DISCOVER, RECYCLE AND REUSE FREQUENT PATTERNS IN ASSOCIATION RULE MINING

GAO CONG

(Master of Engineering, Tianjin University, China)

A DISSERTATION SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

First of all, I feel very privileged and grateful to have had my supervisor Dr Anthony K.H. Tung as my supervisor. He deserves more thanks than I can properly express for his continuous encouragement, his support as not only my advisor but my friend, sharing with me his knowledge and the great deal of time he gave me for discussion. My endeavors would not have been successful without him. I also thank him for his kindness in involvement of me in projects of various topics, which expands my horizons.

I feel very grateful to Dr Bing Liu who is my supervisor when he was NUS for his continuous supports, many insightful discussions, directing me how to find research topics and especially his patience and comments in directing my paper writing.

I would like to express my deep gratitude to Dr Beng Chin Ooi for all his nice assistance when I study at NUS. Without his assistance, I might not have had the chance to study here. I would like to express my gratitude to Dr Kian-Lee Tan and Dr Wee Sun Lee for their helps in my Ph.D. study. I would like to thank Dr Mong Li Lee and Dr Sam Yuan Sung for their comments on my draft thesis. I also thank NUS for providing scholarship and facilities for my study. I would like to thank the reviewers for their highly valuable suggestions to improve the quality of this thesis.

I would like to thank my team-workers, CuiPing Li, Xin Xu, Feng Pan, Haoran Wu and Lan Yi. I am also grateful to all my good friends in CHIME lab(S17-611), Database Lab (S16-912) and other labs in NUS, especially Ziyang Zhang, Minqing Hu, Ying Hu, KaiDi Zhao, Bei Wang, Baohua Gu, Xiaoli Li, Patric Phang, Jing Liu, Qun Chen, Jing Xiao, Rui Shi, Wen Wu, Hang Cui couple, Gang Wang, Cheng Zhang, Xia Cao, Zonghong Zhang, Rui Yang, Jing Zhang, and Manqin Luo etc. Please forgive me for not listing all of you here but you are all in my heart. You gave me quite a lot of happy hours and made my hard and boring Ph.D. life a bit better. It is my pleasure to get to know all of you.

I would like to express my deep appreciation to my parents and young sister for their unselfish support and love. I never can thank you enough. But I know that you will feel proud of my achievements, which are the best reward to you.

Foremost, I want to thank my wife who always shares my good and bad moods, endures my awful time, and supports me with her care and love. I would like to dedicate this thesis to you with love.

Contents

List of Tables

List of Figures

Summary

With the fast-growing and huge amount of data and information, Knowledge Discovery in Databases has become one of the most active and exciting research areas in the database research community because of its promise in efficiently discovering interesting or unexpected knowledge from large databases. Association rule is one of the most important subareas of data mining since it provides a concise and intuitive description of knowledge and has wide applications. In this thesis, a framework of mining, recycling and reusing frequent patterns for association rule mining is proposed. Within the framework, several open technical problems are examined and addressed.

First, an approach is proposed to recycle the intermediate mining results and frequent patterns from the previous mining process to speed up the subsequent mining process when the mining constraints are changed. The main component of the approach is a new concept "tree boundary" and a recycling technique based on the new concept. On the basis of the recycling technique, two mining algorithms are adapted for the recycling task. An extensive experiment is conducted and the experimental results show that the technique is able to reduce the amount of computation greatly in the iterative mining with constraint changes.

Second, an approach to recycling frequent patterns from previous round of mining is proposed. The proposed method is operated in two phases. In the first phase, frequent patterns obtained from an early iteration are used to compress a database. In the second phase, subsequent mining processes operate on the compressed database. Two compression strategies are proposed. One strategy, MLP, mainly considers space and the other, MCP, considers the mining cost. In this thesis, three existing frequent pattern mining techniques are adapted to exploit the compressed database. The experimental results show that the proposed recycling algorithms outperform their non-recycling counterparts by an order of magnitude. Another interesting finding from experimental results is that the strategy MCP is more effective than MLP for recycling.

Third, in order to efficiently mine frequent patterns in microarray datasets, which are usually characterized by a large number of columns and a small number of rows, several algorithms are proposed on the basis of row enumeration strategy. A series of pruning strategies are designed to speed-up the proposed algorithms. The experimental results on real-life microarray data show that the proposed algorithms outperform existing algorithms by orders of magnitude.

Finally, based on row enumeration strategy, algorithm FARMER is proposed to mine interesting association rule groups (IRGs) given user specified rule consequent by identifying their upper bounds and lower bounds. The IRGs can reduce the number of discovered rules greatly. FARMER exploits the user specified constraints including minimum support, minimum confidence and minimum chi-square to pruning rule search space. Several experiments on real microarray datasets show that FARMER is orders of magnitude better than previous association rule mining algorithms.

In summary, this thesis describes a framework for mining and recycling frequent patterns in association rule mining. Within the framework, the mining results from the previous mining process are shown to be helpful for subsequent constrained mining and the proposed algorithms of mining frequent patterns and IRGs are shown to be efficient.

The publications that have arisen from the material described in this thesis are listed in the reverse chronological order as follows.

- Gao Cong, Anthony K.H. Tung, Xin Xu, Feng Pan, Jiong Yang. *FARMER: Fining*

Interesting Association Rule Groups by Row Enumeration in Biological Datasets. In 23rd ACM International Conference on Management of Data, 2004.

- Gao Cong, Beng Chin Ooi, Kian-Lee Tan, Anthony K.H. Tung. *Go Green: Recycle and Reuse Frequent Patterns*. In IEEE 20th International Conference on Data Engineering, 2004.

- Feng Pan, Gao Cong, Anthony K.H. Tung, Jiong Yang, Mohammed J. Zaki. *CARPENTER: Finding Closed Patterns in Long Biological Datasets*. In the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2003.

- Gao Cong, Bing Liu. *Speed-up Iterative Frequent Itemset Mining with Constraint Changes*. In IEEE International Conference on Data Mining, 2002.

Chapter 1

Introduction

With the popular use of the World Wide Web as well as the widespread use of new technologies for data generation and collection, we are flooded with huge amounts of fast-growing data and information. The explosive growth mainly comes from business transactional data, medical data, scientific data, demographic data, and web data. These data, collected and stored in numerous large databases, are far beyond our human ability for comprehension without powerful tools. So, we must find ways to automatically analyze, summarize, cluster and classify the data, and to discover and characterize the properties in the data. In this situation, Knowledge Discovery in Databases (or KDD in short) has become one of the most active and exciting research areas in the database community.

KDD is the "process of discovering interesting knowledge from large amounts of data stored in databases, data warehouses or other information repositories"[41]. The discovered knowledge should be interesting to users. Moreover, the discovered knowledge is usually implicit, previously unknown or unexpected, and potentially useful information. KDD can be viewed as the natural evolution of information technology. As shown in Figure 1.1 [41], the development of KDD in the database industry follows the path from data collection and database creation to database management system, and to data warehouse and KDD.

KDD involves an integration of techniques from multiple disciplines, such as database

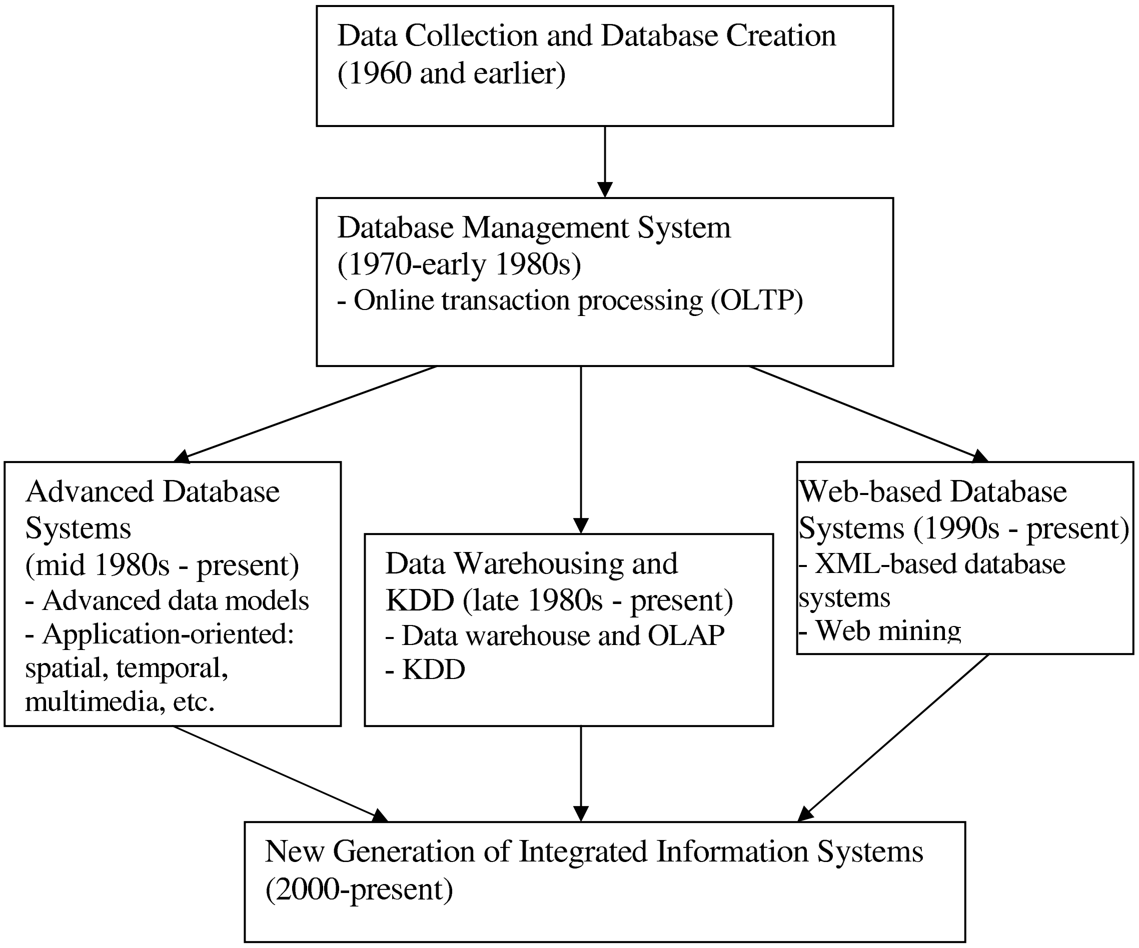

Data Collection and Database Creation
(1960 and earlier)
Database Management System
(1970-early 1980s)
- Online transaction processing (OLTP)
Advanced Database Systems
(mid 1980s - present)
- Advanced data models
- Application-oriented: spatial, temporal, multimedia, etc.
Data Warehousing and KDD (late 1980s - present)
- Data warehouse and OLAP
- KDD
Web-based Database Systems (1990s - present)
- XML-based database systems
- Web mining
New Generation of Integrated Information Systems
(2000-present)

Figure 1.1: The evolution of database technique

technology, machine learning, statistics, expert systems and data visualization. Some techniques of KDD, such as clustering and classification, have already been extensively studied in machine learning. However, the emphasis of KDD is placed on **efficiency** and **scalability** of algorithms to discover interesting knowledge from emerging huge datasets and other new types of datasets, such as web data and biological data.

KDD aims to find interesting knowledge for users in an efficient and effective way. Only the users can judge whether the discovered knowledge is interesting or not. Moreover, it is difficult to know exactly what can be discovered from database before mining is done [41] since one of the most attractive aspects of data mining is to find unexpected patterns. As a result, data mining should be a human-centered, interactive and iterative process.

The discovered knowledge or patterns can be represented with various formats, among which association rules, classification, clustering and outlier detection are the most investigated topics in the field of KDD. A short introduction of them is given as follows.

- **Association rules.** Association rule mining discovers the attribute-value conditions that occur frequently together in a given database [5]. A typical example of association rule problem is market basket analysis, in which the typical question addresses the sets of items that customers are likely to purchase together in a trip to the store.

- **Classification.** Classification is "the process of finding a set of models (or functions) that describe and distinguish between data classes or concepts for the purpose of being able to use the model to predict the class of objects whose class label is unknown"[41, 76].

- **Clustering.** Unlike classification that is supervised, clustering analyzes data objects without consulting a known class label since such a class label does not always exists. Clustering is used to generate such a label and group similar objects together based on the principle of maximizing the intraclass similarity and minimizing the interclass similarity [41, 44, 104].

- **Outlier Analysis.** Outlier analysis finds deviation from the expected values since the rare events may be more interesting than the more regularly occurring ones [41]. The expected value may be given by users or estimated by some statistic method, such as regression analysis.

Many people treat Knowledge Discovery in Databases, (KDD in simple) as a synonym for another popular term, Data Mining [41]. Others consider Data Mining as an essential step in the whole process of KDD, which includes the following steps: (1) Data cleaning; (2) Data integration; (3) Data selection; (4) Data transformation; (5) Data Mining; (6) Pattern evaluation; and (7) Knowledge presentation [41]. Like [41], this thesis does not distinguish between the two terms, KDD and Data Mining and views both terms as the whole process of knowledge discovery in database.

This thesis concentrates on association rule mining. The rest of this chapter first gives some background knowledge about the association rule mining, and then discusses the motivations and contributions of this thesis.

1.1 Background

This section will introduce the association rule and its applications as well as research progress of association rule mining.

1.1.1 Association rules and their applications

Association rule mining was first introduced in [5] to address a class of problems typified by a market-basket analysis. One example of association rule is "80 percent of all transactions in which *beer* and *peanut* were purchased also included *potato chips*." Classic market-basket analysis treats the purchase of a number of items (for example, the contents of a shopping basket) as a single transaction. The goal is to find trends across large numbers of transactions that can be used to understand and exploit natural buying patterns. This information can be used to adjust inventories, modify floor or shelf layouts, or introduce targeted promotional activities to increase overall sales.

Before introducing the other applications of association rule mining, this chapter first gives an informal definition of association rule and the formal concept will be given in Chapter 2.

Let I=$\{i_1, i_2, .., i_m\}$ be a set of items. Let D be the dataset (or table) which consists of a set of rows (transactions) R=$\{r_1, .., r_n\}$. Each row r_i consists of a set of items in I and is associated with a unique identifier, called *RID* (row ID) or TID (transaction ID). Note that we can also treat D as a boolean dataset, where each column of the boolean dataset corresponds an item in I and each row r_i consists of a value of either 1 or 0 for each item in I.

An association rule has the form $X \rightarrow Y$, where the X and Y are sets of items, and $X \cap Y = \emptyset$. For simplicity, a set of items is called as an **itemset** or a **pattern**[1]. Let $freq(X)$ be the number of rows (transactions) containing X in the given database. The **support** of an itemset X is defined as the fraction of all rows containing the itemset, i.e. $freq(X)/|D| \times 100\%$ [2]. The **support** of an association rule is the support of the union of X and Y, i.e. $freq(X \cup Y)/|D| \times 100\%$. The **confidence** of an association rule is defined as the percentage of rows in D containing itemset X that also contain itemset Y, i.e. $(freq(X \cup Y)/freq(X)) \times 100\%$. An itemset (or a pattern) is frequent if its support is equal to or more than a user specified minimum support (a statement of generality of the discovered association rules). Association rule mining is to identify all rules meeting user-specified constraints such as minimum support and minimum confidence (a statement of predictive ability of the discovered rules). One key step of association mining is frequent itemset (pattern) mining, that is to mine all itemsets satisfying user-specified minimum support.

While association rule approaches have their origins in the retail industry, they can be applied equally well to services that develop targeted marketing campaigns or determine common (or uncommon) practices. In the financial sector, association approaches can be used to analyze customers' account portfolios and identify sets of financial services that people often purchase together. They may be used, for example, to create a service "bundle" as part of a promotional sale campaign. The association relations can also help in many business decision making processes, such as cross-marketing, category design

[1]This thesis does not distinguish the two terms itemset and pattern

[2]It is noticed that support of an itemset X is defined as $freq(X)$ in some research

and loss-leader analysis.

Besides, association rules and frequent itemsets can be applied for many other data mining functionalities. One of successful applications is classification. CBA [55] chose a special subset of association rules to build classifiers. CBA was shown to be, in general, more accurate than well known rule based classification system C4.5. Some other classification methods using association rules include CMAR [54] and [10]. Similarly, CAEP [33] built classifiers from emergent patterns [32], which can be regarded as special association rules.

In [39], clustering was done using association rule hypergraphs. Association rules have also been widely used in web mining and text mining. For example, frequent itemset mining was applied to build Yahoo-like information hierarchies [94]. In [28, 93], frequent itemset mining was used to mine common substructures from semi-structured datasets. [36, 84] mine text documents with the help of association rules. More applications include building intrusion detection models [60], recommending services in E-commerce[78], and mining sequence patterns [8, 98], etc.

1.1.2 Association rule mining algorithms

Generally, association rule mining is divided into two subtasks: (1) find all itemsets whose supports are no less than the user-specified minimum support. Such itemsets are called *large itemsets* or *frequent itemsets*; (2) generate association rules satisfying user-specified minimum confidence from the frequent itemsets.

Since step (1) is usually the most time consuming step in association rule mining, researchers mainly focused on frequent pattern mining. A phenomenal number of algorithms have been developed for frequent pattern mining, such as [2, 5, 6, 7, 17, 34, 40, 43, 45, 46, 57, 58, 68, 72, 74, 80, 82, 91, 97, 100, 103]. Many of these proposed frequent pattern (itemset) mining algorithms, such as [5, 6, 7, 17, 40, 45, 58, 68, 80, 82, 91, 103], are variants of Apriori [7]. Apriori employs a bottom-up, breadth-first search and uses the downward closure property of pattern support to prune the search space. Intuitively, the downward closure property means that all subsets of a frequent pattern must be frequent. Such a property is widely applied in frequent mining algorithms to prune the candidates whose subsets are not frequent. Some recently proposed algorithms do not strictly follow the downward property to prune the search space, but discover frequent patterns by extending patterns organized in an enumeration tree. The TreeProjection algorithm [2] proposes a database projection technique which explores the projected database with different frequent patterns. On the basis of projected database, some new algorithms are designed, such as FP-growth algorithm [43], H-Mine [72] and opportunistic projection [46].

The data mining research community has put great effort to develop efficient algorithms to discover frequent itemsets or association rules. In recent years, researchers have realized that the involvement of users in data mining process is vital for discovering only patterns that are interesting and relevant to the users.

One way of involving users in data mining is to allow the users to express their preferences for mining via constraints. One simple constraint is that the LHS of discovered rule contains item *beer*. (Note that minimum support and confidence can also be regarded as constraints specified by users.) [86] first introduced item constraints to produce only the useful patterns for constructing association rules. [63] classified the constraints to be imposed in association mining and proposed an effective solution for succinct constraints, anti-monotone constraints and monotone constraints. In a later work [64], more complicated constraints problems were investigated. [71] successfully integrated convertible constraints into some frequent pattern mining algorithms. The method reordered the items, which makes it possible to apply the anti-monotone and monotone properties to the convertible constraints. [19, 49] transformed the constraint problems into optimization problems.

The introduction of constraints into mining process is valuable in two aspects. On one hand, constrained mining tries to return knowledge that is of interest to users and it does not bother the users with uninteresting patterns or rules. On the other hand, constraints can be used to guide the mining process instead of being imposed after patterns or rules are discovered. In case that constraints are imposed to check whether return patterns or rules satisfy the given constraints, the performance cannot be improved since the computation has already been wasted in discovering uninteresting patterns. Researchers try to push constraints deep into association rules mining, improving the search efficiency by pruning the search spaces that do not satisfy the constraints.

Besides the introduction of constraints into data mining process, strengthening user interaction in mining process has also been studied. [63] proposed placing breakpoints in the mining process to accept user feedback to guide the mining. The idea was to divide the mining task into several sub-tasks and to place a breakpoint between two subtasks. The mining task can be adjusted at the breakpoint as early as possible if the user is not satisfied, thus avoiding unnecessary computation of the whole task. Although the idea seems to be promising in strengthening user interaction, it is difficult to operate in practice to divide association rule mining into smaller subtasks than two subtasks, mining frequent patterns and generating rules. Furthermore, online association rule mining [3] also allowed the user to make dynamic changes (with limitation) to the parameters of computation to improve interaction.

In addition to frequent itemsets, two other concepts, maximal frequent itemsets and frequent closed itemset, are proposed. The maximal frequent itemset is proposed under the background that existing frequent itemset mining algorithms usually degrade greatly when the discovered itemsets are long and are large in number although they usually show good performance in market basket datasets where the discovered itemsets are usually short. In many real applications, such as bio-sequences, census data, etc, finding long itemsets (the length can be more than 30) is not uncommon [12, 101] and has great requirements for both CPU and I/O. The maximal frequent itemsets are typically orders of magnitude fewer than all frequent itemsets and usually much faster than existing frequent pattern mining algorithms as shown in [1, 12, 20]. The maximal itemsets can help in understanding the datasets. But they can not be used to generate association rules directly since maximal itemset mining does not compute the frequency of subsets.

Unlike maximal frequent itemsets, frequent closed itemsets are lossless in that the frequencies of all frequent subsets can be obtained from frequent closed itemsets. Frequent closed itemsets are shown to be orders of magnitude fewer than frequent itemsets on some datasets (especially those dense datasets and datasets in which items are highly correlated), which results in faster algorithms [11, 70, 73, 92, 101]. The representative algorithms of finding frequent itemsets, maximal frequent itemsets and frequent closed itemsets will be reviewed in next chapter.

Although there are a large number of algorithms for frequent itemset mining (or maximal frequent itemsets or frequent closed itemsets), frequent pattern mining is still a time consuming computation, which is often beyond the users' expectation. The performance of algorithms often relies on the underlying datasets. None of these algorithms can outperform the others in all cases. This has been shown by the results reported in [38, 106][3].

Considering that frequent itemset mining is a time consuming process and the underlying database may change, some researchers have proposed some incremental mining methods [22, 35, 69, 88, 89] to utilize previous mining results when database is changed.

Considering that the number of association rules can be extremely large for users to understand, some researchers proposed to discover *interesting* or *optimized* association rules instead of all association rules. There are various definitions of interestingness according to different metrics, such as [13, 37, 50, 56, 77]. Some of these methods post-processed discovered association rules to prune those uninteresting rules while some integrated the pruning process into mining process to improve algorithm efficiency.

[3]In their experiments, the datasets are assumed to loaded into memory. It is still not clear of the performance comparisons of state of the art algorithms when dataset cannot fit in memory

This subsection gave a brief introduction of the research on association rule mining. The next subsection will discuss the motivations of this thesis.

1.2 Motivations

Association rule mining is an iterative process. For a given mining task, the user will set some initial constraints and run a mining algorithm. At the end of the mining, the user will check the results. In the case where the user is satisfied with the output, the mining task ends. In the case that the user is not satisfied with the results, the discovered knowledge will be abandoned and another round of mining is required after the user changes some constraints. The user often needs to run the mining algorithm several times before he/she is satisfied with the final results in most practical applications. The iterative process is illustrated with a frequent pattern mining task with only the *minimum support* constraint (also called the *frequency* constraint). The user may initially set the minimum support to 5% and run a mining algorithm. After inspecting the returned results, s/he finds that 5% is too high. S/he then decides to reduce the minimum support to 3% and runs the algorithm again. This process is usually repeated several times before s/he is satisfied with the final mining results.

This interactive and iterative mining process is very time consuming. Mining a dataset from scratch in each iteration is clearly inefficient because a large portion of the computation from previous mining is repeated in the new mining process. This results in enormous waste in computation and time. Iterative computation means that it is possible to integrate the consecutive iterations to speed up the computation. In other words, the subsequent iterations can turn to previous mining results besides mining algorithms. However, it is noticed that, so far, limited work has been done to address this problem. [89] mentioned the possibility that the incremental mining algorithm can be adapted to make use of previous computation to speed up new round of mining and [75] considered the change of minimum support in incremental mining. The careful analysis to be presented in Chapter 2 will show that it may not feasible to adapt existing incremental mining algorithms to handle the above problem.

As discussed in last subsection, there are many other constraints imposed in association rule mining besides minimum support and minimum confidence. On one hand, these additional constraints give the user more freedom to express his/her preferences. On the other hand, however, it often prolongs the mining process because the user may want to see the results of various combinations of constraint changes by running the mining algorithm more times. This makes mining using previous results for efficiency even more important.

In a multi-user data mining system (that may run on a peer-to-peer platform), exploiting and recycling frequent itemsets mined previously is more valuable. This is because users might share and recycle the mining results of other users that are mined under different constraints.

When a constraint imposed on association rule mining is *tightened*, e.g., minimum support is increased. In this case, it is straightforward to obtain the new set of frequent patterns under the new constraint by simply checking the frequent patterns obtained from the old mining to filter out the patterns that do not satisfy the new constraints. This filtering process is sufficient because the set of new frequent patterns is only a subset of the old set.

