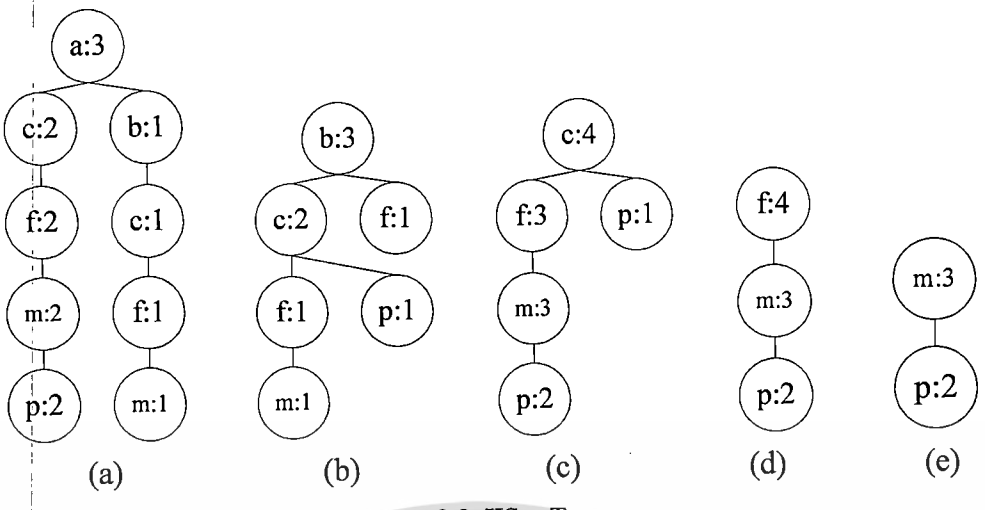

Figure 3.2. IIS_{item}Tree

Definition 4 (A_1-Tree). Let DB be a transaction database; let T be a tree such that each node of the tree is of the form $(A:s)$, where A is a frequent itemset in DB and s is the support of A; and let A_1 be a frequent itemset in the DB. T is called an *A_1-tree* if, for any path P of T, P is of the form $P=((A_1:s_1),(A_2:s_2),...,(A_k:s_k))$, where k is a positive integer; A_i and A_j are pairwise disjoint frequent itemsets for all $i,j=1,2,\ldots,k$ with $i \neq j$; s_i is the support of $\bigcup_{m=1}^{i} A_m$ for $i=1,2,\ldots,k$; $(A_1:s_1)$ is the root of T; and $(A_k:s_k)$ is a leaf of T.

Example 3.3. According to Figure 3.2 (a), the *ac*-tree is shown in Figure 3.3.

Definition 5 (Prefix subpath). Let T be a tree, a_1 be the root of T, and $P=(a_1,...,a_m)$ be a path of T, where m is a positive integer. Every path $Q=(a_1,...,a_k)$ of T is then called a *prefix subpath* of P, where $1 \leq k \leq m$.

Example 3.4. According to Figure 3.2 (a), the paths ((*a*:3),(*c*:2)) and ((*a*:3),(*c*:2),(*f*:2)) are prefix subpaths of ((*a*:3),(*c*:2),(*f*:2),(*m*:2)).

Definition 6 (Subheader). Let T be an *A-tree* and $(x{:}s(x))$ be a node of T, where x is a frequent item in a given transaction database and $s(x)$ is the support of x. Suppose that all of the nodes (of T) containing x are only in the paths $P_1,\ldots,P_k$ of T from the root $(A{:}s)$ to some leafs of T, and suppose that Q_i is a prefix subpath (of P_i) from the root $(A:s)$ to $(x:s(Q_i))$ for all $i=1,...,k$. The *subheader* of T, denoted by $SH(A)$, is defined as the order set $SH(A) = \{x|x$ is a frequent item not contained in A, $(x:s(Q_i))$ is a node in Q_i for $i=1,\ldots,k$ and $\sum_{i=1}^{k} s(Q_i) \geq minsup\}$.

Example 3.5. According to Figure 3.2 (a), SH(a) = {*c*,*f*,*m*}, where SH(a) is the Subheader of the tree in Figure 3.2 (a), $supp(c)=3$, $supp(f)=3$, and $supp(m)=3$.

Definition 7 (Conditional itemset-tree). Let A_1 be a frequent itemset, T be an A_1-*tree*, x_2 be a frequent item with $x_2 \in SH(A_1) - A_1$, and $A_1x_2 := A_1 \cup \{x_2\}$ be a frequent itemset. A *conditional* A_1x_2-*tree*, denoted by $T(A_1x_2)$, is a tree that satisfies the following:

(i) $(A_1x_2 : s_2)$ is the root of $T(A_1x_2)$, where s_2 is the support of A_1x_2 and where $s_2 \geq$ *minsup*.

(ii) The number of items in $SH(A_1) > 1$.

(iii) All of the paths are derived from T in the following way: $Q = ((A_1x_2 : s_2),(x_3 : s_3),...,(x_k : s_k))$ is a path of $T(A_1x_2)$ if and only if a path $P = ((A_1 : s_1),(x_2 : s_2),...,(x_l : s_l))$ exists from the root $(A_1 : s_1)$ to the leaf $(x_l : s_l)$ of T, where $l \geq k$, x_k is in $SH(A_1)$; and if there is a node $(x_r : s_r)$ of P such that $r > k$ and $((A_1 : s_1),(x_2 : s_2),...,(x_r : s_r))$ is a prefix subpath of P, then x_r must not be in $SH(A_1)$.

Notably, every nonempty subset of a frequent itemset is also frequent. This fact leads to the following definition.

Example 3.6. According to Figure 3.2 (a), the conditional itemset-tree, or conditional *ac*-tree, is shown in Figure 3.4.

Definition 8 (Frequent itemset* of length m derived from tree). Let A be a frequent itemset, x be a frequent item, and $x \notin A$. Suppose that T is a conditional Ax-tree, m is a positive integer greater than 1, and $Ax = A \cup \{x\}$ has m elements. Ax is called a *frequent itemset* of length m derived from* T, denoted by $FS^*_m(Ax)$, if T contains precisely two nodes and Ax is a frequent itemset or if $SH(A) = \{x\}$.

Example 3.7. According to Figure 3.4, the frequent itemset* of length 4 derived from the conditional *acf*-tree is $FS^*_4(acfm) = acfm$.

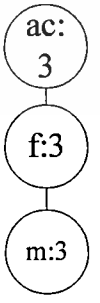

Figure 3.3. *ac*-tree

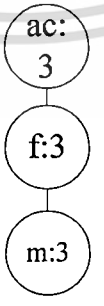

Figure 3.4. Conditional *ac*-tree

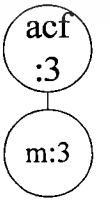

Figure 3.5. Conditional *acf*-tree

Definition 9 (Extendable frequent itemset*). Let m be a positive integer greater than 1, T be a $T(Ax)$ defined as in definition 7 with $A_1 = A$ and $x_1 = x$, and Ax be a frequent itemset* of

length m derived from T. Each itemset in $FS^*_m(Ax)$ is said to be *extendable* if $m \geq 3$. For every k = 2, 3,..., m, we let $Ext^*_k(Ax)$ denote the set of all itemsets containing exactly k items of $FS^*_m(Ax)$, where $m \geq 3$. Each element of $Ext^*_k(Ax)$ is called an *extendable frequent itemset** *(derived from T) of length k* for all $k \geq 2$. The set of all extendable frequent itemsets* of length k that is denoted by $Ext^*_k = \cup\{Ext^*_k(Ay) \mid Ay$ is a frequent itemset* of length at least k derived from a conditional *Ay-tree*$\}$ for all $k \geq 2$.

Example 3.8. According to previous example, the length of $FS^*_4(acfm)$ is 4; we then find that $Ext^*_2(acfm)$ = {*ac*, *af*, *am*, *cf*, *cm*, *fm*} and $Ext^*_3(acfm)$ = {*acf*, *acm*, *afm*, *cfm*}. Therefore, Ext^*_2 = {*ac*, *af*, *am*, *cf*, *cm*, *fm*, and all members of other $Ext^*_2(Ay)$ } and Ext^*_3 = {*acf*, *acm*, *afm*, *cfm*, and all members of other $Ext^*_3(Ay)$ }.

Definition 10 (Frequent itemset* of length *m*). Let k be the maximum length of all frequent itemsets in a transaction database, $FS^*_m(Ax)$ and Ext^*_k be given as in definitions 8 and 9, respectively; let $A_m = \{FS^*_m(Ax) \mid Ax$ is a frequent itemset* of length m derived from $T\}$ for m = 2, 3,..., k; define $FS^*_2 = A_2 \cup \{Ax \mid Ax$ is a frequent itemset of length 2$\}$; and define $FS^*_m = A_m$ for m = 2, 3,..., k}. Let FI^*_m be defined by $FI^*_1 = \{\{x\} \mid x$ is a frequent item} and $FI^*_m = FS^*_m \cup Ext^*_m$ for m = 2, 3,..., k}. Any element of FI^*_m is called the *frequent itemset* of length m*.

Example 3.9. According to the previous example, we find that FI^*_2 = {*ac*, *af*, *am*, *cf*, *cm*, *fm*}, FI^*_3 = {*acf*, *acm*, *afm*, *cfm*}, and FI^*_4 = {*acfm*}.

Definition 11 (Frequent itemset*). Let FI^*_m be given as in definition 10, and let FI^* denote $\cup^k_{m=1} FI^*_m$, where k is the maximum length of all frequent itemsets in a transaction database. Any element of FI^* is called a *frequent itemset**.

Example 3.10. According to the previous example, FI^* is {*ac*, *af*, *am*, *cf*, *cm*, *fm*, *acf*, *acm*, *afm*, *cfm*, *acfm*}.

On the basis of the above definitions and examples, Algorithm 3.3 presents the IIS-Mine algorithm to show how it can be used to mine all frequent itemsets.

Algorithm 3.3: (IIS-Mine: Mining frequent itemsets using IIS_{item}Tree)

Input: IIS, IIS_{item} Trees constructed according to Algorithm 3.2, and *minsup*

Output: *FI**

Procedure AllFreqItemset (IIS, *FI**)

1 Begin

2 For each frequent item x in the IIS do

3 $FI_1^* = \{\{x \mid x \in I, supp(x) \geq minsup\}$

4 Call $IIS_x Tree$, which is constructed from Algorithm 3.2

5 If Tree ≠{}

6 Call IIS-Mine (Tree, x)

7 End //If

8 End //For

9 Find $FI^* = \cup_{m=1}^{k} FI_m^*$ // FI^* is given in definition 11

10 End //Begin

Procedure IIS-Mine(Tree, x)

1 Begin

2 Call SubHeader (A-tree, subheader$A$)

3 Generate all Ax with its support

//All Ax are the frequent itemsets where A is the root of A-*tree* and $x \in$subheader$A$

4 For each $|\lambda| > 1$ do // λ is Ax

5 Flag=1

6 While Flag = 1 do

7 Call SkipFreqItemset (subheader$A$, λ, δ ,FlagRepeat, FlagTree)

8 If FlagRepeat=0 and FlagTree=0 then // ($\delta \notin FI_m^*$)

9 Call CondItemsetTree (A-tree, δ , conditional δ -*tree*)

10 Else

11 Flag=0

12 End // If

13 If conditional Ax-tree contains greater than two nodes

14 Call IIS-Mine(conditional δ -*tree*, x)

15 Else Flag=0

16 End //If

17 End //While

18 If $|FS_m^*(\delta)| \geq 3$ and $FS_m^*(\delta) \notin FI_m^*$ then

19 Call ExtFreqItemset($FS_m^*(\delta)$, Ext_k^*) // FS_m^* is given in definition 8

20 Save $FS_m^*(\delta)$ and all Ext_k^* to FI_m^* // FI_m^* is given in definition 10

21 Else

22 If $FS_m^*(\delta) \notin FI_m^*$ then

23 Save $FS_m^*(\delta)$ to FI_m^*

24 End //If

25 End //If

26 End //For

27 End //Begin

Procedure SubHeader (A-tree, subheaderA)

1 Begin

2 Each frequent item x of A-tree // SH(A) is given in definition 6

3 Find $\{(x{:}s(x))|x \in SH(A), s(x)$ is the support of x in A-tree$\}$

4 End // Begin

Procedure CondItemsetTree(Tree, δ, conditional δ -tree)

1 Begin

2 Scan tree once to collect the paths that have an association with root δ

3 For all paths are derived from tree do

4 Connect all paths to δ

5 End //For

6 End //Begin

Procedure SkipFreqItemset (SubheaderA, λ, δ, FlagRepeat, FlagTree)

1 Begin // To skip the construction of conditional item tree

2 // x is the frequent item in subheaderA // λ is the root of tree; or a frequent itemset

3 // FlagRepeat=0 means $\delta \notin FI_m^*$ // FlagTree=0 means the conditional item-tree is

4 // constructed , // n is the maximum number of elements of itemsets in subheaderA

5 FlagRepeat =1, FlagTree=1

6 $\beta = \lambda, \alpha = \beta$

7 While(FlagRepeat=1 and (order of $x \le n$)) do

8 If $\beta \notin FI_m^*$ then

9 FlagRepeat=0, FlagTree=0

10 If x is the last item

11 If $|\delta| = 2$ then FlagTree=1

12 End //If

13 $\delta = \alpha$

14 End //If
15 Else
16 If x is the last item then
17 Increment order of item x
18 Else
19 Increment order of item x
20 $\beta = \delta \cup x$
21 $\alpha = \delta$
22 End// If
23 $\delta = \beta$
24 End //If
25 End //while
26 End //Begin

Procedure ExtFreqItemset($FS^*_m(\delta)$, Ext^*_k)
1 Begin // Ext^*_k is given in definition 9
2 Generate all subsets of $FS^*_m(\delta)$ and save to Ext^*_k
3 End // Begin

Example 3.11. This example is given to demonstrate how the proposed IIS-Mine algorithm can be used to mine all frequent itemsets. Assume *minsup* = 3 for all of the definitions above, Example 2.1, Example 3.1, Example 3.2, Figure 3.2, and Algorithm 3.3. For simplicity, this example is divided into five main steps ordered by five frequent items in the IIS; the proposed mining algorithm proceeds as follows.

Let step 1 be the first main step. According to Procedure AllFreqItemset, the frequent item *a* is the first frequent item in the IIS that is mined. The $IIS_a Tree$ is constructed, as illustrated in Figure 3.2 (a). The algorithm calls Procedure IIS-Mine, so Procedure Subheader is called in order. The SH(*a*) is (*c*,*f*,*m*), where *supp*(*c*) = 3, *supp*(*f*) = 3, and *supp*(*m*) = 3. After line 3 in the procedure, IIS-Mine generates all of the frequent itemsets with its support, which are *ac*, *af*, and *am*; all of these frequent itemsets have support equal to three. Let steps 1.1, 1.2 and 1.3 represent each of the frequent itemsets.

The step 1.1, the first frequent itemset is *ac*, which has support equal to three. The algorithm checks |*ac*|>1 and then calls Procedure SkipFreqItem. At this procedure, *ac* is not

in FI_2^*, so the algorithm rolls back to line 9 of Procedure IIS-Mine. The frequent itemset *ac* with its support is defined to be the root; then, the conditional *ac*-tree, which has root "<*ac*:3>", is constructed using the input *IIS$_a$Tree*. The conditional *ac*-tree is illustrated in Figure 3.4. The algorithm is iterated by calling Procedure IIS-Mine again because the conditional *ac*-tree contains more than two nodes; let this call be step 1.1.1.

In step 1.1.1, the Procedure IIS-Mine is called; SH(*ac*) = (*f*,*m*), where *supp*(*f*) and *supp*(*m*) are then equal to 3. At line 3, the algorithm generates frequent itemsets, which are *acf* and *acm*, where *supp*(*acf*) and *supp*(*acm*) are then equal to 3. The first frequent itemset in this step is *acf* and $|\ acf\ | > 1$, so the procedure SkipFreqItem in line 7 is processed, and it finds that *acf* is not in FI_3^*. Next, the algorithm rolls back to line 9 to construct a conditional *acf*-tree that has a conditional *ac*-tree as an input tree, which is illustrated in Figure 3.5. The condition of line 13 is that the conditional *acf*-tree contains only two nodes, so the algorithm obtains $FS_4^*(acfm) = acfm$. At line 18, the size of $FS_4^*(acfm)$ is greater than three and $FS_4^*(acfm) \notin FI_4^*$; thus, at line 19, Procedure ExtFreqItemset is called to find all of the subsets of $FS_4^*(acfm)$, which are $Ext_2^*(acfm)$ and $Ext_3^*(acfm)$. $Ext_2^*(acfm) = \{ac, af, am, cf, cm, fm\}$, and $Ext_3^*(acfm) = \{acf, acm, afm, cfm\}$. Therefore, $FI_2^* = \{ac, af, am, cf, cm, fm\}$, $FI_3^* = \{acf, acm, afm, cfm\}$, and $FI_4^* = \{acfm\}$.

In step 1.1.2, the next frequent itemset generated together with step 1.1.1 is *acm*. At line 7 of Procedure IIS-Mine, the algorithm calls Procedure SkipFreqItem and obtains *acm*, which is already a member of FI_3^*; *m* is the last frequent item in SH(*ac*), so we exit from this step without the construction of a conditional *acm*-tree.

In step 1.2, at line 4 of Procedure IIS-Mine, the next frequent itemset is *af*, and at line 7, Procedure SkipFreqItem is called and obtains *af*, which is contained in FI_2^*. In the loop in line 20 of Procedure SkipFreqItem, after *af* is combined with the next frequent item in SH(*a*), which is *m*, *afm* is obtained, which is contained in FI_3^*, and item *m* is the last item in SH(*a*). The algorithm rolls back to line 8 of Procedure IIS-Mine, so the conditional *af*-tree is not constructed.

In step 1.3, at line 4 of Procedure IIS-Mine, the next frequent itemset is *am*. At line 7, Procedure SkipFreqItem is called and obtains *am*, which is contained in FI_2^*, and there is no frequent item in SH(*a*). Therefore, the conditional *am*-tree is not constructed.

Let step 2 be the second main step. According to line 2 of Procedure AllFreqItemset, *b* is the second frequent item in the IIS that is mined. After line 4, the *IIS$_b$Tree* is constructed, as illustrated in Figure 3.2 (b). The algorithm calls Procedure IIS-Mine at line 6. At line 2 of Procedure IIS-Mine, Procedure Subheader is called and obtains an empty SH(*b*), so the size of the

frequent itemset generated with *b* is one. The processing of this step is terminated, and we return to line 2 of Procedure AllFreqItemset.

Let the third main step be step 3. According to line 2 of Procedure AllFreqItemset, *c* is the third frequent item in IIS that is mined. Line 4 is called to construct the $IIS_c Tree$, as illustrated in Figure 3.2 (c). At line 6, the algorithm calls Procedure IIS-Mine to mine frequent itemsets. At line 2 of Procedure IIS-Mine, Procedure Subheader is called to obtain the SH(*c*) that is (*f*,*m*,*p*). Next, at line 3, the algorithm generates frequent itemsets, which are *cf*, *cm*, and *cp*, where *supp*(*cf*) = 3, *supp*(*cm*), and *supp*(*cp*) = 3. Let steps 3.1, 3.2 and 3.3 represent each of the frequent itemsets.

In step 3.1, at line 4 of Procedure IIS-Mine, the first frequent itemset is *cf*, where $|cf| > 1$; line 7 then calls Procedure SkipFreqItemset. At Procedure SkipFreqItemset, *cf* unions with the next frequent item in SH(*c*), which is *m*, so *cfm* with a support of 3 is obtained after processing lines 19 to 23. Next, line 8 is checked, and as *cfm* is already obtained in FI_3^*, lines 19 to 23 are checked again, and *cfmp* is obtained. The frequent itemset *cfmp* is not a member in FI_4^*, and *p* is the last frequent item in SH(*c*), so *cfm* is set to be a root for a conditional *cfm*-tree. The algorithm goes back to line 8 of Procedure IIS-Mine to construct a conditional *cfm*-tree, where the $IIS_c Tree$ is an input tree, which is illustrated in Figure 3.6. *supp*(*p*) is less than *minsup*, so $FS_3^*(cfm) = cfm$. Procedure ExtFreqItemset in line 18 is not called because $FS_3^*(cfm)$ is contained in FI_3^*. In this step, the algorithm skips the construction of the conditional *cf*-tree.

For step 3.2, according to line 4 of the Procedure IIS-Mine, the next frequent itemset is *cm*, and the Procedure SkipFreqItemset in line 7 is called. Line 8 of Procedure SkipFreqItemset checks that *cm* is a member in FI_2^*. Next, lines 19 to 23 are checked, so *cm* unions with the next item in SH(*c*), which is *p*, and we obtain *cmp* with a support of 3. The frequent itemset *cmp* is not in FI_3^*, and *p* is the last frequent item in SH(*c*), so *cm* is set to be a root for a conditional *cm*-tree. The algorithm goes backs to line 8 of the Procedure IIS-Mine to construct a conditional *cm*-tree, where the $IIS_c Tree$ is an input tree, which is illustrated in Figure 3.7. *supp*(*p*) is less than *minsup*, so $FS_2^*(cm) = cm$ and $FS_2^*(cm)$ are already members in FI_2^*.

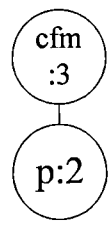

Figure 3.6. Conditional *cfm*-tree

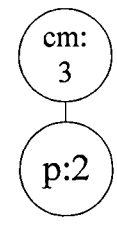

Figure 3.7. Conditional *cm*-tree

In step 3.3, according to line 4 of the Procedure IIS-Mine, the next frequent itemset is *cp*. Next, at line 5, Procedure SkipFreqItemset is called, and *cp* is not a member in FI_2^*. There is no frequent item in SH(*c*), so this procedure is terminated, and we return to line 8 of the Procedure IIS-Mine. After lines 22 to 23 are checked, the new answer is contained in FI_2^*, which is *cp*. This step skips the construction of a conditional *cp*-tree.