On the other hand, when constraints are *relaxed*, the problem becomes non-trivial as re-running the mining algorithm is needed to find the additional frequent patterns. For instance, if minimum support is decreased, more patterns may be generated. The problem becomes even more complicated when multiple constraints are changed at the same time. There is still no effective solution for this complex problem.

The first motivation of this thesis is *to examine how the previous mining results can be utilized to speed up re-mining when constraints are changed*.

The second motivation of this thesis is *to present some novel algorithms to effectively and efficiently mine frequent patterns, and thus association rules from the microarray datasets*.

Microarray gene expression profiling technology [81] provides the opportunity to measure the expression levels of tens of thousands of genes in cell simultaneously, which results in a large amount of high-dimensional data at both the transcript level and the protein level. These microarray datasets [4] typically have a large number of columns but a small number of rows. For example, many gene expression datasets may contain up to tens of thousand of columns but only tens or hundreds of rows. The high-dimensional datasets also become popular in scientific datasets, census and text datasets.

In [29], association rules are discovered from microarray data to find associations between different genes as well as genes and their environments/categories (for instance cancer cell), thus helping gene pathway regulations. The associations between genes can discover the knowledge of how a particular gene is affected by other genes. The associations between genes and categories can describe what genes are expressed as a result of certain cellular environments, thus providing great help in the search for gene predictors of the sample categories. [29] also suggested to construct gene network from association rules. Moreover, frequent patterns are expected to be useful for clustering and bi-clustering microarray datasets and [105] applied frequent pattern mining algorithms in mining bi-clusters.

After introducing the usefulness of association rules and frequent patterns in microarray datasets, we now examine the problems of discovering association rules from microarray data. Most of state of the art algorithms for association rule mining or frequent pattern mining usually work well when the average number of items in each transaction (row) is small (the number is usually less than 100). However, they do not scale well with high-dimensional datasets and are not practical to mine such datasets. Even

[4]Each column of the original microarray datasets represents the expression level of a gene, which is continuous value. In this thesis, the continuous value will be discretized.

the algorithms of mining closed itemsets face the same dilemma. The intuitive reason for the poor performance of existing techniques on high-dimensional datasets is that they usually search frequent (closed) patterns in the item enumeration space, which is very large for microarray data. The detailed analysis will be discussed in Chapter 4. [66] proposed to mine frequent closed patterns in row enumeration space [5] and proposed algorithm CARPENTER. Considering that many algorithms have been proposed to mine frequent (closed) patterns by item enumeration, it would be interesting to investigate whether some ideas can be borrowed from these algorithms to search row enumeration space more efficiently.

Although the proposed frequent pattern mining algorithms in this thesis are usually much faster than existing algorithms, mining all association rules satisfying minimum support and minimum confidence is still time consuming sometimes. Moreover, due to high dimensions of microarray data and the combinatorial explosion of frequent itemsets (from which rules are generated), it is not uncommon to have billions of rules generated, in which many are redundant. The huge discovered rules are beyond the ability of human to understand. Fortunately, it is noticed that people may often be interested in rules with given consequent, which will be much smaller in number than all rules. For instance, recent studies have shown that association rules with given class label RHS are very useful in the analysis of microarray data. For example, it is shown in [33, 53] that classifiers built from association rules are rather accurate in identifying cancerous cell. But fixing the consequent of rules only partly alleviates but not solve the above problems, i.e., the huge number of rules and long runtime. Another motivation of this thesis is *to efficiently discover interesting rules for given consequent from micorarray data*.

[5][34] called the counting method in Apriori algorithm [7] as row-wise counting. [34] also proposed to count supports of candidates by intersecting of the list of row ids of items in a candidate and called this counting method as column-wise counting. Both row-wise and column-wise in [34] have different meaning with the item enumeration and row enumeration in this thesis and their methods are completely different from the proposed methods in this thesis

1.3 Contributions

Based on the above gaps identified in current research of association rule mining, this thesis will present an extended framework for mining and recycling frequent itemsets. Two open problems in the framework will be addressed in this thesis. One is to recycle and reuse previous mining results. The other is to efficiently mine frequent patterns and interesting association rules with given consequent from microarray datasets.

Extended framework for association rule mining. The proposed framework is designed for association rule mining in the environments of individual user or multiple users. There are two main features of the framework. One is to choose appropriate algorithms according to the properties of datasets to be mined. The other is to recycle previous mining results for new round of mining if there exist such previous mining results.

Recycling frequent patterns with constraint changes. This thesis will present two novel techniques to solve the problem of recycling previous patterns for new rounds of mining when (individual or multiple) constraints are changed.

Using the relaxation of frequency constraint (the decrease of *minimum support*) as an example, the first technology is based on the proposed concept of *tree boundary* to summarize and to reorganize the previous mining results which include both final mining results and some intermediate results. The additional frequent patterns can be generated in the new mining process by extending only the patterns on the *tree boundary* without re-generating the frequent patterns produced in the previous mining. The proposed technique has been implemented in the contexts of two frequent pattern mining algorithms, FP-tree [43] and Tree Projection [2]. This results in two augmented pattern mining algorithms RM-FP (re-mining using FP-tree) and RM-TP (re-mining using Tree Projection). Extensive experiments on both synthetic data and real-life data are conducted to compare the performance of RM-FP and FP-tree algorithm, as well as RM-TP and Tree Projection algorithm. Finally, it is also addressed how the proposed technique can be applied to handle the changes of other types of constraints given in previous studies [63, 71]. This work is published in [24].

The first technique utilizes both the frequent patterns and some intermediate results from previous mining while the latter may not be available sometimes. For example, in a multi-user environment, one user may not be willing to output the intermediate results for other users to recycle since it will take extra time and space to output these intermediate results, which may not be useful for the user him/herself.

The second proposed recycling technique only utilizes frequent patterns. The proposed technique uses the frequent patterns generated from previous mining to compress the datasets to be mined. And the new round of mining is done on the compressed datasets. Two compression strategies are designed. While the first attempts to minimize cost(computation cost measured by runtime), the second minimizes storage space. The strategy of minimizing cost is novel in that a function is designed to estimate the potential saving of using a pattern to do the compression for subsequent mining. The strategy of minimizing storage space is relatively straightforward. Three existing frequent pattern mining algorithms, H-Mine [72], FP-tree [43] and Tree Projection[2], are adapted to mine the compressed databases. Extensive experiments are conducted to compare the recycling algorithms with their counterparts as well as the effectiveness of two compression techniques for recycling patterns. This work is published in [25].

New mining algorithms for microarray datasets. This thesis will present three algorithms, CARPENTER [6], RERII and REPT, to mine frequent closed patterns for microarray databases with a large number of columns and a small number of rows, thus obtaining all association rules or using the discovered patterns for other tasks as discussed in last section. Unlike the traditional frequent (closed) pattern mining algorithms that mine the datasets in column enumeration strategy, the proposed techniques adopt row enumeration strategy. With the change, the well-known pruning techniques, such

[6]Feng Pan played a primary role in developing CARPENTER algorithm

as downward property, cannot be used in the proposed technique. Instead, a series of new pruning techniques are introduced to improve the efficiency of the proposed algorithms. The three algorithm explore various possibilities of implementing row enumeration approach. The proposed algorithms are tested on some publicly available microarray datasets to compare its efficiency with the performance of some other established approaches for frequent pattern mining. The basic idea of row enumeration and algorithm CARPENTER are published in [66].

The number of association rules generated from microarray datasets is usually enormous. The concept of interesting association group is proposed to greatly reduce the number of discovered rules while keeping the necessary information of all interesting rules. A rule group is identified by its unique upper bound and a rule group has lower bounds. The concept will be introduced in Chapter 5. Algorithm FARMER is designed to mine interesting association rule groups with given RHS from microarray datasets. In FARMER, the basic idea of row enumeration is combined with efficient search pruning strategies based on user-specified thresholds (minimum support, minimum confidence and minimum chi-square value), yielding a highly optimized algorithm. The algorithm FARMER is compared with the existing rule mining algorithms in terms of efficiency on real-life biological datasets. This work is published in [27].

1.4 Outline

In the next chapter, state of the art algorithms will be reviewed. In Chapter 3, a framework for mining and recycling frequent itemsets is presented. In Chapters 4 and 5, two methods of recycling previous frequent patterns are proposed. Three algorithms that mine frequent closed itemsets from microarray datasets are described in Chapter 6. The algorithm FARMER that mines interesting rule groups from microarray datasets is presented in Chapter 7. This thesis is concluded in Chapter 8.

Chapter 2

State of the Art

This chapter will first introduce some preliminary of association mining algorithms in Section 2.1, then present the frameworks of association rule mining in Section 2.2, and state of the art of association rule mining algorithms in Section 2.3.

2.1 Preliminaries

The concepts of association rules, maximal frequent itemsets and closed frequent itemsets will be given first and then two kinds of dataset layouts are explained.

Formally, the association rule mining problem can be stated as follows. Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items. Let **dataset** [1] D be a set of variable length **transactions**, where each **transaction** t (or **row**) is a set of **items** (or **columns**) such that $t \subseteq I$. An association rule is an implication of the form $X \rightarrow Y$, where $X \subset I, Y \subset I$ and $X \cap Y = \emptyset$. X is called the antecedent and Y is called the consequent of the rule. In general, a set of items (such as X and Y) is called an **itemset** (or **a pattern**). The number of items in an itemset is called the length of the itemset. Itemset with length k is called k-itemset (or k-pattern). Note that this thesis does not distinguish *itemset* and *pattern* as well as *item* and *column*.

[1] In this thesis, boolean database is considered as underlying dataset

TID	Items
100	a,b,c,d,e,f
200	a,b,d,h
300	a,b,d,e,f
400	a,b,c
500	c,e

Table 2.1: The example database *DB* in horizontal layout.

Each itemset has an associated support that is a statement of statistical significance or generality. For an itemset $X \subset I$, its support is $s = sup(X)$ if the percentage of transactions that contain X equals to s. Each rule has an associated confidence that is a statement of strength or predictive ability. For a rule $X \rightarrow Y$, its confidence is defined as $c = sup(X \cup Y)/sup(X) \times 100\%$, which means that the percentage of transactions in D that contain X also contain Y.

Definition 2.1.1. Mining association rules

The problem of mining association rules is to generate all rules that have support and confidence greater than some user specified minimum support,, denoted by ξ, and minimum confidence, denoted by $minconf$, thresholds respectively.$\square$

The algorithms for mining the complete association rules usually work in the following two steps:

1. Generate all **frequent** (or **large**) itemsets. An itemset is called *frequent* (or *large*) if it has support above the user specified *minimum support*.
2. Generate all association rules that satisfy *minimum confidence* threshold using the frequent itemsets.

Example 2.1.1. Given the database as shown in table 2.1, minimum support $\xi = 80\%$ and minimum confidence $minconf = 80\%$. The set of frequent itemsets is $\{a : 4, b : 4, c : 3, d : 3, e : 3, ab : 4, ad : 3, abd : 3\}$. Symbol ":" separates an itemset and its associated support. For simplicity, we use abd to represent itemset $\{a, b, d\}$. The same is for itemsets ab and ad. The set of association rules includes $\{a \rightarrow b : 100\%, b \rightarrow a : 100\%, d \rightarrow a : 100\%, d \rightarrow ab : 100\%\}$. As a comparison, the rule $\{a \rightarrow d : 75\%\}$ does not satisfy the minimum confidence threshold.$\square$

The second step, i.e. generating rules that have minimum confidence given the set of frequent itemsets and their supports is relatively straightforward. The general idea given in [7] is as follows: given a frequent itemset X and any $Y \subset X$, if $sup(X)/sup(X - Y) \geq$ *minimum confidence*, then the rule $(X - Y) \rightarrow Y$ satisfies the minimum confidence. However, the first step, i.e. discovering all frequent itemsets, is a non-trivial problem. Nearly all proposed algorithms of mining association rule focus on the first step while using the method in [7] for the second step. The algorithms introduced in the remaining chapter use the method in [7] for the second step if there is no special specification.

Since the discovered frequent patterns may be huge in number, maximal frequent patterns and closed frequent patterns are proposed as two complementary concepts of frequent patterns.

Definition 2.1.2. maximal frequent pattern

Given database D, a frequent pattern X is called as a maximal pattern if there exists no frequent pattern Y such that $X \subset Y$.$\square$

Definition 2.1.3. closed frequent pattern

Given database D, a frequent pattern X is called as a closed frequent pattern if there exists no frequent pattern Y such that $X \subset Y$ and $sup(X) = sup(Y)$.$\square$

Example 2.1.2. Continue with the last example, itemset $\{a, b, d\}$ is a maximal frequent pattern while $\{a, b\}$ and $\{a, d\}$ are not because they are subsets of $\{a, b, d\}$. Itemset $\{a, b, d\}$ is also a closed frequent itemset while $\{a, d\}$ is not. Here itemset $\{a, b\}$ is also a closed frequent itemset while it is not a maximal frequent itemset.$\square$

Item	TID-list
a	100,200,300,400
b	100,200,300,400
c	100,400,500
d	100,200,300
e	100,300,500
f	100,300
h	200

Table 2.2: The example database *DB* in vertical layout.

Maximal frequent patterns can be regarded as the boundary of all frequent patterns and the frequency information of other frequent patterns cannot be inferred from the maximal frequent patterns. However, the closed frequent patterns are information lossless and the frequency information of all frequent patterns can be inferred from them. Closed frequent pattern is a way of avoiding the discovery of a large set of redundant patterns as explained in [70].

Horizontal layout data and vertical layout data. Both kinds of data layouts are widely adopted in existing association mining algorithms. One example of horizontal layout dataset is shown in Table 2.1, where each row of database represents a transaction which has a transaction identifier (TID), followed by a set of items. The vertical layout of dataset in Table 2.1 is illustrated in Table 2.2. In a vertical layout dataset, each tuple corresponds to an item, followed by a TID list[2], which is the list of rows that the item appears. For clarity, we call row for a record in horizontal data layout database while we call $tuple$ for a record in a vertical layout database.

2.2 Frameworks

As discussed in Section 2.1, association rules mining algorithms usually work in two steps, i.e. first finds frequent itemsets, and then derives association rules from frequent

[2]The TID list can also be represented in bit vector in some algorithms, such as[82]

itemsets. The two phases form the framework of most of existing association rule mining algorithms. [63] suggests to place break points in a whole mining task. In an association rule mining task, break point can be placed between the two steps. Only if users are satisfied with the returned frequent itemsets, the next step i.e. generating association rules is initiated.

There are some studies that do not strictly follow the two phase framework. OPUS [95] searched association rules directly. In [95], it was shown that OPUS is good to mine top-k association rules but its performance is poor when mining all association rules. The study in [99] proposed a framework to discover non-redundant rules using frequent closed itemsets (rather than frequent itemsets) although it still follows the two phase framework. Research in [37, 77] aimed to discover one optimized association rule from dataset with numeric attributes.

Constrained association rule mining extends the two step framework of association rule mining. Some constraints in [63, 71, 86] are introduced for the two-step association rule mining. Users can express their preference by means of constraints. These constraints can be divided into the following categories in previous studies [63, 71, 86].

- If a constraint C_a is *anti-monotone*, for any pattern S that does not satisfy C_a, none of the supersets of S satisfies C_a.
- If a constraint C_m is *monotone*, for any pattern S that satisfies C_m, every superset of S also satisfies C_m.
- If a constraint C_s is a *succinct* constraint, we can succinctly characterize the set of all patterns that satisfy C_s and only generate patterns that satisfy C_s.
- Given an order R over the set of items I, if a constraint C_{ca} is a *convertible anti-monotone* constraint w.r.t. R, for any pattern S that violates the constraint C_{ca}, any pattern that has S as its prefix w.r.t. R also violates C_{ca}. *Convertible monotone* constraint is defined similarly.

Constraint	Anti-monotone	Succint	Monotone
$v \in S$	no	yes	yes
$S \supseteq V$	no	yes	yes
$S \subseteq$	yes	yes	no
$min(S) \leq v$	no	yes	yes
$min(S) \geq v$	yes	yes	no
$max(S) \leq v$	yes	yes	no
$max(S) \geq v$	no	yes	yes
$count(S) \leq v$	yes	weakly	no
$count(S) \geq v$	no	weakly	yes
$sum(S) \leq v, \forall a \in S, a \geq 0$	yes	no	no
$sum(S) \geq v, \forall a \in S, a \geq 0$	no	no	yes
$range(S) \leq v$	yes	no	no
$range(S) \geq v$	no	no	yes
$avg(S)\theta v, \theta \in \{\leq, \geq\}$	conv.	no	conv.
$range(S) \geq v$	yes	no	no

Table 2.3: The representative constraints

- For any constraints that do not have any above property are classified as *hard* constraints since they are hard to be pushed into frequent pattern mining algorithms effectively.

As a summary, Table 2.3 lists the representative subsets of constraints introduced in [63, 71, 86].

Incremental mining technique further extends the framework of association rule mining. Incremental mining algorithms are developed to update the set of discovered association rules when one of the following cases occurs or both cases occur: (1) new tuples are inserted into the database; (2) existing tuples are deleted from the database.

The earliest proposed incremental mining techniques are based on Apriori-like algorithm. FUP in [22] is a straightforward extension of Apriori algorithm for incremental mining. If an itemset is counted in old dababase DB and is a candidate in the updated database DB', FUP only needs to count the frequency of the itemset in $DB' - DB$. Therefore, the saving of FUP comes from the frequencies of itemset counted in DB. FUP method follows the Apriori framework and may need $O(n)$ passes over the data where n is the size of the maximal frequent itemset.

Some incremental mining methods [35, 69, 88, 89] utilize the concept of negative border (proposed in [90]) to update the mining results when additional data becomes available. A negative border consists of all the itemsets that are candidates of the Apriori algorithm that do not have sufficient support. [35, 88, 89] utilize negative borders as follows: (1) from frequent itemsets LDB and its negative border $NB(LDB)$ for old dataset DB, the algorithm obtains the itemsets LDB' that are frequent under the updated dataset DB'; (2) it then computes the negative border $NB(LDB')$; (3) it sets $LDB' = LDB + NB(LDB+)$, and goes to step (2) until $NB(LDB') = LDB'$; (4) it counts the supports of the itemsets in $NB(LDB')$ but not in LDB and $NB(LDB)$. Although the methods in [35, 69, 88, 89] only need one scan of the updated dataset, they could not avoid the disadvantage of negative border, i.e., maintaining a negative border is very memory consuming and is not well adapted for very large databases [69].

One inherent assumption of the above incremental techniques [22, 35, 69, 88, 89] is that the new set of frequent patterns after database is changed mainly comes from the set of frequent patterns and infrequent patterns counted before database is changed. The assumption usually can be true because the updating of database is usually small compared with the original database. As a result, existing incremental techniques rely on previous frequent patterns and the infrequent patterns computed previously.

DELTA [75] is another incremental method that is based on negative border. Unlike the previous incremental technique based on negative border, DELTA does not generate the complete negative border as candidates, which can be very large in number. Instead, DELTA first counted the set of 2-itemset and then generated negative border for the itemsets longer than 2. DELTA usually scanned the incremental database three times and the original database once. Because it was observed that the set of 2-itemset is typically much smaller than the overall number of all possible 3-itemsets pairs, it is expected that the generated negative border is much smaller than that generated by previous methods.

Another contribution of DELTA [75] is that it considered the case when the distribution of the update is skewed compared with original database. In such case, the experiments in [75] showed that the algorithms [35, 88] nearly cannot achieve saving comparing with running Apriori from scratch while DELTA can still improve performance.

As discussed in Chapter 1, association rule mining is an iterative process, especially the constrained association rule mining. Previous association rule mining frameworks usually fail to utilize the previous mining results to speed up subsequent rounds of mining when constraints are changed in the iterative mining process.

The concept of reusing previous frequent patterns has already been used in incremental mining. However, our concept is different in two aspects: 1) we consider to recycle and reuse of previous frequent pattern when constraints are changed between two data mining iterations instead of dataset change in the incremental mining; 2) we also consider the case that one user recycles and reuses the mining results from the other users in a data mining system.

With the similarity between our proposed concept and incremental mining, can the existing incremental mining algorithms be adapted to efficiently handle the recycling problem when constraints are changed? [89] actually mentioned the possibility of using negative border [90] but no detailed algorithm was proposed. However, one significant shortcoming of the negative border based incremental mining approach is that generating candidates under new constraints using the negative border under old constraints usually results in over-generation of a huge number of useless candidates (although the dataset is only scanned once). This makes the approach in [35, 88] impractical for our constraint relaxation problem for large datasets, especially when the number of frequent patterns is large. For example, if 10^5 frequent patterns are obtained given minimum support of 1% and 50 1-itemsets become frequent after minimum support is reduced to 0.9%, the number of candidate itemsets generated using the above approach is $(2^{50}\text{-}1)*10^5 \approx 10^{20}$ even if we do not consider the expansions of 10^5 frequent itemsets themselves. This is clearly impractical.

DELTA [75] tried to extend the framework of incremental mining by considering that the minimum support may be changed after database is updated. The method is established on the assumption that the frequent 2-itemset is only a small fraction of 2-itemset candidates and it becomes possible to generate the negative border when the set of frequent 2-itemsets is known. However, the assumption may not always hold and it is very common that the itemsets with length larger than 2 is very large in number, thus generating huge number of candidates as analyzed in last paragraph.

FUP is not designed for mining with constraint changes. If it is applied to our task, it basically re-runs the Apriori algorithm without re-counting the supports of the itemsets generated previously (they still need to be re-generated). Moreover, FUP [22] is criticized that it may need $O(n)$ passes over the data where n is the size of the maximal frequent itemset. There, the computation saving of adapting FUP to recycle previous results with constraint change is thus very limited, if any, because of some overheads to check which itemsets have already been counted.

2.3 Algorithms

This section will introduce and compare some representative algorithms of mining association rules and frequent itemsets. Finally, algorithms of finding interesting or optimized rules are introduced.

2.3.1 Apriori and Apriori-like algorithms

Apriori. Apriori [7] is the first algorithm that efficiently fulfills the task of mining all association rules satisfying minimum support and minimum confidence from large databases. Apriori algorithm makes multiple passes over the database for discovering frequent itemsets. In the first scan, all frequent singleton items together with their counts are obtained. The set of singleton items are used to generate new potential frequent 2-itemsets, called **candidate** itemset and one more database scan is required to compute the support of these candidates of size 2 to determine whether they are frequent. Then from frequent 2-itemsets, candidates of size 3 are generated and evaluated. The process continues until no new frequent itemsets are found.

The efficiency of Apriori algorithm is based on a simple but effective observation that **any subset of a frequent itemset must be frequent**. Such a property is called as **downward closure property**. With the property, any candidate that contains a subset that is not frequent can be removed and there is no need to count its support. More specifically, Apriori generates k-candidates by joining the $(k-1)$-frequent itemsets; a k-candidate is pruned off if any of its $(k-1)$ subsets is not frequent. The process can be explained by the following example. Given the set of frequent 2-itemsets $\{a, b\}, \{a, c\}, \{a, d\}, \{c, d\}$, by joining 2-itemsets, the set of potential candidates of size 3, $\{a, b, c\}, \{a, c, d\}$ can be generated. Because that the subset $\{b, c\}$ of $\{a, b, c\}$ is not frequent, $\{a, b, c\}$ surely cannot be frequent and is pruned from the set of candidates. In this way, Apriori algorithm avoids wasting computation in counting the itemsets that must not be frequent by judging from their subsets.

The above property is also utilized in [58], a concurrent work of Apriori algorithm[7] and, to some extent, in nearly all subsequent frequent pattern mining algorithms. After Apriori algorithm [7], some subsequently proposed algorithms adopt similar database scan level by level while the methods of candidate generation and pruning, support counting and candidate representation may differ. We classify these algorithms as Apriori-like algorithms and will describe them briefly with a stress on their difference with Apriori algorithm.

Instead of using a hash-tree to represent candidates for support counting as proposed in [68], some efficient and well-known implementations of Apriori algorithm, such as [16], adopt prefix tree data structure to represent candidates for generating and pruning candidates as well as counting supports.

DHP. It was observed in [68] that to determine frequent itemsets from a huge number of candidates is one of the dominating factors for the performance of association rule mining. A hash-based technique DHP [68] was proposed to further reduce both the number of candidates in the early passes of database and the size of database, thus improving performance. It was shown that the the number of candidates of size 2 generated by the hash-based method can be orders of magnitude smaller than that of Apriori algorithm according to the experimental results [68].

Partition. The Partition algorithm [80] computed all frequent itemsets only in two passes over the database. In the first scan of database, the algorithm partitioned the database into sections that are small enough to be handled in main memory. Each partitioned database was loaded into memory and mined level by level to obtain the locally frequent itemsets in the partitioned database. All locally frequent itemsets made up the candidates of global frequent itemsets. The second scan over the database would compute the global frequency of each locally frequent itemset. Another originality of Partition algorithm is its method of counting support. Partition algorithm associated each frequent itemset with a *TID list*, which is the list of transaction (or row) ids that contain the itemset, and computed the frequencies of itemsets by *TID-list* join.

Sample. Sample [90] reduced the I/O overhead of association rule mining from a different angle. It aimed to only scan database once to generate all frequent itemsets. It worked by first finding all association rules that probably hold in the given database by mining a random sample of the database using a reduced minimum support threshold, and then verifying the results with the rest of the database.

DIC. DIC [18] algorithm also tried to improve performance by reducing the number of database scans. DIC utilized the downward closure property for candidate pruning and adopted similar counting method as Apriori algorithm does. Two tricks were used in DIC in order to reduce the number of database scans. Unlike most of previous algorithms that mine frequent itemsets strictly level by level, DIC maintained a set of candidates that can have different sizes and dynamically changed them when scanning a database. The underlying database was divided into several sections. Unlike Apriori algorithm that generates candidates for next-level scan at the end of one scan, DIC detected a new candidate (i.e. all its subsets are frequent so far) after scanning one section, and started to count frequency of the new candidate at the point of the scan. In this way, DIC could reduce the number of complete database scans.

Comments. It is noted in our study and also in other studies, such as [43], that the size of candidates can be the memory bottleneck and even the performance bottleneck of Apriori algorithm. The number of database scans is the other factor that dominates the performance of Apriori algorithms. DHP[68], Partition[80], Sampling[90] and DIC[18] algorithms are variants of Apriori. DHP tries to reduce the number of candidates while the other three algorithms address to reduce the number of database scans.

DHP [68] adopted a hash-based technique to reduce the number of candidates. The technique was shown to work in the early scans, especially the second level. However, the hash technique may deteriorate the performance of later levels. Therefore, DIC algorithm still followed the Apriori algorithm to generate candidates in later level. For datasets of market basket type, the number of candidates of size 2 is often the largest of all sizes and they become the memory bottleneck. The hash-based technique of DHP can relieve the problem although it cannot solve it completely. But in some other datasets as used in [106, 12], it is not the case and the length of candidates of the largest size is hard to tell in advance. It is not clear whether DHP can improve the performance of Apriori algorithm in this case.

Another originality of the DHP algorithm is to progressively reduce the size of database for subsequent database scans. But this will result in additional I/O overhead.

We find that algorithms Partition, Sampling and DIC usually generate more candidates than Apriori algorithm does although they are able to reduce the number of database scans. As a result, if the time used to compute the frequencies of these extra candidates is more than the saving in database scans, these algorithms will preform worse. In many dense datasets used in [12, 20], generating the large number of candidates and computing their frequencies are the bottleneck of performance while the database scans only take a small part of runtime. In this case, Partition, Sampling and DIC may make the problem worse because they consider more candidates than Apriori algorithm does.

The performance of the three algorithms relies on an implicit assumption that the database is homogenous and thus they will not generate too many extra candidates than Apriori algorithm does. For example, if all partitions in Partition algorithm are not homogenous and nearly completely different sets of local frequent itemsets are generated from them, the performance cannot be good. The effect of data homogenization on performance is noticed in DIC algorithm. The datasets used by most of these studies are synthetic datasets generated by the IBM data generator, which are usually homogenous. However, the real-life datasets may display different properties and only DIC studied performance on real-life datasets while the Partition and Sample's performance on real-life datasets are not studied.

Of all the algorithms described above, Partition utilizes the vertical layout data while all others mine association rules from horizontal layout data. Some algorithms [34, 82, 102] are proposed to mine association rules from vertical layout data, which is the topic of next subsection.

2.3.2 Mining from vertical layout data

This section will introduce some algorithms that successfully mine vertical layout database, such as the Eclat and MaxClique algorithms [102] and VIPER algorithm [82].

Eclat and MaxClique. Eclat and MaxClique utilized maximal frequent itemsets in discovering frequent itemsets although the two methods did not aim to discovering maximal frequent itemsets. The Eclat and MaxClique used two different methods to generate potential maximal candidates, identified true maximal frequent itemsets and computed the supports of their subsets by means of TID list intersection. The two methods also explored the orders of top-down, bottom-up and their hybrid to compute frequent itemsets. The limitation of two methods is the memory usage. The frequent itemsets together with their TID lists will make two methods infeasible especially when there are a large number of frequent itemsets and database has a large number of transactions.

VIPER. VIPER [82] represented TID list with compressed bit-vector. The representation, together with some optimization of storage, generation and intersection of TID list, formed the core of VIPER algorithm. Although VIPER claims that there are no special requirements for underlying database, it still has several limitations. First, it assumes that candidates of size 2 are the most large in number and TID list intersection can not be effective to generate frequent 2-itemsets. VIPER uses similar method as Apriori to generate frequent 2-itemsets and has the same problem that the candidates of size 2 cannot fit in memory. Moreover, it is possible that the number of candidates of size 2 is not the largest while the number of candidates of larger size, e.g 10, is.

2.3.3 Projected database based algorithms

In Tree Projection algorithm[2], the concept of projected database was proposed and applied to frequent itemset mining. FP-tree growth and H-Mine also used the projected database as underlying framework. But they proposed original data structures to represent projected databases. These methods do not strictly follow the Apriori-like algorithms. For example, if itemsets $\{a, b\}$ and $\{a, b\}$ are frequent, itemset $\{a, b, c\}$ will be counted while Apriori-like method will check whether $\{b, c\}$ is frequent before counting the support of $\{a, b, c\}$ (if $\{b, c\}$ is not, $\{a, b, c\}$ is not counted). Therefore, the projected database algorithms usually compute frequencies for more itemsets than Apriori-like algorithm does. However, the projected database algorithms usually are more efficient in counting support since they use divide-and-conquer method. As experiments showed in [2, 43, 72, 46], these methods usually perform better than earlier algorithms. Before introducing these algorithms, we first explain the concept of projected database.

Definition 2.3.1. Projected database

Projected database for an itemset X are composed of the set of records (or rows) containing X.□

For example, given the database shown in table 2.1, the projected database of itemset $\{a, b\}$ is $\{100 : abcde, 200 : abdh, 300 : abdef, 400 : abc\}$. The symbol ":" separates a record ID and a record.

The size of projected database of X can be further reduced by removing the items that will not be useful in extending X to get longer frequent itemsets. For example, the projected database of itemset $\{a, b\}$ can be reduced to $\{100 : d, 200 : d, 300 : d, 400 : \emptyset\}$ given the minimum support ξ =3.

Tree Projection. Tree Projection algorithm [2] represented frequent itemsets as nodes of a lexicographic tree [3]. The lexicographic order usually followed the ascending order of frequency of items for better performance. All frequent items formed the first level nodes of a lexicographic tree.

Tree Projection algorithm explored two kinds of orders of generating frequent itemsets: breadth-first and depth-first. For breath-first order, Tree Projection algorithm generated frequent itemsets by successive construction of nodes of a lexicographic tree level by level. In order to compute frequencies of nodes (corresponding frequent itemsets) at k level, tree projection algorithm maintained matrices at nodes of the $k-2$ level and one database scan was required for counting support. Transactions were projected on each node of the tree from the root on one by one. The reduced set of transactions after being projected were used to compute frequencies, thus improving efficiency. For depth-first order, databases were required to fit in memory and projected along the paths of a lexicographic tree. The advantage is that the projected database will become smaller along the branch of the lexicographic tree while the breadth-first needs to project the database from the scratch at each level. The disadvantage of depth-first is obvious that it needs to load database and projected databases in memory. The breadth-first method will also meet the memory bottleneck when the number of frequent items is large and the matrix is too large to fit in memory.

FP-tree. In the FP-tree algorithm[43], the set of frequent individual items was discovered first and items in the set were sorted based on their frequencies in the descending order to form a list, denoted by $F\text{-}list$. Then, the dataset was scanned to construct a frequent pattern tree (or FP-tree in short), which was a prefix tree. A FP-tree represented compressed but complete frequent itemset information in database. Each path of the FP-tree followed the order of $F\text{-}list$. The transactions with the same prefix shared the portion of the path from a root. The FP-tree was explored with a pattern fragment

[3]More explanation of lexicographic tree can be found in Chapter 4

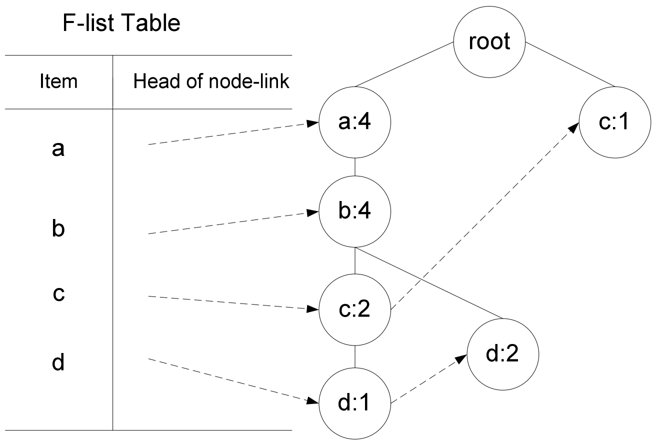


Figure 2.1: Example of FP-tree

growth method to discover all frequent itemsets.

Example 2.3.1. Given the database as shown in Table 2.1, minimum support ξ = 3. The database is represented with FP-tree in Figure 2.1.□

Definition 2.3.2. Conditional pattern base

For an itemset X, the set of prefix paths of X forms the conditional pattern base of X which co-occurs with X.□

Definition 2.3.3. Conditional FP-tree

The FP-tree built on the conditional pattern base of X is called the conditional FP-tree of X and denoted by $FP|_X$. Conditional FP-tree of X contains the complete information of candidate extensions of X.□

The FP-tree algorithm worked by (1) identifying the list of frequent items F-$list$ in the dataset, (2) for each item α in F-$list$, constructing its conditional pattern base to mine the set of frequent items E_α, and constructing conditional FP-tree $FP|_\alpha$ with items in E_α, and (3) mining frequent itemsets beginning with α on $FP|_\alpha$ by recursively constructing conditional FP-tree for its descendant itemsets.

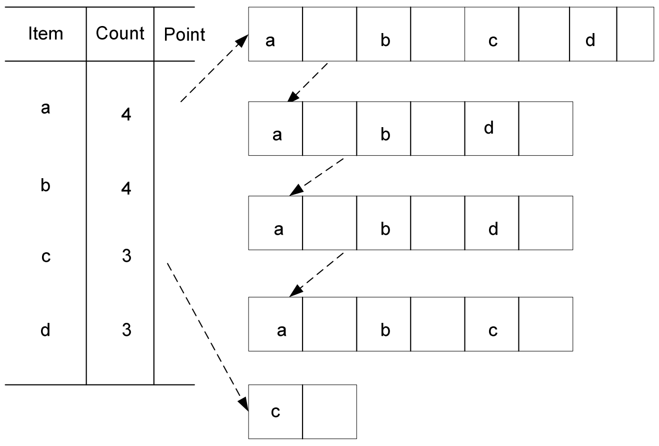


Figure 2.2: Example of data structure of H-Mine

H-Mine. H-Mine algorithm adopted in-memory pointers [72] to construct projected databases. This is different from FP-tree and Tree Projection that physically constructed projected databases. The advantage of in-memory pointers is that the projected database itself do not need memory and the extra memory usage is only for the set of in-memory pointers. The disadvantage is that the size of projected database cannot be further reduced by removing those useless items for subsequent mining.

Example 2.3.2. Given the database as shown in table 2.1, minimum support $\xi = 3$. The database is represented with the data structure of H-Mine in Figure 2.2. It is obvious that we can get the a-projected database by following the pointers. After a-projected database is mined, the tuples are assigned to *b*, *c* and *d* projected databases.□

As discussed in Chapter 1, when the discovered frequent itemsets are long (more than 15 to 20 items), the number of all frequent patterns may be too large so that algorithms introduced above may become infeasible. Maximal frequent patterns and frequent closed patterns are proposed to address the problem.

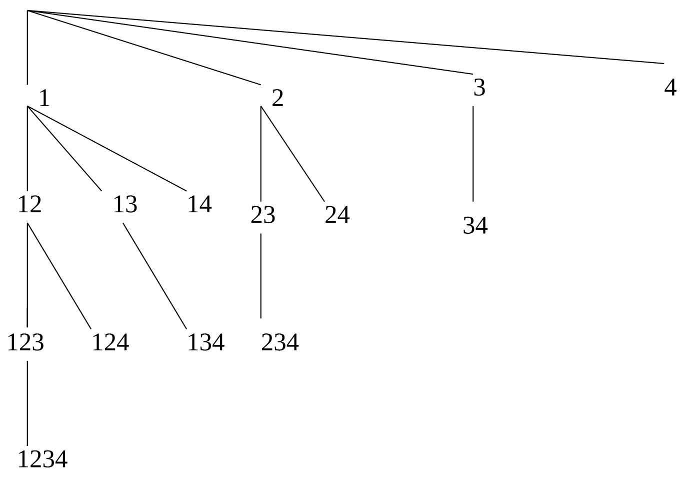


Figure 2.3: Column enumeration space of four items

2.3.4 Maximal frequent pattern mining

This section will introduce Max-Miner[12], the first maximal frequent pattern mining algorithm, DepthProject[1] and MAFIA[20].

Max-Miner. Figure 2.3 shows the itemset enumeration tree representing all itemsets over $\{a, b, c, d\}$. Similar to Apriori algorithm, Max-Miner [12] enumerated frequent itemsets level by level. However, at each node, Max-Miner also computed the frequencies of the itemset that is the union of all its children. For example, at node $\{b\}$, the support of itemset $\{b, c, d\}$ is counted together with $\{b, c\}$ and $\{b, d\}$. If itemset $\{b, c, d\}$ is frequent, all subsets of $\{b, c, d\}$ are frequent. Then there is no need to explore the node below $\{c\}$. Max-Miner also explored a lot of optimizations to improve efficiency, such as item ordering.

DepthProject. DepthProject algorithm was similar to a depth-first version of Tree Projection algorithm. DepthProject algorithm required the database to be loaded in memory. DepthProject algorithm represented database with bitstring, which can save memory usage and speed up counting computation. DepthProject algorithm used some heuristics to determine at which nodes to maintain projected databases.

MAFIA. MAFIA algorithm mined maximal frequent patterns using vertical layout database. Similar to other maximal frequent pattern mining algorithm, MAFIA also needed the database to be loaded in memory. MAFIA explored different alternative techniques proposed by previous research on mining vertical layout databases and chose the best combination to form the MAFIA algorithm.

2.3.5 Frequent closed pattern mining

Close and Pascal. Close [70] and Pascal [11] are two algorithms which discover closed patterns by performing breadth-first, column enumeration. Close [70] is an Apriori-like algorithm that first found all *generators* which were the smallest frequent patterns that determine a closed itemset. After finding all frequent patterns of length k, the support of a pattern was compared to that of its subsets. A pattern was removed if it had the same support with any of its subsets since it was proven that such a pattern cannot be a generator. In the second step, Close computed the closure of all generators by intersecting the set of rows that contained the generators. Pascal [11] was an improved algorithm of Close. Pascal found the set *key patterns* and the authors showed that all other frequent patterns could be directly inferred from the key patterns. Key patterns are a superset of the frequent closed patterns. Due to the level by level approach of Close and Pascal that is like Apriori, the number of database scans will be large when the length of discovered patterns is large.

CLOSET and CLOSET+. In [73], the CLOSET algorithm was proposed for mining closed frequent patterns. Unlike Close and Pascal, CLOSET performed depth first, column enumeration. CLOSET used a novel frequent pattern tree (FP-tree) to give a compressed representation of the datasets. It then performed recursive computation of conditional tables to simulate the search on the column enumeration tree.

CLOSET was usually unable to handle biological datasets with a large number of columns because of two reasons. First, the FP-tree is unable to give good compression for long rows. Second, there are too many combinations when performing column enumerations.

CLOSET+ was an improved algorithm on CLOSET algorithm. It adopted the FP-tree structure to represent datasets for datasets but adopted two different methods for dense datasets and sparse datasets. For dense datasets, CLOSET+ explored FP-tree using a method similar to that used in FP-tree algorithm [43] but for sparse data CLOSET+ used a set of in-memory pointers to explore FP-tree, which is similar to the method used in H-Mine algorithm.

CHARM. Another algorithm for mining frequent closed pattern is CHARM [101]. Like CLOSET, CHARM performed depth-first, column enumeration. However, unlike CLOSET, CHARM stored the dataset in a vertical format where a list of row ids was stored for each item. These row id lists were then merged during the column enumeration to generate new rows id lists that represented nodes in the enumeration tree. In addition, a technique called *diffset* was used to reduce the size of the row id lists and the computational complexity for merging them. Although performance studies in [101] showed that CHARM was substantially faster than all other algorithm on most datasets, CHARM is still unable to handle microarray dataset efficiently because it still performed column enumeration.

2.3.6 Analysis of algorithms

In this subsection, we analyze frequent pattern mining algorithms in terms of both CPU time and memory usage.

The Apriori algorithm has been quite successful on the basket-type datasets, for which it is designed. The basket-type datasets usually contain a large number of rows and the lengths of discovered frequent itemsets are usually short. Moreover, the basket-type datasets are usually sparse. In such kinds of datasets, it is usually the case that the dominating CPU loads come from scanning database, and generating and counting candidates. As a result, subsequent variants of Apriori algorithms try to improve efficiency by reducing the number of database scans, such as Partition, Sample and DIC, or the number of candidates, such as DHP.

Apriori-like algorithms typically do not work well on datasets that have different properties with basket-type data, such as census datasets and telecom datasets. These datasets usually contain high correlated items and may be dense. As a result, the frequent patterns discovered from these datasets are long and are large in number. If the longest frequent itemset has k items, the number of its subsets is 2^k. Therefore, the number of candidates may increase exponentially with the length of longest frequent itemset. This will result in huge increase of CUP load and deteriorate the performance of Apriori algorithms. It is usually the case to mine such datasets that compared with the time used to generate candidates and compute their supports, the time of database scanning can be ignored if the database is not very large. Algorithms Partition, Sample and DIC usually consider more candidates than Apriori does and might become worse.

The Apriori-like algorithms do not need load database into memory. Therefore, the size of database will not be the bottleneck of memory. However, the Apriori-like algorithms will meet memory problem when the number of candidates is too large to fit in memory.

Algorithms that mine association rules from vertical layout database basically generate the same set of candidates as Apriori algorithm does. Therefore, large number of candidates will result in poor performance of these algorithms although these algorithms claim that TID list intersection is faster to compute frequencies than Apriori algorithm. The category of algorithms usually require more memory than Apriori algorithm does.

They need to load both the database (although it may not be the whole database), candidates and the TID list of some frequent itemsets.

Projected database based algorithms are usually faster in computing the supports of itemsets. For breath-first Tree Projection, its memory usage is mainly limited by the number of candidates. The number of candidates is usually not a problem in depth-first Tree Projection, FP-tree and H-Mine algorithms since they usually do not consider a large number of candidates at one time. However, they are limited by the size of database. FP-tree needs to load a series of projected FP-trees in memory while depth-first Tree Projection needs to load a series of projected databases. Although FP-tree usually can compress the underlying datasets, the compression is not obvious when the number of items in dataset is large and the average length of rows is large since the pointers for FP-tree structure take a lot of memory. H-Mine needs to load the database in memory but subsequent projected databases do not take much memory since it uses in-memory pointers.

The algorithms for maximal frequent itemsets and closed frequent itemsets are usually developed on the basis of some algorithms of frequent itemsets. For example, DepthProject is developed using the depth-first TreeProjection, CLOSET algorithm is based on FP-tree and CLOSE is based on Apriori. Therefore, they have the similar problem with their corresponding frequent itemset mining algorithms although maximal frequent itemsets and closed frequent itemsets can alleviate the problem since their size is usually smaller then frequent itemsets. Although CHARM [101] algorithm proposes the diffset technique to reduce the size of tidlist, it still needs more memory than CLOSET and CLOSET+ as shown in [92]. One problem of closed pattern mining algorithms CLOSE and CHARM is that they need the whole or part of discovered closed patterns to be held in memory. This is not feasible in some cases although the number of frequent closed itemsets maybe much smaller than that of frequent itemsets.

In summary, all existing frequent itemset mining algorithms (inclusive of maximal frequent itemsets and closed itemsets) mine frequent itemsets using column-enumeration (or item-enumeration). As shown in Figure 2.3, all existing methods enumerate certain combination of columns (or items) although different pruning strategies are designed to prune some of the search space to improve performance.

It is obvious that the search space of column-enumeration will increase exponentially with the number of items in a row. As a result, all existing algorithms cannot work well for high-dimensional datasets. The detailed analysis will be given in Chapter 6.

2.3.7 Mining the optimized association rules

The number of discovered association rules can be huge because of the combinational explosion of frequent itemsets. The large number of rules is often beyond the users' understanding. Moreover, there are a lot of redundant rules in all discovered association rules.

There are numerous proposals about the definition of ”interestingness”. Various approaches are proposed to mine the interesting rules according to respective definitions of interestingness. Studies of mining interesting rules can be roughly classified into two categories. First, some studies [56, 83] are proposed to do a postprocessing to identify and remove redundant and less interesting rules. Second, others try to generate only interesting rules, thus saving a lot of computation.

[56] proposed a two phase method of postprocessing. The method worked in two steps. First it pruned the discovered association rules by removing the insignificant association while preferring to general and simple rules. The insignificant associations can be explained using the example in [56]. Given two rules: R1: *Job* = *yes* $\rightarrow$ *Loan* = *approved* at support = 60% and confidence = 90%; and R2: *Job* = *yes*, *Credit-history* = *good* $\rightarrow$ *Loan* = *approved* at support = 40% and confidence = 91%. Given R1, R2 is insignificant and will be pruned off. From the remaining rules after insignificant rules are removed, a special subset of the unpruned association rules are selected to form a

summary of the discovered association rules. The rules in the selected subset give the essential relationships of the dataset while the other rules are not surprising given the selected set of rules.

Bayardo and Agrawal [13] defined two kinds of orders for interesting rules according to confidence and support measures. One kind is called SC-Optimality and the other is called PC-Optimality.

Definition 2.3.4. SC-Optimality.

Given rules r_1 and r_2, order $r_1 < r_2$ if and only if sup(r_1) $\leq$ sup(r_2) $\wedge$ conf(r_1) $<$ conf(r_2), or sup(r_1) $<$ sup(r_2) $\wedge$ conf(r_1) $\leq$ conf(r_2). Additionally, $r_1 = r_2$ if and only if sup(r_1) = sup(r_2) $\wedge$ conf(r_1) = conf(r_2).$\Box$

The concept of SC-optimality is useful in finding the support-confidence borders. It tends to produce rules that primarily characterize only a specific subset of records. In other word, the discovered rules do not consider the coverage of rules on the underlying tuples. To remedy the deficiency, the concept of PC-Optimality is proposed in [13]. Before giving the definition of PC-Optimality, we first explain the concept *population* or *coverage*. The population of a rule $A \rightarrow C$ is simply the set of records that are covered by the rule. We denote the population of a rule r as pop(r). Clearly, $|$pop(r)$|$=sup(r).

Definition 2.3.5. PC-Optimality.

Given rules r_1 and r_2, order $r_1 < r_2$ if and only if pop(r_1) $\subseteq$ pop(r_2) $\wedge$ conf(r_1) $<$ conf(r_2), or pop(r_1) $\subset$ pop(r_2) $\wedge$ conf(r_1) $\leq$ conf(r_2). Additionally, $r_1 = r_2$ if and only if pop(r_1) = pop(r_2) $\wedge$ conf(r_1) = conf(r_2).$\Box$

When some rules have the same PC-Optimality, one of them is placed into the set of interesting rules.

Besides the minimum confidence and minimum support, some other metrics have been proposed and used to define the ”interestingness” or ”goodness” of discovered association rules. These metrics include lift [18, 30] (also known as interest and strengthen), conviction[18], gain [37], chi-square value[62], entropy gain [61], gini [61], and laplace[96].

There are some research [37, 77] that mines an optimized rule from datasets with numeric attributes. The mined rule is in the form of $salary \in [s_1, s_2] \rightarrow Loan = approved$.

There is also some research, such as [85], that mine quantitative rule from datasets with quantitative and categorical attributes and some research that mine hierarchical association rules, such as [40].

Having described and analyzed the state of the art association rule mining algorithms and frameworks, this chapter comes to the end. In next chapter, we will describe a framework of mining and recycling frequent patterns considering the problems identified in this chapter.

Chapter 3

A Framework for Association Rule Mining

This chapter will describe an extended framework for association rule mining. Below, this chapter first present a typical data mining system and some important problems that need to be addressed in the system.

Figure 3.1 shows a typical association rule mining system on a data warehouse. There are m association mining algorithm modules and n users in the system. In the multiple user system, user may have access to not only mining results from previous rounds of mining but also mining results from other users.