The remaining steps, such as the fourth main step, which constructs the IIS_fTree and is illustrated in Figure 3.2 (d), and the fifth main step, which constructs the IIS_mTree and is illustrated in Figure 3.2 (e), are performed in the same way in sequence.

We show the complete frequent itemsets by item in Table 3.1 and by length in Table 3.2.

Table 3.1. Complete frequent itemsets by item

Items	Frequent Itemsets
a	*a, acfm, ac, af, am, cf, cm, fm, acf, acm, amf, cfm*
b	*b*
c	*c, cp*
f	*f*
m	*m*
p	*P*

Table 3.2. Complete frequent itemsets by length

m-length	Frequent Itemsets
1	*a, b, c, f, m, p*
2	*ac, af, am, cf, cm, cp, fm*
3	*acf, acm, afm, cfm*
4	*acfm*

We show the advantages of our algorithm. First, according to step 1.1.1 of Example 3.11, we obtained frequent itemsets *acfm,* which are derived from a conditional *acf*-tree. This step shows the properties of an extendable frequent itemset*, which are given in Definition 9 and Procedure ExtFreqItemset in Algorithm 3.3. This step then finds all of the subsets of *acfm*, so we derive ten frequent itemsets, which are *ac*, *af*, *am*, *cf*, *cm*, *fm*, *acf*, *acm*, *amf*, and *cfm*, without

contributing more trees or using recursion to mine. Therefore, our algorithm can reduce many subsequent steps in mining frequent itemsets. Notice that if $|FS^*_m(Ax)|$ is large, then the property of an extendable frequent itemset* is frequently used. Second, our method is also good for reducing runtime and space consumption, and its performance will be shown in the experiment section. Last, according to step 3.1 of Example 3.11, we obtain the conditional *cfm*-tree, which reduces one node in the tree because of the step that sets frequent itemsets to be the root node in the Procedure SkipFreqItemset in Algorithm 3.3. The algorithm reduces the number of nodes, levels and size of the tree, thus reducing space consumption. In general, users change many minimum support thresholds to make their decision. Our method supports the ability to make these changes without rescanning the database using an IIS and the restart algorithm shown in Algorithm 3.3.

3.3 Correctness

The following theorem and proof are given to demonstrate that the proposed IIS-Mine algorithm can mine frequent itemsets completely and correctly.

Theorem. The set of all frequent itemsets* derived from IIS-Mine is the complete set of all of the frequent itemsets.

Proof. Let I be the nonempty finite set of all items in a given transaction database, FI denote the set of all frequent itemsets in a transaction database, and $\alpha = minsup > 0$. We must prove that $FI = FI^*$.

First, we prove that $FI \subseteq FI^*$. We let $F = \{a_1,...,a_k\} \in FI$ with $a_1 <_l ... <_l a_k$, and $s_i = supp\{a_1,...,a_i\}$ for all $i = 1,...,k$. Then, $s_1 \geq ... \geq s_k \geq \alpha$, and an item b exists such that $b \geq a_1$, and $IIS_b tree$ contains $a_1,...,a_k$. We can assume that b is the item in I having these properties because I is the nonempty finite set of all items. Because $s_i \geq ... \geq s_k \geq \alpha$ for all $i = 1,...,k$, there exists a path $((b_1 : s_1'),...(b_l : s_l'))$ of $IIS_b tree$ containing $(a_1 : s_1),...,(a_k : s_k)$; that is, for all $i = 1,...,k$, there exists $j = 1,...,l$ such that $(a_i : s_i) = (b_j : s_j)$. It is obvious from the IIS-Mine algorithm that $FI \subseteq FI^*$.

We also show that $FI^* \subseteq FI$. We let $F = \{a_1,...,a_k\} \in FI^*$ with $a_1 <_l ... <_l a_k$. Then, for some positive integer m, $F \in FI^*_m$. It is evident from definition 10 that if $m = 1$, $F \in FI^*_m$ implies $F \in FI$. Now, suppose $m \geq 2$; then, $F \in FI^*_m$ or $F \in Ext^*_m$. In the first case, $F \in FS^*_m$, and from Definition 8, $SH\{a_1,...,a_{k-1}\}=(a_k)$ or the conditional $\{a_1,...,a_{k-1}\}a_k$ tree contains exactly

two nodes, $(\{a_1,...,a_{k-1}\}:s_{k-1})$ and $(a_k:s_k)$, where $s_i = supp\{a_1,...,a_i\}$ for $i = k-1, k$. Then, from Definition 7 and 8, we obtain $\{a_1,...,a_{k-1}\}a_k = \{a_1,...,a_k\} = F \in FI$. In the other case, suppose that $F \in Ext^*_m$ for $m \geq 2$. Then, from Definition 9, an Ax exists such that $F \in Ext^*_m(Ax)$, and Ax is a frequent itemset* of length at least m derived from the conditional Ax-tree. Again, from Definition 9, a positive integer k greater than 2 exists such that F has itemsets containing precisely m items of $FS^*_k(Ax)$, and from Definition 6 and 8, $FS^*_k(Ax)$ is a frequent itemset; therefore, $F \in FI$. The proof is complete.

3.4 Time and Space Complexity

Let f be 1-frequent itemsets which specific a *minsup* and m be 1-frequent itemsets with maximum length of transactions. In the worst case of time and space complexity, we found that both algorithm, FP-growth and IIS-Mine, are equal. Two algorithms mine all frequent itemsets by scanning tree, so all f in tree are scanned. Therefore, the time complexity is $O(f \times \log f)$. The space complexity is $O(f \times m)$ because all f and m are scanned. However, IIS-Mine performs to mine all frequent itemsets with extendable itemset property. Theoretically, both algorithms will generate equal Big O, however, when dealing with dense datasets, IIS-Mine algorithm can use the property of extendable itemset more frequently which will reduce run time and space consumption. This will result in better performance of IIS-Mine as compare with FP-growth.

CHAPTER 4

RESULTS AND DISCUSSIONS

This chapter presents the experiments in which the run time, memory consumption and scalability are tested for the IIS-Mine algorithm and FP-growth algorithm with different datasets and varying minimum support thresholds. The experiments were performed on a Microsoft Windows XP Professional Version 2002 Service Pack 3 operating system on a personal computer with 1 GB of main memory and Pentium (R) CPU 3.00 GHz. All algorithms were coded using C language. The two groups of benchmark datasets are used: two synthetic datasets and two real datasets.

4.1 Datasets

For the first group of datasets, we also present the experimental results for two synthetic datasets generated by the IBM Almaden Quest research group [24-25]. The datasets serve as the FIMI repository, which is the result of the workshops on Frequent Itemset Mining Implementations [26-27]. The two original databases of synthetic datasets are T10I4D100K and T40I10D100K, which are sparse datasets. The notation *TxIyDzK* denotes a dataset where *K* is 1,000 transactions. Table 4.1 lists the parameters of the synthetic datasets. The synthetic datasets varied in the number of transactions, i.e., 20%, 40%, 60%, and 80% of the original database.

Table 4.1 Parameters of the synthetic datasets

$\|T\|$	The average number of items per transaction
$\|I\|$	The average length of a frequent itemset
$\|D\|$	The number of transactions

For the second group of datasets, the real datasets from the UCI Machine Learning Repository [28] has been used to test the proposed method. The real datasets used in the experiment are Chess [29] and Mushroom [30], which are dense datasets with a great number of long frequent itemsets. The characteristic of real datasets are shown in Table 4.2.

Table 4.2 Characteristics of real datasets

Real datasets	Description
Chess	The average number of items per transaction is 37, the number of transactions is 3,196, and the number of items is 75.
Mushroom	The average number of items per transaction is 23, the number of transactions is 8,124, and the number of items is 119.

4.2 Experimental Results

4.2.1 Run time

Table 4.3 presents all experimental results of the run time by using the synthetic datasets which compare the run time of two algorithms. The experiment results are separated by testing varying minimum support thresholds. Each of minimum support presents the total number of frequent itemsets and run time of two algorithms.

Table 4.3 The experimental results of the run time by using the synthetic datasets

Datasets	Minimum support (%)	Total number of frequent itemsets	Run time (s)	
			FP-growth	IIS-Mine
T10I4D100K	5	10	1.53	0.05
	3	60	2.94	1.53
	2	155	25.73	12.55
	1	385	61.12	29.90
T40I10D100K	20	5	10.17	0.09
	17.5	9	10.97	0.14
	15	19	11.75	1.20
	12.5	40	22.70	15.08
	10	82	207.23	61.11
	7.5	157	824.88	305.13
	5	316	3,651.14	2,704.15

Table 4.4 presents all experimental results of the run time by using the real datasets which compare the run time of two algorithms. The experiment results are separated by testing varying minimum support thresholds. Each of minimum support presents the total number of frequent itemsets and run time of two algorithms.

Table 4.4 The summary of the run time

Datasets	Minimum support (%)	Total number of Frequent Itemsets	Run time (s)	
			FP-growth	IIS-Mine
Chess	90	622	0.22	0.003
	80	8,227	0.92	0.07
	70	48,731	2.96	0.58
	60	254,944	9.31	5.11
	50	1,272,932	44.52	31.53
Mushroom	40	565	0.30	0.006
	30	2,735	0.39	0.11
	20	53,583	2.97	1.96
	10	574,431	6.02	4.22

Figure 4.1 (a) and Figure 4.1 (b) show the performance of the algorithms on two synthetic datasets, T10I4D100K and T40I10D100K, respectively. In Figure 4.1 (a), IIS-Mine performs better than FP-growth in every support threshold. The gap in the graph becomes larger as the support threshold decreases. In Figure 4.1 (b), when the minimum support is set at 20%, 17.5%, 15%, or 12.5%, the run time between the two algorithms is not very different. However, when the minimum support is set at 10%, 7.5%, or 5%, the run time of FP-growth increases significantly when compared to that of IIS-Mine, which confirms that IIS-Mine performs better than FP-growth. The results shown in Figures 4.1 (a) and 4.1 (b) can be explained as follows. With sparse datasets, when the minimum support is high, the number of frequent itemsets is low. However, when the minimum support is low, many frequent itemsets are obtained. IIS-Mine was always faster than the FP-growth method, especially when the minimum support was low, because FP-growth constructs bushy and wide FP-trees when the minimum support is low, so FP-growth is computationally expensive for tree traversing the FP-trees. IIS-Mine has the step of finding the root node from previous frequent itemsets, which can reduce the number of nodes and

levels of the conditional itemset-tree. Therefore, traversing in the conditional itemset-tree is on a reduced tree, which can result in a low run time consumption. However, the run times of both algorithms rely on the length of the transaction (as observed in a comparison of the graphs in Figures 4.1 (a) and 4.1 (b) with a minimum support of 5); when the transaction is long, the run time of both algorithms is also long.

Figures 4.1 (c) and 4.1 (d) show the performance of algorithms on two dense datasets, chess and mushroom. Figure 4.1 (c) shows that the run time of IIS-Mine is better than that of FP-growth in every support threshold. The run time of both algorithms increases when the minimum support threshold is reduced to 50. In Figure 4.1 (d), IIS-Mine again performs better than FP-growth in every minimum support. The run time of FP-growth increases significantly compared with IIS-Mine when the minimum support is less than 30%. The results shown in Figures 4.1 (c) and 4.1 (d) can be explained as follows. In the two figures, IIS-Mine is faster than FP-growth for dense datasets. The main work in FP-growth is traversing FP-trees and constructing new conditional FP-trees after the first FP-tree is constructed from the original database. For dense datasets, we found from numerous experiments that the time spent on traversing FP-trees was very long. IIS-Mine improved this problem using the property of extendable itemsets to reduce the number of recursive mining steps so that the size and number of constructing and traversing the trees are reduced. The run time of IIS-Mine is then less than that of FP-growth.

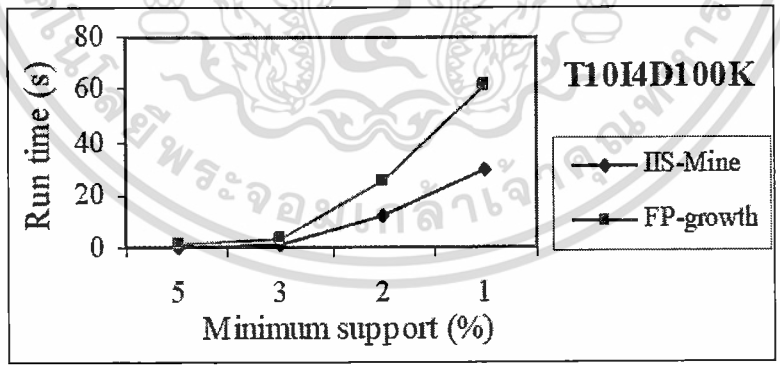

Figure 4.1 (a). Run time of mining on T10I4D100K

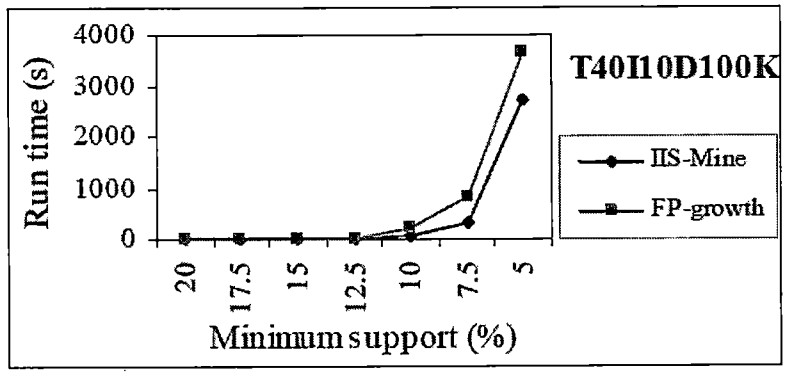

Figure 4.1 (b). Run time of mining on T40I10D100K

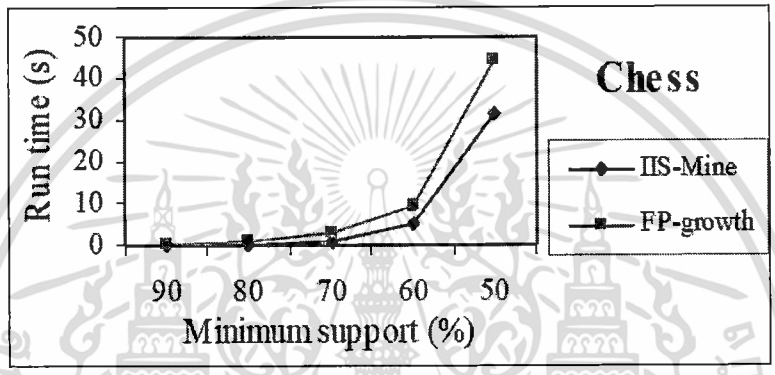

Figure 4.1 (c). Run time of mining on Chess

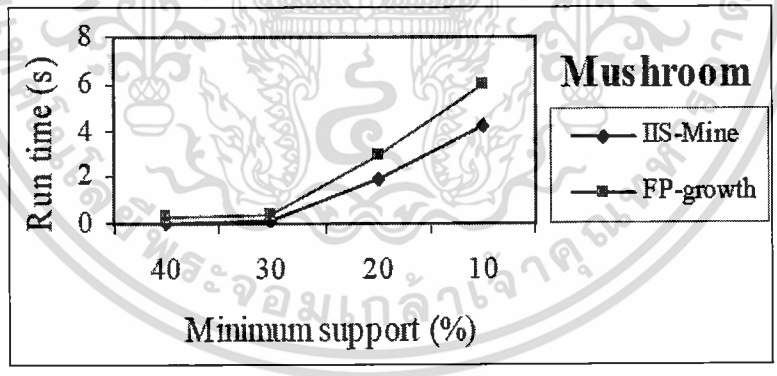

Figure 4.1 (d). Run time of mining on Mushroom

4.2.2 Memory Consumption

Table 4.5 presents the summary result of all experiments which compare memory consumption of two algorithms. The experiment results are separated by testing varying minimum support thresholds. Each of minimum support presents the total number of frequent itemsets and memory consumption of two algorithms.

Table 4.5 The Summary of the Memory Consumption

Datasets	Minimum Support (%)	Total Number of Frequent Itemsets	Main Memory (K)	
			FP-growth	IIS-Mine
T10I4D100K	5	10	1,770	1,140
	3	60	296,760	129,040
	2	155	1,376,406	333,476
	1	385	3,093,066	1,120,392
T40I10D100K	20	5	716	340
	17.5	9	10,210	1,943
	15	19	170,370	76,840
	12.5	40	1,500,306	730,456
	10	82	4,278,810	1,087,556
	7.5	157	8,106,426	2,289,006
	5	316	13,446,408	4,111,319
Chess	90	622	9,096	3,744
	80	8,227	101,640	10,576
	70	48,731	516,036	88,596
	60	254,944	2,301,942	328,088
	50	1,272,932	9,261,852	1,599,868
Mushroom	40	565	6,912	5,480
	30	2,735	25,548	5,876
	20	53,583	95,226	11,220
	10	574,431	384,852	58,408

Figures 4.2 (a) and 4.2 (b) show the memory consumption of the algorithms on the synthetics datasets. In Figure 4.2 (a), FP-growth consumes more memory than IIS-Mine. The graph is clearly separated when the minimum supports are less than 3. In Figure 4.2 (b), in comparing memory consumption when the minimum supports are 20%, 17.5% or 15%, there is no difference until the minimum support is less than 15%. As illustrated in Figures 4.2 (a) and 4.2 (b), we can see that IIS-Mine consumes less memory than FP-growth on synthetic datasets because the memory is mainly used by FP-trees constructed in FP-growth. The large memory

consumption of FP-growth when running on synthetics datasets can be explained by the fairly low minimum support and presence of many single items in the datasets; therefore, FP-growth constructs wide and bushy trees to mine all frequent itemsets. However, IIS-Mine uses the property of extendable itemsets, which reduces the construction of conditional itemset-trees, and uses the step of finding the root node from previous frequent itemsets. Therefore, the node construction and tree sizes are reduced, resulting in a reduction in memory consumption.

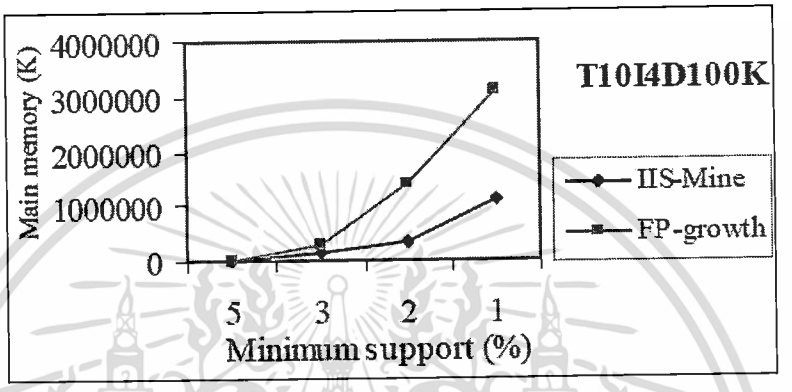

Figure 4.2 (a). Memory consumption of mining on T10I4D100K

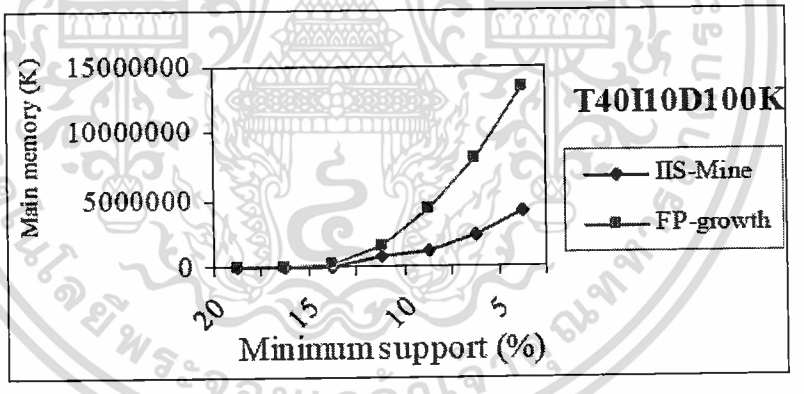

Figure 4.2 (b). Memory consumption of mining on T40I10D100K

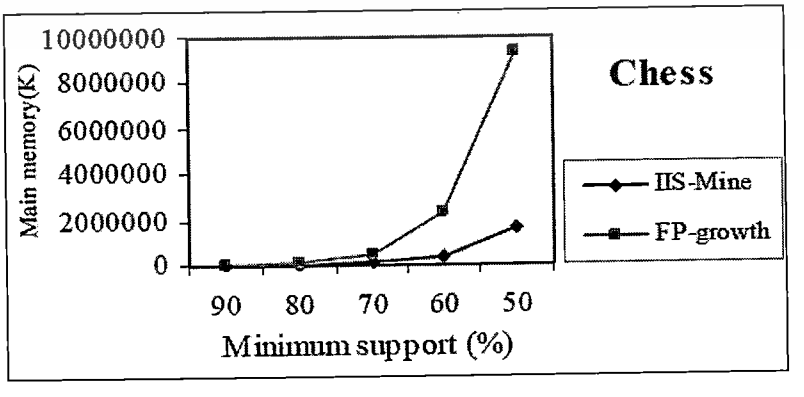

Figure 4.2 (c). Memory consumption of mining on Chess

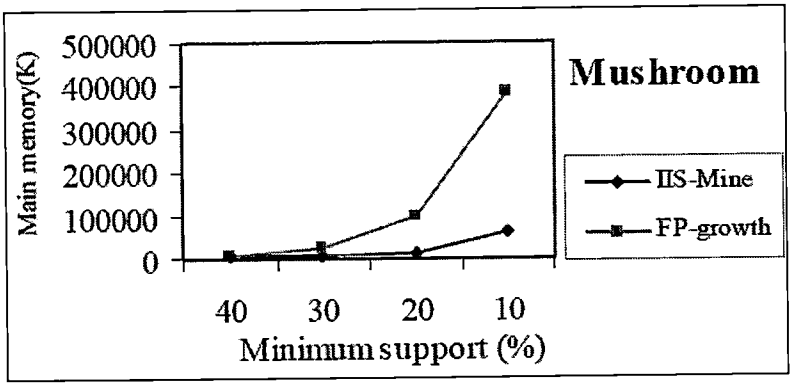

Figure 4.2 (d). Memory consumption of mining on Mushroom

Figures 4.2 (c) and 4.2 (d) show that the memory consumption of IIS-Mine is less than that of FP-growth on dense datasets. Figure 4.2 (c) shows that when the minimum support is less than 80%, the memory consumption of FP-growth increases significantly compared with that of IIS-Mine. Figure 4.2 (d) also shows that when the minimum support is less than 30%, the gap of the graph clearly separates, which confirms that FP-growth consumes more memory than IIS-Mine. In both figures, FP-growth consumes a great deal more memory when the minimum support is low, which is not observed in IIS-Mine because FP-growth has constructed large FP-trees for mining all frequent itemsets, whereas IIS-Mine uses the property of extendable itemsets, which performs better for dense datasets. Consequently, the recursions of mining frequent itemsets in the next loops are reduced. Therefore, the construction of nodes and the sizes of the conditional item-trees are reduced.