As discussed in Chapter 1, there are two important open questions identified in a data mining system shown in Figure 3.1. The two questions are reviewed as follows:

1. The first problem comes from the consideration that association rule mining is an iterative process and multiple users in a data mining system can share mining results. This poses a challenge for association rule mining: whether the mining results from the previous mining by the same user or different users can be recycled and reused in the new round of mining to avoid the waste of previous computation.

2. The analysis in the last two chapters shows that there is still no efficient algorithm to mine association rules from dataset with long columns despite the numerous

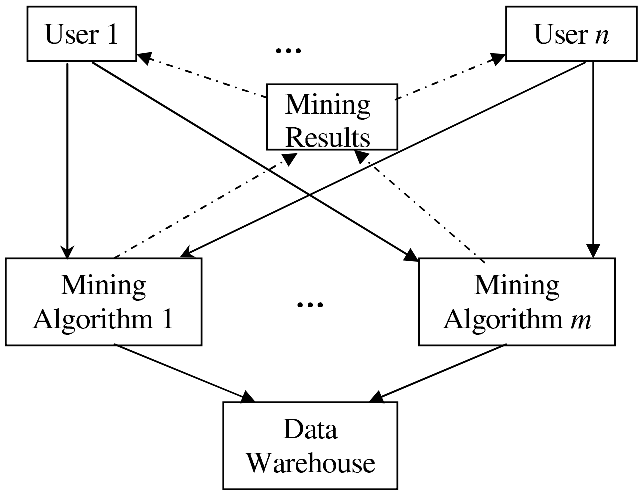


Figure 3.1: A typical data mining system

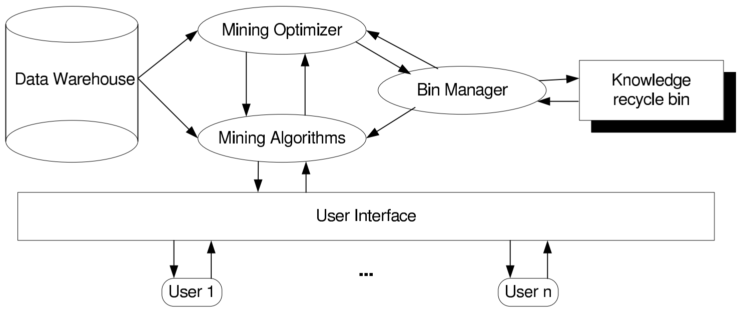


Figure 3.2: Framework for association rule mining and recycling.

algorithms available for association rule mining. Moreover, given so many algorithms, which one do users choose to mine a specific dataset?

The remaining chapter will introduce a framework of association rule mining. Within the framework, the above two problems can be examined. Note that the framework will be described in an abstract way in this thesis and the implementation of such a framework will be discussed in Chapter 8.

3.1 Overview

Figure 3.2 shows the proposed framework of mining and recycling association rules. There are four main components in the framework, the **mining algorithms**, the **knowledge recycle bin**, the **bin manager** and the **mining optimizer**. Below, each of them will be looked at individually.

Mining Algorithms. This component includes the available algorithms of mining frequent patterns and association rules. Some of representative algorithms have been reviewed in the last two chapters. This thesis also *describes some novel algorithm to mine frequent patterns in Chapter 6 and a novel algorithm to mine interesting rule groups in Chapter 7 from microarray data* that have a few of rows and a large number of columns.

Knowledge Recycle Bin. As the name implies, the **knowledge recycle bin** stores information about data mining processes that have been previously executed. For each association rule mining process, some of the useful information that should be stored is as follows:

- **Data Selection Predicates and Mining Parameters.** Data Selection predicates and mining parameters will be used to evaluate the usefulness of the association rules or frequent patterns for recycling in a different mining context. Typically, a set of selection predicates are used to specify the relevant set of rows and columns from the database for mining. For example, users may want to mine sale data of the last 3 months, which means to select a set of rows to mine. Storing the data selection predicates helps us to determine the relevancy between one mining process and another. Knowledge discovered on one set of data is usually not useful to recycle when a completely different set is mined.

Mining parameter here means the constraints imposed on frequent pattern mining and association rule mining. For example, minimum support and minimum confidence are two common constraints.

- **Knowledge/Patterns Discovered and Algorithms.** The adopted mining algorithm, discovered frequent itemsets together with their frequencies, and the infrequent itemsets computed together with their frequencies are important information for recycling. Given a dataset and a set of constraints, all algorithms generate the same set of frequent itemsets, but the set of computed infrequent itemsets depends on the algorithms. These infrequent itemsets are also called as **intermediate results** in the rest thesis.

  One issue that should be addressed is that the amount of storage for these output can be large. As such, it is important to group the common output of different KDD processes together so as to reduce storage for them. Efficient retrieval of these output is also an important issue which is addressed by a **bin manger** that will be introduced later.

- **Time of Mining.** Since the database might be changed after a mining process, storing the time of mining can help us to determine whether the result of a mining process is still valid. Storing the time of mining also helps the **bin manager** to determine which process should be removed should there be a need to reduce storage for the knowledge recycle bin.

- **Efficiency Measurement.** The CPU and I/O for each process are stored to help selection of mining algorithms in the future.

Bin Manager. The task of maintaining the knowledge recycle bin is handled by the bin manager. The main functions of the bin manager are listed as follows:

- **Storage to Bin.** Information on terminated mining algorithms will be passed to the bin manager for storage in the knowledge recycle bin. The bin manager ensures that there is minimum redundancy (user specified threshold) among the KDD processes in the knowledge recycle bin. To do so, a matching must be done to compare the incoming KDD process against those in the recycle bin based on some similarity measure. For this purpose, a coarse filtering can first be done by matching the data selection predicates and the mining algorithm/parameters that are used. Further comparison of the output can be conducted for those which pass the coarse filter possibly with the help of indexes.

The discovered knowledge/patterns should be explicitly stored if minimum redundancy is satisfied. Otherwise, it will be more desirable to use virtual pointers to link a KDD process to a set of common discovered knowledge in the bin and store only those portion of the knowledge which is different for the process.

- **Retrieval from Bin.** Given a mining query by the user, the **mining optimizer** will try to retrieve relevant knowledge from the knowledge recycle bin through the bin manager by sending **bin retrieval query** to the bin manager. A bin retrieval query from the mining optimizer can consist of the desired data selection predicates, mining algorithms/parameters and the knowledge that is required. Again, a coarse filtering can be done initially by comparing the data selection predicates and the mining algorithm/parameters.

  Further refinement of the search will depend on the type of knowledge being discovered in the recycle bin entries. In the context of frequent pattern discovery, the subject of interest could be patterns satisfying certain constraints. Supporting efficient execution of such queries will be one of the main challenges for implementing the bin manager.

- **Removal from Bin.** In the case where the knowledge recycle bin becomes saturated, heuristic must be adopted to determine the entries that should be cleared from the recycle bin. A simple strategy to do this is to remove the oldest entries in the recycle bin. Alternatively, it can be done by monitoring the usage of each entry and removing those that are less frequently used. Since patterns or knowledge in the common area might be relevant to multiple entries, they are removed only if no entries are referencing them.

Mining Optimizer. The role of the mining optimizer is to develop a mining plan using existing knowledge or patterns in the knowledge recycle bin such that the cost of the association rule mining is minimized. An analogy can be drawn between the database query optimizer and the mining optimizer in terms of functions. Despite the large amount of research that has been done on data mining, the equivalence of a database query optimizer for data mining is largely unheard of. While there is work on implementing data mining algorithms based on a series of SQL statements [65, 79], the query optimizer in such case is only optimizing each SQL statement individually without being aware of the mining algorithm.

- **Mining Plan.** Mining plan is made from two aspects. First, in the existence of frequent patterns in the knowledge recycle bin, the mining optimizer will check whether these patterns are useful for new mining process and whether these are some methods to recycle the previous mining patterns. This thesis will present *two recycling methods in Chapters 4 and 5* respectively.

  On the other hand, if there are no patterns to recycle or there are no good ways to recycle, mining optimizer tries to select appropriate algorithms from available algorithms based on the database properties.

In the next two subsections, this chapter will further discuss the two aspects of making mining plans.

3.2 Recycle and reuse frequent patterns

From the *knowledge recycle bin*, a set of frequent patterns and intermediate results, together with the constraints imposed on them, may be available for new mining process. Both the two kinds of itemsets may be frequent in a new round of mining when constraints are changed. Therefore, one natural solution of speeding up the new round of mining is to recycle these itemsets counted before in an effective way. *The proposed method in Chapter 4 tries to recycle both frequent itemsets and infrequent itemsets from previous mining to effectively speed up subsequent mining*.

Considering the case that intermediate results are not always available from *knowledge recycle bin* while the set of frequent itemsets are available*, the proposed method in Chapter 5 recycle only the discovered frequent itemsets*.

3.3 Select appropriate mining algorithms

It is difficult to tell which is the best algorithm to mine a dataset before every algorithm is tried and compared on the dataset. One possible way is to sample the given database and try to mine the sample database using candidate algorithms. The algorithm that performs the best on the sample is selected to mine the whole database. Moreover, this section tries to give some heuristic rules for selecting appropriate algorithms according to dataset properties on the basis of the analysis in Chapter 2.

When the user specified minimum support is relatively high, i.e. the discovered frequent itemsets are small in number and short in length, nearly all algorithms perform very well and do not have much difference. Therefore, it does not matter too much to choose any algorithm. This can be shown in the study [106] on the performances of association rule mining algorithms.

When the specified minimum support is relatively low, i.e. the discovered frequent itemsets are long or large in number, it is desirable to choose a good algorithm since the performance of various algorithms can be of several orders of magnitude difference.

When the dataset, such as the market basket dataset, is large and sparse, Apriori and apriori-like algorithms are usually sufficient for the association rule mining. However, these algorithms will meet problems when the candidates cannot fit in memory. Projected database based algorithms, such as H-Mine algorithm and Tree Projection algorithm, usually can get better performance than Apriori algorithm does according to [2, 72] if the dataset can be held in memory (required by H-mine) or candidates can be held in memory (required by breadth-first Tree Projection algorithm).

When the dataset is dense, FP-tree algorithm and its variations usually perform good if the FP-tree and its conditional FP-tree can fit in memory.

When the dataset is highly correlated (usually dense datasets are highly correlated), the frequent closed itemsets can usually reduce the number of frequent itemsets greatly, thus achieving better performance. Therefore, the algorithms of mining frequent closed patterns can be used. However, if it is not the case, the algorithm of mining closed itemsets may deteriorate performance since it needs additional computation to determine whether an itemset is closed. Moreover, as discussed in Chapter 2.3, the algorithms of mining frequent closed itemsets usually originate from some algorithm of frequent itemset mining and will meet similar problems with their corresponding frequent itemset mining algorithms although they usually can alleviate the problems to some extent.

When the average number of columns (items in a row) is large (for example > 100) and the minimum support is relatively low, all existing algorithms usually cannot perform well. As discussed in Chapter 1, microarray data usually have tens of thousands columns and only tens or hundreds of rows. Even the proposed recycling techniques in Chapters 4 and 5 usually will not be helpful for mining frequent patterns from microarray data and the reason will be discussed in Chapter 8. *This motivates the design of new algorithms described in Chapters 6 and 7 to mine microarray data*.

These rules can be helpful for the **mining optimizer** to select algorithm if the property of database, such as dense or sparse, is known in advance. It is noticed that algorithm in [46] first scans dataset once to judge wether the dataset is dense and use different strategy for dense and sparse data. Algorithm CLOSET+ [92] performs the similar step to mine frequent closed patterns. But when the property of database is unknown, more operative rules are required for the **mining optimizer** to work. We notice that it is still an open problem, but not the focus of this thesis.

Note that the framework to be proposed applies to mining systems with multiple mining tasks although this thesis only considers association rule mining algorithms. Some recent research tried to combine several kinds of data mining task. This means that there are opportunities to recycle one mining result in another mining results. For example, classification approach CBA [55] used association rule to build classifier. Obviously, previous association rule mining results, if have, can be recycled for classification purpose. Netcube [59] used bayesian network for the purpose of database compression, that could be extended for query and mining. [9] integrated different data mining functions for one task by utilizing set of decision trees for compression. [23] utilized the clustering results to improve the classification accuracy. [48] gave a model and algebra about the integration of data mining models.

Chapter 4

Speed-up Iterative Frequent Pattern Mining with Constraint Changes

This chapter will address the question of how to make use of previous mining results to speed up subsequent mining when constraints are changed.

4.1 Introduction

In constrained data mining, users can specify constraints to prune the search space to avoid mining uninteresting knowledge. This is typically done by specifying some initial values of the constraints that are subsequently refined iteratively until satisfactory results are obtained. Most of existing schemes fail to exploit the frequent patterns from an early round of mining for subsequent iterations.

In this chapter, a novel technique is proposed to solve this problem. Using the relaxation of frequency constraint (the decrease of minimum support) as an example, this chapter first proposes the concept of tree boundary to summarize and to reorganize the previous mining results. This chaper then shows that the additional frequent itemsets can be generated in the new mining process by extending only the itemsets on the tree boundary without re-generating the frequent itemsets produced in the previous mining.

The proposed technique has been implemented in the contexts of two frequent itemset mining algorithms, FP-tree [43] and Tree Projection [2]. This results in two augmented itemset mining algorithms RM-FP (re-mining using FP-tree) and RM-TP (re-mining using Tree Projection). Extensive experiments on both synthetic data and real-life data show that RM-FP and RM-TP dramatically outperform FP-tree and Tree Projection algorithm respectively. Finally, it is also addressed how the proposed technique can be applied to handle the changes of other types of constraints given in previous studies [63, 52, 71, 86].

The proposed tree boundary is different from the negative border proposed in [90] although they are both related to the previous mining results. Moreover, tree boundary is utilized in a completely different way from the previous works [35, 69, 88, 89] that utilize negative border (the concept has been introduced in Section 2.2). The proposed approach can avoid the disadvantages of negative border (e.g. maintaining the negative border can consume huge memory). It is also shown that a simple combination of tree boundary with existing mining approaches can not be effective and an effective solution is proposed in Section 4.4.

This rest of this chapter is organized as follows: Section 4.2 describes the problem of iterative mining of frequent patterns. The proposed technique will be described in Section 4.3 and the experimental analysis is presented in Section 4.4. Finally, Section 4.5 extends the proposed technique to all kinds of constraint changes and Section 4.6 concludes this chapter.

4.2 Problem statement

Let I be the set of all items, and D be a transaction database. This section first reviews the constraints used in frequent pattern mining. It then states the problem of iterative mining of frequent patterns with constraint changes.

4.2.1 Constraints in frequent pattern mining

Let each item in I be an object with some predefined attributes (e.g., *price*, *type*, etc). A is used to denote an attribute of items in the set I. Let $S.A$ be the set of values of attribute A for the items in pattern S. Constraints can be imposed on both pattern S itself and its attributes. For example, in a market basket case, the constraint $S \supseteq \{Carlsberg\_beer, budweiser\_ beer\}$ means that a pattern S must contain *Carlsberg_beer* and *Budweiser_beer*. The constraint *S.Type* $\supseteq \{beer\}$ means that a valid pattern S must contain some items whose type is *beer*.

There are many types of constraints that can be imposed on frequent pattern mining. Four categories of constraints: *anti-monotone*, *monotone*, *succinct*, and *convertible* constraints have been effectively integrated into some mining algorithms [52, 63, 71, 86]. They have been introduced briefly in Section 2.2.

4.2.2 Iterative mining of frequent patterns with constraint changes

As discussed in Chapter 1, a typical data mining application is an iterative process. The user of the application often runs the mining algorithm many times, and in each time s/he changes some constraints.

Given a transaction database D, the whole process of *iterative* (*and interactive*) *mining of frequent patterns with constraint changes* is captured with the following iterative steps:

(1) specify the initial set of constraints *SC*;

(2) run the mining algorithm;

(3) check the returned results to determine whether they are satisfactory. If so, the mining process ends. Otherwise, the user changes one or more constraints in *SC* (including deletion and addition of constraints), and the process then goes to (2).

(1) and (3) will not be discussed further in this thesis as it is the user's responsibility to devise and to change constraints. This chapter presents a framework designed for the

mining algorithm in (2) so that it is able to leverage on the mining results from the previous mining iteration to improve the efficiency of the current mining, and consequently speed up the whole data mining process.

Constraint changes: Change of a constraint includes two cases:

(1) Tighten the constraint: The solution space is reduced. For example, the *minimum support* is increased.

(2) Relax the constraint: The solution space is expanded. For example, the *minimum support* is reduced.

Constraint changes mean changes to one or several constraints in a set of pre-defined constraints. The changes cover deletion or addition of constraints. Adding a new constraint corresponds to tightening the constraint, while deleting an existing constraint corresponds to relaxing the constraint.

As discussed in Chapter 1, if a constraint C is tightened to C', the set of patterns that satisfy the new constraint C' is only a subset of the patterns that satisfy the old constraint C. Thus, the set of patterns that satisfy C' can be obtained by filtering the set of patterns that satisfy C. The challenge comes when a constraint C is relaxed to C'. The set of patterns that satisfy the old constraint C is only a subset of the patterns that satisfy the new constraint C'. The problem is how to efficiently discover the set of patterns F_n that satisfy the new constraint C' but not the old constraint C. Note that the patterns that satisfy C do not need to be generated again. The rest of this chapter focuses on this problem, i.e., taking advantage of the previous mining results to speed up the mining of F_n when C is relaxed to C'. This chapter also presents how to utilize the previous mining results to efficiently discover the set of patterns when multiple constraints are changed at the same time.

4.3 Proposed technique

The minimum support constraint is used as an example to present the proposed technique for finding the set of patterns F_n that satisfy the new but not the old *minimum support* when the *minimum support* is reduced (relaxed) from one mining process to the next. Minimum support is an essential constraint in frequent pattern mining. The relaxation problems of the other constraints can be solved within the proposed framework (to be discussed in Section 4.5), although the technical details may vary.

Let ξ_{old} be the minimum support used in the previous (or old) mining, and ξ_{new} be the relaxed (or new) minimum support. This section first introduces the useful information that can be obtained from the previous mining process (under $\xi_{old})$ using a *projected database* based pattern mining framework. The reason that the proposed approach uses a projected database based framework will become clear later. Then a method is described to represent the old information for the purpose of mining under ξ_{new}. Next, a naïve approach and the proposed technique are presented for discovering the set of patterns F_n that are frequent under ξ_{new} but not ξ_{old}.

4.3.1 Useful information from previous mining

This subsection discusses what kind of information from previous mining is essential for the current mining.

After running a mining algorithm using ξ_{old}, the set of frequent patterns can be discovered. One byproduct of the process is the set of patterns that are checked against ξ_{old} (supports are counted) but are not frequent. Let L_f be the set of frequent patterns under ξ_{old}, and L_{if} be the set of patterns that are counted, but found infrequent (the byproduct). The byproduct is also called *intermediate results*. Although all frequent pattern mining algorithms generate the same set L_f, the set of infrequent patterns L_{if} checked in the process varies according to algorithms.

Algorithms, such as those in [2, 43], do not strictly follow the candidate generation of Apriori-like algorithms. Instead, they are based on projected databases as discussed in Chapter 1. These algorithms are classified as projected database based algorithms. They will count the support of a pattern S = $\{i_1, i_2, \ldots, i_k\}$ if two proper subsets of S, namely S_1 = $\{i_1, \ldots, i_{k-2}, i_{k-1}\}$ and S_2 = $\{i_1, \ldots, i_{k-2}, i_k\}$, are frequent.

The projected database based mining algorithms are used as the underlying mining framework of the proposed technique because they can give the proposed technique sufficient information, while Apriori-like algorithms do not (see Remark 4.3.1). Experimental studies in [2, 43] also show that projected database based algorithms are actually more efficient than Apriori algorithm.

As in [2], a lexicographic tree is used to represent the set of frequent patterns L_f. Given the set of items I, it is assumed that a lexicographic order R exists among the items in I. The order R is important for efficiency and for the organization of mining results. The notation $i \leq_L j$ denotes that item i occurs lexicographically earlier than j.

Definition 4.3.1. Lexicographic Tree

A node in a lexicographic tree corresponds to a frequent pattern. The root of the tree corresponds to the *null* pattern.

Definition 4.3.1 is extended so that the patterns in L_{if} can also be represented with a lexicographic tree. An example lexicographic tree is shown in Figure 4.1. Those nodes enclosed in circles are frequent patterns under ξ_{new} but not ξ_{old}, i.e. the patterns in F_n. The nodes enclosed by dotted squares are the patterns in L_{if} that are not frequent under either ξ_{old} or ξ_{new}. The other nodes are patterns that are frequent under both ξ_{old} and ξ_{new}. Let P and Q be two patterns and Q be the parent of P.

Definition 4.3.2. Tree Extensions

A frequent 1-extension of a pattern such that the last item is the contributor to the extension is called a *tree extension*. The list of *tree extension*s of a node P is denoted by $E(P)$.

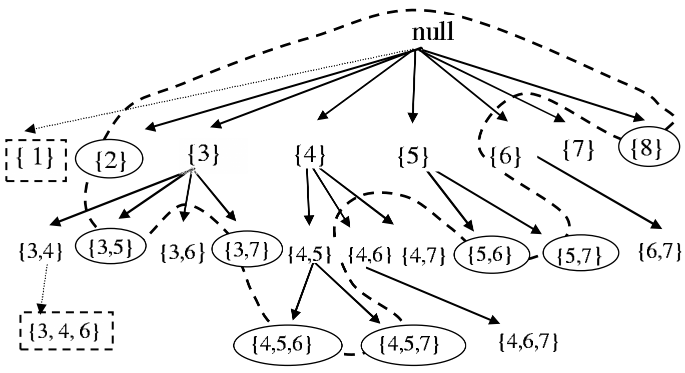


Figure 4.1: A lexicographic tree

In Figure 4.1, under ξ_{old}, the list of tree extensions of node 3 E(3) = <4, 6>.

Definition 4.3.3. Candidate Extensions

The list of candidate extensions of a node P is defined to be the items in $E(Q)$ that occur lexicographically after the node P. The list is denoted by $C(P)$. Note that $E(P)$ is a subset of $C(P)$.

Items in $C(P)$ are possible frequent extensions of P. Under ξ_{old}, the candidate extensions of *null* node C(*null*) = <3, 4, 5, 6, 7> (note that 2 is not frequent under ξ_{old}), and the candidate extensions of node 3 C(3) = <4, 5, 6, 7>.

Extensions of Lexicographic Tree The following extends the lexicographic tree with some new conceptions, which will be used in the proposed technique.

Definition 4.3.4. Infrequent Borders

If a 1-extension i of pattern P is not frequent, i is called an *infrequent border*. The list of *infrequent borders* of a node P is denoted by *IB*(P). We have the relationship: *IB*$(P) = C(P) - E(P)$.

Note that the union of infrequent border will be superset of the negative border. In Figure 4.1, under ξ_{old}, the infrequent borders of node 3 *IB*(3) = <5, 7>.

Definition 4.3.5. New Tree Extensions

If pattern $P\cup$ $\{i\}$, i $\in$ *IB*(P), becomes frequent after minimum support ξ is reduced from ξ_{old} to ξ_{new}, i is called a new tree extension of node *P w.r.t.* ξ_{new}. The list of new tree extensions of node P *w.r.t.* ξ_{new} is denoted by *NTE*(P).

In Figure 4.1, the list of new tree extensions of node 3 $w.r.t.$ ξ_{new} *NTE*(3) = <5, 7>.

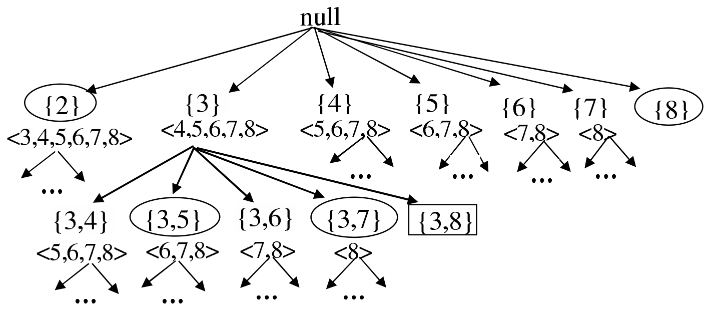
For any frequent pattern P (which can be *null*) under ξ_{old}, its *tree extensions* $E(P)$ and *infrequent borders IB*(P) are stored for mining under ξ_{new}. Its *new tree extensions NTE*(P) *w.r.t.* ξ_{new} can be obtained by checking the list of infrequent borders of P, *IB*(P). Under ξ_{old}, the set of *tree extensions* of all frequent tree nodes makes up L_f, and the set of *infrequent borders* of all frequent nodes in the tree makes up L_{if}.