4.2.3 Scalability

The scalability of two algorithms is tested by running them on datasets generated from T10I4 and T40I10. The number of transactions in the datasets ranged from 20*K* to 100*K*, where *K* is 1000 transactions. Table 4.5 presents the summary result of all experiments which compare the scalability of run time of two algorithms. The experiment results are separated by testing the number of transactions which specific a minimum support. Each of number of transactions presents the run time of two algorithms.

Table 4.6 The summary of the scalability of run time

Datasets	Minimum support (%)	Number of transactions (K)	Run time (s)	
			FP-growth	IIS-Mine
T10I4	1	20	8.61	2.73
		40	23.09	7.06
		60	45.56	15.51
		80	59.24	23.18
		100	67.12	29.90
T40I10	5	20	47.08	39.21
		40	288.83	172.56
		60	597.94	181.71
		80	2159.52	784.66
		100	3651.14	2704.14

Table 4.7 presents the summary result of all experiments which compare the scalability of run time of two algorithms. The experiment results are separated by testing the number of transactions which specific a minimum support. Each of number of transactions presents the run time of two algorithms.

Table 4.7 The summary of the scalability of memory consumption

Data sets	Minimum support (%)	Number of transactions (K)	Memory consumption (K)	
			FP-growth	IIS-Mine
T10I4	1	20	687,642	72,540
		40	1,238,766	122,092
		60	1,923,270	379,320
		80	2,517,688	614,712
		100	3,093,066	1,120,392
T40I10	5	20	2,750,010	442,356
		40	5,434,884	1,074,690
		60	8,104,572	1,890,441
		80	10,802,250	2,969,361
		100	13,446,408	4,111,319

In Figure 4.3 (a) and Figure 4.4 (a), the algorithms were run on all of the datasets generated from T10I4 for a minimum support of 1. Both run time and memory consumption were recorded. Figure 4.3 (a) shows the speed scalability, which means that the number of transactions increases as the run time increases. Figure 4.4 (a) shows the memory scalability of the algorithms; the curve of FP-growth is over the curve of IIS-Mine, which means that FP-growth consumes more memory than IIS-Mine. Moreover, the figure shows that the memory consumption of the algorithms increases linearly with the size of the datasets.

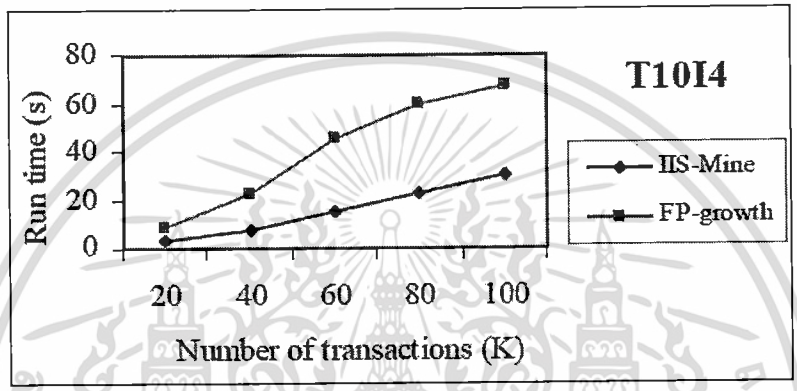

Figure 4.3 (a). Scalability of run time on T10I4

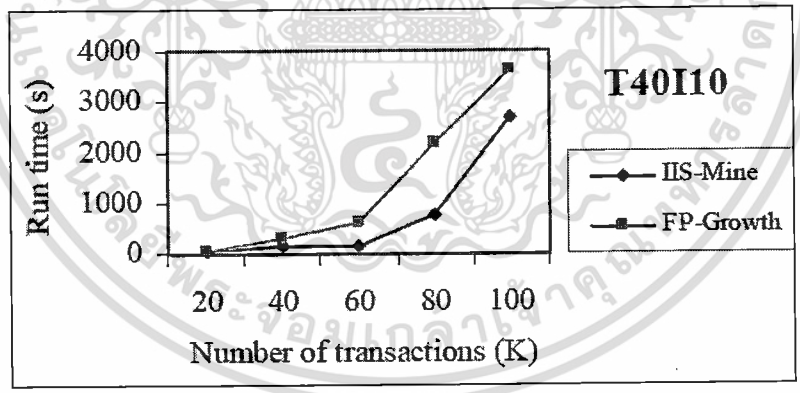

Figure 4.3 (b). Scalability of run time on T40I10

In Figures 4.3 (b) and 4.4 (b), the algorithms were run on all of the datasets generated from T40I10 for a minimum support of 5. Both run time and memory consumption were recorded. Figure 4.3 (b) confirms that the run times of both algorithms rely on the length and the number of transactions; if the length or number of transactions increases, the run times of both algorithms also increase. However, the run time of IIS-Mine was better than that of FP-growth for every number of transactions. Figure 4.4 (b) confirms that the memory consumption of algorithms

increases with the length and the number of transactions. However, the memory consumption of IIS-Mine was better than that of FP-growth for every number of transactions.

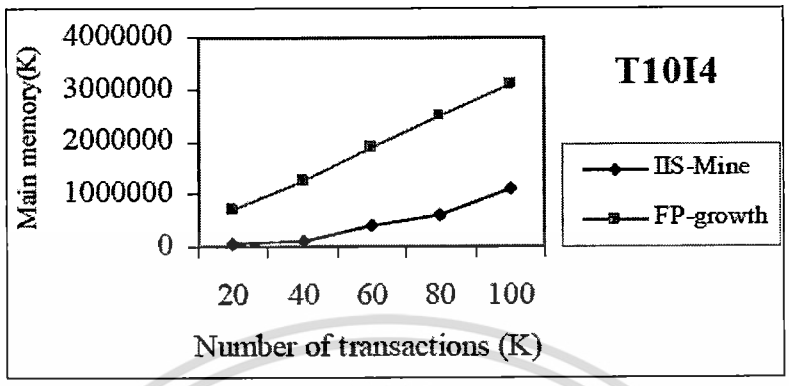

Figure 4.4 (a). Scalability of memory consumption on T10I4

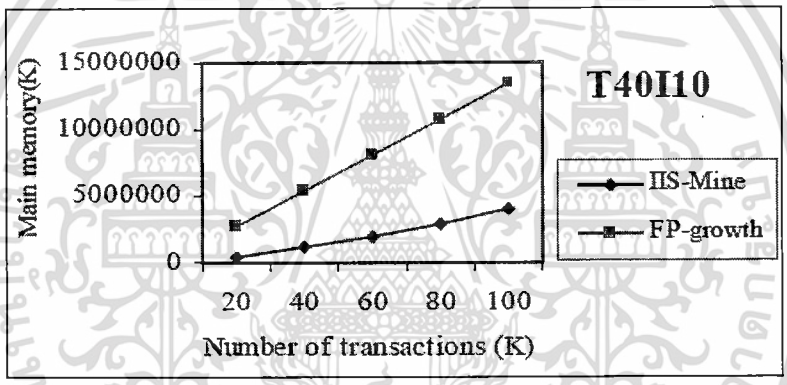

Figure 4.4 (b). Scalability of memory consumption on T40I10

CHAPTER 5

CONCLUSIONS AND SUGGESTIONS

5.1 Conclusions

A data structure called an IIS has presented to store transaction data by scanning a database only once. Changing the minimum support does not affect the IIS, and rescanning the database is not required. We also present a new algorithm called IIS-Mine that is able to find frequent itemsets without generating candidate itemsets. It employs a more efficient use of the extendable-itemset property to reduce the number of recursive steps of mining. The node construction and the size of trees are then reduced, thereby reducing the runtime and memory consumption. The completeness and correctness of the proposed method was confirmed by a mathematical proof, and the simulated experiments compared it with the FP-growth algorithm using both synthetic and real datasets. Experiments were presented to analyze the runtime, memory consumption and scalability with different datasets when the minimum support threshold is varied. Although the proposed method accesses the IIS structure multiple times, the experimental results demonstrate that the IIS-Mine algorithm is better than the FP-growth algorithm in runtime and space consumption and for dense datasets.

5.2 Suggestions

The proposed method has some drawbacks: first, the proposed method needs to scan an IIS many times to construct the IIS_{item}Trees which is time consuming. Second, the run time performance of the IIS-Mine algorithm depends on the size of the IIS_{item}Trees. If each of IIS_{item}Tree is very small, then the performance of the algorithm is deceased because the property extendable-itemset is used rarely.

In the future research, the data structure and algorithm of mining will be improved to find new ways to mine all frequent itemsets. For example, mining of frequent itemsets without minimum support threshold but by using automate mining with intelligence method such as soft computing techniques. Moreover, mining on stream data may be another interesting area.

REFERENCES

1. J. Han and M. Kamber, "Data Mining: Concepts and Techniques", Elsevier, Maryland Heights (MO), **2006**, pp.227-231.
2. J. Han, J. Pei, Y. Yin and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach", *Data Mining Knowl. Discov.*, **2004**, *8*, 53-87.
3. G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using FP-Trees", *IEEE Trans. Knowl. Data Eng.*, **2005**, *17*, 1347-1362.
4. R. Agrawal, T. Imielinski and A. Swami, "Mining association rules between sets of items in large databases", Proceedings of ACM SIGMOD International Conference on Management of Data, **1993**, Washington, DC, USA, pp.207-216.
5. R. Agrawal and R. Srikant, "Fast algorithms for mining association rules", Proceedings of 20th International Conference on Very Large Data Bases, **1994**, Santiago de Chile, Chile, pp.487-499.
6. T. H. N. Vu, J. W. Lee and K. H. Ryu, "Spatiotemporal pattern mining technique for location-based service system", *ETRI J.*, **2008**, *30*, 421-431.
7. J. Han, J. Pei and Y. Yin, "Mining frequent patterns without candidate generation", Proceedings of ACM SIGMOD International Conference on Management of Data, **2000**, Dallas, (TX), USA, pp.1-12.
8. J. Pei, J. Han, H. Lu, S. Nishio, S. Tang and D. Yang, "H-mine: Hyper-structure mining of frequent patterns in large databases", Proceedings of IEEE International Conference on Data Mining, **2001**,San Jose, California, USA, pp.441-448.
9. A. Pietracaprina and D. Zandolin, "Mining frequent itemsets using Patricia tries", Proceedings of 3rd IEEE International Conference on Data Mining, **2003**, Melbourne, (FL), USA.
10. G. Grahne and J. Zhu, "Efficiently using Prefix-Trees in mining frequent itemsets", Proceedings of 3rd IEEE International Conference on Data Mining, **2003**, Melbourne, (FL), USA.
11. Q. Zhu and X. Lin, "Depth first generation of frequent patterns without candidate generation", Proceedings of 11th Pacific-Asia Conference on Knowledge Discovery and Data Mining, **2007**, Nanjing, China, pp.378-388.

12. S. Sahaphong and V. Boonjing, "The combination approach to frequent itemsets mining", Proceedings of 3rd International Conference on Convergence and Hybrid Information Technology, **2008**, Busan, Korea, pp.565-570.

13. V. K. Shrivastava, P. Kumar and K. R. Padasani, "FP-tree and COFI based approach for mining of multiple level association rules in large database", *Int. J. Comp. Sci. Inf. Secur.*, **2010**, *7*, 273-279.

14. L. Huang, J. Z. Liang, Y. Pan and Y. Xian, "A complete attribute reduction algorithm based on improved FP tree", Proceedings of International Conference on Circuits, Communications and System, **2010**, Beijing, China, pp.1-4.

15. U. Yun and K. H. Ryu, "Approximate weighted frequent pattern mining with/without noisy environments", *Knowl.-Based Syst.*, **2011**, *24*, 73-82.

16. M. J. Zaki, "Scalable algorithms for association mining", *IEEE Trans. Knowl. Data Eng.*, **2000**, *12*, 372-390.

17. M. J. Zaki and K. Gouda, "Fast vertical mining using diffsets", Proceedings of 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, **2003**, Washington, DC, USA, pp.236-355.

18. D. J. Chai, L. Jin, B. Hwang and K. H. Ryu, "Frequent pattern mining using bipartite graph", Proceedings of 18th International Conference on Database and Expert Systems Applications, **2007**, Regensburg, Germany, pp.182-186.

19. J. Dong and M. Han, "BitTableFI: An efficient mining frequent itemsets algorithm", *Knowl.-Based Syst.*, **2007**, *20*, 329-335.

20. W. Yen, "A new mining algorithm based on frequent item sets", Proceedings of International Workshop on Knowledge Discovery and Data Mining, **2008**, Adelaide, Australia, pp.410-413.

21. W. Song, B. Yang and Z. Xu, "Index-BitTableFI: An improved algorithm for mining frequent itemsets", *Knowl.-Based Syst.*, **2008**, *21*, 507-513.

22. S. Sahaphong and V. Boonjing, "Mining of frequent itemsets by using the property of extendable-itemset", Proceedings of 7th International Joint Conference on Computer Science and Software Engineering, **2010**, Bangkok, Thailand, pp.168-173.

23. S. Sahaphong and G. Sritanratana, "Mining of frequent itemsets with JoinFI-Mine algorithm", Proceedings of 10th WSEAS International Conference on Artificial

Intelligence, Knowledge Engineering and Database, **2011**, Cambridge, United Kingdom, pp.73-78.

24. Frequent Itemset Mining Dataset Repository, "T10I4D100K", http://fimi.cs.helsinki.fi/data/, **2003** (Accessed: January 11, 2010).
25. Frequent Itemset Mining Dataset Repository, "T40I10D100K", http://fimi.cs.helsinki.fi/data/, **2003** (Accessed: January 11, 2010).
26. Workshop on Frequent Itemset Mining Implementations, http://fimi.ua.ac.be/fimi03/, **2003** (Accessed: January 2, 2010).
27. Workshop on Frequent Itemset Mining Implementations, http://fimi.ua.ac.be/fimi04/, **2004** (Accessed: January 2, 2010).
28. UCI Machine Learning Repository, http://archive.ics.uci.edu/ml/, **2007** (Accessed: March 15, 2010).
29. UCI Machine Learning Repository, "Chess", http://archive.ics.uci.edu/ml/datasets.html, **1989** (Accessed: July 19, 2010).
30. UCI Machine Learning Repository, "Mushroom", http://archive.ics.uci.edu/ml/datasets.html, **1987** (Accessed: July 19, 2010).

APPENDIX

APPENDIX A

Publications

1. S. Sahaphong and V. Boonjing, "IIS-Mine: A new efficient method for mining frequent itemsets", *Maejo Int. J. Sci. Technol.*, **2012**, *6*, 130-151.
2. S. Sahaphong and G. Sritanratana, "Mining of frequent itemsets with JoinFI-Mine algorithm", Proceedings of the 10th International Conference on Artificial Intelligence, Knowledge Engineering and Database, **2011**, Cambridge, United Kingdom, pp.73-78.
3. S. Sahaphong and V. Boonjing, "Mining of frequent itemsets using extendable-itemset property", Proceeding of the 7th International Join Conference on Computer Science and Software Engineering, **2010**, Bangkok, Thailand, pp. 168-173.
4. S. Sahaphong and K. H. Ryu, "A modify SL-Mine algorithm approach to frequent itemsets mining", Proceeding of the 2nd International Conference on Frontiers of Information Technology, Application and Tools, **2009**, Cheongju, Korea, pp. 18-22.
5. S. Sahaphong, "Frequent itemsets mining using vertical index list", Proceeding of the 2nd International Conference on Computer Science and Information Technology, **2009**, Beijing, China, pp. 480-484.
6. S. Sahaphong and V. Boonjing, "A combination approach to frequent itemsets mining", Proceedings of the 3rd International Conference on Convergence and Hybrid Information Technology, **2008**, Busan, Korea, pp. 565-570.
7. S. Sahaphong and N. Hiransakolwong, "Unsupervised image segmentation using automated Fuzzy c-Mean", Proceeding of the 7th International Conference on computer and Information Technology, **2007**, Fukushima, Japan, pp. 690-694.

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

Maejo International Journal of Science and Technology

ISSN 1905-7873
Available online at www.mijst.mju.ac.th

Full Paper

IIS-Mine: A new efficient method for mining frequent itemsets

Supatra Sahaphong* and Veera Boonjing

Department of Mathematics and Computer Science, Faculty of Science, King Mongkut's Institute of Technology Ladkrabang, Bangkok 10520, Thailand

* Corresponding author, e-mail: supatra@ru.ac.th

Received: 11 September 2011 / Accepted: 4 March 2012 / Published: 23 April 2012

Abstract: A new approach to mine all frequent itemsets from a transaction database is proposed. The main features of this paper are as follows: (1) the proposed algorithm performs database scanning only once to construct a data structure called an inverted index structure (IIS); (2) the change in the minimum support threshold is not affected by this structure, and as a result, a rescan of the database is not required; and (3) the proposed mining algorithm, IIS-Mine, uses an efficient property of an extendable itemset, which reduces the recursiveness of mining steps without generating candidate itemsets, allowing frequent itemsets to be found quickly. We have provided definitions, examples, and a theorem, the completeness and correctness of which is shown by mathematical proof. We present experiments in which the run time, memory consumption and scalability are tested in comparison with a frequent-pattern (FP) growth algorithm when the minimum support threshold is varied. Both algorithms are evaluated by applying them to synthetics and real-world datasets. The experimental results demonstrate that IIS-Mine provides better performance than FP-growth in terms of run time and space consumption and is effective when used on dense datasets.

Keywords: association rule mining, data mining, frequent itemsets mining, frequent patterns mining, knowledge discovering

INTRODUCTION

The objective of frequent itemset mining is to identify all frequently occurring itemsets using a support threshold. Decision-makers are interested in all itemsets associated with high frequencies. Association rule mining algorithms can be broken down into two major phases. The first phase finds all of the itemsets that satisfy the minimum support threshold, which are the frequent itemsets. The

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

second phase is rule generation, in which all the high confidence rules from the frequent itemsets found in the previous phase are extracted [1]. Many previous investigations focused on the first phase. Early algorithms based on generated and tested candidate itemsets have two major defects. First, the database must be scanned multiple times to generate candidate itemsets, which increases the I/O load and is time-consuming. The search space of itemsets that must be explored grows exponentially. Second, enormous candidate itemsets are generated and calculated from their supports, which consumes a large amount of CPU time [2].

To overcome the above-mentioned problems, a next generation of algorithms using a compact tree structure was proposed, called a frequent-pattern (FP) tree [3], which finds frequent itemsets directly from the data structure. However, most of the FP-tree-based algorithms have the following weaknesses. First, the mining of frequent itemsets from the FP-tree to generate a huge conditional FP-tree requires a large amount of run time and space. The best case is when a database has the same set of transactions; an FP-tree then contains only a single branch of nodes. The worst case is when a database has a unique set of transactions [3]. Second, when the users change to a new minimum support threshold for their new decision, the algorithm restarts the whole operation and scans the database twice.

Many researchers have tried to solve the above problems using a vertical data layout. However, most of the algorithms have the drawback of increasing the run time and space consumption due to the following reasons. First, when the users change to a new minimum support threshold for their new decision, the algorithm restarts the whole operation more than one time to scan a database and construct their data structure. Rescanning the database for a new minimum support threshold wastes both run time and space. Second, all of the FP-tree-based algorithms generate a huge conditional FP-tree, which has a large number of recursive processing steps and requires a large amount of run time and space consumption.

In this paper we present a new, efficient method to solve the above-mentioned problems by proposing both a data structure and a mining algorithm for decreasing the consumption of run time and space. First, the proposed method performs database scanning to construct a data structure called an inverted index structure (IIS) only once. In addition, changing the minimum support threshold does not affect the IIS; therefore, database rescanning is not required. Second, IIS-Mine is a new algorithm that mines all of the frequent itemsets without generating candidate itemsets and uses a new tree structure called the IIS_{Item}Tree. IIS-Mine employs an efficient property of the extendable itemset, which decreases the number of recursive processing steps when mining frequent itemsets. The completeness and correctness of the algorithm is proved using a mathematical proof. Last, the efficiency of IIS-Mine is compared with that of FP-growth in terms of run time and space consumption through simulation experiments. Our experiments show that IIS-Mine is more efficient than FP-growth in run time and space consumption for dense datasets.

RELATED WORK

The first algorithm to generate all frequent itemsets is the AIS algorithm, which was first introduced by Agrawal et al [4]. However, this algorithm constructs a list of all of the possible

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

itemsets at each level of traversal, so infrequent itemsets that are not needed are also generated. Later, the algorithm was improved upon and renamed the Apriori algorithm by Agrawal et al [5]. The Apriori algorithm uses a level-wise and breadth-first search approach for generating association rules. Many efficient association mining techniques have been developed based on the Apriori algorithm. Vu et al. [6] proposed a rule-based location prediction technique to predict the user's featured location, but this proposal generates more candidate itemsets than are required. These algorithms are also expensive in terms of I/O load and run time when the database must be scanned multiple times to generate candidate itemsets.

The above-mentioned problems can be improved upon by using a compact tree structure and finding frequent itemsets directly from the data structure. The algorithms scan a database twice. The first scan of the database is to discard infrequent itemsets; the second is to construct a tree. The FP-growth algorithm, developed by Han et al. [7], is the most popular method. It performs a depth-first search approach in a search space. It encodes a dataset using a compact data structure called an FP-tree or prefix tree and extracts frequent patterns directly from the FP-tree. Many approaches have been proposed to extend and improve upon this algorithm. Pei et al. [8] developed the H-mine algorithm using array- and tree-based data structures to improve the main memory cost. The PatriciaMine algorithm [9] compressed Patricia tries to store datasets, which is space efficient for both dense and sparse datasets. The FP-growth algorithm [10] reduces the FP-tree traversal time using an array technique. Zhu [11] proposed a new method to compress a large database into an FP-tree with a children table but not a header table, and applied a depth-first search with this tree for the mining step, which reduces both the run time and the space consumption. Sahaphong and Boonjing [12] proposed a new algorithm which constructs a pattern base using a new method that is different from the pattern base in the FP-growth and mined frequent itemsets using a new combination method without the recursive construction of a conditional FP-tree. An approach based on the FP-tree and co-occurrence frequent items (COFI) was proposed to find frequent items in multilevel concept hierarchy by using a non-recursive mining process [13]. A new data structure called improved FP tree was proposed, which can reduce space consumption and enhance the efficiency of an attribute reduction algorithm [14]. To maintain the anti-monotone property of approximate weighted frequent patterns, a robust concept was proposed to relax the requirement for exact equality between the weighted supports of patterns and a minimum threshold [15]. However, most of the FP-tree-based algorithms require a large amount of run time and space to generate the huge conditional FP-trees. Moreover, the algorithm restarts the whole operation and requires that a database be scanned twice when the minimum support threshold is changed.

As mentioned above, most of the algorithms that mine frequent itemsets use a horizontal data layout. However, many researchers use a vertical data layout. The Eclat algorithm was proposed [16] to generate all frequent itemsets in a breadth-first search using the joining step from the Apriori property when no candidate items can be found. The Eclat algorithm is very efficient for large itemsets but is less efficient for small ones. The diffset technique [17] was introduced to improve the memory requirement. Chai et al. [18] detailed a data structure called large-item bipartite graph to accommodate the data when a database is scanned. Similar to the FP-growth algorithm, this method

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

mines frequent patterns using the recursive conditional FP-tree. The BitTableFI algorithm [19] uses horizontal and vertical data layouts to compress a database. Yen [20] presented an algorithm based on an undirected itemset graph that finds frequent itemsets by searching undirected graphs. When the database and minimum support change, this algorithm requires that the graph structure be re-searched to generate new frequent itemsets. The Index-BitTableFI [21] was developed to reduce the cost of candidate generation and to support counting. Sahaphong and Boonjing [22] proposed a new algorithm that reduces the run time. The drawback of this algorithm is its large memory consumption from generation of many repeated nodes. The JoinFI-Mine algorithm [23] uses a sorted-list structure constructed from the vertical data layout and finds all frequent itemsets using a depth-first search for joining frequent itemsets. Therefore, this algorithm consumes time and space in its joining step.

METHODS

Frequent-Itemsets Mining Problem

We introduce the basic concepts of mining frequent itemsets. All terminologies in this section are proposed by Han et al [2].

Let $I=\{x_1,x_2,...,x_m\}$ be a set of items and $DB=\{T_1,T_2,...,T_n\}$ be a transaction database, where $T_1,T_2,...,T_n$ are transactions that contain items in I. The support, or *supp* (occurrence frequency), of a pattern A, where A is a set of items, is the number of transactions containing A in DB. A pattern A is frequent if A's support is no less than a predefined minimum support threshold, *minsup*.

Given a DB and a minimum support threshold *minsup*, the problem of finding a complete set of frequent itemsets is called the frequent-itemsets mining problem.

For a greater understanding, we provide an example to illustrate the above definitions.

Example 1. An example of the database by Han et al. [2] is used here. Table 1 is a DB. It consists of 5 transactions (T_1, T_2, T_3, T_4, and T_5) and 17 items (*a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p*, and *s*). For example, the first transaction is T_1, which contains *f, a, c, d, g, i, m*, and *p*.

Table 1. A transaction database

Transaction	Item
T_1	*f, a, c, d, g, i, m, p*
T_2	*a, b, c, f, l, m, o*
T_3	*b, f, h, j, o*
T_4	*b, c, k, s, p*
T_5	*a, f, c,* e, *l, p, m, n*

IIS: Design and Construction

We present a data structure that contains transaction data called an inverted index structure (IIS). The IIS is a structure that holds a relationship between items and the transactions included within. The IIS is constructed from one scan of the DB. This original IIS can support every *minsup*;

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

therefore, it does not need to rescan the DB when the *minsup* is changed. According to the definitions in the previous section, we present a new definition with an example and an algorithm to demonstrate how to construct this IIS.

Definition 1 (IIS). Let DB be a transaction database and I be a non-empty finite set of all items in each transaction in the DB, where each transaction is a set of items in I associated with an identifier, and let S be a set of all non-empty subsets of DB. An IIS is the function $f: I \rightarrow S$ defined by $f(a) = S_1$ if T contains a for each $T \in S_1$. This function can be identified as a table consisting of the attributes of the items in I and the corresponding transactions in the DB. That is, each row in the IIS contains an item in I as well as the transactions in the DB that contain that item. The set of transactions are written in the order of their ascending identification numbers.

With the above definition, the IIS represents the relationship between each item in I and its corresponding transactions; therefore, the IIS can apply to all minimum support thresholds, and a rescan of the database is not required. We demonstrate the steps to construct the IIS through the following example.

Example 2. We use the example of a DB in Table 1. The DB is scanned once to create the IIS. The scan of the first transaction is T_1, which consists of items *f, a, c, d, g, i, m* and *p*. The transaction T_1 will be inserted for each corresponding item sorted in ascending order (*a, c, d, f, g, i, m, p*). T_1 will be the first transaction inserted in the transactions of item *a*. The second examined item is *c*, so we insert T_1 in item *c*. Next, we examine item *d*; we then subsequently insert T_1 in item *d*. The remaining items (*f, g, i, m* and *p*) in T_1 can be similarly inserted. The remaining transactions (T_2, T_3, T_4 and T_5) in the DB are performed in a similar manner.

Algorithm 1 shows how to construct the IIS. Figure 1 shows all of the items of the IIS after scanning the DB once, and the bold items are all frequent items that have a support greater than or equal to the *minsup*, which is assumed to be 3.

```
Algorithm 1 (IIS construction)
Input: DB.
Output: IIS.
Method: The IIS is constructed as follows.
1 Begin
2   Create header that contains all items.
3   For each transaction T in DB do  // scanning DB once
4      Sort items in T                // ascending order
5      Create transaction to each corresponding item
6   End //For
7 End //Begin
```

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

Item	Transaction			
a	T_1	T_2	T_5	
b	T_2	T_3	T_4	
c	T_1	T_2	T_4	T_5
d	T_1			
e	T_5			
f	T_1	T_2	T_3	T_5
g	T_1			
h	T_3			
i	T_1			
j	T_3			
k	T_4			
l	T_2	T_5		
m	T_1	T_2	T_5	
n	T_5			
o	T_2	T_3		
p	T_1	T_4	T_5	
s	T_4			

Figure 1. An example of the IIS

IIS-Mine Algorithm

We present a new algorithm called IIS-Mine. This algorithm uses a new tree structure, called the IIS_{item}Tree, to mine frequent itemsets. The main features of this algorithm are as follows: (1) every frequent itemset is found without generating candidate itemsets; (2) the algorithm reduces the recursion of mining steps using the property of extendable itemset; and (3) the algorithm supports the mining of frequent itemsets with any value of the *minsup* without needing to rescan the database. From the above features, we can quickly find the frequent itemsets and completely and correctly obtain them. We now introduce the terminologies of the IIS_{item}Tree, its construct, the theorem, the examples and the algorithms to describe how to mine frequent itemsets.

Definition 2 (Itemset-tree structure). An itemset-tree structure is a tree structure constructed from the IIS. It is a finite set of one or more nodes with the following structure:

(i) It consists of the root which contains an item, a set of item subtrees as the children of the root, and a set of header tables.

(ii) Each node in this tree comprises five fields: *item-name*, which registers which item this node represents; *support*, which registers the number of transactions represented by the portion of the path reaching this node; *same-item*, which represents a pointer that points to the node in the itemset-tree structure that carries the same item-name; *parent*, which represents a pointer that points to the previous node in the same path; and *child*, which represents a pointer that points to the child node.

(iii) Each member of the *header table* consists of two fields, *item-name* and *head of node link*, where *head of node link* represents a pointer that points to the first node in the itemset-tree structure carrying the *item-name*.

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

Definition 3 (IIS_{item}tree). Let x_0 be an arbitrary frequent item in a given transaction database and IIS be the inverted index structure of the transaction database. A tree T constructed from the IIS is called an inverted index structure-x_0 tree, denoted by $IIS_{x_0}tree$, if it satisfies the following:

(i) Each node of T is of the form $(A:s)$, where A is a frequent itemset and s is its support. If $(A:s)$ is a node of T and $A=\{a\}$, where a is a frequent item, then $(A:s)$ is simply written by $(a:s)$.

(ii) Let $(x_0:s_0)$ be its root, where s_0 is the support of x_0.

(iii) Let $x_0,x_1,...,x_k$ be frequent items in the IIS, and $s_i = supp\{x_0,x_1,...,x_i\}$ for all $i=0,1,...,k$. In this case, $P=((x_0:s_0),(x_1:s_1),...,(x_k:s_k))$ is a path from the root $(x_0:s_0)$ to a leaf $(x_k:s_k)$ of the tree T if and only if $s_0 \geq s_1 ... \geq s_k > 0$, $x_0 <_l x_1 <_l ... <_l x_k$, where $<_l$ is the lexicographic order. If a is a frequent item in the IIS and if $(a:s)$ is a node of T such that $x_k < a$ or $x_i <_l a <_l x_{i+1}$ for all $i=0,1,...,k$, then $supp\{x_0,x_1,...,x_i,a\}=0$ and $(a:s)$ is not a node of P.

The header table of $IIS_{x_0}tree$ is a set of all frequent items a of a node $(a:s)$ of this tree.

Based on the above definition, we have the IIS_{item}Tree construction algorithm, as shown in Algorithm 2. It is evident that if *minsup*>0 is a minimum support threshold, then every frequent itemset can be derived from an IIS_{item}Tree.

```
Algorithm 2 (IIS_item Tree construction)
Input: IIS.
Output: IIS_item Tree.
Method: An IIS_item Tree is constructed as follows.
1Begin
2  Create header table
3  Read frequent item x in IIS
4  Create root R and initial supp(R) to 1
5  Link R to header table
6  For each transaction T of root R where T = T_1 to T_n do
7    While next frequent item≠(last frequent item)+1 do
8      Read next frequent item (N) that has same T with R
9      Call InsertTree (N,R)
10   End//While
11 End//For
12End //Begin
Procedure InsertTree (N,R)
1Begin
2  If IIS_RTree has a node C such that C.item-name = N.item- name then
3    Increment supp(N) by 1
4  Else
5    Create new node N and initial supp(N) to 1
6    Link N to N's parent
7    Link N to N's header table
8    Link N to same-item
9  End //If
10End //Begin
```

If no confusion arises, then $\{x_1,x_2,...,x_n\}$ and $(\{x_1,x_2,...,x_n\}:s)$ are replaced by $x_1x_2...x_n$, and $<x_1x_2...x_n:s>$ respectively, where $x_1,x_2,...,x_n$ are items.

Example 3. In this example, we describe the steps to construct all of the IIS_{item}Trees except the last frequent item p using the IIS in Figure 1 and *minsup* = 3. The first frequent item in the IIS is

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

a; therefore, we first construct the *IIS$_a$Tree*, and item *a* is a root. We obtain two paths: (<*a*:2>, <*c*:2>, <*f*:2>, <*m*:2>, <*p*:2>) and (<*a*:1>, <*b*:1>, <*c*:1>, <*f*:1>, <*m*:1>). The first path consists of frequent items (*a*,*c*,*f*,*m*,*p*) that appear twice in the DB. Similarly, the second path indicates frequent items (*a*,*b*,*c*,*f*,*m*) that are contained in only one transaction in the DB. These two paths share the frequent item *a*; thus, *a* appears three times in the DB. Other IIS$_{item}$Trees, except the *IIS$_p$Tree*, can be similarly constructed. In Figure 2, all of the IIS$_{item}$Trees, except the last *IIS$_p$Tree*, are illustrated.

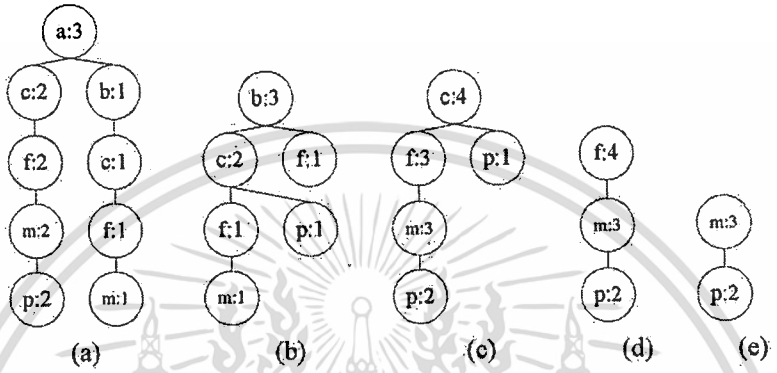

Figure 2. IIS$_{item}$Trees

Definition 4 (A$_1$-tree). Let DB be a transaction database; let *T* be a tree such that each node of the tree is of the form $(A:s)$, where *A* is a frequent itemset in DB and *s* is the support of *A*; and let A_1 be a frequent itemset in the DB. *T* is called an *A_1-tree* if, for any path *P* of *T*, *P* is of the form $P=((A_1:s_1),(A_2:s_2),...,(A_k:s_k))$, where *k* is a positive integer; A_i and A_j are pairwise disjoint frequent itemsets for all $i, j = 1,2,...,k$ with $i \neq j$; s_i is the support of $\bigcup_{m=1}^{i} A_m$ for $i = 1,2,...,k$; $(A_1:s_1)$ is the root of *T*; and $(A_k:s_k)$ is a leaf of *T*.

Example 4. According to Figure 2 (a), the *ac*-tree is shown in Figure 3.

Definition 5 (Prefix subpath). Let *T* be a tree, a_1 be the root of *T*, and $P=(a_1,...,a_m)$ be a path of *T*, where *m* is a positive integer. Every path $Q=(a_1,...,a_k)$ of *T* is then called a prefix subpath of *P*, where $1 \leq k \leq m$.

Example 5. According to Figure 2(a), the paths ((*a*:3),(*c*:2)) and ((*a*:3),(*c*:2),(*f*:2)) are prefix subpaths of ((*a*:3),(*c*:2),(*f*:2),(*m*:2)).

Definition 6 (Subheader). Let *T* be an *A-tree* and $(x:s(x))$ be a node of *T*, where *x* is a frequent item in a given transaction database and $s(x)$ is the support of *x*. Suppose that all of the nodes (of *T*) containing *x* are only in the paths $P_1,...,P_k$ of *T* from the root $(A:s)$ to some leafs of *T*, and suppose that Q_i is a prefix subpath (of P_i) from the root $(A:s)$ to $(x:s(Q_i))$ for all $i = 1,...,k$. The subheader of *T*, denoted by *SH*(*A*), is defined as the order set *SH(A)* = $\{x|x$ is a frequent item not contained in *A*, $(x:s(Q_i))$ is a node in Q_i for $i = 1,...,k$ and $\sum_{i=1}^{k} s(Q_i) \geq minsup\}$.

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

Example 6. According to Figure 2(a), SH(a) = {c,f,m}, where SH(a) is the subheader of the tree in Figure 2 (a), $supp(c) = 3$, $supp(f) = 3$, and $supp(m) = 3$.

Definition 7 (Conditional itemset-tree). Let A_1 be a frequent itemset, T be an *A_1-tree*, x_2 be a frequent item with $x_2 \in SH(A_1) - A_1$, and $A_1x_2 := A_1 \cup \{x_2\}$ be a frequent itemset. A *conditional A_1x_2-tree*, denoted by $T(A_1x_2)$, is a tree that satisfies the following:

(i) $(A_1x_2 : s_2)$ is the root of $T(A_1x_2)$, where s_2 is the support of A_1x_2 and $s_2 \geq minsup$.

(ii) The number of items in $SH(A_1) > 1$.

(iii) All of the paths are derived from T in the following way: $Q = ((A_1x_2 : s_2),(x_3 : s_3),...,(x_k : s_k))$ is a path of $T(A_1x_2)$ if and only if a path $P = ((A_1 : s_1),(x_2 : s_2),...,(x_l : s_l))$ exists from the root $(A_1 : s_1)$ to the leaf $(x_l : s_l)$ of T, where $l \geq k$ and x_k is in $SH(A_1)$; and if there is a node $(x_r : s_r)$ of P such that $r > k$ and $((A_1 : s_1),(x_2 : s_2),...,(x_r : s_r))$ is a prefix subpath of P, then x_r must not be in $SH(A_1)$.

Example 7. According to Figure 2 (a), the conditional itemset-tree, or conditional *ac*-tree, is shown in Figure 4.

Notably, every non-empty subset of a frequent itemset is also frequent. This fact leads to the following definition.

Definition 8 (Frequent itemset* of length m derived from tree). Let A be a frequent itemset, x be a frequent item, and $x \notin A$. Suppose that T is a conditional Ax-tree, m is a positive integer greater than 1, and $Ax = A \cup \{x\}$ has m elements. Ax is called a frequent itemset* of length m derived from T, denoted by $FS^*_m(Ax)$, if T contains precisely two nodes and Ax is a frequent itemset, or if $SH(A) = \{x\}$.

Example 8. According to Figure 5, the frequent itemset* of length 4 derived from the conditional *acf*-tree is $FS^*_4(acfm) = acfm$.

Figure 3. *ac*-tree

Figure 4. Conditional *ac*-tree

Figure 5. Conditional *acf*-tree

Definition 9 (Extendable frequent itemset*). Let m be a positive integer greater than 1, T be a $T(Ax)$ defined as in definition 7 with $A_1 = A$ and $x_1 = x$, and Ax be a frequent itemset* of length m derived from T. Each itemset in $FS^*_m(Ax)$ is said to be extendable if $m \geq 3$. For every $k = 2, 3,...,$ m, we let $Ext^*_k(Ax)$ denote the set of all itemsets containing exactly k items of $FS^*_m(Ax)$, where $m \geq 3$. Each element of $Ext^*_k(Ax)$ is called an extendable frequent itemset* (derived from T) of length k for all $k \geq 2$. The set of all extendable frequent itemsets* of length k that is denoted by $Ext^*_k = \cup\{Ext^*_k(Ay) \mid Ay$ is a frequent itemset* of length at least k derived from a conditional Ay-*tree*$\}$ for all $k \geq 2$.

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

Example 9. According to the previous example, the length of $FS_4^*(acfm)$ is 4; we then find that $Ext_2^*(acfm)$ = {*ac*, *af*, *am*, *cf*, *cm*, *fm*} and $Ext_3^*(acfm)$ = {*acf*, *acm*, *afm*, *cfm*}. Therefore, Ext_2^* = {*ac*, *af*, *am*, *cf*, *cm*, *fm*, and all members of other $Ext_2^*(Ay)$ } and Ext_3^* = {*acf*, *acm*, *afm*, *cfm*, and all members of other $Ext_3^*(Ay)$ }.

Definition 10 (Frequent itemset* of length *m*). Let *k* be the maximum length of all frequent itemsets in a transaction database, $FS_m^*(Ax)$ and Ext_k^* be given as in definitions 8 and 9 respectively; let $A_m = \{FS_m^*(Ax) \mid Ax$ being a frequent itemset* of length *m* derived from $T\}$ for $m = 2, 3, \ldots, k$; define $FS_2^* = A_2 \cup \{Ax \mid Ax$ being a frequent itemset of length 2}; and define $FS_m^* = A_m$ for $m = 2, 3, \ldots, k\}$. Let FI_m^* be defined by $FI_1^* = \{\{x\} \mid x$ being a frequent item} and $FI_m^* = FS_m^* \cup Ext_m^*$ for $m = 2, 3, \ldots, k\}$. Any element of FI_m^* is called the frequent itemset* of length *m*.

Example 10. According to the previous example, we find that FI_2^* = {*ac*, *af*, *am*, *cf*, *cm*, *fm*}, FI_3^* = {*acf*, *acm*, *afm*, *cfm*}, and FI_4^* = {*acfm*}.

Definition 11 (Frequent itemset*). Let FI_m^* be given as in definition 10, and let FI^* denote $\cup_{m=1}^{k} FI_m^*$, where *k* is the maximum length of all frequent itemsets in a transaction database. Any element of FI^* is called a frequent itemset*.

Example 11. According to the previous example, FI^* is {*ac*, *af*, *am*, *cf*, *cm*, *fm*, *acf*, *acm*, *afm*, *cfm*, *acfm*}.