4.3.2 Naïve approach

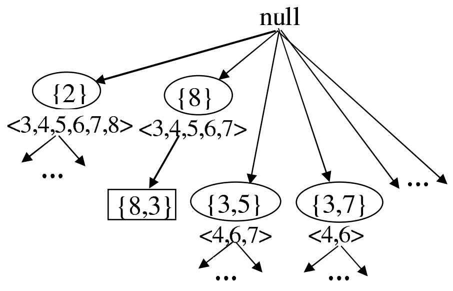
With the two sets L_f and L_{if} from the mining under ξ_{old}, this subsection first looks at a naïve (or straightforward) approach to making use of previous mining results for the new mining.

The naïve approach checks all patterns in L_f and L_{if} one by one to find the change of their candidate extensions under ξ_{new}, and to extend them to obtain the complete set F_n (in which patterns are frequent under ξ_{new} but not ξ_{old}). Figure 4.2(a) shows the children patterns of *null* node and the children patterns of pattern $\{3\}$ in the naïve approach. To make the figure manageable, this thesis assumes that pattern $\{3, 8\}$ is frequent under ξ_{new} but $\{4, 8\}$, $\{5, 8\}$, $\{6, 8\}$, and $\{7, 8\}$ are not. Candidate extensions of each node are shown under the node in Figure 4.2(a). The only saving in the new mining comes from the reuse of the count information saved previously for those patterns in L_f and L_{if}.

However, this saving in computation is very limited in a tree-based algorithm. Thus, the computation is basically the same as re-mining from scratch. In tree-based algorithms, the main computation comes from the generation of projected transactions for each node. Project transactions for a pattern S are the set of transactions containing S.



(a) The naïve approach



(b) The approach with *tree boundary*

Figure 4.2: Part of mining results under ξ_{new}

Tree-based algorithms use this sub-transaction set for counting support and for all subsequent pattern (containing S) generations. Therefore, this naïve approach still requires the same computation to generate the projected transactions as running a tree-based algorithm from scratch. For instance, in Figure 4.2(a), this naïve approach still needs to create projected transactions for $\{3\}$ to count the support for pattern $\{3, 8\}$ although the supports of its other children patterns $\{3, 4\}$, $\{3, 5\}$, $\{3, 6\}$ and $\{3, 7\}$ are known previously (the projected transactions for $\{3\}$ are also used to generate the projected transactions for children patterns of $\{3\}$). A similar computation is required for creating projected transactions for $\{2\}$, $\{3\}$, $\{4\}$, $\{5\}$, $\{6\}$ and $\{7\}$.

Another shortcoming of the naive approach is that it cannot avoid re-generating patterns in L_f because they need to be extended in the new mining. For example, in Figure 4.2(a), patterns $\{3\}$, $\{4\}$, $\{5\}$, $\{6\}$ and $\{7\}$ still need to be generated to check whether item 8 is in their tree extensions although their supports are already counted in previous mining.

Based on the above discussion, it can be seen that saving by the naive approach is limited. It is thus not efficient. Below, the next subsection first presents the concept of *tree boundary*, which provides an effective and efficient framework for mining under ξ_{new}. Then the proposed approach is presented.

4.3.3 Proposed approach

Definition 4.3.6. Tree Boundary

A tree boundary $w.r.t.$ ξ_{new} is defined to be the set of patterns *TB* = $\{$*tb* $\mid$ *tb* $\in L_{if}$, *Support*(*tb*) $\geq \xi_{new}\}$, where L_{if} is the set of counted but infrequent patterns under ξ_{old}, and *Support*(*tb*) is the *support* of pattern *tb*. □

For example, the patterns on the dotted line shown in Figure 4.1 make up the *tree boundary* $w.r.t.$ ξ_{new}. Patterns $\{1\}$ and $\{3, 4, 6\}$ are not in *TB* although they are in L_{if} because they are not frequent under ξ_{new}.

Tree boundary is different from the negative border proposed in [90] although they are both related to the previous mining results. For example, the itemset $\{3, 4, 6\}$ will be in the negative border under the ξ_{old} while it is not in the tree boundary. Unlike previous works on incremental mining that maintain the negative border in the new mining process (many itemsets will be put in the negative border under ξ_{new}, for example, $\{3, 5, 6\}$), the proposed approach uses the tree boundary as the starting point and framework for the new mining (explained below). Thus, the shortcoming of negative border (e.g. maintaining the negative border can take up huge memory, as given in the example of Section 2.2) can be avoided.

The proposed approach discovers the complete set of F_n by extending only the patterns on the *tree boundary*. The basic idea is to eliminate the effect of minimum support decrease on patterns in L_f, i.e., no pattern will be extended if it has been extended in the previous mining. This is achieved by changing the order of *tree extensions* of every node (including the *null* node) in L_f (under ξ_{old}).

Let S_p be *null* node or any pattern in L_f. Tree extensions of S_p under ξ_{new}, denoted by $E_{new}(S_p)$, contain two parts:

- tree extensions of S_p under ξ_{old},$E_{old}(S_p)$, e.g., E_{old}(3) = <4, 6>, and
- new tree extensions of S_p ($w.r.t.$ ξ_{new}), *NTE*(S_p), e.g., *NTE*(3) = <5, 7>.

The item order of $E_{new}(S_p)$ is changed as follows: move items from the new tree extensions, *NTE*(S_p), to the front of the (old) tree extensions of S_p under ξ_{old}, $E_{old}(S_p)$. For example, in Figure 4.1, the tree extensions of *null* under ξ_{new} are changed from <**2**, 3, 4, 5, 6, 7, **8**> to < **2**, **8**, 3, 4, 5, 6, 7 >.

With the new ordering, for a child pattern of S_p such that S_c = $S_p \cup \{i\}$, where$i \in E_{old}(S_p)$ ($S_c \in L_f$), the *candidate extensions* of S_c are the same under ξ_{old} and ξ_{new}. For a child pattern of S_p such that $S_n = S_p \cup \{i\}$, where $i \in$ *NTE*(S_p), the *candidate extensions* of S_n consists of :

- the items j such that $i \leq_L j$, where $j \in$ *NTE*(S_p), and
- the items j, $j \in E_{old}(S_p)$.

Due to the re-ordering, candidate extensions of the patterns in L_f are not affected. For instance, after the tree extensions of *null* node under ξ_{new} are changed into <2, 8, 3, 4, 5, 6, 7>, the tree extensions of patterns $\{3\}$, $\{4\}$, $\{5\}$, $\{6\}$ and $\{7\}$ under ξ_{new} are the same as those under ξ_{old}. The tree extensions of pattern $\{8\}$ become <3, 4, 5, 6, 7> from $\emptyset$ under ξ_{old}. The proposed approach only needs to compute the projected transactions for pattern $\{8\}$ to decide whether items 3, 4, 5, 6, and 7 are tree extensions of $\{8\}$. There is no need to compute projected transactions for $\{3\}$, $\{4\}$, $\{5\}$, $\{6\}$ and $\{7\}$ (they were computed in the previous mining).

Another example is given in Figure 4.2(b), which shows the corresponding part of Figure 4.2(a) in our approach. After the order of tree extensions of the *null* node is changed, there is no need to extend patterns $\{3\}$, $\{4\}$, $\{5\}$, $\{6\}$ and $\{7\}$ with item 8. Tree extensions of pattern $\{3\}$ is changed from <4, 5, 6, 7> to <5, 7, 4, 6>. The candidate extensions of node $\{3, 5\}$ are <4, 6, 7>. The candidate extensions of node $\{3, 7\}$ are <4, 6>. As a result, the proposed approach only needs to compute projected transactions for patterns $\{3, 5\}$ and $\{3, 7\}$ (which are not computed in the previous mining) while the naïve approach needs to compute projected transactions for patterns $\{3, 4\}$, $\{3, 5\}$, $\{3, 6\}$ and $\{3, 7\}$.

Notice that those patterns on the *tree boundary* whose candidate extensions are empty can be removed from the *tree boundary*, e.g., patterns $\{4, 5, 7\}$ and $\{5, 7\}$ in Figure 4.1.

Let us summarize the advantages of the *tree boundary* based extension with ordering change.

1) The approach is able to avoid the computation of counting the supports of patterns in L_f and L_{if}. The approach does not re-generate the patterns in L_f to extend them in the new mining process.

2) The approach is able to avoid the generation of projected transactions that were done in the previous mining while the naïve approach is unable to.

The ordering change is the key of the proposed technique. It also brings some additional benefits when integrating tree-based algorithms with *tree boundary*. These benefits will be further discussed in the next subsection.

Now, let us prove the correctness and completeness of *tree boundary* approach.

Property 4.3.1. Given *tree boundary TB* $w.r.t.$ ξ_{new}, the complete set of patterns F_n (frequent under ξ_{new} but not ξ_{old}) can be generated by extending the patterns in *TB*.

Proof: As discussed above, F_n can be obtained by extending frequent patterns in $L_f$and

L_{if} in the naïve approach. It is proved that extending patterns on *TB* with their candidate extensions can achieve the same results as the naïve approach does. Let S_p be the *null* node or any pattern in L_f. Changing the order of tree extensions of S_p under ξ_{new},$E_{new}(S_p)$, affects the generation of candidate extensions of the children patterns of S_p. Consider any two items i, $j \quad \in E_{new}(S_p)$. We have two cases.

The ordering change does not affect the relative order of i and j. If j is a candidate extension of pattern $S_p \cup i$ before the ordering change, it is still the case after the order change. Therefore, the ordering change will not affect the final results in this case.

If the relative order of i and j is changed, it must be the case that $i \in E_{old}(S_p)$, $j \in$ *NTE*(S_p), and $i \leq_L j$ (or $j \in E_{old}(S_p)$, $i \in$ *NTE*(S_p), and $j \leq_L i$), where $E_{old}(S_p)$ is the list of tree extensions of S_p under ξ_{old} and *NTE*(S_p) is the list of new tree extensions of S_p, because of the way that the order is changed. j is a *candidate extension* of pattern $S_p \cup i$ before the order change while i is a *candidate extension* of pattern $S_p \cup j$ after the order change. Therefore, the order change will not affect the results in the case.

Thus, based on (1) and (2), it is known that the approach of extending the patterns in *TB* can generate the same set of results as the naïve approach does, i.e., the complete set of patterns F_n. $\Box$

Besides mining frequent patterns with constraint changes, *tree boundary* can be applied to mining other kinds of patterns with constraint changes when integrated with their corresponding tree-based algorithms. For example, *tree boundary* can be used to discover *maximal frequent patterns with constraint changes* when it is integrated with the algorithms in [4, 8], to discover *closed frequent patterns with constraint changes* when integrated with the algorithm [20], and to discover *disjunction-free sets with constraint changes* when integrated with the algorithm in [7].

Remark 4.3.1. In Apriori-like algorithms, previous mining results under ξ_{old} do not provide sufficient information to build the *tree boundary* for re-mining under ξ_{new}. Moreover, even if we could build a *tree boundary*, Apriori-like algorithms could not be easily modified to extend patterns on *tree boundary* to discover F_n.

Apriori-like approaches count the support of a child pattern of S iff all proper subsets of the child pattern are frequent. As a result, previous mining results generated by Apriori-like approaches do not provide sufficient information to build *tree boundary*. For example in Figure 4.1, although patterns $\{4, 5, 6\}$ and $\{4, 5, 7\}$ are on the *tree boundary*, Apriori-like approaches do not count their supports because patterns $\{5, 6\}$ and $\{5, 7\}$ are not frequent.

Furthermore, even if it is assumed that *tree boundary* could be built, Apriori-like algorithms are unable to discover the set of patterns F_n by extending patterns on *tree boundary*. Unlike tree-based algorithms that generate *candidate extensions* node by node, Apriori-like approaches generate candidate k-patterns by joining frequent (k-1)-patterns with themselves. Apriori-like approaches do not allow change in the order of *tree extensions* as proposed above. For example in Figure 1, pattern $\{3, 5\}$ cannot be extended with item 4 in Apriori-like approaches as the new technique does. Thus, a different technique is needed for Apriori-like algorithms, which is beyond the scope of this thesis. $\square$

4.3.4 Tree boundary based re-mining

The proposed technique is realized using the FP-tree frequent pattern mining and the Tree Projection algorithms, which are both tree-based algorithms. The algorithm using FP-tree is called Re-Mining using FP-tree (in short RM-FP), and the algorithm using Tree Projection is called RM-TP (Re-Mining using Tree Projection). This section first presents the RM-FP algorithm, and then RM-TP algorithm.

4.3.4.1 Re-Mining using FP-tree (RM-FP)

This subsection will present the approaches to constructing FP-tree under ξ_{new} based on the previous mining results and to extending *tree boundary* using adapted FP-tree algorithm to generate the complete set of pattern F_n that are frequent under ξ_{new} but not under ξ_{old}.

FP-tree algorithm works by recursively building conditional databases and mining the corresponding FP-trees. This thesis has given a brief introduction of FP-tree algorithm in Section 2.3.

Constructing FP-tree under ξ_{new}: As discussed in Section 4.3.1, for any frequent pattern α (or *null*) under ξ_{old}, its tree extensions$E(\alpha)$ and infrequent borders *IB*(α) are stored for new mining. Another piece of valuable information from previous mining is the initial FP-tree constructed under ξ_{old}. It is called the *old FP-tree*. The *old FP-tree* includes complete information about items in E(*null*) (the set of frequent items under $\xi_{old})$ in data *D*. The initial FP-tree built under ξ_{new} is called the *new FP-tree*. The *new FP-tree* includes complete information about items in E(*null*) and *NTE*(*null*) (the set of items that are frequent under ξ_{new} but not $\xi_{old})$ in dataset *D*. With the help of the *old FP-tree* and the set of infrequent items under ξ_{old}, *IB*(*null*), the new FP-tree is obtained as follows.

(1) If *NTE*(*null*) = $\emptyset$, the *new FP-tree* is the same as the *old FP-tree*.

(2) If *NTE*(*null*) $\neq$ $\emptyset$, items in *NTE*(*null*) have lower frequencies than items in E(*null*). Since the items of each path of the FP-tree are in the order of decreasing frequencies, in the *new FP-tree*, items in *NTE*(*null*) are in the lower part of a path while items in E(*null*) are in the upper part. Because the *old FP-tree* includes complete information about the items in E(*null*), the new FP-tree can be obtained by appending the paths of the *old FP-tree* with the information about the items in *NTE*(*null*).

Integration of tree boundary with FP-tree: For each pattern *tb* on *tree boundary TB*, its conditional FP-tree, FP$|_{tb}$, is built. FP$|_{tb}$ contains all items in the candidate extensions of pattern *tb*, which contain two parts as discussed in the Section 4.3.3:

(a) items from new tree extensions $w.r.t.$ ξ_{new} of the parent pattern of *tb*, and

(b) items from tree extensions of the parent pattern of *tb* under ξ_{old}.

The items from (b) are ordered in front of items from (a) in each path of FP$|_{tb}$.

Note that the order plays the same role as the order that the Section 4.3.3 discussed for *tree boundary* although they are different apparently. This is because FP-tree algorithm works backwards.

Compared with the initial order O_d, the item order discussed above can bring two additional advantages because items from (b) have higher supports than items from (a):

First, there are more chances for prefix paths of FP$|_{tb}$ to be shared. Thus a smaller FP-tree which still contains the complete information could be built.

Second, the ordering increases the possibility that the conditional FP-tree built from FP$|_{tb}$ contains a single path. According to a lemma given in [43], if the occurrences of items are in a single path in FP$|_{tb}$, the complete set of frequent patterns extended from pattern *tb* can be generated by enumerating all the combinations of the items in FP$|_{tb}$. Thus there is no need to build conditional FP-tree for children patterns of pattern *tb*, which improves efficiency greatly.

Conditional FP-trees built in such a way ensure that FP$|_{tb}$ contains the complete information about candidate extensions of *tb* for discovering all frequent patterns under *tb*. With property 4.3.1 and the definition of FP-tree, conditional FP-trees of the patterns on the *tree boundary* provide sufficient information to discover the set F_n (frequent under ξ_{new} but not under ξ_{old}).

Based on the above discussion, the mining algorithm RM-FP is given as follows.

Step 1: For each pattern *tb* on *tree boundary TB*, RM-FP first constructs its conditional pattern base from the new FP-tree to discover its tree extension E(*tb*), and then construct its conditional FP-tree FP$|_{tb}$, which contains items in the set E(*tb*).

Step 2: If FP$|_{tb}$ contains a single path, for each combination of items (denoted as c) in the path, frequent pattern *tb* $\cup c$ is generated. Otherwise, for each $\alpha \in E$(*tb*), RM-FP generates frequent pattern β = *tb* $\cup$ α, and for each β, RM-FP constructs its conditional pattern base and its conditional FP-tree FP$|_{\beta}$ from FP$|_{tb}$.

Step 3: If FP$|_{\beta} \neq \emptyset$, let *tb* = β, and go to Step 2.

4.3.4.1 Re-Mining using Tree Projection (RM-TP)

This subsection will present the approach to using *tree boundary* to generate the complete set of pattern F_n that are frequent under ξ_{new} but not under ξ_{old}. A brief introduction about Tree Projection [2] can be found in Section 2.3.

Integration of Tree Boundary with Tree Projection: RM-TP discovers the set of itemsets F_n (which satisfy the new constraints but not the old) by extending *tree boundary* with a breadth-first strategy. RM-TP first finds frequent 1-exentsions of the itemsets on *tree boundary* by one database scan. These frequent 1-extensions form new leaf nodes of the lexicographic tree. Then the new leaf nodes are extended. The process continues until the leaf nodes cannot be extended any more.

RM-TP has the advantages of *tree boundary*. Moreover, it has two additional advantages: (1) *Tree boundary* makes up the initial leaf nodes of a lexicographic tree. Unlike Tree Projection algorithm that builds the kth level of the lexicographic tree at the kth database scan, RM-TP extends its leaf nodes that cut across various levels of the lexicographic tree. With such a strategy, RM-TP (usually) requires fewer database scans than re-running the Tree Projection algorithm (up-to 8 times fewer in our experiments). This saving is significant for real datasets that are large. (2) RM-TP can reduce the size of the dataset by removing those transactions that do not contain any items useful for re-mining after scanning the dataset one time. These features make running RM-TP much more efficient than re-running Tree Projection.

4.4 Experimental evaluation

In order to evaluate the effectiveness and efficiencies of the techniques that utilize previous mining results to speed-up current mining, this section reports the performance studies of RM-FP and RM-TP over a variety of datasets.

4.4.1 Experimental setup

Both synthetic and real-life datasets were used in experiments to compare the efficiency of FP-tree algorithm and RM-FP as well as that of Tree Projection and RM-TP. All experiments were performed on a 750-Mhz Pentium PC with 512 MB main memory, running on Microsoft Windows 2000. All the programs were written in Microsoft Visual C++ 6.0.

The synthetic datasets were generated using the procedure described in [7]. Experimental results are reported with two synthetic datasets: one is T25.I20.D200k [43], denoted as D1, with 1K items. In D1, the average transaction size and the average maximal potentially frequent pattern size are 25 and 20 respectively. The number of transactions is 200k. The other dataset is T20.I6.D100k [7], denoted as D2, also with 1K items.

Experiments were also conducted on one real-life dataset obtained from the UC-Irvine Machine Learning Database Repository [1]. The *Mushroom* dataset has 8124 transactions, and each transaction has 23 items chosen from 119 items.

4.4.2 RM-FP vs FP-tree

Figure 4.3 shows the performances of RM-FP with FP-tree algorithm run on datasets D1. In the curves for RM-FP, the CPU time for each point (except the first point) is obtained by running RM-FP (with the value of that point as ξ_{new}) based on the previous mining results (including frequent patterns and patterns in the *tree boundary*) under ξ_{old} just before that point. For example, in Figure 4.3, the CPU time of RM-FP at ξ_{new} = 1.75% is based on the old mining results with ξ_{old} = 2%, and the CPU time for RM-FP at ξ_{new} = 1.5% is based on the old mining results with ξ_{old} = 1.75%, and so on. Note that when ξ_{new} of RM-FP is the same as ξ_{old} of the previous mining, e.g., at ξ = 2% in Figure 4.3, the extra running time of RM-FP as compared to FP-tree shows the overhead

[1] http://www.ics.uci.edu/~mlearn/MLRepository.html

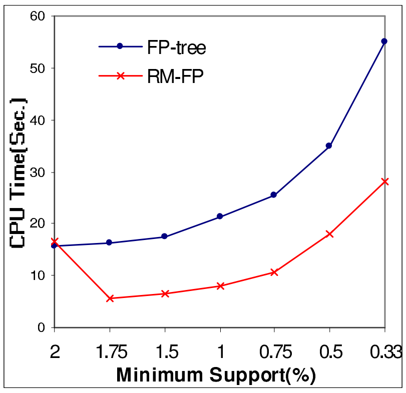


Figure 4.3: Interactive mining on D1

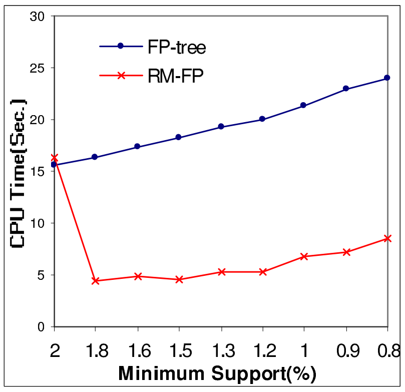


Figure 4.4: Interactive mining on D1(smaller decrease)

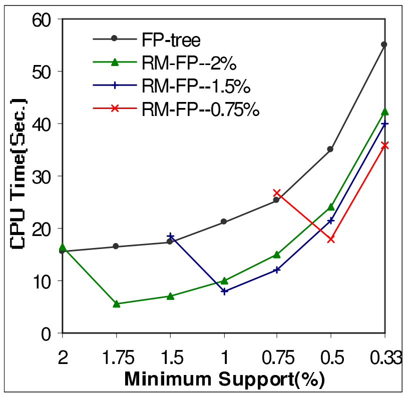


Figure 4.5: RM-FP performance on D1

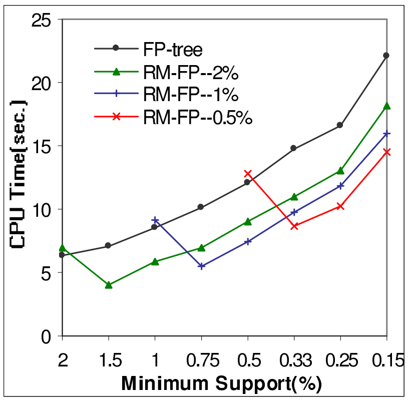


Figure 4.6: RM-FP performance on D2

of RM-FP to output patterns in L_{if}. The extra time is very small as shown in Figures 4.3-4.7.

From Figures 4.3, it can be observed that RM-FP is able to save more than 40% of running time of FP-tree in each iteration. The saving is very significant in practice. In fact, RM-FP can achieve even better results if the decrease of minimum support is smaller in each iteration as shown in Figure 4.4. In Figure 4.4, the minimum support is reduced by 10% each time (the decrease is smaller than that in Figures 4.3). At each point, again RM-FP is run based on the mining results of the previous point except for 2%. In each iteration, more than 70% of the running time is saved.

More performance curves on datasets D1, D2, and *Mushroom* are given in Figures

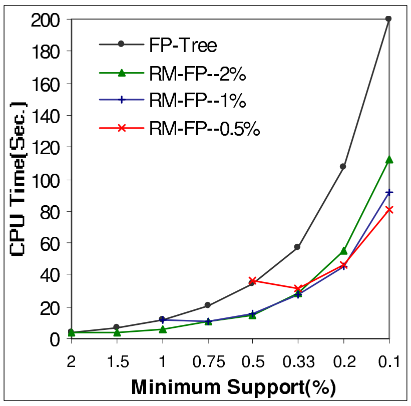


Figure 4.7: RM-FP performance on *mushroom*

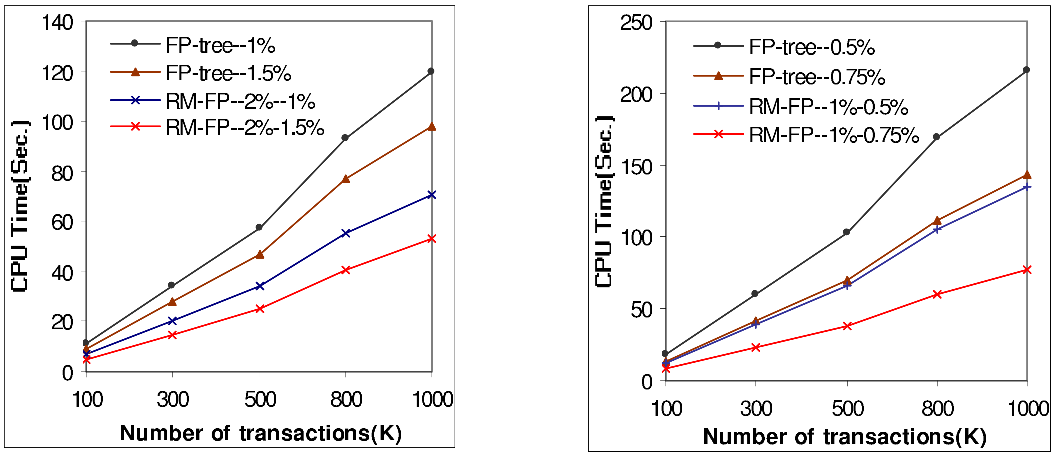


Figure 4.8: Scalability with the number of transactions

4.5, 4.6, and 4.7 respectively. In Figure 4.5, RM-FP is run based on the initial mining results of the FP-tree algorithm with ξ_{old} = 2%, 1.5% and 0.75%. In each case, a few reduced ξ_{new} values are used. In Figure 4.6, RM-FP is run based on the mining results of ξ_{old} = 2%, 1% and 0.5%. In Figure 4.7, RM-FP is run based on the mining results at ξ_{old} = 2%, 1%, and 0.5%. In each of Figures 4.5, 4.6, and 4.7, results with different ξ_{new} values are shown.