On the basis of the above definitions and examples, Algorithm 3 presents the IIS-Mine algorithm to show how it can be used to mine all frequent itemsets.

```
Algorithm 3: (IIS-Mine: Mining frequent itemsets using IIS_itemTree)
Input: IIS, IIS_item Trees constructed according to Algorithm 2, and minsup
Output: FI*
Procedure AllFreqItemset (IIS, FI*)
1 Begin
2   For each frequent item x in the IIS do
3     FI_1* = {{x | x∈I, supp(x) ≥ minsup}
4     Call IIS_xTree , which is constructed from Algorithm 2
5     If Tree ≠{}
6       Call IIS-Mine (Tree, x)
7     End //If
8   End //For
9   Find FI* = ∪_{m=1}^{k} FI_m*                    // FI* is given in definition 11
10 End //Begin
Procedure IIS-Mine(Tree, x)
1 Begin
2   Call SubHeader (A-tree, subheaderA)
3   Generate all Ax with its support
    //All Ax are the frequent itemsets where A is the root of A-tree and x ∈subheaderA
4   For each |λ|> 1 do  // λ is Ax
5     Flag=1
6     While Flag = 1 do
7       Call SkipFreqItemset (subheaderA, λ , δ ,FlagRepeat, FlagTree)
8       If FlagRepeat=0 and FlagTree=0 then  // (δ ∉ FI_m*)
9       Call CondItemsetTree (A-tree, δ , conditional δ -tree)
```

```
    Else
      Flag=0
    End // If
    If conditional Ax-tree contains greater than two nodes
      Call IIS-Mine(conditional δ -tree, x)
    Else Flag=0
    End //If
  End //While
  If |FS*m(δ)|≥3 and FS*m(δ) ∉ FI*m then
    Call ExtFreqItemset( FS*m(δ) , Ext*k )          // FS*m is given in definition 8
    Save FS*m(δ) and all Ext*k to FI*m              // FI*m is given in definition 10
  Else
    If FS*m(δ) ∉ FI*m then
      Save FS*m(δ) to FI*m
    End //If
  End //If
 End //For
End //Begin
Procedure SubHeader (A-tree, subheaderA)
Begin
 Each frequent item x of A-tree                   // SH(A) is given in definition 6
 Find {(x:s(x))|x ∈SH(A), s(x) is the support of x in A-tree}
End // Begin
Procedure CondItemsetTree(Tree, δ , conditional δ -tree)
Begin
 Scan tree once to collect the paths that have an association with root δ
 For all paths are derived from tree do
  Connect all paths to δ
 End //For
End //Begin
Procedure SkipFreqItemset (SubheaderA, λ , δ , FlagRepeat, FlagTree)
Begin // To skip the construction of conditional item tree
// x is the frequent item in subheaderA. // λ is the root of tree, or a frequent itemset
// FlagRepeat=0 means δ ∉ FI*m        // FlagTree=0 means the conditional item-tree is constructed
// n is the maximum number of elements of itemsets in subheaderA
 FlagRepeat =1, FlagTree=1
 β = λ, α = β
 While( FlagRepeat=1 and (order of x ≤ n )) do
  If β ∉ FI*m then
    FlagRepeat=0, FlagTree=0
    If x is the last item
      If |δ|= 2 then FlagTree=1
      End //If
      δ = α
    End //If
   Else
    If x is the last item then
      Increment order of item x
    Else
      Increment order of item x
      β = δ ∪x
      α = δ
    End// If
```

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

23 $\delta = \beta$
24 End //If
25 End //while
26 End //Begin
Procedure ExtFreqItemset($FS^*_m(\delta)$, Ext^*_k)
1 Begin // Ext^*_k is given in definition 9
2 Generate all subsets of $FS^*_m(\delta)$ and save to Ext^*_k
3 End // Begin

Example 12. This example is given to demonstrate how the proposed IIS-Mine algorithm can be used to mine all frequent itemsets. Assume *minsup* = 3 for all of the definitions above, and for Examples 1-3, Figure 2 and Algorithm 3. For simplicity, this example is divided into five main steps ordered by five frequent items in the IIS. The proposed mining algorithm proceeds as follows.

Let step 1 be the first main step. According to Procedure AllFreqItemset, the frequent item *a* is the first frequent item in the IIS that is mined. The *IIS$_a$Tree* is constructed, as illustrated in Figure 2(a). The algorithm calls Procedure IIS-Mine, so Procedure Subheader is called in order. The SH(*a*) is (*c*,*f*,*m*), where *supp*(*c*) = 3, *supp*(*f*) = 3 and *supp*(*m*) = 3. After line 3 in the procedure, IIS-Mine generates all of the frequent itemsets with its support, i.e. *ac*, *af* and *am*; all of these frequent itemsets have support equal to three. Let steps 1.1, 1.2 and 1.3 represent each of the frequent itemsets.

The step 1.1, the first frequent itemset is *ac*, which has support equal to three. The algorithm checks |*ac*|>1 and then calls Procedure SkipFreqItem. At this procedure, *ac* is not in FI^*_2, so the algorithm rolls back to line 9 of Procedure IIS-Mine. The frequent itemset *ac* with its support is defined to be the root; then, the conditional *ac*-tree, which has root "<*ac*:3>", is constructed using the input *IIS$_a$Tree*. The conditional *ac*-tree is illustrated in Figure 4. The algorithm is iterated by calling Procedure IIS-Mine again because the conditional *ac*-tree contains more than two nodes; let this call be step 1.1.1.

In step 1.1.1, the Procedure IIS-Mine is called; SH(*ac*) = (*f*,*m*), where *supp*(*f*) and *supp*(*m*) are then equal to 3. At line 3, the algorithm generates frequent itemsets, which are *acf* and *acm*, where *supp*(*acf*) and *supp*(*acm*) are then equal to 3. The first frequent itemset in this step is *acf* and |*acf*|>1, so the procedure SkipFreqItem in line 7 is processed, and it finds that *acf* is not in FI^*_3. Next, the algorithm rolls back to line 9 to construct a conditional *acf*-tree that has a conditional *ac*-tree as an input tree, which is illustrated in Figure 5. The condition of line 13 is that the conditional *acf*-tree contains only two nodes, so the algorithm obtains $FS^*_4(acfm) = acfm$. At line 18, the size of $FS^*_4(acfm)$ is greater than three and $FS^*_4(acfm) \notin FI^*_4$. Thus, at line 19, Procedure ExtFreqItemset is called to find all of the subsets of $FS^*_4(acfm)$, which are $Ext^*_2(acfm)$ and $Ext^*_3(acfm)$: $Ext^*_2(acfm)$ = {*ac*, *af*, *am*, *cf*, *cm*, *fm*} and $Ext^*_3(acfm)$ = {*acf*, *acm*, *afm*, *cfm*}; hence, FI^*_2 = {*ac*, *af*, *am*, *cf*, *cm*, *fm*}, FI^*_3 = {*acf*, *acm*, *afm*, *cfm*}, and FI^*_4 = {*acfm*}.

In step 1.1.2, the next frequent itemset generated together with step 1.1.1 is *acm*. At line 7 of Procedure IIS-Mine, the algorithm calls Procedure SkipFreqItem and obtains *acm*, which is already a

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

member of FI_3^*; *m* is the last frequent item in SH(*ac*), so we exit from this step without the construction of a conditional *acm*-tree.

In step 1.2, at line 4 of Procedure IIS-Mine, the next frequent itemset is *af*, and at line 7, Procedure SkipFreqItem is called and obtains *af*, which is contained in FI_2^*. In the loop in line 20 of Procedure SkipFreqItem, after *af* is combined with the next frequent item in SH(*a*), which is *m*, *afm* is obtained, which is contained in FI_3^*, and item *m* is the last item in SH(*a*). The algorithm rolls back to line 8 of Procedure IIS-Mine, so the conditional *af*-tree is not constructed.

In step 1.3, at line 4 of Procedure IIS-Mine, the next frequent itemset is *am*. At line 7, Procedure SkipFreqItem is called and obtains *am*, which is contained in FI_2^*, and there is no frequent item in SH(*a*). Therefore, the conditional *am*-tree is not constructed.

Let step 2 be the second main step. According to line 2 of Procedure AllFreqItemset, *b* is the second frequent item in the IIS that is mined. After line 4, the *IIS$_b$Tree* is constructed, as illustrated in Figure 2(b). The algorithm calls Procedure IIS-Mine at line 6. At line 2 of Procedure IIS-Mine, Procedure Subheader is called and obtains an empty SH(*b*), so the size of the frequent itemset generated with *b* is one. The processing of this step is terminated, and we return to line 2 of Procedure AllFreqItemset.

Let the third main step be step 3. According to line 2 of Procedure AllFreqItemset, *c* is the third frequent item in IIS that is mined. Line 4 is called to construct the *IIS$_c$Tree*, as illustrated in Figure 2(c). At line 6, the algorithm calls Procedure IIS-Mine to mine frequent itemsets. At line 2 of Procedure IIS-Mine, Procedure Subheader is called to obtain the SH(*c*) that is (*f*,*m*,*p*). Next, at line 3, the algorithm generates frequent itemsets, which are *cf*, *cm* and *cp*, where *supp*(*cf*), *supp*(*cm*) and *supp*(*cp*) = 3. Let steps 3.1, 3.2 and 3.3 represent each of the frequent itemsets.

In step 3.1, at line 4 of Procedure IIS-Mine, the first frequent itemset is *cf*, where $|cf| > 1$; line 7 then calls Procedure SkipFreqItemset. At Procedure SkipFreqItemset, *cf* combines with the next frequent item in SH(*c*), which is *m*, so *cfm* with a support of 3 is obtained after processing lines 19-23. Next, line 8 is checked, and as *cfm* is already obtained in FI_3^*, lines 19-23 are checked again, and *cfmp* is obtained. The frequent itemset *cfmp* is not a member in FI_4^*, and *p* is the last frequent item in SH(*c*), so *cfm* is set to be a root for a conditional *cfm*-tree. The algorithm goes back to line 8 of Procedure IIS-Mine to construct a conditional *cfm*-tree, where the *IIS$_c$Tree* is an input tree, which is illustrated in Figure 6. *Supp*(*p*) is less than *minsup*, hence $FS_3^*(cfm) = cfm$. Procedure ExtFreqItemset in line 18 is not called because $FS_3^*(cfm)$ is contained in FI_3^*. In this step, the algorithm skips the construction of the conditional *cf*-tree.

In step 3.2, according to line 4 of Procedure IIS-Mine, the next frequent itemset is *cm*, and Procedure SkipFreqItemset in line 7 is called. Line 8 of Procedure SkipFreqItemset checks that *cm* is a member in FI_2^*. Next, lines 19-23 are checked, so *cm* combines with the next item in SH(*c*), which is *p*, and we obtain *cmp* with a support of 3. The frequent itemset *cmp* is not in FI_3^*, and *p* is the last frequent item in SH(*c*), so *cm* is set to be a root for a conditional *cm*-tree. The algorithm goes back to line 8 of Procedure IIS-Mine to construct a conditional *cm*-tree, where the *IIS$_c$Tree* is an input tree, which is illustrated in Figure 7. *Supp*(*p*) is less than *minsup*, hence $FS_2^*(cm) = cm$ and $FS_2^*(cm)$ are already members in FI_2^*.

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

Figure 6. Conditional *cfm*-tree

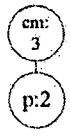

Figure 7. Conditional *cm*-tree

In step 3.3, according to line 4 of Procedure IIS-Mine, the next frequent itemset is *cp*. Next, at line 5, Procedure SkipFreqItemset is called, and *cp* is not a member in FI_2^*. There is no frequent item in SH(*c*), so this procedure is terminated, and we return to line 8 of Procedure IIS-Mine. After lines 22-23 are checked, the new answer is contained in FI_2^*, which is *cp*. This step skips the construction of a conditional *cp*-tree.

The remaining steps such as the fourth main step, which constructs the *IIS$_f$Tree* and is illustrated in Figure 2(d), and the fifth main step, which constructs the *IIS$_m$Tree* and is illustrated in Figure 2(e), are performed in the same way in sequence.

The complete frequent itemsets are shown by item in Table 2 and by length in Table 3.

Table 2. Complete frequent itemsets by item

Item	Frequent itemset
a	*a, acfm, ac, af, am, cf, cm, fm, acf, acm, amf, cfm*
b	*b*
c	*c, cp*
f	*f*
m	*m*
p	*p*

Table 3. Complete frequent itemsets by length

m-Length	Frequent itemset
1	*a, b, c, f, m, p*
2	*ac, af, am, cf, cm, cp, fm*
3	*acf, acm, afm, cfm*
4	*acfm*

The advantages of our algorithm are as follows. First, according to step 1.1.1 of Example 12, the frequent itemsets *acfm* are obtained, which are derived from a conditional *acf*-tree. This step shows the properties of an extendable frequent itemset*, which are given in Definition 9 and Procedure ExtFreqItemset in Algorithm 3. This step then finds all of the subsets of *acfm*, so we derive ten frequent itemsets, which are *ac*, *af*, *am*, *cf*, *cm*, *fm*, *acf*, *acm*, *amf* and *cfm*, without contributing more trees or using recursion to mine. Therefore, our algorithm can reduce many

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

Professional Version 2002 Service Pack 3 operating system on a personal computer with 1 GB of main memory and Pentium (R) CPU 3.00 GHz. All algorithms were coded using C language. Two groups of benchmark datasets, i.e. two synthetic datasets and two real datasets, were used.

For the first group of datasets, we also presented the experimental results for two synthetic datasets generated by the IBM Almaden Quest research group [24-25]. The datasets serve as the FIMI repository, which is a result of the workshops on frequent itemset mining implementations [26, 27]. The two original databases of synthetic datasets are T10I4D100K and T40I10D100K, which are sparse datasets. The notation *TxIyDzK* denotes a dataset where *K* is 1,000 transactions. Table 4 lists the parameters of the synthetic datasets, which vary in the number of transactions, i.e. 20%, 40%, 60% and 80% of the original database.

Table 4. Parameters of the synthetic datasets

$\|T\|$	Average number of items per transaction
$\|I\|$	Average length of a frequent itemset
$\|D\|$	Number of transactions

For the second group of datasets, the real datasets from the UCI machine learning repository [28] were used to test the proposed method. The real datasets used in the experiment were Chess [29] and Mushroom [30], which are dense datasets with a great number of long frequent itemsets. The characteristics of the real datasets are shown in Table 5.

Table 5. Characteristics of real datasets

Real dataset	Description
Chess	Average number of items per transaction = 37, number of transactions = 3,196, and number of items = 75
Mushroom	Average number of items per transaction = 23, number of transactions = 8,124, and number of items = 119

Run Time

Figures 8(a) and 8(b) show the performance of the algorithms on two synthetic datasets, T10I4D100K and T40I10D100K respectively. In Figure 8(a), IIS-Mine performs better than FP-growth in every support threshold. The gap in the graph becomes larger as the support threshold decreases. In Figure 8(b), when the minimum support is set at 20%, 17.5%, 15% or 12.5%, the run time between the two algorithms is not very different. However, when the minimum support is set at 10%, 7.5% or 5%, the run time of FP-growth increases significantly when compared to that of IIS-Mine, which confirms that IIS-Mine performs better than FP-growth. The results shown in Figures 8(a) and 8(b) can be explained as follows. With sparse datasets, when the minimum support is high, the number of frequent itemsets is low. However, when the minimum support is low, many frequent itemsets are obtained. IIS-Mine is always faster than FP-growth method, especially when the

minimum support is low, because FP-growth constructs bushy and wide FP-trees when the minimum support is low. So FP-growth is computationally expensive for tree traversing the FP-trees. IIS-Mine has the step of finding the root node from previous frequent itemsets, which can reduce the number of nodes and levels of the conditional itemset-tree. Therefore, traversing in the conditional itemset-tree is on a reduced tree, which can result in a low run time consumption. However, the run time of both algorithms relies on the length of the transaction (as observed in a comparison of the graphs in Figures 8(a) and 8(b) with a minimum support of 5%); when the transaction is long, so it the run time of both algorithms.

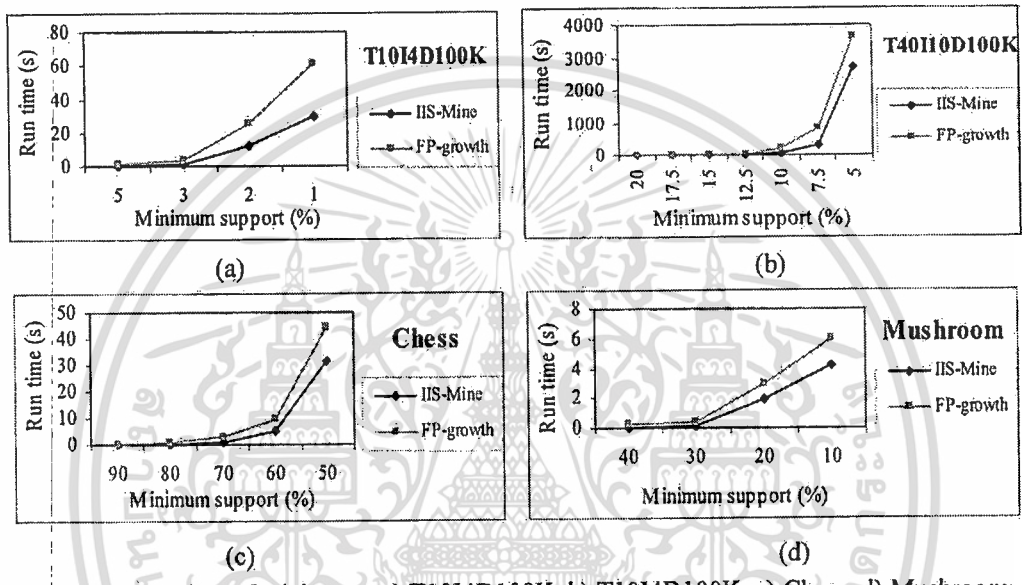

Figure 8. Run time of mining on: a) T10I4D100K; b) T10I4D100K; c) Chess; d) Mushroom

Figures 8(c) and 8(d) show the performance of algorithms on two dense datasets: chess and mushroom. Figure 8(c) shows that the run time of IIS-Mine is better than that of FP-growth in every support threshold. The run time of both algorithms increases when the minimum support threshold is reduced to 50%. In Figure 8(d), IIS-Mine again performs better than FP-growth in every minimum support. The run time of FP-growth increases significantly compared with IIS-Mine when the minimum support is less than 30%. The results shown in Figures 8(c) and 8(d) can be explained as follows. In the two Figures, IIS-Mine is faster than FP-growth for dense datasets. The main work in FP-growth is traversing FP-trees and constructing new conditional FP-trees after the first FP-tree is constructed from the original database. For dense datasets, we have found from numerous experiments that the time spent on traversing FP-trees is very long. IIS-Mine improves this problem using the property of extendable itemsets to reduce the number of recursive mining steps so that the size and number of constructing and traversing the trees are reduced. The run time of IIS-Mine is then less than that of FP-growth.

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

Memory Consumption

Figures 9(a) and 9(b) show the memory consumption of the algorithms on the synthetic datasets. In Figure 9(a), FP-growth consumes more memory than IIS-Mine. The graphs clearly separate out when the minimum support is less than 3%. In Figure 9(b), there is no difference in memory consumption until the minimum support is less than 15%. Thus, we can see that IIS-Mine consumes less memory than FP-growth on synthetic datasets. The large memory consumption of FP-growth when running on synthetic datasets can be explained by the fairly low minimum support and the presence of many single items in the datasets; therefore, FP-growth constructs wide and bushy trees to mine all frequent itemsets. However, IIS-Mine uses the property of extendable itemsets, which reduces the construction of conditional itemset-trees, and uses the step of finding the root node from previous frequent itemsets. Therefore, the node construction and tree sizes are reduced, resulting in a reduction in memory consumption.

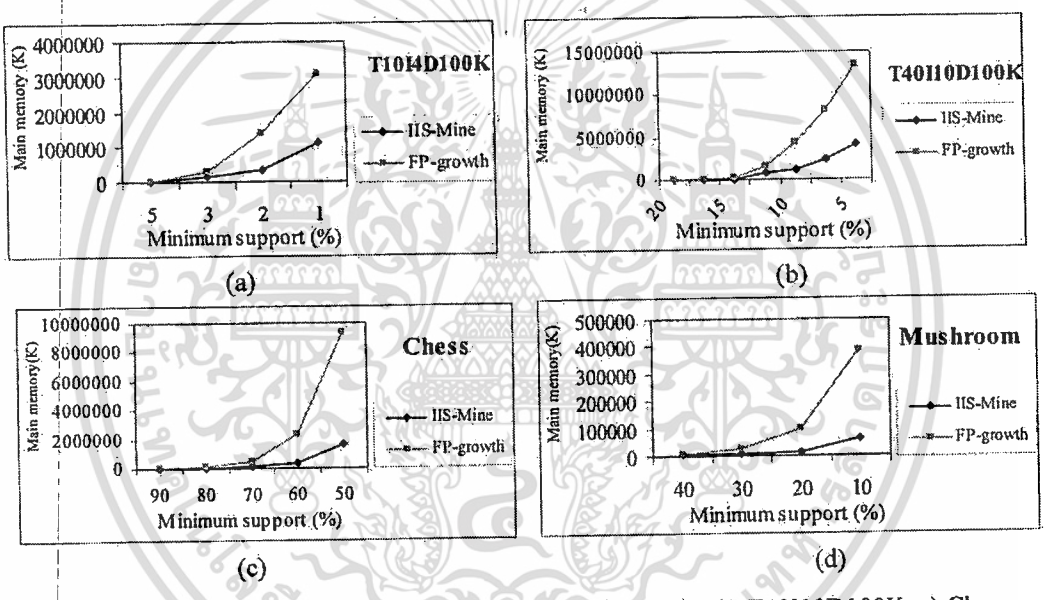

Figure 9. Memory consumption of mining on: a) T10I4D100K; b) T40I10D100K; c) Chess; d) Mushroom

Figures 9(c) and 9(d) show that the memory consumption of IIS-Mine is less than that of FP-growth on dense datasets. Figure 9(c) shows that when the minimum support is less than 80%, the memory consumption of FP-growth increases significantly compared with that of IIS-Mine. Figure 9(d) also shows that when the minimum support is less than 30%, the gap of the graphs clearly widens, which confirms that FP-growth consumes more memory than IIS-Mine. In both figures, FP-growth consumes a great deal more memory when the minimum support is low because FP-growth has constructed large FP-trees for mining all frequent itemsets, whereas IIS-Mine uses the property of extendable itemsets, which perform better for dense datasets. Consequently, the recursion of

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

mining frequent itemsets in the next loops is reduced. Therefore, the construction of nodes and the sizes of the conditional item-trees are reduced.

Scalability

The scalability of the algorithms was tested by running them on datasets generated from T10I4 and T40I10. The number of transactions in the datasets ranged from 20*K* to 100*K*, where *K* is 1000 transactions. In Figure 10(a) and Figure 11(a), the algorithms were run on all of the datasets generated from T10I4 at a minimum support of 1%. Both run time and memory consumption were recorded. Figure 10(a) shows the speed scalability, which means that the number of transactions increases as the run time increases. Figure 11(a) shows the memory scalability of the algorithms; the curve of FP-growth is over that of IIS-Mine, which means that FP-growth consumes more memory than IIS-Mine. The figure also shows that the memory consumption of the algorithms increases linearly with the size of the datasets.

In Figures 10(b) and 11(b), the algorithms were run on all of the datasets generated from T40I10 at a minimum support of 5%. Both run time and memory consumption were recorded. Figure 10(b) confirms that the run time of both algorithms relies on the length and number of transactions: if the length or number of transactions increases, so does the run time of both algorithms. However, the run time of IIS-Mine was better than that of FP-growth for every number of transactions. Figure 11(b) confirms that the memory consumption of the algorithms increases with the length and number of transactions. However, the memory consumption of IIS-Mine was better than that of FP-growth for every number of transactions.

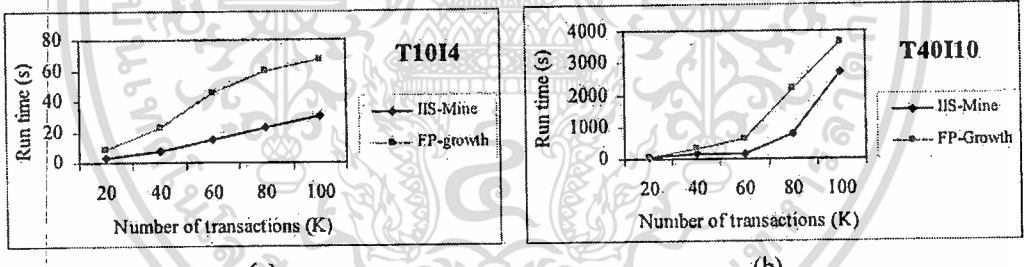

Figure 10. Scalability of runtime on a) T10I4; b) T40I10

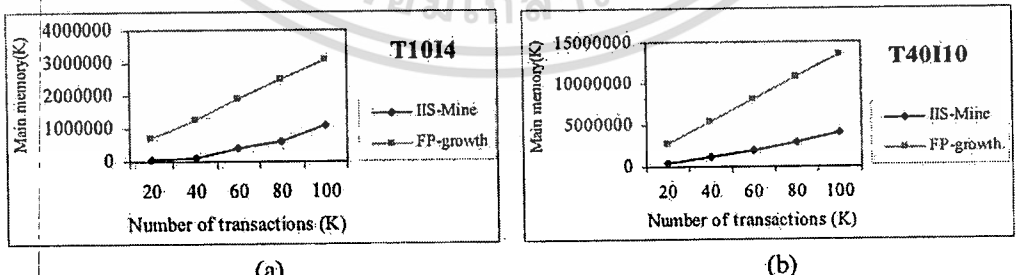

Figure 11. Scalability of memory consumption on: a) T10I4; b) T40I10

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

CONCLUSIONS

A data structure called inverted index structure (IIS) can store transaction data by scanning a database only once. Changing the minimum support does not affect the IIS and rescanning of the database is not required. A new algorithm called IIS-Mine can find frequent itemsets without generating candidate itemsets. It employs a more efficient use of the extendable-itemset property to reduce the number of recursive steps of mining. The node construction and the size of trees are then reduced, thereby reducing the run time and memory consumption. Although the proposed method accesses the IIS structure multiple times, experimental results demonstrated that for dense datasets the IIS-Mine algorithm is better than FP-growth algorithm in run time and space consumption.

ACKNOWLEDGEMENTS

This work was supported by the National Centre of Excellence in Mathematics, PERDO, Office of the Higher Education Commission, Thailand.

REFERENCES

1. J. Han and M. Kamber, "Data Mining: Concepts and Techniques", Elsevier, Maryland Heights (MO), **2006**, pp.227-231.
2. J. Han, J. Pei, Y. Yin and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach", *Data Mining Knowl. Discov.*, **2004**, *8*, 53-87.
3. G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using FP-Trees", *IEEE Trans. Knowl. Data Eng.*, **2005**, *17*, 1347-1362.
4. R. Agrawal, T. Imielinski and A. Swami, "Mining association rules between sets of items in large databases", Proceedings of ACM SIGMOD International Conference on Management of Data, **1993**, Washington, DC, USA, pp.207-216.
5. R. Agrawal and R. Srikant, "Fast algorithms for mining association rules", Proceedings of 20th International Conference on Very Large Data Bases, **1994**, Santiago de Chile, Chile, pp.487-499.
6. T. H. N. Vu, J. W. Lee and K. H. Ryu, "Spatiotemporal pattern mining technique for location-based service system", *ETRI J.*, **2008**, *30*, 421-431.
7. J. Han, J. Pei and Y. Yin, "Mining frequent patterns without candidate generation", Proceedings of ACM SIGMOD International Conference on Management of Data, **2000**, Dallas (TX), USA, pp.1-12.
8. J. Pei, J. Han, H. Lu, S. Nishio, S. Tang and D. Yang, "H-mine: Hyper-structure mining of frequent patterns in large databases", Proceedings of IEEE International Conference on Data Mining, **2001**, San Jose (CA), USA, pp.441-448.
9. A. Pietracaprina and D. Zandolin, "Mining frequent itemsets using Patricia tries", Proceedings of 3rd IEEE International Conference on Data Mining, **2003**, Melbourne (FL), USA.
10. G. Grahne and J. Zhu, "Efficiently using Prefix-Trees in mining frequent itemsets", Proceedings of 3rd IEEE International Conference on Data Mining, **2003**, Melbourne (FL), USA.

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

11. Q. Zhu and X. Lin, "Depth first generation of frequent patterns without candidate generation", Proceedings of 11th Pacific-Asia Conference on Knowledge Discovery and Data Mining, **2007**, Nanjing, China, pp.378-388.
12. S. Sahaphong and V. Boonjing, "The combination approach to frequent itemsets mining", Proceedings of 3rd International Conference on Convergence and Hybrid Information Technology, **2008**, Busan, Korea, pp.565-570.
13. V. K. Shrivastava, P. Kumar and K. R. Padasani, "FP-tree and COFI based approach for mining of multiple level association rules in large database", *Int. J. Comp. Sci. Inf. Secur.*, **2010**, *7*, 273-279.
14. L. Huang, J. Z. Liang, Y. Pan and Y. Xian, "A complete attribute reduction algorithm based on improved FP tree", Proceedings of International Conference on Circuits, Communications and System, **2010**, Beijing, China, pp.1-4.
15. U. Yun and K. H. Ryu, "Approximate weighted frequent pattern mining with/without noisy environments", *Knowl.-Based Syst.*, **2011**, *24*, 73-82.
16. M. J. Zaki, "Scalable algorithms for association mining", *IEEE Trans. Knowl. Data Eng.*, **2000**, *12*, 372-390.
17. M. J. Zaki and K. Gouda, "Fast vertical mining using diffsets", Proceedings of 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, **2003**, Washington, DC, USA, pp.236-355.
18. D. J. Chai, L. Jin, B. Hwang and K. H. Ryu, "Frequent pattern mining using bipartite graph", Proceedings of 18th International Conference on Database and Expert Systems Applications, **2007**, Regensburg, Germany, pp.182-186.
19. J. Dong and M. Han, "BitTableFI: An efficient mining frequent itemsets algorithm", *Knowl.-Based Syst.*, **2007**, *20*, 329-335.
20. W. Yen, "A new mining algorithm based on frequent item sets", Proceedings of International Workshop on Knowledge Discovery and Data Mining, **2008**, Adelaide, Australia, pp.410-413.
21. W. Song, B. Yang and Z. Xu, "Index-BitTableFI: An improved algorithm for mining frequent itemsets", *Knowl.-Based Syst.*, **2008**, *21*, 507-513.
22. S. Sahaphong and V. Boonjing, "Mining of frequent itemsets by using the property of extendable-itemset", Proceedings of 7th International Joint Conference on Computer Science and Software Engineering, **2010**, Bangkok, Thailand, pp.168-173.
23. S. Sahaphong and G. Sritanratana, "Mining of frequent itemsets with JoinFI-Mine algorithm", Proceedings of 10th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Database, **2011**, Cambridge, United Kingdom, pp.73-78.
24. Frequent Itemset Mining Dataset Repository, "T10I4D100K", http://fimi.cs.helsinki.fi/data/, **2003** (Accessed: January 11, 2010).
25. Frequent Itemset Mining Dataset Repository, "T40I10D100K", http://fimi.cs.helsinki.fi/data/, **2003** (Accessed: January 11, 2010).
26. Workshop on Frequent Itemset Mining Implementations, http://fimi.ua.ac.be/fimi03/, **2003** (Accessed: January 2, 2010).

Maejo Int. J. Sci. Technol. **2012**, *6*(01), 130-151

27. Workshop on Frequent Itemset Mining Implementations, http://fimi.ua.ac.be/fimi04/, **2004** (Accessed: January 2, 2010).
28. UCI Machine Learning Repository, http://archive.ics.uci.edu/ml/, **2007** (Accessed: March 15, 2010).
29. UCI Machine Learning Repository, "Chess", http://archive.ics.uci.edu/ml/datasets.html, **1989** (Accessed: July 19, 2010).
30. UCI Machine Learning Repository, "Mushroom", http://archive.ics.uci.edu/ml/datasets.html, **1987** (Accessed: July 19, 2010).

Mining of Frequent Itemsets Using Extendable-Itemset Property

Supatra Sahaphong[1,2] and Veera Boonjing[1]
[1]Department of Mathematics and Computer Science, Faculty of Science,
King Mongkut's Institute of Technology Ladkrabang, Bangkok, 10520, Thailand.
[2]Department of Computer Science, Faculty of Science,
Ramkhamhaeng University, Bangkok, 10240, Thailand. supatra@ru.ac.th

Abstract

In this paper, we propose a new approach to mine all frequent itemsets from a transaction database. The main features of this paper are: (1) the proposed data structure is performing only one time database scanning (2) the changing of the minimal support threshold is not affected by this structure and rescanning of database is not required, and (3) reduction of the subsequence steps to mine frequent itemsets by using an efficient property of extendable-itemset which can quickly find frequent itemsets. We have conducted the examples of all definitions, measured the performance and simulated experiment by comparison with the FP-growth algorithm. The results demonstrate that our method can mine frequent itemsets faster than the FP-growth algorithm.

Key Words: association rule mining, data mining, frequent itemset mining, frequent pattern mining

1. Introduction

Frequent pattern mining is the essential role for discovering the interesting associations and relations in various kinds of database, including transaction database, time-series database and etc. The frequent pattern mining problem was first introduced by Agrawal et al. in [1]. The algorithm was then improved by Agrawal et al., called the Apriori algorithm which is proposed in [2]. The Apriori algorithm uses a level-wise and breadth-first search approach for generating association rule. The Apriori algorithm [3] based on generated and tested candidate itemsets have problems of increasing I/O load and time consuming. The problems in the above mentioned are improved. The FP-growth algorithm is the most popular of these improved algorithms which is developed by Han J. et al in [4], [5]. The FP-growth algorithm mines frequent itemsets without generating candidate sets and scans database only twice. The FP-growth performs depth-first search approach in the search space. It encodes the data set using a compact data structure called FP-tree and extracts frequent pattern directly from this prefix tree. There are many approaches that have been proposed to extend and improve this algorithm [6]-[9]. Most of the FP-tree algorithm base may restart and scan database twice when the changing of minimal support. Many researchers have proposed ways to scan database once. Eclat algorithm was proposed in [10] to generate frequent patterns in a depth-first search and used the join step from the Apriori property, since no candidate itemsets can be found. In [11], they proposed a data structure, called LIB-graph to contain data when database is scanned. However, these algorithms have some drawbacks such as: (1) the algorithms need to construct their data structure when the minimal support is changed so when the user would like to test new a support threshold, the algorithms must restart again, and (2) all of the FP-tree based algorithms have problem of many recursive sub-trees.

From the above mentioned, our motivations are to find new way to improve the running time of mining steps such as: (1) the changing of the minimal support threshold is not affected with the data structure and rescanning of database is not required, and (2) the reduction of the recursive steps of mining of frequent itemsets.

In this paper, we propose an efficient data structure and major steps to mine frequent itemsets in transaction database. The data structure performs database scanning only once. The new step to mine frequent itemsets which reduces subsequence of mining by using an efficient property, called *extendable-itemset*, so that we can quickly find frequent itemset faster than the classic algorithm, named the *FP-growth* algorithm. This paper is organized as follows. The basic concept is presented in section 2, followed by the approach which is

presented in section 3, the experimental result is shown in section 4 and the finally, the conclusion is addressed in section 5.

2. Basic Concept

This section introduces basic concepts for mining of frequent itemset. All definitions in this section are proposed by J. Han et al in [5][12], as follows.

Let $I=\{x_1,x_2,...,x_m\}$ be a set of items and a *transaction database* $DB=\{T_1,T_2,...,T_n\}$, where $T_i(i\in[1..n])$ is a transaction which contains items in I. The *support or supp* (or occurrence frequency) of a pattern A, where A is a set of items, is the number of transactions containing A in DB. A pattern A is *frequent* if A's support is no less than a predefined minimum support threshold, *minsup*. An item x is called a *frequent item* if $supp(x)\geq minsup$, otherwise it is called *infrequent item*.

Given a transaction database DB and a minimum support threshold *minsup*, the problem of *finding the complete set of frequent itemsets* is called the *frequent itemset mining problem* and any element of the complete set is called a *frequent itemset*. For greater understanding, we provide an example to describe the above definitions.

Example 1. We use an example of the database which was used in [5]. Here Table 1 is a *DB*. It consists of five transactions T_1,T_2,T_3,T_4,and T_5, labeled as *Transactions* in the table, and seventeen items *a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p* and *s*, labeled as *Items* in the table. For example, the first transaction is T_1 containing *f, a, c, d ,g, i, m,* and *p*.

Table 1. A transaction database

Transactions	Items
T_1	f,a,c,d,g,i,m,p
T_2	a,b,c,f,l,m,o
T_3	b,f,h,j,o
T_4	b,c,k,s,p
T_5	a,f,c,e,l,p,m,n

3. The Approach

3.1 The Proposed Data Structure

In this subsection, we present a data structure that contains transaction data, called *Inverted Index Structure* or *IIS*. The *IIS* is a form of structure which has a relationship between items and the transactions included within. The *IIS* is constructed from a scan of *DB* once, and the original *IIS* can support every *minsup* threshold. Therefore it does not need to rescan the *DB* when *minsup* threshold is changed. According to definitions in the previous section, we give a new definition, an example and construction steps to present how to make the *IIS*.

Definition 1 (Inverted Index Structure)
Let $T_i=\{x_1,x_2,x_3,...,x_m\}$ be a transaction in *DB*, where $i=1,2,...,m$ *and* x_j is an item for $j=1,2,...,m$. An *inverted index structure* (or *IIS* in short) is the structure constructed from a scan of each T_i in *DB* only once. The *IIS* consists of the attribute of items in *I* and the set of transactions in *DB*, where *I* is the set of all items in *DB*. Each row in *IIS* contains an item in *I* and transactions in *DB* which contain such an item. The set of transaction will be written in order according to the assending of its identification number.

With the above definition, we present the steps to construct the *IIS* by using the following example.
Example 2. We use an example transaction database *DB* in Table 1. The *DB* is scanned once to create the *IIS*. The scan of the first transaction is T_1 which consists of items *f, a, c, d, g, i, m* and *p*. The transaction T_1 will be inserted into a corresponding item named ordering by assending sort of items (*a, c, d, f, g, i, m, p*). The T_1 will be the first transaction inserted at the transaction of item <*a*>. The second examined item is <*c*>, we insert T_1 at item <*c*>. Next, we examine item <*d*>, we then subsequently insert T_1 at item <*d*>. The remaining items (*f, g, i, m* and *p*) in T_1 can be done in the same way. The remaining transactions (T_2,T_3,T_4, and T_5) in *DB* can also be done in the same way. In figure 1, we present only rows of frequent items (*a, b, c, f, m, p*) and omit the infrequent items (*d, e, g, h, i, j, k, l, n, o*).

Items	Transactions			
a	T_1	T_2	T_5	
b	T_2	T_3	T_4	
c	T_1	T_2	T_4	T_5
...	...			
f	T_1	T_2	T_3	T_5
...	...			
m	T_1	T_2	T_5	
...	...			
p	T_1	T_4	T_5	
...	...			

Figure 1.Items of the *IIS* after scanning *DB* once

The construction of the *IIS* has five major steps, presented as figure 2.

Step1.	Construct the header table of a set of items (*I*).
Step2.	Check all transactions (*T*) in database (*DB*). If there is any *T* in *DB*, then step 3 and step 4 are repeated until there is no *T* in *DB*, otherwise perform step 5.
Step3.	Read *T* from *DB* and sort items in *T*.
Step4.	Link *T* to each corresponding item in the header table.
Step5.	End step.

Figure 2. The construction steps of the *IIS*

Based on the above definition and figure 2, the *IIS* structure is constructed after a scan of *DB* only once and the frequent items are found. The *IIS* represents the relationship between each item in *I* and transactions *T* so that this structure can support all of *minsup*.

3.2 The Proposed Mining Steps

In this subsection, we present new steps that use a new tree structure, called the $IIS_{item}Tree$, to mine frequent itemsets. The main features of this proposed method are: (1) the frequent itemset is found without generation of candidate itemsets, (2) the subsequence of the mining steps are reduced by using the property of *extendable-itemset*, and (3) the proposed method supports mining of frequent itemsets with any *minsup* threshold without rescanning of database. From the above features, we can quickly find the frequent itemsets and obtain completely and correctly frequent itemsets. We give the definitions, the examples and the mining steps for presentation of how to mine frequent itemsets.

Definition 2 (IIS_{item} *Tree*)
An IIS_{item} *Tree* is a tree structure which is constructed from frequent items in the *IIS*. It is a finite set of one or more nodes with the following structure.
(1) It consists of a root and a set of the children of the root.
(2) Each node in this subtree consists of five fields: *item-name*, *support*, *same-item*, *parent* and *child* where *item-name* registers which item this node represents, *support* registers the number of transactions represented by the portion of the path reaching this node, *same-item* represents a pointer which points to the node in the IIS_{item} *Tree* carrying the same item-name, *parent* represents a pointer which points to previous node in the same path, and *child* represents a pointer which points to the child node.
(3) The *header table* consists of two fields: *item-name* and *head of node link* where *head of node link* represents a pointer which points to the first node in an IIS_{item} *Tree* carrying the *item-name*.

Example 3. In this example, we describe the steps of construction of all the $IIS_{item}Trees$ except the last frequent item <*p*> by using the *IIS* in figure 1 and *minsup*=3. The first frequent item in the *IIS* is <*a*>, so we first construct theIIS_aTree and item <*a*> is a root. We obtain two paths which are (<*a*:2>,<*c*:2>,<*f*:2>,<*m*:2>,<*p*:2>) and (<*a*:1>, <*b*:1>, <*c*:1>, <*f*:1>,<*m*:1>). The first path consists of frequent items (<*a*>,<*c*>,<*f*>,<*m*>,<*p*>) which appears twice in *DB*. Similarly, the second path indicates the frequent items (<*a*>,<*b*>,<*c*>,<*f*>,<*m*>) which are contained in only one transaction in *DB*. These two paths share the frequent item <*a*> together, so <*a*> appears three times in *DB*. After the mining of frequent itemsets process which will be explained in the next example, we construct the next IIS_{item} *Tree*. It is the IIS_bTree, the prior frequent item,<*a*>, is omitted. In figure 3 all of the IIS_{item} *Trees* except the last IIS_pTree are illustrated.

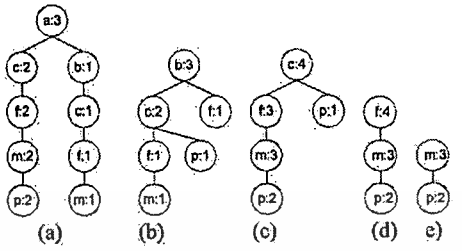

Figure 3. The IIS_{item} *Tree*

Base on the above definition, we have the main steps of IIS_{item} *Tree* construction.

Step1.	Construct the header table of frequent items (x)
Step2.	Check all x in the *IIS*. If there is any x in the *IIS*, then step 3 to step 6 are repeated until there are no x in the *IIS*, otherwise perform step 9.
Step3.	Read the frequent item in the *IIS*.
Step4.	Create root node (R) and initial support of R to 1.
Step5.	Link R to header table.
Step6.	Check all x which has transaction (T) similar R. If the next x is not the last x in the *IIS*, then perform step7 and step8, otherwise perform step9.
Step7.	Perform reading the next x (N) in *IIS* which similar T with R until meet last similar T of last x.
Step8.	Construct of the IIS_xTree. If tree has child node C which item-name of C is the same item-name of N then increment support of N by 1 and link N to header table.
Step9.	End step.

Figure 4. Construction steps of the IIS_{item} *Tree*

Definition 3 (Combine-itemset)
Let $b_1$be arbitrary frequent item, *minsup*>0 be a minimum support threshold, and let P:=(<b_1:s_1>,...,<b_k:s_k>) be a path of a conditional tree T, where <b_1:s_1>,...,<b_k:s_k> are all nodes of T and b_1 is the root of T. Then a *combine-itemset* of $b_1,...,b_k$ with respect to P is denoted by $ci_k(P)$ or $<b_1...b_k>_P$ and is defined by the set of $b_1,...,b_k$. The set of all combine-itemsets of $b_1,...,b_k$ with respect to P is denoted by $CI_k(P)$. If no confusion arises, $<b_1...b_k>_P$ is replaced by $<b_1...b_k>$.