Note that Figures 4.3 and 4.5 are different although they are obtained using the same dataset D1. In Figure 4.5, for each curve of RM-RP, at all points (corresponding to different new minimum support) of the curve, RM-FP utilizes the mining result of the starting point of the curve, but not the previous point as RM-FP did in Figure 4.3.

All the experiments show that RM-FP consistently outperforms the FP-tree algorithm even when minimum support drops to a very low level from a very high level. The reasons for the improvement have been given in Section 4.3. With the same initial (old) mining results, it can be observed from the experimental results that the lower the ξ_{new} is in the new mining, the smaller is the percentage of saving in computation. The observation can be explained by the fact that the number of frequent patterns at ξ_{new} is much larger than the number of patterns in L_f from old mining. For example, for D1, the number of frequent patterns discovered at 2% is 521 while the number at 0.33% is 519,704. For D2, the number of frequent patterns discovered at 2% is 381 while the number at 0.15% is 558,834. However, in practice, the same user typically will not reduce the minimum support so drastically from one mining process to the next. For example, in most cases, it is unlikely that the user uses ξ_{old} = 2% first, and then changes it to ξ_{new} = 0.15% suddenly for the next mining. Instead, the decrease each time is usually small as in the cases of Figures 4.3 and 4.4.

Note that in Figure 4.7, RM-FP based on 1% support takes more time than RM-FP based on 2% support at ξ_{new} = 0.75%. The likely reason is that the time used to check previous mining results offsets part of the benefit from utilizing previous mining results when the number of the previous mining results is very large.

Another finding from the experiment is that that RM-FP requires less memory than FP-tree algorithm does. This is because the number of nested FP-trees built in RM-FP is smaller than that in the FP-tree algorithm. In real data, the saving can be 50%. Because the FP-tree algorithm usually requires more memory than Tree Projection and Apriori algorithms, the memory saving is valuable.

The scalability experiments were conducted by increasing the number of transactions of dataset D1. As shown in Figure 4.8, both FP-tree and RM-FP have linear scalability with the number of transactions, but RM-FP is more scalable.

4.4.3 RM-TP vs Tree Projection

The experiments presented in this subsection were conducted on the same datasets as those in last subsection. Figure 4.12 shows the experimental result of interactive mining on D1. Figure 4.12 shows that RM-TP achieves much better improvement than Tree Projection in interactive and iterative mining. The saving is more than 70%. The lowest curve in Figure 4.12 used a smaller decrease of *minimum support* than the curve above it. These results clearly show the usefulness of previous mining results in interactive and iterative mining.

More experimental results are given in Figures 4.9, 4.10 and 4.11. In Figure 4.9, RM-TP was run based on the initial mining results of the FP-tree algorithm with ξ_{old} = 2%, 1.5% and 0.75%. In each case, a few reduced ξ_{new} values were used. In Figure 4.10, RM-TP was run based on the mining results of ξ_{old} = 2%, 1% and 0.5%. In Figure 4.11, RM-TP was run based on the mining results of ξ_{old} = 20%, and 14% (very high minimum support have to be used because the dataset is very dense). In each of Figures 4.9, 4.10 and 4.11, results with different ξ_{new} values are shown.

From Figures 4.9 to 4.11, it is clearly shown that RP-TP is able to improve the efficiency of Tree Projection algorithm, which is in a similar way that RM-FP is able to improve FP-tree algorithm shown in Section 4.4.2. The likely reason for the improvement is that RM-TP makes use of previous mining results as RM-FP does. Moreover, with the help of the previous mining results, RM-TP (usually) requires fewer database scans than Tree Projection algorithm does (it can be 8 times fewer for dataset Mushroom in our experiments). These features make RM-TP much more efficient than re-running Tree Projection.

Figures 4.9 to 4.11 also show that the improvement of RM-TP as compared to Tree Projection is very small when the new minimum support drops sharply from the old minimum support. The reason for this is that RM-TP can hardly save the number of database scans required by Tree Projection when the minimum support drops sharply.

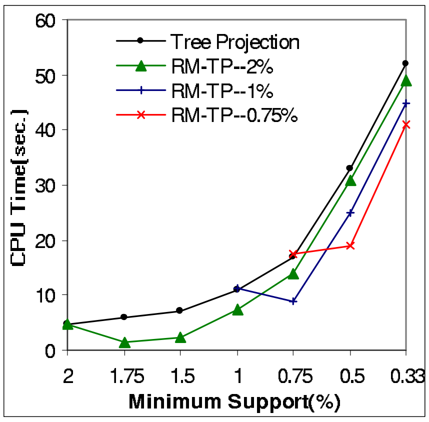


Figure 4.9: RM-TP performance on D1

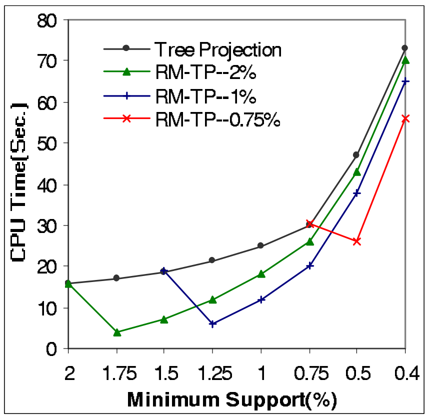


Figure 4.10: RM-TP performance on D2

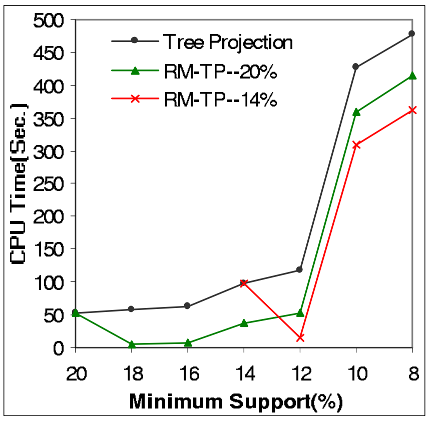


Figure 4.11: RM-TP performance on *mushroom*

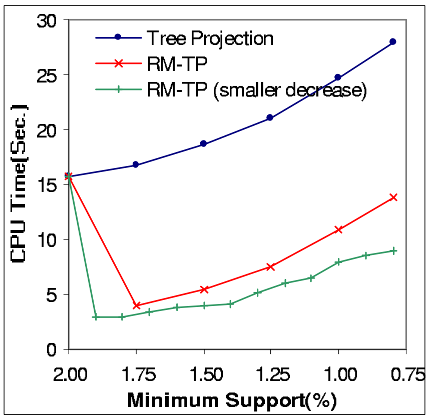


Figure 4.12: Interactive mining on D1

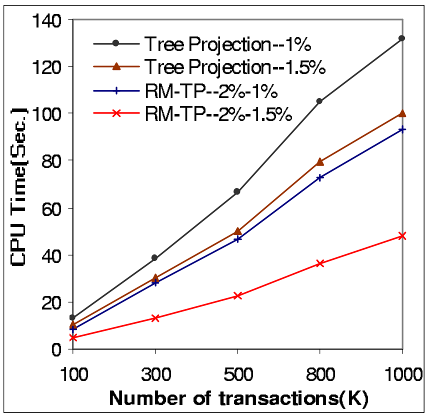


Figure 4.13: Scalability with the number of transactions

In this case, the only saving comes form the previous mining results.

Figure 4.13 shows the scalability test of RM-TP with the number of transactions.

Both Tree Projection and RM-TP have linear scalability with the number of transactions, but RM-TP is more scalable.

4.5 Application to other constraints

This section shows that the proposed approach is also applicable to discovering the set F_n (which satisfies the new constraints but not the old) when any other single or multiple constraints are changed.

The detailed techniques of dealing with the mining with changes of these constraints differ. The followings present methods of dealing with the change of individual constraints and multiple constraints intuitively.

4.5.1 Dealing with individual constraint changes

This subsection discusses the methods for discovering the set F_n when a single constraint is changed.

Method 1: Filtering previous mining results

The set F_n can be obtained by filtering previous results in the following two cases:

- tightening of a constraint of any kind, which has been discussed in Section 4.1;
- relaxation of a convertible monotone or monotone constraint.

For example, assume that a monotone constraint *sum*(*S.price*) $\geq$ 100 is relaxed to *sum*(*S.price*) $\geq$ 50. The set F_n can be obtained by checking the previous mining results because in the old mining those patterns that do not satisfy the old constraint are also checked. Thus, all checked patterns can be stored. In the new mining, each of them is simply checked against the new constraint.

Method 2: Tree boundary based re-mining

This method applies to the relaxation of a convertible anti-monotone or anti-monotone constraint. Minimum support is an anti-monotone constraint. Our *tree boundary* approach is able to discover the set F_n. An example of the convertible anti-monotone constraint is *avg*(*S.price*) $\geq$ 25 when items are listed in the descending order of *price*. When the *avg* value is relaxed, e.g., to 20, the set F_n can be discovered basically following the approach for anti-monotone constraint relaxation. Unlike an anti-monotone constraint relaxation, the relaxation of a convertible anti-monotone constraint does not need to change the item order in order to construct *tree boundary*. This is due to the special property of convertible constraints as discussed in [71].

Method 3: Simpler tree boundary based re-mining

Tree boundary in this method is easier to devise than that for Method 2 and usually contains only 1-patterns. It applies to the relaxation of a succinct and anti-monotone constraint, or a succinct and monotone constraint. When one of such constraints is relaxed, it can be dealt with as follows: Let E(*null*) be the list of frequent items that satisfy the old constraint. By checking the old mining results, we first find the list of frequent items *NTE*(*null*) that satisfy the new constraint but not the old constraint. Patterns made of individual items in *NTE*(*null*) make up the *tree boundary*. Note that if items in *NTE*(*null*) are not counted (the minimum support constraint is imposed after other constraints), they are needed to be counted by scanning the dataset once. Below, the application of this method to the two kinds of constraints will be explained.

When a succinct and anti-monotone constraint is relaxed, the patterns made of individual items in *NTE*(*null*) form the *tree boundary*, and the candidate extensions of patterns on the *tree boundary* come from the lists E(*null*) and *NTE*(*null*). For example, assume we have a set of items I_f = $\{i_1, i_2, i_3, i_4, i_5, i_6\}$, where i_1.*price* = 30, i_2.*price* = 50, i_3.*price* = 80, i_4.*price* = 100, i_5.*price* = 110, and i_6.*price* = 130. Each item in I_f is a frequent 1-pattern and any combination of items in I_f is frequent. Assume the old constraint *min*(*S.price*) $\geq$ 100 is relaxed to the new constraint *min*(*S.price*) $\geq$ 50. Under the old constraint the list E(*null*) is $< i_4, i_5, i_6 >$. With regard to the new constraint, the list

NTE(*null*) is $< i_2, i_3 >$. Patterns $\{i_2\}$ and $\{i_3\}$ are on the *tree boundary*. The candidate extensions of $\{i_2\}$ are $< i_3, i_4, i_5, i_6 >$ and those of $\{i_3\}$ are $< i_4, i_5, i_6 >$. Then, the set F_n can be discovered by extending the patterns on *tree boundary* with items of their candidate extensions.

For the relaxation of a succinct and monotone constraint (e.g., $min(S.price) \leq 100$ is relaxed to $min(S.price) \leq 150$), the method is similar except for the generation of candidate extensions.

In summary, both RM-FP and RM-TP achieve better performance than their counterparts that do not utilize previous results. This proves the effectiveness and efficiency of the proposed technique.

4.5.2 Dealing with multiple constraint changes

Although users usually change one constraint at a time to see the effect of the change, it is also possible that multiple constraints are changed at the same time. Table 4.1 shows the methods for discovering F_n when two constraints are changed at the same time. Most of the combined cases can be handled by combining the approaches to handle the change of individual constraint. For example, tightening a succinct & anti-monotone constraint and relaxing a succinct & monotone constraint require Method 1 (handling the tightening) and 3 (handling the relaxation). The exceptional cases in table 4.1 are explained as follows.

Adapted: Consider the case that a succinct & anti-monotone constraint is relaxed and a succinct & monotone constraint is tightened. A tree boundary cannot be constructed from the previous mining results by combining methods 1 and 3. Instead, the dataset is required to be scanned once to find patterns on the tree boundary before applying method 3. We call this method adapted Method 3 (M3). For example, given a set of items $I_f = \{i_1, i_2, i_3, i_4, i_5, i_6\}$, where i_1.*price* = 120, i_2.*price* = 100, i_3.*price* = 80, i_4.*price* = 60, i_5.*price* = 50, and i_6.*price* = 30. Each item in I_f is a frequent 1-pattern.

Constraint 1	Constraint 2	Tighten 1&2	Relax 1 tighten 2	Tighten 1 relax 2	Relax 1&2
Succinct & Anti-monotone	Succ. & Anti.	M1	M1&M3	M1&M3	M3
	Succ. & Mono.	M1	M1& adapted M3	M1&M3	adapted M3
	Anti.	M1	M1&M3	M1&M2	M2&M3
	Mono.	M1	M1&M3	M1	M1&M3
	Convertible Anti.	M1	depends	M1&M2	depends
	Convertible Mono.	M1	depends	M1	depends
Succinct & Monotone	Succ. & Mono.	M1	M1&M3	M1&M3	M3
	Anti.	M1	M1&M3	M1&M2	M2&M3
	Mono.	M1	M1&M3	M1	M1&M3
	Convertible Anti.	– \ M1	– \ M1&M3	– \ M1&M2	– \ M2&M3
	Convertible Mono.	– \ M1	– \ M1&M3	– \ M1	– \ M1&M3
Anti-monotone	Anti.	M1	M1&M2	M1&M2	adapted M2
	Mono.	M1	M1&M2	M1	M1&M2
	Convertible Anti.	M1	violates	M1&M2	violates
	Convertible Mono.	M1	violates	M1	violates
Monotone	Mono.	M1	M1	M1	M1
	Convertible Anti.	M1	M1	M1&M2	M1&M2
	Convertible Mono.	M1	M1	M1	M1
Convertible Anti-monotone	Convertible Anti.	– \ M1	– \ M1&M2	– \ M1&M2	– \ M2
	Convertible Mono.	– \ M1	– \ M1&M2	– \ M1	– \ M1&M2
Convertible Monotone	Convertible Mono.	– \ M1	– \ M1	– \ M1	– \ M1

Table 4.1: Handling the change of two combined constraints

The succinct & anti-monotone constraint *min*(*S.price*) $\geq$ 50 is relaxed to *min*(*S.price*) $\geq$ 20 and the constraint *max*(*S.price*) $\geq$ 80 is tightened to *max*(*S.price*) $\geq$ 100. To discover F_n, we first count the supports for $\{i_6, i_1\}$, $\{i_6, i_2\}$, $\{i_6, i_3\}$, $\{i_6, i_4\}$ and $\{i_6, i_5\}$. If $\{i_6, i_1\}$, $\{i_6, i_3\}$, $\{i_6, i_4\}$ are frequent, then the tree boundary is composed of pattern $\{i_6, i_1\}$ whose candidate extensions are $< i_3, i_4 >$.

Depends: Consider the case that a succinct & anti-monotone and a convertible anti-monotone constraint are both relaxed. If the order required by the convertible constraint cannot be maintained in the new mining with the tree boundary approach, the convertible property is violated, i.e., using the tree boundary approach may cause the loss of the convertible property. If the order can be maintained, the tree boundary approach can be applied while the convertible anti-monotone property is also exploited.

– : When a convertible constraint (anti-monotone or monotone) is combined with another constraint that requires items to follow a different order from the order required by the convertible constraint, the combination will cause the loss of the convertible property [71] or the loss of the property of the other constraint. In [71], it is shown that one of the two constraints cannot be pushed into frequent pattern mining even without considering constraint changes. In this case, the *tree boundary* based method faces the same problem.

Violates: Consider the case that an anti-monotone and a convertible anti-monotone constraint are relaxed at the same time. When an anti-monotone constraint is relaxed, item order is needed to changed to construct the tree boundary. Because of the order change, the convertible anti-monotone constraint is not convertible any longer. Therefore, the convertible property will be violated if the tree boundary *approach* is applied. Note that the above case (–) is different from this case as in this case the properties of the constraints can be utilized in mining if old mining results are not employed to speed up new mining.

For the cells containing "\ " in Table 4.1, e.g., -\ M1, it means that the combined changes require the method on the left of "\" in some cases and on the right of "\" in other cases.

Finally, when more than two constraints are changed at the same time, they can be handled by combining the methods for their respective changes, taking into account the exceptional cases.

4.6 Summary

Practical data mining is often a highly interactive and iterative process. Users change constraints and run the mining algorithm many times before they are satisfied with the final results. Using the minimum support constraint as an example, this chapter first describes the concept of *tree boundary* to summarize and reorganize the previous mining results. It then presents an effective and efficient framework for re-mining under the reduced minimum support. Experimental results demonstrate that the proposed technique is highly effective. Finally, this chapter also shows that when any other individual constraint is changed, the new set of frequent patterns can also be mined efficiently using the proposed technique. When multiple constraints are changed at the same time, the *tree boundary* technique usually can provide an effective solution.

The proposed technique in this chapter assumes that a user utilizes the previous mining results to speed-up the current round of results. With the assumption, it is usually the case that the constraints are not changed significantly at one time. However, in the multi-user environment, mining results from other users can be recycled. In such a case, the constraints of two mining process may differ significantly and the proposed technique in this chapter may not work well. Moreover, the proposed method of this Chapter relies on the intermediate results, which may not always be available. Considering the problem, the next chapter will propose a different solution.

Chapter 5

Recycle and Reuse Frequent Patterns

In this chapter, a novel solution to recycle frequent patterns to speed up subsequent frequent pattern mining will be described. Unlike the technique in last chapter, the solution to be proposed does not need the intermediate results from previous mining process.

5.1 Introduction

The proposed recycling scheme comprises two phases. In the first phase, the frequent patterns from an early iteration of mining are used to compress the database. In the second phase, the compressed database is mined. The compression here aims to speed up subsequent mining by utilizing the knowledge encapsulated in previous frequent patterns, rather than to save space although it does. Two compression strategies are designed. While the first attempts to minimize cost, the second minimizes storage space. The strategy of minimizing cost is novel in that a function is designed to estimate the potential saving of using a pattern to do the compression for subsequent mining. The strategy of minimizing storage space is relatively straightforward. This chapter also presents a naive mining algorithm that operates on the compressed database using the projected database technique. It will be shown how the naive algorithm can be combined with algorithms that use the projected database as the underlying framework easily. In this thesis, the H-Mine [72], FP-tree [43] and Tree Projection [2] are adapted to mine compressed databases.

Extensive experiments are conducted to study the performance of the proposed recycling technique. The experimental results show that the proposed recycling algorithms outperform their non-recycling counterparts by an order of magnitude. The experimental study also shows that the compression strategy that minimizes cost is more effective than the compression strategy which minimizes storage space. Another interesting finding is that the saving of recycling algorithms over non-recycling counterparts is much greater than the time that is used to mine the set of frequent patterns for recycling.

The rest of this chapter is organized as follows. In Section 5.2, the problem of recycling patterns is described. Section 5.3 presents the compression techniques and how to apply the projected database techniques to mine the compressed database. Section 5.4 shows how existing frequent mining algorithms can be adapted to mine a compressed database. Section 5.5 presents performance studies. Section 5.6 gives some discussion and concludes this chapter.

5.2 Problem statement

As discussed before, a typical data mining application is an iterative process. A user often runs a mining algorithm many times, each with more refined constraints. Such an iterative process provides the opportunity to recycle frequent patterns obtained in early iterations. Moreover, when there are many users in a data mining system, the frequent patterns discovered by one user also provide opportunity for the others to recycle.

Recycling frequent patterns: Given a database DB and a set of constraints C, the problem of *recycling frequent patterns* is to find the complete set of frequent patterns with the help of the set of frequent patterns, denoted as FP, discovered at a set of constraints C_{old}.

Compared with C_{old}, the set of constraints C might be tightened (e.g., the *minimum* *support* is increased), or relaxed (e.g., the *minimum support* is decreased). When constraints are tightened from C_{old}, the new set of frequent pattern can be filtered from the old set easily.

The challenge comes when constraints are relaxed. The new set of frequent patterns cannot be obtained from the old ones. The main approach to recycling previous patterns is to carefully select a set of frequent patterns from an early iteration and compress the data to be mined using these patterns. The selection criteria take into account the estimated saving that could occur when the database is compressed with a particular pattern. A series of algorithms using projected database as the underlying framework can be adapted to mine the compressed database.

It is noted that many frequent pattern discovery algorithms have been developed [7, 12, 43, 46, 72, 101] and it is not the intention of this chapter to develop yet another efficient algorithm for finding frequent patterns. *Instead, the aim of this chapter is to show that the concept of recycling patterns is useful and practical in an interactive data mining environment.* More specifically, it is hoped to illustrate two points: (1) Frequent patterns can be used to estimate the cost for visiting some portion of the search space that have been visited before. (2) It is possible to use such estimation to develop a mining plan such that the cost of a new round of mining is reduced.

Note that the problem statement can be extended by two cases: (1) The constraints C and C_{old} are the same while a set of FP may be discovered on a database that contains more or fewer tuples than DB. This is essentially the incremental update problem. (2) Both constraints and database are changed. It should be pointed out that the proposed technique can be applied to these two cases.

5.3 Recycling frequent patterns through compression

This section will present the proposed strategy to recycle frequent patterns by first looking at how compression can optimize subsequent mining. Then this section will present two compression strategies, and a naive algorithm to mine the compressed database.

The *minimum support* constraint relaxation is used as an example to present the proposed technique of recycling frequent patterns. Let ξ_{old} be the *minimum support* corresponding to the set of frequent patterns FP, and ξ_{new} be the current *minimum support* (relaxed from ξ_{old}). Recycling with other constraint changes can be similarly addressed. The proposed technique uses projected database concept as the underlying mining framework. Algorithms based on projected database concept include Tree Projection, FP-tree, H-Mine and their variations [46, 73].

5.3.1 Recycling frequent patterns via compression

This subsection will illustrate how compression can be used to speed up the mining of frequent patterns with an example. The following three definitions will be used in the example.

Definition 5.3.1. Frequent List

Given a database DB, a *frequent list* is a list in which frequent items in the database are ordered in support ascending order. Frequent list is denoted as *F-list*. □

For example, with $\xi_{new} = 2$, the *F-list* of the database DB in Table 5.1 is $< d : 2, f : 3, g : 3, a : 3, e : 4, c : 4 >$, where the number after ":" indicates the support of the item.

Definition 5.3.2. Projected Database

Consider a database DB and its F-list. Let *i* be a frequent item in DB. The *i*-projected database is the subset of tuples in DB containing i, where all the occurrences of infrequent items, item *i* and items before *i* (i.e., lower support values) in the F-list are omitted. The *i*-projected database is denoted as $PROJ_i$ □

For instance, the a-projected database in Table 5.1 is $< 100 : ec, 400 : ec, 500 : e >$ where ":" separates the tuple ID and tuple.

Definition 5.3.3. Candidate Extension

Consider a (projected) database DB and its *F-list*. Let *i* be an item in *F-list*. The candidate extensions of *i* (or the corresponding pattern of i) in DB are defined to be the items following *i* in the F-list. Candidate extensions of i are denoted as C_i □

Example 5.3.1. For the database in Table 5.1, the set of frequent patterns under $\xi_{old} = 3$ is $FP = \{f : 3, fg : 3, fgc : 3, g : 3, gc : 3, a : 3, ae : 3, e : 4, ec : 3, c; 4\}$. Table 5.2 is the corresponding compressed database using the set *FP* (It will be explained shortly how to get the compressed database). The outlying items are the remaining items in each tuple after compression.

With $\xi_{new} = 2$, the fourth column in Table 5.2 is obtained by ranking the left items according to *F-list* after removing the infrequent items (not in *F-list*) in the third column of Table 5.2. It is observed that compression can help to save computation in two ways.