Definition 4 (Extendable-itemset)
Let b_1, *minsup* and P be given as in Definition 3, where $k\geq 2$. An *extendable-itemset* of length m with

respect to P is a nonempty subset of the set $<b_1,...,b_k>$ which has at least two items. If an *extendable-itemset* of length m is $\{b_{i_1},...,b_{i_m}\}$ and $b_{i_1}<...<b_{i_m}$ where $<$ is the lexicographic order, then it is denoted by $<b_{i_1}...b_{i_m}>_p$. The set of all *extendable-itemsets* of length m is with respect to P is denoted by $EI_m(P)$.

Definition 5 (Frequent-itemset* of m-length)
Let b_i, *minsup*, and P be given in Definition 3. Then a *frequent-itemset** of length 1 is given by $FS_1 := \{\{x\}|x \in I, supp(x) \geq minsup\}$ and a *frequent-itemset** of length $m \geq 2$ with respect to P is given by $FS_m(P) := CI_m(P) \cup EI_m(P)$, wher $CI_m(P) = f_{\emptyset}^{ci_k(P);\, m=k}$ $;m<k$, and $EI_k(P) = \emptyset$. Any element $\{b_i\}$ of $FS_1(P)$ is denoted by $<b_i>$. The *frequent-itemset** of length m is given by $FS_m := \cup\{FS_m(P)|P$ is a path of condition tree of length at least $m\}$.

Definition 6 (Frequent-itemset*)
A *frequent-itemset** is given by $FI := \cup_{m=1}^{n} FS_m$, where n is the maximum depth of conditional tree for which the root is at level one.

Example 4. This example explains the details of processing steps to mine frequent itemset by examining the FS_1 item in the figure 1 in sequence, according to an example 3 and figure 3.

First, the *IIS$_a$ Tree* has two paths which are (<*a*:2>,<*c*:2>,<*f*:2>,<*m*:2>,<*p*:2>) and (<*a*:1>,<*b*:1>, <*c*:1>,<*f*:1>, <*m*:1>). We construct the conditional tree, as presented in figure 5(a). We discard the nodes which have support less than *minsup*, (<*b*:1> and <*p*:2>). We get a path P_1 = (<*a*:3>, <*c*:3>, <*f*:3>, <*m*:3>) which is no less than *minsup*, so we obtain $ci_4(P_1)$ is <*acfm*:3>. We then save <*afcm*> to $FS_4(P_1)$. Therefore, $FS_4(P_1) = CI_4(P_1) \cup EI_4(P_1) = \{<acfm>\} \cup \emptyset = \{<acfm>\}$. Then we will find frequent itemsets which is generated by <*acfm*> from the following steps.

The first step, we shall start finding all of *extendable-itemsets* of length 2 as follow. Since $EI_2(P_1) = \{ac,af,am,cf,cm,fm\}$ and $CI_2(P_1) = \emptyset$, $FS_2(P_1) = \{ac,af,am,cf,cm,fm\}$.

Next, we shall start finding all of *extendable-itemsets* of length 3 as follows. Since $EI_3(P_1) = \{acf,acm,afm,cfm\}$, we save $EI_3(P_1)$ to $FS_3(P_1) = CI_3(P_1) \cup EI_3(P_1) = \{acf,acm,afm,cfm\}$ because $CI_3(P_1) = \emptyset$.

We can obtain the frequent itemsets in $FS_2(P_1)$ and $FS_3(P_1)$ by using the property of *extendable-itemset*. We can skip the remaining recursive steps for construction of the conditional tree, so we reduce subsequent steps of mining. We can quickly find $EI_2(P_1)$ and $EI_3(P_1)$ from $ci_4(P_1)$. Therefore, the frequent itemsets are obtained from the *IIS$_a$ Tree* is {<*a*:3>,<*acfm*:3>,<*ac*:3>,<*af*:3>,<*am*:3>,<*cf*:3>, <*cm*:3>, <*fm*:3>, <*acf*:3>,<*acm*:3>,<*amf*:3>,<*cfm*:3>} in sequence.

Second, the *IIS$_b$ Tree* has multiple paths. We omit the prior FS_1(<*a*>) and construct the conditional tree, as seen in figure 5(b), we obtain only one frequent itemset, <*b*:3> because the supports of frequent items <*c*:2>, <*f*:2>, <*m*:1> and <*p*:1> are less than *minsup*.

Third, the *IIS$_c$ Tree* has two paths which are (<*c*:3>, <*f*:3>, <*m*:3>, <*p*:2>) and (<*c*:1>, <*p*:1>). Notice that we have omitted the prior FS_1 (<*a*> and <*b*>). We examine at the node <*p*> and construct the conditional tree (<*c*:3>, <*p*:3>) as shown in figure 5(c) on the left hand and get $ci_2(P_1)$ = <*cp*:3> so that we save <*cp*> to $FS_2(P_1)$. Next, we examine at the node <*m*> and construct the conditional tree (<*c*:3>, <*f*:3>, <*m*:3>) as shown in figure 5(c) on the right hand, we obtain $ci_3(P_2)$ = <*cfm*:3>, we have found that $ci_3(P_2)$ is saved in FS_3 so that we exit for this step. Therefore, the frequent itemsets that are obtained from the *IIS$_c$ Tree* is {<*c*:4>, <*cp*:4>}.

Fourth, the *IIS$_f$Tree* is a path which is (<*f*:4>, <*m*:3>, <*p*:2>). Notice that we have omitted the prior FS_1(<*a*>, <*b*> and <*c*>). We construct the conditional tree which has one path, (<*f*:3>, <*m*:3>), as shown in figure 5(d). We obtain $ci_2(P_1)$ = <*fm*:3>, we have found that $ci_2(P_1)$ is saved in FS_2 so that we exit for this step. The frequent itemset that is obtained from the *IIS$_f$Tree* is only {<*f*:4>}.

Fifth, the *IIS$_m$Tree* has one path which is (<*m*:3>, <*p*:2>). In figure 5(e), we get only <*m*:3> is no less than minsup, so <*m*> is the frequent itemset.

The last frequent item is <*p*>, so we do not construct the *IIS$_p$Tree* because it is the last frequent item. Therefore, <*p*> is the frequent itemset.

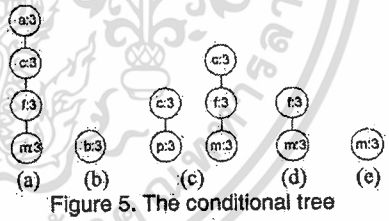

Figure 5. The conditional tree

Based on the above definitions and examples, we have the major steps to mine frequent itemset, seen as figure 6.

Step1.	Check all items (x) in IIS by sequential reading. If the support of x is greater than *minsup*, then perform next step, otherwise read the next x.
Step2.	Save x to frequent itemset which has length of frequent item is one (FS_1).
Step3.	Construction $IIS_{item}Tree$ of x using steps in figure 4
Step4.	If $IIS_{item}Tree$ of x is not an empty tree, then perform the next step, otherwise perform step1.
Step5.	Construction of the conditional tree (β) of root (x)
Step6.	Generate frequent itemset with *combine-itemset* of path ($ci(P)$) from β.
Step7.	If the size of $ci(P)$ is greater than 2, then find non empty subset of $ci(P)$ and save to *extendable-itemset* of path ($EI(P)$), otherwise move empty set to $EI(P)$ and perform next step.
Step8.	Save all $ci(P)$ and $EI(P)$ to frequent itemset of path ($FS(P)$) which separate by size of $FS(P)$.
Step9.	Union all size of $FS(P)$ to each length (m) of frequent itemset (FS_m).
Step10.	Union all FS_m to answer set (FI).
Step11.	End step.

Figure 6. The major steps to mine frequent itemsets

3.3 The Performance Analysis

In this section, we measure the performance of our method with the *FP-growth* by using the time complexity analysis. We consider several components such as the scanning of *DB*, the generation of the new data structure and the mining steps of frequent itemset. The complexity of two methods is compared in Table 2 and Table 3. Our method reduces two operations which are scanning of *DB* once for creation of FP-tree and doing operations without the sorting for FS_1. The performance analysis results demonstrate that our method is faster than the *FP-growth* in the number of times of operation. For time complexity analysis, we let T be the number of transaction, I be the average number of items in each transaction, and N be the number of FS_1.

The operation of the *FP-growth* algorithm consists of four main steps. (1) The scanning of the database once to find FS_1, (2) the construction of the header table from FS_1 by descending support sorted and infrequent itemsets are discarded, (3) the scanning of the database for the second times to construct the FP-tree, and (4) the exploring of the frequent pattern from FP-tree by using the FP-growth algorithm. The time complexity for the four operations is illustrated in Table 2.

The operation of our method is divided into three steps. (1) The scanning of the database to construct the IIS (2) the generation of the $IIS_{item}Tree$ by scanning frequent items in the IIS, and (3) the scanning of the $IIS_{item}Tree$ to mine frequent itemsets with our mining steps. The time complexity for the three operations is illustrated in Table 3.

Table 2. The time complexity of the FP-growth algorithm

Step	The operation	Time complexity
1	The scanning of *DB* to find FS_1	*DB* scanning: $O(T\times I)$
2	The pruning of $\overline{FS}$ and the sorting of FS_1	Pruning: $O(T\times I)$ Sorting: $O(T\times N\log N)$
3	The Generation of the FP-tree	*DB* scanning: $O(T\times I)$ Tree generation: $O(N\times I)$
4	The scanning of the FP-tree and mining with the *FP-Growth*	Tree scanning: $O(N\times logN)$ Pattern mining: $O(N^2)$

Table 3. The time complexity of the proposed method

Step	The operation	Time complexity
1	The scanning of *DB* to construct the IIS	DB scanning: $O(T\times I)$
2	The Generation of the $IIS_{item}Tree$	IIS scanning: $O(N\times T)$ $\overline{FS}$ pruning: $O(T\times I)$ Tree generation: $O(N\times I)$
3	The scanning of the $IIS_{item}Tree$ and the proposed mining steps	Tree scanning: $O(N\times logN)$ Mining steps: $O(N^2)$

4. The Experimental Result

This section presents performance measurement by the simulation experiment. The performance of our method is compared with the *FP-growth* algorithm under Microsoft Windows XP Professional Version 2002 Service Pack 3, with a personal computer consisting of 1 GB main memory, and Pentium (R) CPU 3.00 GHz. All algorithms are coded by using C language. The experiments show the runtime of both algorithms with difference datasets when minimum support threshold is varied. We use synthetic datasets which are generated by using the generator from the IBM Almaden Quest research group and available on [13]. The synthetic datasets are *T10I4D100K* and *T40I10D100K*. These two datasets are sparse datasets, as shown in Table 4.

Table 4. The datasets used in the experiments

Dataset	Description
T10I4D100K	The average item per transaction is 10, the average length of maximal frequent itemset is 4, and the number of transaction is 100,000.
T40I10D100K	The average item per transaction is 40, the average length of maximal frequent itemset is 10, the number of transaction is 100,000.

The graph in figure 7 shows that our method is better than the *FP-growth* algorithm in every support threshold. Clearly, the gap of the graph becomes larger as the support threshold goes down. In figure 8 confirms that when the length of transaction is long, the runtime of both algorithms are also long. This is because the runtime of both algorithms rely on the

length of transaction. We compare the graphs between figure 7 and figure 8 with minimum support is 5. The runtime of both algorithms increases when the minimal support decreases. However, the runtime of our method is still faster than the *FP-growth* algorithm every support threshold because it has the operation of *extendable-itemset* so that the number of recursive of mining are reduced.

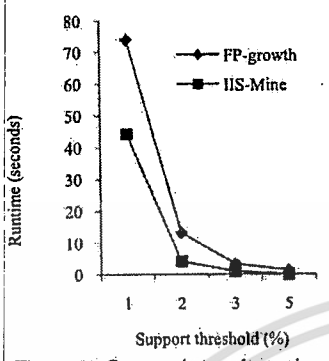

Figure 7. Comparison of running time (*T10I4D100K*)

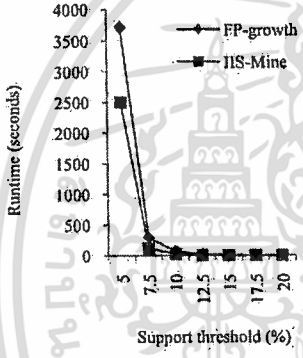

Figure 8. Comparison of running time (*T40I10D100K*)

5. Conclusion

We have presented a data structure that contains transaction data by scanning a database only one time, named *IIS*. The changing of the minimal support is not affecte *IIS* and rescanning of the database is not required. This new mining steps has the ability to find frequent itemsets without generation of candidate itemsets. It employs more efficient use property of *extendable-itemset* which reduces recursive steps of mining. We measure the performance of our method with the *FP-growth* algorithm by using the time complexity analysis and the experimental. The results demonstrate that our method is faster than the *FP-growth* algorithm in the number of times of operation. However, our method requires more space. In the future research, we will try to improve both space consumption and time consumption.

Acknowledgment

We appreciate Associate Professor Dr. Tawesak Tanwandee for prove reading.

References

[1] Agrawal, R., Imielinski, T., and Swami, A, "Mining Association Rule between Sets of Items in Large Database," SIGMOD'93. ACM Press, Washington, DC, United state, June 1993, pp. 207-216.

[2] Agrawal, R., and Srikant R, "Fast Algorithm for Mining Association Rules," VLDB 1994. Morgan Kaufmann, Chile, September 12-15, 1994, pp. 487-499.

[3] Ping-Ning Tan, Michael Steinbach and Vipin Kumar, Introduction to Data Mining. Pearson Education Inc., 2006.

[4] J. Han, J. Pei, and Y. Yin, "Mining Frequent Pattern Without Candidate Generation," ACM Press, Texas, May 14-19, 2000, pp. 1-12.

[5] J. Han, J. Pei, Y. Yin and R. Mao, "Mining Frequent Pattern Without Candidate Generation: A Frequent Pattern tree," Springer, Vol. 8, no. 1, 2004, pp. 53-87.

[6] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "Hmine: Hyper-Structure Mining of Frequent Patterns in Large Databases," ICDM 2001, IEEE, California, USA, 29 November - 2 December 2001, pp. 441-448.

[7] A. Pietracaprina and D.Zandolin, "Mining Frequent Itemsets Using Patricia Tries," FIMI 03, IEEE ,Florida, USA, November 19, 2003.

[8] G. Grahne and J. Zhu, "Efficiently Using Prefix Tree in Mining Frequent Itemsets," FIMI 03, IEEE, Florida,USA, November 19, 2003.

[9] S. Sahaphong and V. Boonjing, "The Combination Approach to Frequent Itemsets Mining," ICCIT 2008, IEEE, South Korea, November 11-13, 2008, pp. 565-570.

[10] M.J. Zaki, "Scalable Algorithms for Association Mining," IEEE Transaction on Knowledge and Data Engineering, IEEE, pp. 372-390.

[11] D. J. Chai, L. Jin, B. Hwang, and K. H. Ryu, "Frequent Pattern Mining Using Bipartite Graph," Regensburg, Germany, IEEE, September 3-7, 2007, pp. 182-186.

[12] J. Han and M. Kamber, Data Mining: Concepts and Techniques. second editions. Elsevier, 2006.

[13] http://fimi.cs.helsinki.fi/data/.

Third 2008 International Conference on Convergence and Hybrid Information Technology

A Combination Approach to Frequent Itemsets Mining

Supatra Sahaphong[1, 2] *Veera Boonjing*[1]
[1]*Department of Mathematics and Computer Science, Faculty of Science, King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand.*
[2]*Department of Computer Science, Faculty of Science, Ramkhamhaeng University, Bangkok, Thailand. supatra@ru.ac.th*

Abstract

In this paper, we propose a new mining of frequent itemsets algorithm, called SFI-mine algorithm. The SFI-mine constructs pattern-base by using a new method which is different from the conditional pattern-base in FP-growth, mines frequent itemsets with a new combination method without recursive construction of conditional FP-trees. It obtains complete and correct frequent itemsets. We have conducted the examples of all definitions and correctness proving.

1. Introduction

Frequent itemsets mining is useful for discovering the interesting associations hidden in large database. The uncovered associations can be represented in the form of association rule or sets of frequent items. The association rule was first introduced in [1]. The classic application is market basket analysis which is the analysis how the items purchased by customers are associated. The association rule is wildly applied in many research fields such as decision support, financial forecasts, bioinformatics, web mining, etc. Moreover, it helps in data classification, clustering and others data mining task as well. The association rule mining algorithms is to decompose into two major subtasks:

(1) To generate all the frequent itemsets that satisfy the *minsup* threshold.

(2) To extract all the high confidence rules from frequent itemsets found in previous step.

In recent years, many researches have been focusing on the mining of frequent itemsets. The most popular and classic algorithm is the Apriori algorithm which was first introduced in [2]. There are many approaches based on this algorithm such as [3] [4] [5]. However, Apriori based algorithm has problem of scanning database many times to generate candidate sets especially when dealing with the database with a lot of long itemsets. To overcome this problem, many algorithms have been proposed. The FP-growth algorithm is one of these improved algorithms and is among the most popular which is introduced by [6]. The FP-growth algorithm mines frequent itemsets without generating candidate sets and scans database only twice. Many approaches have been proposed to mine frequent itemsets based on this algorithm. In [7], they proposed depth first algorithm for mining of frequent itemsets. This method compresses database into FP-tree by using Children table to avoid generating many sub trees. In [8], they proposed TreeITL-Mine for combines the features both horizontal and vertical data layout for frequent pattern mining. In [9], CP-mine algorithm was introduced for mining the set of frequent items from MFP-tree. The drawback of this method is the updating of the marked value which increases space and runtime if many marked nodes contain in pruning tree.

We propose a new frequent itemsets mining algorithm, named SFI-mine algorithm. This algorithm mines frequent itemsets from FP-tree, constructs pattern-base by using a new method which is different from conditional pattern-base in FP-growth but still maintains the correctness. The SFI-mine mines frequent itemsets with a new combination method without recursive construction of conditional FP-trees. As the result, the SFI-mine obtains complete and correct frequent itemsets. This paper is organized as follows. In section 2, background is presented. The proposed SFI-mine algorithm is illustrated in section 3. An illustrate example is given in section 4. The correctness proving is shown in section 5. Finally, the conclusion is addressed in section 6.

2. Background

2.1. The Basic Concept

The basic concept for mining of frequent pattern [10], [11], [12] presents as follow:

978-0-7695-3407-7/08 $25.00 © 2008 IEEE
DOI 10.1109/ICCIT.2008.383

Definition 1 (Transaction database). Let $I=\{x_1,x_2,...,x_m\}$ be a set of items, and a *transaction database* $DB=\{T_1,T_2,...,T_n\}$, where $T_i(i\in[1,...,n])$ is a transaction which contains a set of I. □

Example 1. Let the first two columns of Table 1 is a *DB*. It consists of ten transactions, labeled as *TID* in the table, and eleven items. For example, The first of T (or *TID* = 1) is {*a,b,c,d,f,g,k*}.

Definition 2 (Support). The support (or occurrence frequency) of a *pattern A* is the number of transactions containing *A* in *DB*, where *A* is a set of items. □

Example 2. According to Table 2, there are eleven items with their support.

Table 1. A transaction database

TID	Items	Sorted Frequent Items
1	*a,b,c,d,f,g,k*	*f,c,a,b,k*
2	*b,c,e,f,j,k*	*f,c,b,j,k*
3	*a,f,h,j*	*f,a,j*
4	*a,b,c,d,j*	*c,a,b,j*
5	*f,i*	*f*
6	*a,b,c,h,k*	*c,a,b,k*
7	*c,e,f,i*	*f,c*
8	*a,b,c,d,f,k*	*f,c,a,b,k*
9	*a,f,j*	*f,a,j*
10	*c,f,h,i*	*f,c*

Table 2. The support all of items

Items	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*
Support	6	5	7	3	2	8	1	3	3	4	4

Table 3. The head table with *minsup* = 4

Items	*f*	*c*	*a*	*b*	*j*	*k*
Support	8	7	6	5	4	4

Definition 3 (Minimum support). The minimum support or *minsup* is the threshold that user defines for selection of frequent items. □

Example 3. The *DB* in Table 1, user defines *minsup* = 4. Therefore, all items that have support not less than *minsup* are presented in Table 3, so those items are frequent items.

Definition 4 (Frequent itemsets and infrequent itemsets). For any Item x, support(x) is defined to be a number of transaction T for which $x\subseteq T$. For a given parameter *minsup*, an item x is called *frequent item* if support(x) $\geq$ *minsup*, others called *infrequent item*. The set of frequent item, called frequent itemsets, is denoted by FS, others called infrequent itemsets, denoted by $\overline{FS}$. □

Example 4. All itemsets and its supports illustrated in Table 2. There are FS and $\overline{FS}$. In Table 3, it is the header table that contains only FS, excludes $\overline{FS}$. Because of all items in Table 3 have support not less than *minsup*.

Definition 5 (Frequent pattern tree). A frequent pattern tree or FP-tree is a tree structure with the following properties:

(1) It consists of a *root*, a set of *items-prefix subtrees* which are the children of root and a *header table*.

(2) Each node in the item-prefix subtrees consist of three fields: *item-names*, *support* and *node-link*, where item-name is the name of node represents, support is the number of transactions represented by the portion of the part reaching this node, and node-link links to the next node in FP-tree carrying the same item-name or null.