First, computation can be saved when counting the support of a pattern. When mining frequent patterns extended from item *f* (it is in group fgc), there is no need to scan the items in the group fgc (in the uncompressed database, they have to be scanned tuple by tuple). Instead the group count (here it is 3) can be utilized to compute the frequent items in *f*-projected database. When mining frequent patterns extended from *d* (it does not belong to any group), group fgc is associated with a counter when scanning *d*-projected database, and the counter value (here it is 2) is then added to the counter of each item in fgc. In this way, it requires less computation to mine the compressed database than the uncompressed one. The saving is significant in practice where each group contains a large number of tuples. Similarly, it requires less computation to construct *F-list* by scanning Table 5.2 (the compressed database) instead of Table 5.1 (the uncompressed database).

Second, computation can be saved when constructing a projected database. Consider the construction of *g*-projected database. It can be known all tuples of group fgc belonging to the *g*-projected database by checking fgc once. Again, if we were to operate

ID	Items
100	a,c,d,e, f,g
200	b,c,d,f,g
300	c,e,f,g
400	a,c,e,i
500	a,e,h

Table 5.1: The example database *DB*.

Group	ID	Outlying items	(Ordered) Frequent Outlying Items
fgc	100	a,d,e	d,a,e
	200	b,d	d
	300	e	e
ae	400	c,i	c
	500	h	

Table 5.2: The compressed database *CDB*.

on the uncompressed database, every tuple has to be scanned in Table 5.1. For group ae, it needs to scan ae only once and then scans the outlying items in the group. □

In summary, the new round of mining can benefit from the compression using patterns from the previous round of mining as follows. Computation can be saved when counting supports for candidate extensions of a pattern P in a (projected) database. As shown in the above example, not only items in some groups but also items not in any group can benefit when we compute the supports of candidate extensions. Computation can also be saved when constructing the projected database. Constructing projected databases and computing supports take the main computation in frequent pattern mining algorithms that employ projected database as the underlying framework. The saving from the two aspects can greatly improve mining efficiency as we shall see later in our experimental study.

5.3.2 Compression strategies

We now have some intuition on how compression can help in mining frequent patterns. The remaining problem is to determine good strategies to compress a database given a set of frequent patterns FP. The basic framework for the compression works as follows (see Figure 5.1). Step 1 determines the utility of each frequent pattern. The utility functions used will be discussed shortly. In step 2, the patterns are ordered in descending order of their utility values. For each tuple in the database, Steps 3-5 then select a pattern to compress it. Note that a tuple is left as it is when it has no matching pattern. The pattern picked is the one with the highest utility.

To estimate the potential savings for subsequent mining if a pattern is used for compression, two functions are designed to compute the utility of each frequent pattern X as follows:

Strategy 1: Minimize Cost Principle (MCP)

The utility function is $U(X) = (2^{|X|} - 1) * X.C$, where $X.C$ is the number of tuples that contain pattern *X*.

MCP assumes that the potential savings of pattern X for subsequent mining can be estimated by the cost of visiting the search space to generate the pattern X at ξ_{old}. The assumption is reasonable since the larger the cost used in the old mining to discover X, the larger the potential savings can be derived from using X for the compression. The remaining problem is how to estimate the amount of processing that must be done in order to discover X at ξ_{old}. Since all subsets of X are also frequent patterns in this case and their support are at least X.C, the amount of processing to discover X can be estimated to be $(2^{|X|} - 1) * X.C$. This represents the potential savings for subsequent mining if a tuple is covered with the pattern.

Strategy 2: Maximal Length Principle (MLP)

The utility function is $U(X) = |X| * |DB| + X.C$.

MLP aims to cover each tuple with the longest pattern. Among the patterns with the same maximal length, MLP will choose the pattern with the highest support to do compression. The first part of the utility function, i.e., the product of the pattern length and its frequency of occurrences, ensures that longer patterns always have larger utility values than shorter ones. The second part, i.e., the frequency of occurrences of X in database, ensures that among patterns with the same length, patterns with larger frequency have larger utility values.

The two utility functions essentially give rise to two different compression strategies that will be studied later.

Example 5.3.2. This example illustrates how the compressed database in Table 5.2 is obtained from Table 5.1 using the MCP strategy. The method of using the MCP strategy computes the utility of patterns in FP (e.g. the utility value of $fgc : 3$ is $(2^3-1)*3 = 21$) and sorts them in descending utility value. It is obtained of $\{fgc : 21, fg : 9, gc : 9, ae : 9, ec : 9, e : 4, c; 4, f : 3, g : 3, a : 3\}$ (the number after ":" is the utility value). First, it is found that tuple 100 contains pattern fgc. Thus fgc is used to compress it. The same is done for tuples 200 and 300. Tuple 400 does not contain fgc, fg, and gc, but ae. ae is used to compress it. The same is done for tuple 500. Finally, the results obtained is shown in Table 5.2. □

5.3.3 Naive algorithm for mining compressed databases

This subsection will show how to mine the compressed database using the projected databases in a naive way. Let us illustrate the naive algorithm with an example first.

Example 5.3.3. Figure 5.2 shows the mining process on the compressed database CDB in Table 5.2 with $\xi_{new} = 2$.

Compression Algorithm
Procedure CompressDB(Database: DB, set of frequent patterns: FP)

```
(1)  Compute the utility values of all patterns in $FP$;
(2)  Sort patterns in $FP$ according to the descending order of utility values;
(3)  **for each** tuple $t$ in $DB$ **do**
(4)      **for each** pattern $X$ in $FP$ **do**
(5)          **if** tuple $t$ contains pattern $X$ **then** Use $X$ to compress $t$, **break**;
```

Figure 5.1: The compression algorithm

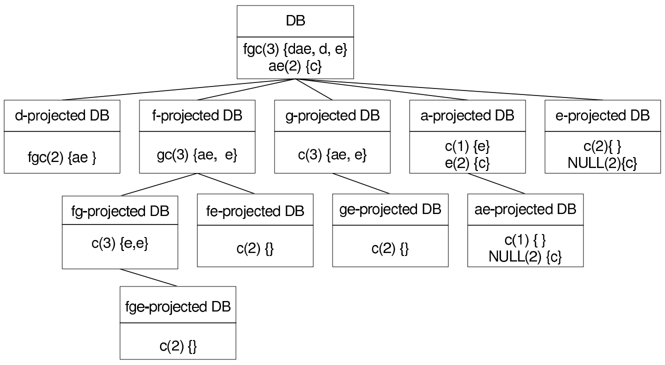


Figure 5.2: Mining from compressed DB

The CDB is first scanned to find frequent items to construct the F-list. Following the order of F-list, the complete search space of frequent patterns is mined as follows (the mining process can be regarded as a depth-first traversal of all nodes of Figure 5.2): **(1)** Find those containing item *d*. The candidate extensions for *d* are the items after *d* in F-list, i.e. f, g, a, e, c. The naive approach first constructs *d*-projected database, which is fgc(2)$\{ae\}$, where 2 registers the frequency of the group fgc in *d*-projected database. Each candidate extension is associated with a counter and each group is also associated with a counter. In the process of constructing projected database, these counters are filled. The naive approach then adds the values of group counters to the corresponding counters for candidate extensions. It is obtained of the set of frequent items $\{f : 2, g : 2, c : 2\}$ in *d*-projected database (the count of a is 1). Because all occurrences of f, g, c belong to group fgc, the frequent patterns can be generated by enumerating any combination of f, g, c, i.e. $\{dc : 2, df : 2, dg : 2, dcf : 2, dcg : 2, dfg : 2, dcfg : 2\}$.

(2) Find those containing item *f* but not *d*. The naive approach first constructs the *f*-projected database. The support for candidate extensions of *f* is counted as in (1). The set of frequent items in f-projected database is $\{g : 3, e : 2, c : 3\}$. Then the naive approach constructs *fg*-projected database. The set of frequent items in *fg*-projected database is $\{e : 2, c : 3\}$. The naive approach needs to construct the *fge*-projected database. In this step, the naive approach can get the set of frequent patterns $\{fg : 3, fge : 2, fgec : 2, fgc : 3, fe : 2, fec : 2, fc : 3\}$.

(3) Find those containing *g*, but not *f* and *d*. The mining process is similar to (2) and is ignored here.

(4) Find those containing *a* but not *g*, *f* and *d*. The naive approach constructs the *a*-projected database and gets the set of frequent items $\{e : 3, c : 2\}$. Then *ae*-projected database is constructed. Finally, the frequent patterns in the step are $\{ae : 3, aec : 2, ac : 2\}$.

(5) Finally, the other frequent patterns are computed in a similar way and are ignored here. These include those patterns containing *e* but no *a*, *g*, *f* and *d* as well as those only containing *c*. □

Lemma 5.3.1. (Single group pattern generation)

Suppose that all occurrences of frequent items in a projected database is in a single group. The complete set of frequent patterns can be generated by the enumerations of all the combinations of frequent items with the count of the group as support. □

The above example assumes that the compressed database fit in memory. Although the compressed database is smaller than the original database, it is possible that it may still be too large for the available memory. In this case, the compressed database can be projected onto its set of frequent items. There are two methods for doing so. One is the partition-based projection as used in [72]. This approach projects each tuple only to its first projected database (according to item ordering). After processing the first projected database, it needs to project the first projected database to subsequent projected

```
**Algorithm Recycling**
**Input:** Compressed database $CDB$, the support
threshold $\xi_{new}$, and available memory $M$.
**Output:** The complete set of frequent patterns.
**Method:** Call Procedure RP-Mine($CDB$, $null$)

**Procedure RP-Mine**(compressed DB: $D$, pattern: $\alpha$)
(1) Scan $D$ to find frequent items $I_f$ and estimate expected memory usage $EM(D)$;
(2) if $(EM(D) > M)$ then
(3)     Project $D$ to items in set $I_f$;
(4)     for each projected database $D_i$ $(i \in I_f)$ do
(5)         Generate pattern $\beta = i \cup \alpha$ with supp = $i.count$
(6)         Call RP-Mine($D_i$, $\beta$);
(7) else Count frequency of items in $D$ & construct $F$-$list$
(8)     Call RP-InMemory($D$, $F$-$list$, $i \cup \alpha$);

**Procedure RP-InMemory**(Projected DB: $PROJ$, List :$list$, Pattern:$\alpha$)
(1) if all occurrences of items in $list$ are in a single group $G$ in $PROJ$ then
(2)     for each combination (denoted as $\beta$) of the items in the $list$ do
            Generate pattern $\beta \cup \alpha$ with supp = the count of group $G$
(3) else for each item $a_i$ in $list$ do
(4)     Generate pattern $\beta = a_i \cup \alpha$ with supp = $a_i.count$;
(5)     Construct $a_i$-projected database $PROJ_{a_i}$ and find the list of frequent items $LF_{a_i}$ in $PROJ_{a_i}$
(6)     if $LF_{a_i} \neq null$
(7)         then Call RP-InMemory($PROJ_{a_i}$, $LF_{a_i}$, $\beta$);
```

Figure 5.3: Algorithm to recycle patterns

databases. The method is not efficient although it saves disk space. Another approach, which this chapter adopted, is to use parallel projection to speed up the computation. This approach projects each tuple into all its projected databases. Based on the above analysis, Figure 5.3 gives the naive algorithm of recycling patterns. In line 1 of procedure RP-Mine(), the estimation of memory usage relies on the representation of the projected database.

5.4 Mining algorithms on compressed database

Three representative frequent pattern mining algorithms (using projected database as the underlying framework) are adapted to mine a compressed database. This section mainly

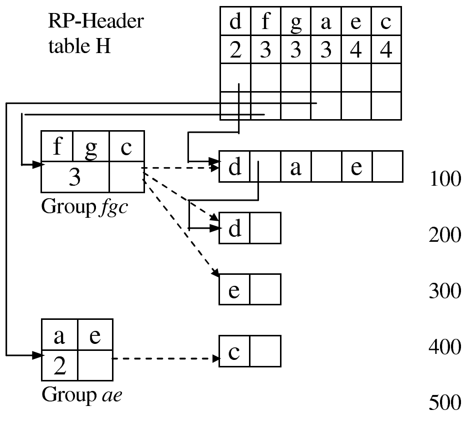


Figure 5.4: The Representation of Table 2 with RP-Struct

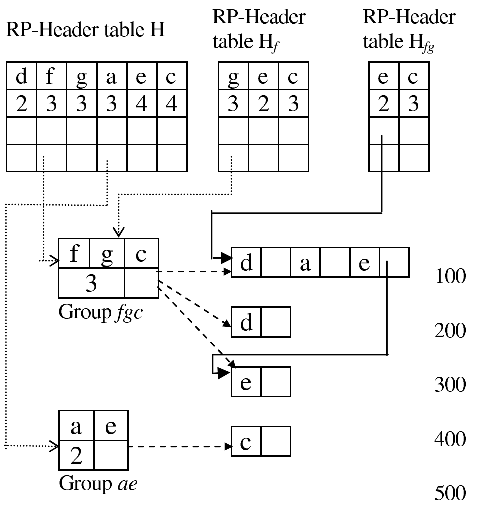


Figure 5.5: RP-Header tables H_f and H_{fg}

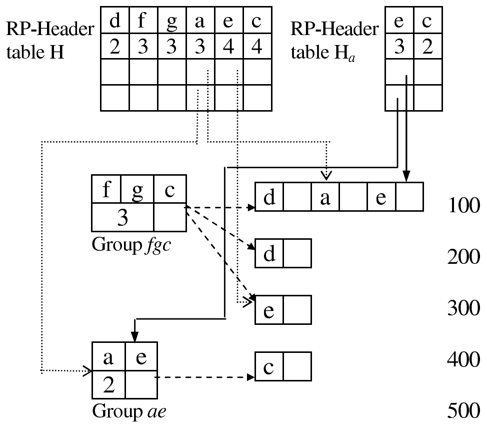


Figure 5.6: RP-Header table H_a

introduces how to adapt H-Mine since it is the most complicated to be adapted. A short introduction about adapting FP-tree and Tree Projection algorithms is also given.

The data structure of H-Mine is used to represent the outlying frequent items (uncompressed part). The integration of such a data structure into recycling algorithm is non-trivial. This subsection first uses an example to illustrate how a compressed database can be mined by adapting H-Mine. Then the algorithm for frequent pattern discovery is given.

Example 5.4.1. Consider the compressed database CDB as shown in Table 2 and $\xi_{new} = 2$. CDB can be organized as shown in Figure 5.4. The RP-Header table *H* contains the same number of items as *F-list* and follows the order of *F-list*. One compressed group *fgc* contains three tuples, and the other group *ae* contains two tuples (one tuple is *null* after compression). □

The representation of the compressed database as in Figure 5.4 is called **RP-Struct**. It has three components:

1. Group Head: Each entry in group head consists of three fields: *group pattern*, *count*, and *tail*, where *group pattern* registers the items contained in the group, *count* registers the number of tuples in the group, and *tail* points to the tuples of the group.

2. Group Tail: It records the frequent items in the uncompressed part of each tuple. The data structure of H-Mine [72] is adapted for *group tail*s. Each frequent item is stored in an entry that contains two fields: *item-name* and *item-link,* where *item-name* registers the item the entry represents and *item-link* is used to link the same *item-name* in different group tails together.

3. RP-Header Table: Each entry in RP-Header table represents a pattern and the entries in RP-Header table follow the same order as F-list. Each entry consists of four fields: *item-name*, *count*, *item-link*, and *group-link*, where *item-name* registers the last item of the pattern represented by the entry, *count* means the number of tuples containing the pattern represented by the entry, *item-link* points to the tuple whose first item is *item-name,* and *group-link* points to the groups containing item-name. By following *item-link,* and *group-link*, the projected database for the pattern represented by each entry can be obtained.

One main originality of H-Mine is to construct the projected database using a set of pointers rather than physically projecting the database. The compressed database makes it non-trivial to do so since both group heads and tails need to be considered. Figure 5.4 shows how to fill the *item-link* and *group-link* of RP-Header table to construct the projected database. The algorithm is described in Figure 5.7.

As in Example 5.3.3, d-projected database is mined first. In filling the RP-Header

```
Algorithm FillTable

Method: Fill-RPHeader($null$, $H$, $F$-$list$, $null$);
Procedure Fill-RPHeader(RP-Header table:$H_1$, PR-Header table:$H_2$, Item List: $LI$, Item:$a_i$)
(1)  for each group $G$ linked by group-link of entry $a_i$ of $H_1$
// if $H_1 = null$, for each group used for compression
(2)      if $G \cap LI \neq \emptyset$
(3)          Let $i$ be the first item of $G \cap LI$;
(4)          Link group $G$ to the group-link of entry $i$ of $H_2$;
(5)      for each group tail $t$ of $G$
(6)          if there exists an item $j, j \in LI \cap t$, $j$ orders before $i$ in the order of $LI$ then
// if there are several such j, we choose the first. When i = null, i is ordered after all items in LI.
(7)              Link entry $j$ in group tail $t$ with entry $j$ of $H_2$
```

Figure 5.7: Algorithm to fill the RP-Header table

table, d-projected database can be obtained while assigning the group heads and group tails that are not in d-projected database to the other entries in RP-Header table. For group head G (lines 2-4), it is assigned to the entry corresponding to the first item of $G \cap F\text{-}list$. For instance, group fgc is assigned to the entry f of RP-Header table H because f is the first item of $fgc \cap F\text{-}list$. For group tail (lines 5-7), two examples are used to illustrate the algorithm *FillTable*: (1)*Group tail* 100 in group fgc is linked by the *item-link* of entry *d* of H. (2) *Group tail* 300 is not linked with entry *e* of H. Note that *group tails* 100 and 300 are handled differently. This is because in 100 item d ranks before f (the first item of group fgc) in $F\text{-}list$ and d-projected database is mined before f-projected database. However, in 300 e ranks after f and e-projected database is mined after f-projected database.

Example 5.4.2. Let us examine the mining process for Example 5.3.3 based on the RP-Struct constructed in Example 5.4.1 as follows:

(1) Find those containing item *d*. There is no *group head* that contains *d*. Therefore, by traversing the *item-link* of *d*, the set of frequent items $\{f : 2, g : 2, c : 2\}$ in *d*-projected database can be found. Because all occurrences of f, g, c belong to group fgc, the frequent patterns can be generated by enumerating any combination of f, g, c.

After discovering frequent patterns in the subset, the recycling algorithm based on

RP-Struct traverses the item-link of *d* again to assign them to the items after *d*, i.e. f, g, a, e (there is no need to fill the item-link and group-link of item c because it can not be further extended). In *group tail* 100, the item after *d* is *a* and *a* ranks after item *f*, the first item of group *fgc*. *Group tail* 100 is not linked with the entry of item *a* of RP-Header table H since *f*-projected database is mined before *a*-projected database. For the similar reason, *group tail* 300 will not be linked with any entry of H, too.

(2) Find those containing item *f* but not *d*. The *item-link* of item *f* is null. The set of frequent items in f-projected database is $\{g : 3, e : 2, c : 3\}$ by checking the group fgc. The RP-Header table H_f is constructed for *f* as shown in Figure 5.5. The group fgc contains the first item g of H_f. Therefore, the group fgc is linked with *group-link* of entry g of H_f. Since all items in *group tails* are after item g according to the order of H_f, there is no need to scan the group tails of fgc to build *item-link* for other items in H_f.

In order to mine the *fg*-projected compressed database, the RP-Header table H_{fg} is constructed as shown in Figure 5.5. The RP-Header table is constructed by traversing the *item-link* and *group-link* of entry *g* of H_f. For group fgc, its first item contained in H_{fg} is *c*. Since its *group tail*s 100 and 300 contain item *e* and *e* is before *c* in H_{fg}, they are linked with entry *e* of H_{fg}. Entry c is not linked with group fgc since pattern (fgc) represented by c can not be extended. The *group tail* 200 does not contain any items in H_{fg}. See Example 5.3.3 for the set of frequent patterns obtained. At the end of the step, the group fgc is assigned to item g of H.

(3) Find those containing *g*, but not *f* and *d*. The mining process is similar to (2) and is ignored here.

(4) Find those containing *a* but not *g*, *f* and *d*. Figure 5.6 shows the RP-Header table H_a. The set of frequent patterns (see Example 5.3.3) can be obtained by traversing the item-link and group-link of entry *e*.

(5) The step is ignored here. □

```
Procedure Recycle-HM(PR-Struct: $Struct$, RP-Header table :$H$, pattern:$\alpha$)
(1)  if RP-Header table $H$ only contains a single group $G$ then
(2)      for each combination of (denoted as $\beta$) the items in group $G$ do
         Generate pattern $\beta \cup \alpha$ with supp = the count of group $G$;
(4)  else for each item $a_i$ in $H$ do
(5)      Generate pattern $\beta = a_i \cup \alpha$ with supp = $a_i.count$;
(6)      Find List of frequent items $LF_{a_i}$ in $a_i$-projected database;
(7)      Construct RP-Header table $H_\beta$ for pattern $\beta$;
(8)      Call Fill-RPHeader($H$, $H_\beta$, $LF_{a_i}$, $a_i$);
(9)      if $H_\beta \neq null$
(10)         then Call Recycle-HM($Struct$, $H_\beta$, $\beta$);
(11)     Let $A_{a_i}$ be the list of items ordered after $a_i$ in $H$;
(12)     Call Fill-RPHeader($H$, $H$, $A_{a_i}$, $a_i$);
```

Figure 5.8: Recycling frequent patterns by adapting H-Mine

Based on the above analysis, Figure 5.8 gives the procedure Recycle-HM that recycles patterns by adapting H-Mine. The procedure Recycle-HM is used to replace the procedure RP-InMemory in algorithm Recycling shown in Figure 5.3. The a_i-projected database in line 6 of procedure Recycle-HM is obtained by following the item-link and group-link of entry a_i of H. The Procedure Fill-RPHeader() called in lines 8 and 12 is given in Figure 5.8. In line 8, the item-link and group-link of a_i are assigned to the RP-Header table in next level while in line 12 they are assigned to the entries after a_i in the same RP-Header table.

In order to adapt FP-tree algorithm [43] to mine compressed databases, the data structure of frequent pattern tree (or FP-tree in short), which is a prefix tree, is used to represent the outlying frequent items (uncompressed part). In the process of recursively constructing projected databases that are represented with FP-tree, each (compressed) group head is treated as a special item, which is in the upper of each prefix tree branch. The frequency can be computed from both the compressed groups and the outlying items as it is done in adapting H-Mine algorithm in last subsection.

In Tree Projection algorithm [2], transactions are projected on each node of the tree from the root on. A matrix is maintained to count the support on the reduced set of transactions after projection. Tree Projection algorithm can mine frequent patterns in both depth-first and breath-first ways. We adapt the depth-first Tree Projection for recycling algorithm. Since the Tree projection does not represent projected databases as algorithms FP-tree and H-Mine do, the implementation is relatively straightforward and can be easily adapted from the naive algorithm in Figure 5.3.

5.5 Performance studies

This section will look at the performance of the approaches to recycling and reusing frequent patterns by comparing recycling algorithms with their corresponding non-recycling algorithms. As it is difficult to simulate the actual constrained mining environment, a simplified method is adopted to conduct our experiments. An initial mining with a support threshold ξ_{old} is performed to generate a set of patterns for recycling and then lower the support threshold to ξ_{new} when trying to recycle the patterns.

An extensive performance study has been performed on a wide range of data sets. A summary of the results is reported here. All the experiments are performed on a 1.4GHz Pentium PC with 512M main memory, running Windows XP [1]. All programs are developed using Microsoft VC++.

Weather [2] and *Forest* [3] are two sparse datasets used to report our results. *Connect-4* [4] and *Pumsb* [5] are dense data sets that we have used. The columns 2-4 of Table 5.3 (a) list the number of tuples, the average tuple length and the total number of items in each data set. Because of the different properties of these datasets, it is not necessary and feasible to choose the same initial support threshold ξ_{old} for all datasets. Instead, the initial support ξ_{old} for each dataset is chose to ensure that there are some frequent patterns to recycle.

[1]The experiments were performed on a different platform with that used in previous chapter because the platform was updated when study in this chapter was done

[2]http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html

[3]http://www.ics.uci.edu/ mlearn/MLRepository.html

[4]http://www.ics.uci.edu/ mlearn/MLRepository.html

[5]http://www.almaden.ibm.com/cs/quest/demos.html

Dataset	#Tuples	Avg. Len.	# Items	ξ_{old}	# pattern	maximal length
Weather	1,015,367	15	7939	5%	1227	9
Forest	581,012	13	15,970	1%	523	4
Connect-4	67,557	43	130	95%	4411	10
Pumsb	49,046	74	7117	90%	2607	8

(a) Properties

Dataset	Run Time(I/O) Sec.		Run Time(Pipeline) Sec.		Compression Ratio	
	MCP	MLP	MCP	MLP	MCP	MLP
Weather	9.61	10.68	4.34	5.31	0.723	0.675
Forest	2.67	4.58	0.45	2.25	0.858	0.785
Connect-4	0.32	0.32	0.06	0.06	0.773	0.773
Pumsb	0.50	0.51	0.10	0.11	0.894	0.894

(b) Compression statistic

Table 5.3: The properties of datasets and compression statistic

The argument for this is that a lack of frequent patterns for recycling will mean that little or no resources are used for the previous round of mining. It thus makes no sense to try to recycle patterns when no resources are used in the first place. This argument, based on the law of conservation, is also consistent with the observation that a lower initial support will usually give better performance of recycling. After all, it is known that mining frequent patterns with low minimum support will typically require more resources in term of both CPU and I/Os. Since more resources are used, it is expected to bring more benefits when reusing the output of the mining.