(3) The header table consists of two fields: *item-name* and *head of node-link* that point to the first node in FP-tree. □

Based on definition 5, the construction of FP-tree algorithm presents as below:

```
Algorithm 1: FP-tree Construction
Input : A transaction Database DB
Output : FP-tree
1  Procedure FP-treeConstruction
2  scan the DB once to collect the frequent itemsets and their
3    support then sort in support descending and create in header
4    table
5  FP-tree = null
6  for each transaction t_i in DB
7    select and sort the frequent items in t_i by order in header
8      table
9    call InsertTree(FP-tree, t_i)
10 end for

11 Procedure Insert_tree(root, DB)
12 for each item x_i in DB
13   if root has a child N then N.item-name = x_i
14     increment N++
15     root = N
16   else
17     create the new node x_i is the child of root
18     link header table to node
19   end if
20 end for
```

Figure 1. The frequent pattern-tree algorithm

Example 5. The *DB* is scanned for second times to create the FP-tree. We first create the root of the tree. The first transaction is leaded to the construction of the first branch of FP-tree: (<*f*:1>,<*c*:1>,<*a*:1>,<*b*:1>,<*k*:1>). The items in the transaction will be inserted into the tree ordering by descending support value. The second transaction (<*f*>,<*c*>,<*b*>,<*j*>,<*k*>) shares the same prefix (<*f*>,<*c*>) with the existing part (<*f*>,<*c*>,<*a*>,<*b*>,<*k*>). Thus, the support of each node along the prefix is increasing by 1 and the remaining item list (<*b*>,<*j*>,<*k*>) in the second transaction will be created as the new node. The new node <*b*:1> is linked to the child of <*c*:2>; node <*j*:1> as a child of <*b*:1>; node

<*k*:1> as a child of <*j*:1>. For the third transaction (<*f*>,<*a*>,<*j*>), it shares only the node <*f*>. Thus, *f*'s support is increased by 1 and remained item list (<*a*>,<*j*>) in the third transaction will be created as the new node. The new node <*a*:1> is linked to the child of <*f*:3>; node <*j*:1> as a child of <*a*:1>. The remaining transaction in *DB* can be done by the same way. After all transactions are scanned, the FP-tree is presented in Figure 2.

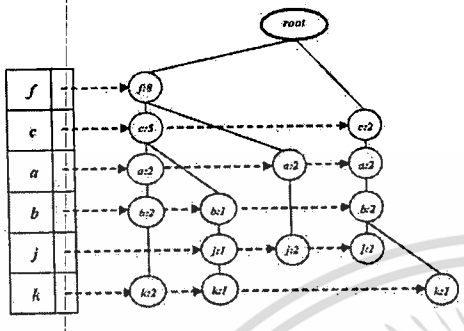

Figure 2. The Frequent Pattern Tree

Table 4. All conditional pattern-base

Items	Conditional pattern base
k	{(*fcab*:2),(*fcbj*:1),(*cab*:1)}
j	{(*fcb*:1),(*fa*:2),(*cab*:1)}
b	{(*fca*:2),(*fc*:1),(*ca*:2)}
a	{(*fc*:2),(*f*:2),(*c*:2)}
c	{(*f*:5)}
f	{}

Definition 6 (Conditional pattern-base). The conditional pattern-base is a sub database which consists of a set of prefix paths in the FP-tree which co-occurrence with a suffix pattern. □

Example 6. Let us examine the mining process for construction of pattern-base by starting from the bottom of the node-link of the header table, according to Figure 2.

For node *k*, there are three paths in the FP-tree: (<*f*:8>,<*c*:5>,<*a*:2>,<*b*:2>,<*k*:2>), (<*f*:8>,<*c*:5>,<*b*:1>,<*j*:1>,<*k*:1>) and (<*c*:2>,<*a*:2>,<*b*:2>,<*k*:1>). All nodes in the first path appear twice together with *k*, so we get only *k*'s prefix path (<*f*:2>,<*c*:2>,<*a*:2>,<*b*:2>). Similarly, all nodes in the second path appear once together with k, so *k*'s prefix path is (<*f*:1>,<*c*:1>, <*b*:1>,<*j*:1>). All nodes in the last path appears together with k once, *k*'s prefix path is (<*c*:1>,<*a*:1>, <*b*:1>). These three prefix paths of *k* are (<*fcab*:2>, <*fcbj*:1>,<*cab*:1>), called *k* 's conditional pattern-base. Similarly, the remaining nodes in the header table such as *j, b, a, c, f* in sequence are constructed in the same way. We summarize all conditional pattern-base in Table 4.

Definition 7 (Conditional FP-tree). The conditional FP-tree is the sub tree which is constructed from a conditional pattern-base as mining pattern. □

Example 7. From previous example, construction of *k* 's conditional FP-tree by doing the following steps. The *k*'s header table is constructed from conditional pattern-base by examining the support of each item that is not less than *minsup*. There are two items, <*c*:4> and <*b*:4>. The conditional FP-tree is constructed by examining the first path in conditional pattern-base, <*fcab*:2> which exclusion of infrequent items (or node which not appear in *k*'s header table) follow by insertion of frequent item <*cb*:2> to *k*'s conditional FP-tree. Similarly, to insert the frequent items of the second path to FP-tree, <*cb*:1>. Lastly, insertion of the frequent items of the third path to FP-tree, <*cb*:1>, as seen in Figure 3.

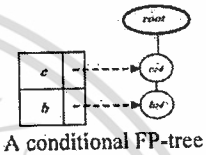

Figure 3. A conditional FP-tree for item *k*

2.2. Frequent Pattern Growth Algorithm

A FP-growth algorithm is introduced by Han J. et al [6], [10]. The algorithm does not subscribe to the generation of candidate itemsets paradigm of Apriori. It encodes the data set using a compact data structure called FP-tree and extracts frequent pattern directly from this prefix tree. This method consists of three main steps. Firstly, to scan the *DB* once to find 1-frequent itemsets. Secondly, to construct the header table from 1-frequent itemsets by descending support sorted. The support value of each item is shown in Table 2, so it consists of frequent itemsets and infrequent itemsets. The frequent itemsets are sorted by descending support and presented in Table 3, infrequent itemsets are discarded. Next, to scan *DB* for the second times to construct the FP-tree, according to definition 5 and example 5. Lastly, to explore frequent pattern from FP-tree by using FP-growth algorithm which present as Figure 5 and summarize in Table 4.

Example 8. In this example, we present the mining of frequent patterns. The mining process illustrate in Figure 4 is based on construction FP-tree shown in Figure 2. Starting mine at node *k*.

For node *k*, the conditional pattern is {(*fcab*:2), (*fcbj*:1),*cab*:1}. Constructing conditional FP-tree on branch (*c*:4,*b*:4), as shown in Figure 2. Then we mine by calling "mine(<*f*:4,*b*:4>|*k*)". The items that involve

mining are *b* and *c* in sequence. The first derives a frequent pattern (*bk*:3), a conditional pattern-base {(*f*:4)} and then call "mine(<*f*:4>|*bk*)"; the second derives a frequent pattern <*fk*:4>; and the third derives a frequent pattern <*fbk*:4>. Therefore, a set of frequent patterns involving k is {<*k*:4>,<*bk*:4>,<*fk*:4>,<*fbk*:4>}.

Similarly, node *j*'s conditional pattern is {(*fcb*:1), (*fa*:2),(*cab*:1)}. All items' support less than *minsup*, so the mining for *j* terminates. Node *b* derives conditional pattern-base {(*fca*:2),(*fc*:1),(*ca*:2)} and conditional FP-tree mine(<*c*:5,*a*:4>|*b*), a set of frequent patterns involving *b* is {<*b*>,<*ab*:4>,<*cb*:4>,<*cab*:4>}. Node a, the conditional pattern is {(*fc*:2),(*f*:2)(*c*:2)} and conditional FP-tree mine(<*f*:4,*c*:4>|*a*) , a set of frequent patterns involving *a* is {<*a*:4>,<*ca*:4>,<*fa*:4>,<*fca*:4>}. Node *c*'s conditional pattern-base is {(*f*:5)} and conditional FP-tree mine(<*f*:5>|*c*) , a set of frequent patterns involving *c* is {<*fc*:5>}. For the last, node *f* derives only <*f*:8> but no conditional pattern-base, so terminates.

The conditional FP-trees and the sets of frequent pattern are summarized in Table 5.

Table 5 All frequent patterns

Items	Conditional FP-tree	Frequent pattern
k	(<*f*:4>,<*b*:4>|*k*)	{*k*, *bk*, *fk*, *fbk*}
j	{}	{}
b	(<*c*:5,*a*:4|*b*)	{*b*, *ab*, *cb*, *cab*}
a	(<*f*:4,*c*:4>|*a*)	{*a*, *ca*, *fa*}
c	(<*f*:5>|*c*)	{*c*, *fc*}
f	{}	{}

Algorithm 2: FP-growth
Input : FP-tree and *minsup*
Output : The set of all frequent patterns
Method: call FP-growth(FP-tree, *null*).

```
1  Procedure FP-growth( Tree, α )
2  if tree contains a single prefix path   // Mining singlepath
3     let P be the single prefix-path part of tree;
4     let Q be the multiple path with the top branching node
5           replaced by a null root;
6     for each combination (denote as β ) of node in P
7     generate itemset β ∪ α with support=minsup of node in β ;
8     let freq_pattern_set(P) be the set of patterns so generated;
9  else let Q be Tree;
10    for each a_i in Q                    // Mining multiplepath
11    generate pattern β = a_i ∪ α with support= a_i .suppot;
12    construct β 's conditional pattern-base and
13          β 's conditional FP-tree Tree_β ;
14    if Tree_β ≠ φ  then call FP-growth(Tree_β, β)
15    let freq_pattern_set(Q) be the set of patterns so generated;
16 return(freq_pattern_set(P) ∪ freq_pattern_set(Q) ∪
17    (freq_pattern_set(P)× freq_pattern_set(Q))
```

Figure 4. The frequent pattern-growth algorithm

3. The Approach

In this section, we propose a new algorithm to mine frequent itemsets, called SFI-mine. The SFI-mine algorithm uses a new method to construct pattern-base from FP-tree which is different from conditional pattern-base in FP-growth and mines frequent itemsets with a new combination method without the need to recursive construction of conditional FP-tree during frequent itemsets mining. As a result, the SFI-mine produces complete and correct frequent itemsets. The approach has essential definitions as follow.

Definition 8: (Postfix item) Let $A \subseteq I$. The postfix item *PI* of *A*, *PI(A)*, is the last item of *A*. □

Example 9. According to the header table in Figure 2, the first *PI* that we examine is *c* and next are *a*, *b*, *j* and *k* in sequence.

Definition 9: (Pattern-base) The pattern-base *PB* of item *x*, PB_x , is the pattern that contains the whole items corresponding to the path start from all the topmost item to item *x*. We denote $PB_x = \{A \mid PI(A) = x\}$. □

Example 10. According to Figure 2, let *PI* = *b*, following the *node-link* for *b*. We get three paths in tree: (<*f*:2>,<*c*:2>,<*a*:2>,<*b*:2>), (<*f*:1>,<*c*:1>,<*b*:1>) and (<*c*:2>,<*a*:2>,<*b*:2>). To build the pattern-base for *b*, we exclude *PI* (or node *b*) in these three paths, (<*f*:2>,<*c*:2>,<*a*:2>), (<*f*:1>,<*c*:1>) and (<*c*:2>,<*a*:2>). Therefore, PB_b = {(fca:2),(fc:1),(ca:2)}

Definition 10: (Pre-frequent item) Let PF_x be the set of summarized item of *PB* in postfix item *x* with *support* ≥*minsup*, so

$$PF_x = \{w \mid w \in PB_x; \sum_{i=1}^{n} \text{support}(w_i) \geq minsup\}$$

; where *n* is the number of *w* in *PB*. □

Definition 11: (Infrequent itemsets) Let IF_x be the set of summarized item of PB_x in postfix item *x* with *support* less than *minsup*, so

$$IF = \{z \mid z \in PB_x; \sum_{i=1}^{n} \text{support}(z_i) < minsup\}$$

; where *n* is the number of *z* in *PB*. □

Example 11. Let we consider *PB* of node *b* in the above example. After we summarized the support of all items in *b*'s paths, we get {<*c*:5>,< *a*:4>,<*f*:3>} Thus, PF_b = {<*c*:5>,< *a*:4>} and *IF* = <*f*:3>}.

Definition 12: (Frequent base) The frequent base *FB* is a set of itemsets in *PB* that excludes *IF*.

$$FB = \{(A\text{-}IF) \mid A \in PB_x; \sum_{i=1}^{m} \text{support}(A\text{-}IF) \geq minsup\}$$

; where *m* is the number of *A-IF* in *PB*. □

Example 12. To consider *PB* in example 9 and *IF* in example 11, so *FB* = {(*ca*:4)}.

Definition 13: (Combination frequent itemsets) The combination frequent pattern F_x is separated into two cases. Case 1, to derive the F_x for second level by combining the postfix item x with item in PF_x. Case 2, to derive the F_x derive for the higher second level by combining the postfix item x with items in FB_x. The item in F_x is *frequent pattern*. □

Example 13. From example 11, there are two items in PF_b. Case 1, we combine item b with item in PF_b for the second level, so F_b for second level are <*cb*:5> and <*ab*:4>. Case 2, according to example 12, we combine b with itemsets in FB. Then, F_b for third level of node b is <*cab*:4>.

The SFI-mine algorithm consists of the following steps as shown in Figure 5, in brief, there are six major steps.

Algorithm 3: SFI-mine
Input: FP-tree, $minsup(x_i)$ of each item x_i in the HeaderTable
Output: The complete and correct sets of all frequent itemsets: FS
Method call SFI-mine(*Tree*,*root*)
1 Procedure SFI-mine(*Tree*,*x*)
2 i=2 // defining the first item
3 repeat
4 for each x_i in the header table from the topmost to the
5 Bottommost
6 // construction of PB
7 Calculate support of all items corresponding to the path
8 start from all the topmost item to x_i and store all
9 corresponding path to PB_{x_i}
10 Check support of each item in $PB_{x_i} \geq minsup$ then
11 store item to PF_{x_i}
12 // combination of the second level
13 Construct $F_{x_i} = x_i.PF_{x_i}$
14 If $|PF_{x_i}| > 1$ then Call High_combine
15 end for
16 until tree = ϕ
17 Result $FS = \cup F_{x_i}$ // union all F_{x_i} to FS
18 **Procedure High_combine** (PF_{x_i} , PB_{x_i})
19 Construct FB_{x_i}
20 //combination of the level higher than the second level
21 Construct $F_{x_i} = x_i.FB_{x_i}$
22 Return F_{x_i}

Figure 5. The SFI-mine algorithm

In line 13 and line 21, we obtain frequent itemsets by using combination of PI with a set of PF and FB. The data structure table of PF and FB consists of *item-name* and *support*.

Step 1. Defining the first item, PI, at the second node in the header table.

Step 2. Construction of $PB(PI)$ and calculation support of each item in PB, if *item.support* is not less than $minsup$, then these items are stored in PF.

Step 3. Construction of F for the second level by combination of PI with PF.

Step 4. Checking member item of PF, if the amount of PF is greater than 1 then FB is constructed from PB with exclusion of IF. To construct F in the level higher than the second level, PI is combined with FB, as the result, F is returned.

Step 5. Checking items in the header table, if there are any items in the header table, then define PI at the next item in the header table and step 2 to step 4 are repeated until there is no item left, then proceed to step 6.

Step 6. Union all F and store the output in FS.

4. An Illustrative Example

In this section, we illustrate an example based on definitions and an algorithm in section 3. The details of all steps are presented as follow:

Example 14.

According to Figure 2 and Algorithm in Figure 5, we start examining the first item at c ($PI = c$), following the node link of item c. To consider each corresponding path starts from the topmost of item c to item c, we get one path in tree: (<*f*:8>,<*c*:5>). To build the PB for c, we exclude item c in path, <*f*:5>. Notice that counter of item in path is set to 5 because f appears five times together with c, so PF_c ={<*f*:5>}. After that, to combine item in PF_c with c. We get F_c = {<*fc*>} because PF_c has only one item, so we terminate considering item c.

Next, we examine item a ($PI = a$), starting at corresponding path of item a from the topmost of item a to item a. We get three paths in the tree: (<*f*:2>,<*c*:2>,<*a*:2>), (<*f*:2>,<*a*:2>) and (<*c*:2>,<*a*:2>). We exclude item a for construct PB_a; (<*f*:2>,<*c*:2>), (<*f*:2>) and (<*c*:2>) or PB_a = {(*fc*:2),(*f*:2),(*c*:2)}. After adding support of these three paths: (<*f*:4>,<*c*:4>), we get PF_a = {(*f*:4),(*c*:4)}. We then combine item in PF_a with a, we get two combination frequent itemsets at the second level F_a = {<*fa*>,<*ca*>}. For PF_a, it has two items, so we construct the third level of combination itemsets by constructing FB_a. In this case, IF={}, so we construct FB_a from PB_a. There is no itemset in the third level because no itemset's support greater than $minsup$, so we terminate considering item a.

The remaining items (b, j and k) in the header table can be done in the same way. After all the items are done, we present all PB_x in Table 5 , all F_x in Table 6.

Last, we present *FS* that separates by level of combination of all F_k, present in Table 7.

Table 5. The Pattern-base

Items	Pattern-base
c	{(*f*:5)}
a	{(*fc*:2), (*f*:2), (*c*:2)}
b	{(*fca*:2),(*fc*:1),(*ca*:2)}
j	{(*fcb*:1),(*fa*:2),(*cab*:1)}
k	{(*fcab*:2),(*fcbj*:1),(*cab*:1)}

Table 6. All combination frequent itemsets

Items	Combination frequent itemsets
c	<*fc*>
a	<*fa*>,<*ca*>
b	<*cb*>,<*ab*>,<*cab*>
j	{}
k	<*ck*>,<*bk*>,<*cbk*>

Table 7. All frequent itemsets

Level	Frequent Itemsets
1	{*f, c, a, b, j, k*}
2	{*fc, fa, ca, cb, ab, ck, bk*}
3	{*cab, cbk*}

5. Correctness

In the following theorem, we show that our algorithm can mine all frequent itemsets completely and correctly.

Theorem 1. The all frequent itemsets that obtained by the SFI-mine algorithm are both complete and correct.

Proof. We separate our proof into two cases.

Case 1. Let the combined itemsets $i = x.a_i$; $a_i \in PF$; $x=PI$. We know that support(a_i) $\geq$ *minsup*, according to Definition 10. When we combine x with a_i , so support(i) $\geq$ *minsup*. Thus, i is frequent itemsets for the second level, according to Definition 13.

Case 2. Let the combined itemsets $j = x.b_i$; $b_i \in FB$; $x=PI$. We know that support(b_i) $\geq$ *minsup*, according to Definition 12. When we combine x with b_i , then support(i) $\geq$ *minsup*. Thus, j is frequent itemsets for the higher second level, according to Definition 13. Therefore, itemsets mined from case 1 and case 2 are frequent itemsets.

6. Conclusion

We have presented a new algorithm to mine frequent itemsets, named SFI-mine algorithm. We illustrate the detail of new definitions, examples and algorithm of the approach. The SFI-mine is developed by mining of frequent itemsets from FP-tree; construction of a new pattern-base which is different from a conditional pattern-base in FP-growth and mining frequent itemsets with a new combination method without recursive construction of conditional FP-trees. The results of this method are still obtaining complete and correct frequent itemset. We conclude by giving examples and proving correctness.

Acknowledgment

We appreciate Tawesak Tanwandee MD. for his help in proof reading of this paper.

7. References

[1] Agrawal, R., Imielinski, T., Swami, A. "Mining Association Rule between Sets of Items in Large Databases.", *SIGMOD 1993*, ACM, 1993, pp. 207-216.

[2] Agrawal, R., and Srikant R. "Fast Algorithm for Mining Association Rules.", *VLDB 1994*, 1994, pp. 487-499.

[3] J. S. Park, M. Chan and P. S. Yu., "An Effective Algorithm for Mining Association Rules.", *SIGMOD 1995*, ACM, May 1995, pp. 175-186.

[4] T. Shintani and M. Kitsuregawa., "Hash Based Parallel and Distributed Info. System.", Miami beach, FL, December 1996.

[5] A. Savasere, E. Omiecinski, and S. Navathe., "An Efficient Algorithm for Mining Association Rules in Large Databases.", *21st VLDB*, Zurich, Swizerland, 1995.

[6] Jiawei Han, Jian Pei and Yiwen Yin, "Mining Frequent Pattern Without Candidate Generation", *SIGMOD 2000*, ACM, Dallas, Texas, USA, May 2000, pp. 1-12.

[7] Qunxiong Zhu, and X, "Depth First Generation of Frequent Patterns Without Candidate Generation.", *PAKDD 2007*, Springer-Verarge, 2007, pp. 378-388.

[8] Raj P. Gopalan, and Yudho Giri Sucahyo, "TreeITL-Mine: Mining Frequent Itemsets Using Pattern Growth, Tid Intersection and Prefix Tree.", *AI 2002*, Springer-Verlarg, pp. 535-546.

[9] Nualsri Denwattana, and Yuttana Treewai, "Discovering of Frequent Itemsets with CP-mine Algorithm.", *DMIN 2006*, Las Vagas, Nevada, June 26-29, 2006.

[10] Jiawei Han, Jian Pei, Yiwen Yin and Runying Mao, "Mining Frequent Pattern Without Candidate Generation: a Frequent-pattern Tree", *Data Mining and Knowledge Discovery*, © 2004 Kluwer Academic Publishers, Netherland, 2004, pp. 53-87.

[11] Jiawei Han and Micheline Kamber, "*Data Mining: Concepts and Techniques, second editions*", Elsevier, 2006.

[12] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar, *Introduction to Data Mining*, Pearson Education Inc., Boston, Ma, 2006.

BIOGRAPHY

Supatra Sahaphong received her B.S. in Computer Science from Ramkhamhaeng University (RU), Thailand, in 1990, a M.S. in Information Technology from King Mongkut's Institute of Technology Ladkrabang (KMITL), Thailand, in 1998, and a Ph.D. in Computer Science from KMITL in 2012. From 1991 to 1999, she worked at Institute of Computer, RU. In 1999 she joined the Department of Computer Science, Faculty of Science, RU, where she is currently an Assistant Professor. Her research interests include data mining, knowledge discovering, and image segmentation.