5.5.1 Analysis of compression strategies

This subsection analyzes compression time and compression ratio of the two proposed compression strategies, MCP and MLP.

Table 5.3 (b) gives some statistic on the patterns that are discovered with a minimum support ξ_{old}. We compress each database using these patterns. The last two columns of Table 5.3 (b) shows the compression ratio using the two strategies. The compression ratio R is computed as S_c/S_o, where S_c is the size of compressed database and S_o is the size of original database. In term of the compression ratio, $MLP \geq MCP$.

It would be stressed again that the compression here provides a way to speed up subsequent mining by utilizing previous frequent patterns, rather than to save space although it does. It can seen that the compression ratio is not very large.

Table 5.3(b) also shows the running time for compressing the dataset in seconds. The column *"run time* (I/O)" in Table 5.3(b) includes the time used to read, write and compress data sets. The column *"run time (pipeline)"* deducts the I/O time from the column *run time* (I/O). We list such a column since the compression step can in fact be directly integrated into the mining algorithm when it is projecting the databases which means that the I/O time will be incurred anyway.

As shown in Table 5.3(b), the compression time is small compared with the mining time as shown in next subsection. This shows that the overhead of compression is not significant. The run time of the two strategies follows the order: $MLP \geq MCP$. The order is consistent with the compression ratio since the better ratio usually means more computation required for compression.

5.5.2 Mining in main memory

In this subsection, it is assumed that both the compressed databases and original databases can fit into the memory. We will evaluate the effectiveness of recycling patterns and the two compression strategies. HM-MCP and HM-MLP are used to represent the two recycling pattern algorithms adapted from H-Mine. HM-MCP and HM-MLP run on compressed database generated with the MCP and MLP strategies respectively. Similarly, FP-MCP and FP-MLP represent two algorithms adapted from FP-tree; TP-MCP and TP-MLP represent two algorithms adapted from Tree Projection.

The reported CPU time does not include the time used to output frequent patterns since it is the same for all algorithms. In any case, the mining cost dominates performance so that including them does not affect the relative performance of the various schemes.

The effectiveness of recycling patterns: Figures 5.9, 5.12, 5.15 and 5.18 compare the performance of recycling algorithm HM-MCP with H-Mine by varying the new support threshold ξ_{new} and plotting the CPU running time for each support threshold. For example, in Figure 5.9, HM-MCP mines the compressed dataset *weather* which is generated using the set of frequent patterns under $\xi_{old} = 5\%$. Readers can refer to Table 5.3 to get the related information for other figures. Note that the vertical axes of Figures 5.15 and 5.18 use logarithmic scale for clarity. These figures clearly show that HM-MCP are performing far better than H-Mine with respect to run time. In Figures 5.15 and 5.18, recycling algorithms are over two orders of magnitude faster than the non-recycling version. It is also observed of similar relative performance between the recycling algorithms and their non-recycling counterparts for FP-based techniques (see Figures 5.10, 5.13, 5.16 and 5.19) and Tree Projection methods (see Figures 5.11, 5.14, 5.17 and 5.20, where the vertical axes of Figures 5.17 and 5.20 use logarithmic scale). The experiment results clearly demonstrate the usefulness of recycling frequent patterns.

There are three interesting observations from our experiment results:

(1) When *minimum support* is low, the savings of HM-MCP against H-Mine are much more than the time used to generate the set of frequent patterns at ξ_{old}.[6] Considering that the compression time with pipeline in Table 5.3 is also small, this suggests the possibility that a new mining task with low minimum support could be split into two steps: (a) first run it with a high minimum support; (b) then compress the database with the strategy MCP and mine the compressed database with the actual low minimum support.

(2) None of H-Mine, FP-tree and Tree Projection algorithms came out as a winner on all the datasets used. However, the proposed recycling algorithms can always improve their performance.

[6]Although we do not show the time when mining with the support threshold ξ_{old}, readers can infer that it will be less than the CPU time for the lowest ξ_{new}.

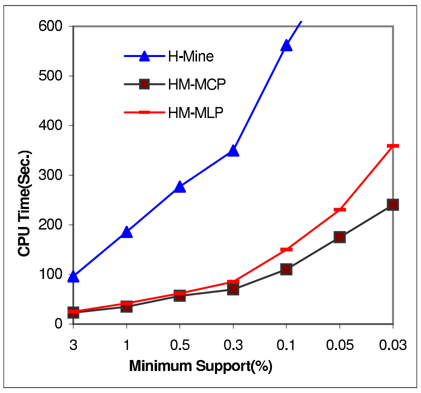


Figure 5.9: Adapting H-Mine on Weather

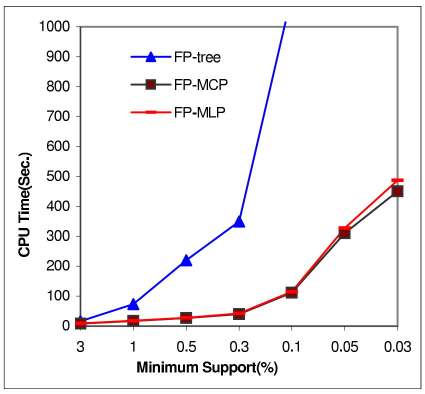


Figure 5.10: Adapting FP-tree on Weather

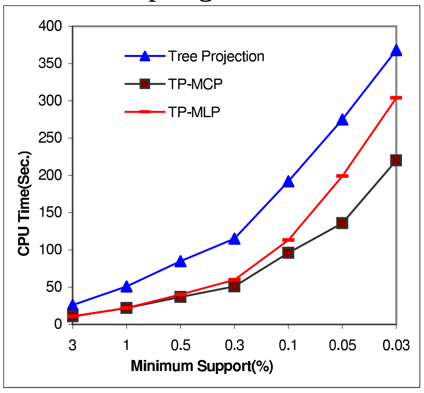


Figure 5.11: Adapting Tree Proj. on Weather

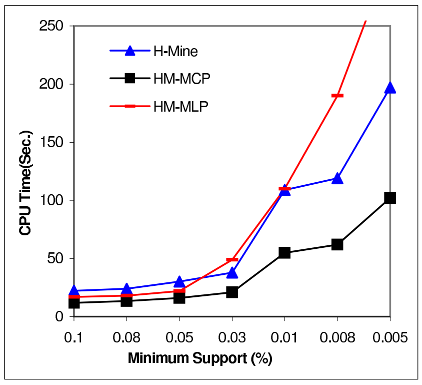


Figure 5.12: Adapting H-Mine on Forest

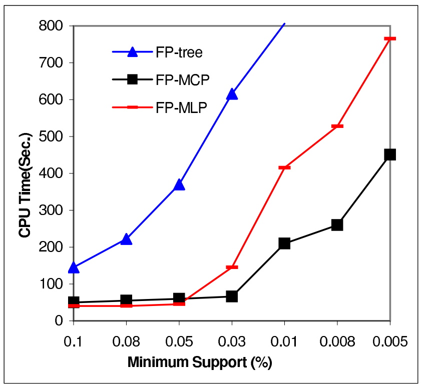


Figure 5.13: Adapting FP-tree on Forest

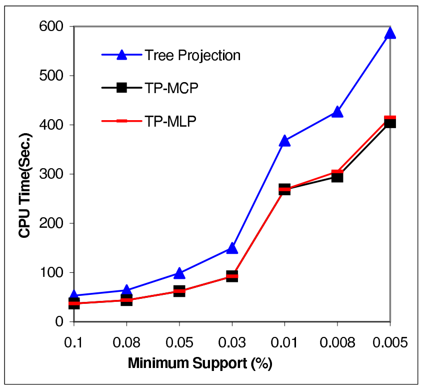


Figure 5.14: Adapting Tree Proj. on Forest

(3) When the *minimum support* is low, recycling patterns using MCP performs better. This is exciting for incremental mining of frequent patterns. Existing incremental mining techniques do not work well when the data set or constraints change dramatically (e.g. sharp decrease in minimum support). HM-MCP can overcome the problem when

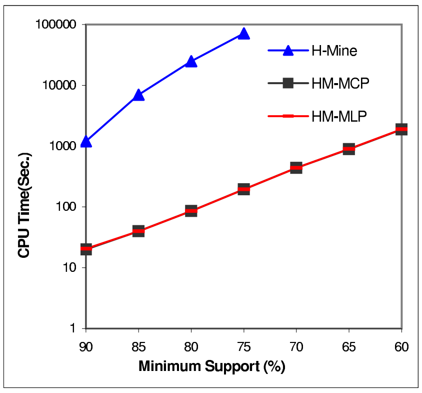


Figure 5.15: Adapting H-Mine on Connect-4

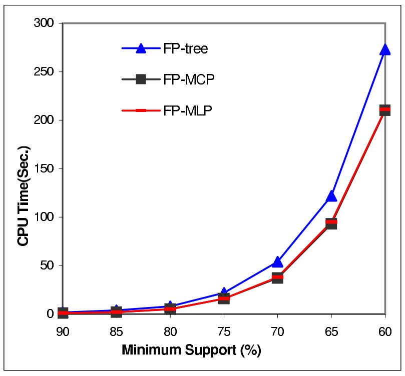


Figure 5.16: Adapting FP-tree on Connect-4

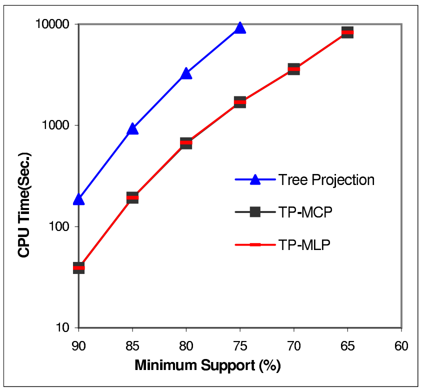


Figure 5.17: Adapting Tree Proj. on Connect-4

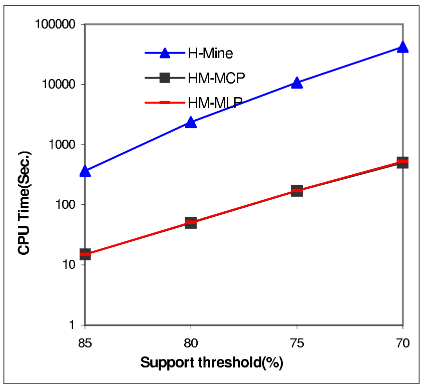


Figure 5.18: Adapting H-Mine on Pumsb

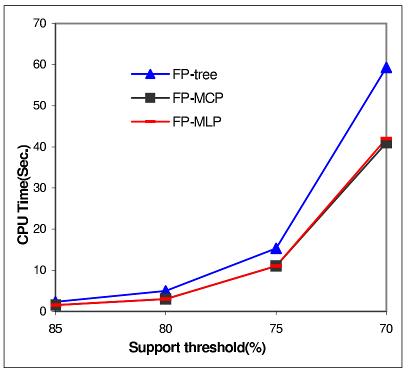


Figure 5.19: Adapting FP-tree on Pumsb

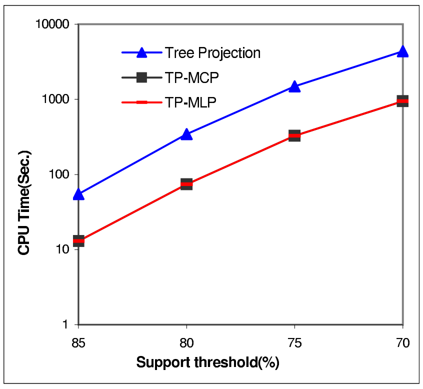


Figure 5.20: Adapting Tree Proj. on Pumsb

it is applied to incremental mining.

Comparison of two compression strategies: Figures 5.9–5.20 compare the usefulness of the two compression strategies in recycling patterns. As shown, in all cases,

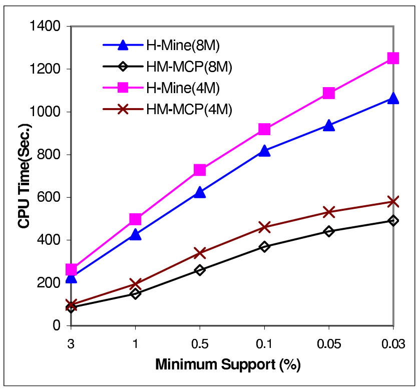


Figure 5.21: Weather with Memory Limitation

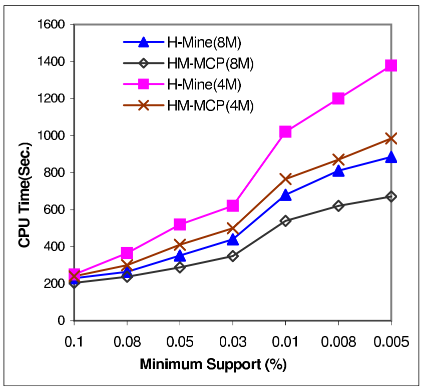


Figure 5.22: Forest with Memory Limitation

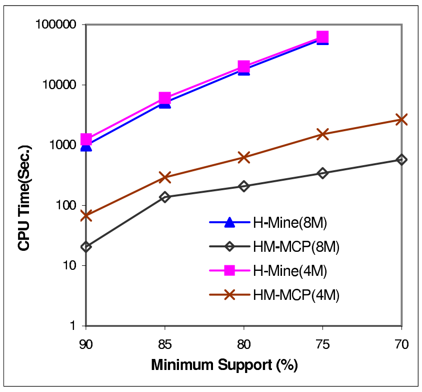


Figure 5.23: Connect-4 with Memory Limitation

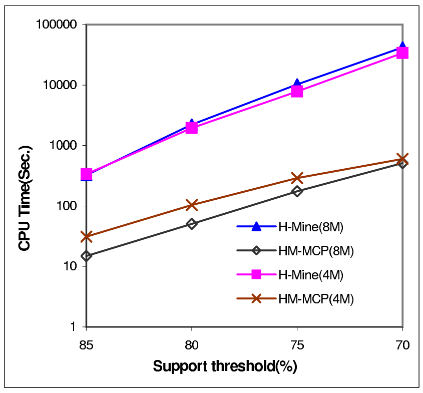


Figure 5.24: Pumsb with Memory Limitation

the compression strategy MCP achieves at least the same or better performance than the other strategy MLP. As shown in Section 5.5.1, MLP usually achieves better compression ratio than MCP. Therefore, it can be concluded that better compression does not necessary means better performance. The experiments also prove that minimizing mining cost (MCP) is more effective than minimizing storage space (MLP)in recycling frequent patterns. In fact, the compression ratios are not very large for both strategies as in Table 5.3. As given in section 5.3.1, the reason for the improvement of recycling algorithms against non-recycling schemes is that recycling algorithms can achieve savings in counting and projecting by means of compression.

In Figure 5.12, HM-MLP that recycles patterns based on MLP performs even worse than H-Mine. This implies that simply maximizing compression can even worsen the situation. It is also observed that the recycling mining algorithms based on the two compression strategies nearly achieve the same performance on dense data sets. This is because the two strategies nearly do the same compression as shown in Table 5.3.

5.5.3 Mining with memory limitation

This subsection will consider the case that the compressed datasets (and hence the original datasets) cannot be held in the main memory. As discussed in [72], the mining algorithms HM-MLP and HM-MCP can compute the size of memory usage in the same manner as the H-Mine algorithm because they adopt similar data structure. The memory usage of FP-tree and Tree Projection algorithms can not be effectively estimated, and it is difficult to enforce memory limitation using FP-tree and Tree Projection algorithms. As a result, this subsection does not compare FP-tree and Tree Projection with their recycling algorithms.

Memory limitation is enforced to 4 and 8 megabytes. Such limitations are used because they can imitate the memory limitation situations considering the size of datasets although it is realized such limitations are small compared to the available memory in current PC. The compressed databases are generated using the same set of recycled patterns as that in Section 5.5.1. Figures 5.21–5.24 show that HM-MCP outperforms H-Mine. Figures 5.23 and 5.24 use logarithmic scale for y-axes.

When comparing Figure 5.15 with Figure 5.23, readers may find that enforcing memory restriction on dense data set *Connect-4* even improves performance of both H-Mine and HP-MCP in some cases . Memory restriction requires that the (compressed) database be projected in the secondary storage until a level where the projected database can fit into memory. The projected (compressed) databases require less time in counting and the savings may be larger than the time used to read and write the projected (compressed) database.

Finally, all experiments show that in nearly all cases the saving of recycling algorithms over non-recycling counterparts is much greater than the time that is used to generate the set of frequent patterns for recycling.

In summary, the experimental results and performance analysis support the claim that recycling patterns is useful. Moreover, the experiments showed that the strategy of minimizing cost (MCP) is usually more effective than MLP for recycling patterns.

5.6 Discussion and summary

Obviously, the proposed technique in this chapter can be applied to incremental problems. Compared with existing incremental techniques [22, 69, 88], the proposed technique overcomes the following disadvantages of existing incremental techniques: (1) existing incremental techniques need to store the negative border or similar information from previous computation, which can take large amount of space; (2) they are not effective when the change of database or constraints are significant; (3) existing techniques become awkward when the size of data set reduces rather than increases. However, existing incremental mining techniques are not practical for recycling with constraint changes as discussed in Section 2.2.

Next, this section will give a qualitative comparison of the recycling approach in last chapter and the approach described in this chapter.

First, the method of handling of constraint changes in last chapter is dependent on the properties of constraints and is not applicable to certain constraints, for example $convertible$ and $hard$ constraints. The proposed technique in this chapter gives a non-intrusive method of reusing patterns in previous computation no matter what type of constraints that are being used.

Second, the approach of this chapter does not make any assumption that old mining process realizes and makes preparation for subsequent mining. Thus, it is more applicable to recycling in a multi-user environment. The method of last chapter relies on intermediate results and the intermediate results can be too large to handle.

Third, the underlying reasons for saving of the two approaches are different. The method of last chapter only mine the set of frequent patterns that satisfy the new constraints BUT NOT the old constraints. Therefore the saving mainly comes from the frequency information to be recycled. It usually performs good in the case that the set of frequent patterns with new constraints mainly comes from the previous mining results when the changes of constrains are not significant. On the contrary, the improvement may not be significant. This is shown in the experiments of last chapter. For the approach described in this chapter, the saving is not directly from the frequency information computed in previous process. The approach will mine the set of complete frequent patterns that satisfy the new constraints (also satisfy the old constraints) no matter how much the constraint are changed.

In a summary, this chapter shows how frequent patterns discovered in the early round of mining (by the same user or different users) can be recycled to enhance subsequent mining. This chapter describes a two phase strategy that first compresses the database based on frequent patterns from an early round of mining and then mine the compressed database. This chapter presents two compression strategies and three existing mining algorithms are adapted to work on compressed databases. The experimental results show that the proposed strategies are effective, and the proposed recycling algorithms outperform their non-recycling counterparts significantly. The results also show that a cost-based compression strategy is preferred over a storage-based strategy.

Chapter 6

Mining Frequent Closed Patterns for Microarray Datasets

This chapter will describe three algorithms adopting row enumeration strategy to mine the microarray datasets with a large number of columns and only a few rows.

6.1 Introduction

6.1.1 Properties of microarray datasets

With recent advances in DNA chip-based technologies [81], we can now measure the expression levels of thousands of genes in cell simultaneously, which results in a large amount of high-dimensional data. These microarray datasets typically have a large number of columns but a small number of rows. For example, many gene expression datasets may contain up to tens of thousands columns but only tens of hundreds rows.

6.1.2 Usefulness of frequent patterns in microarray datasets

In the biological analysis, the frequent patterns have the following potential applications:

1) To discover association rules. Association rules are suggested to be very promising in helping uncover gene networks [29]. Associations rules can describe the associations between the expression of one gene with the expression of a set of other genes. If such discovered association rules indicate reasonable cause and effect relationship that

can be proved by the prior biological knowledge or further investigation, one might easily infer some type of gene networks. Then similar to cluster analysis, one could infer some function for a gene whose function is not known from those genes by means of association rules. However, association rules might reveal more patterns than clustering [31, 87], considering that a gene may belong to many rules while it is usually grouped to one cluster (or a hierarchy of clusters in hierarchical clustering method).

Moreover, when there are two (or more) categories of samples in microarray datasets, association rules can help to identify a subset of genes/proteins which can identify the category that a sample belongs to.

2) To discover bi-clustering of gene expression [4] or help in performing subspace clustering on these datasets [21]. The bi-clustering [21] is a bit similar to frequent patterns and [105] applied frequent pattern mining algorithm to discover bi-clusters.

6.1.3 Feasibility analysis of algorithms

Such new high-dimensional biological datasets pose a great challenge for existing frequent pattern discovery algorithms. While there are a large number of algorithms that had been developed for frequent pattern discovery [2, 7, 43, 58, 72, 82, 102], their basic approaches are all column enumeration in which combinations of columns are tested systematically to search for frequent patterns. As a result, their running time increases exponentially with increasing average length of the records. The high dimensional bioinformatics datasets with thousands of items render most of these algorithms impractical. The same trend holds even for recent work on closed pattern mining [11, 70, 73, 101] which aims to find non-redundant patterns from the data.

Most previous work on (closed) frequent pattern mining assumes that the average number of columns in a dataset is much smaller than the number of rows. The length of a frequent pattern is obviously limited by the row length. If i is the maximum length of a row, the longest frequent pattern could have length i, and the number of possible frequent patterns will be 2^i based on the Apriori principle. Previous pattern mining methods work well for datasets with small average row length (usually $i < 100$). However, for the datasets taken from the bioinformatics domain (or other domains with similar data characteristics), i can be in the range of tens of thousands. As a result, the column search space is simply too large.

On the other hand, the number of rows in such datasets is typically on the order of hundreds to a thousand. If m is the number of rows, the size of the row subset space is 2^m. In the application domain (e.g., microarray datasets), the possible row set space is much less than the possible item set space since $m \ll i$. Therefore, it seems reasonable to devise the algorithm that does not search column set space, but rather search the row set space. One challenge here is whether row enumeration method is able to mine the same set of complete and sound frequent patterns as column enumeration method does. So far, none of existing studies investigate the possibilities of discovering rules in row enumeration space [1]. The other challenge is how to design algorithms for row enumeration if it is feasible for rule mining.

This chapter will describe three algorithms that implement the row enumeration strategy in different ways to mine frequent closed patterns. This first algorithm, called CARPENTER [2], is inspired by algorithms [15] [3] and [72]. The second algorithm called RERII [4] is inspired by algorithms that mine patterns from vertical layout data [82, 101, 102]. The third algorithm called REPT [5] is inspired by algorithms that are based on FP-tree [43, 73]. Although the three row enumeration algorithms are inspired by existing column enumeration methods, the three proposed algorithms are very different from the existing algorithms in that they adopt row enumeration which requires

[1] As discussed in Chapter 1, the row-wise concept in [34] is completely different from the row enumeration concept in this thesis

[2] CARPENTER stands for Closed Pattern Discovery by Transposing Tables that are Extremely Long; the ”ar” in the name is gratuitous.

[3] BUC is developed to compute iceburg cube on relational data.

[4] Row Enumeration by Row ID Intersection

[5] Row Enumeration based on Prefix Tree

i	r_i
1	b,d,e,f
2	a,c,e,f
3	a,c,d,e
4	a,b,c,d,e,g
5	a,b,c,d,e,f

Figure 6.1: Example Table

i_j	$\mathcal{R}(i_j)$
a	2,3,4,5
b	1,4,5
c	2,3,4,5
d	1,3,4,5
e	1,2,3,4,5
f	1,2,5
g	4

Figure 6.2: Transposed Table

i_j	$\mathcal{R}(i_j)$
e	3,4,5
f	5

Figure 6.3: 12-Projected Transposed Table

different implementation and pruning methods. Several experiments are performed on real-life microarray data to show that the new algorithms are much faster than the existing algorithms, including CLOSET [73], CLOSET+[92] and CHARM[101].

The remaining chapter is organized as follows: it first introduces some background and preliminaries of the problem: mining frequent closed patterns from microarray datasets. The three proposed algorithms will be explained in Sections 6.3, 6.4 and 6.5 respectively. To show the superiority of the proposed algorithms for mining microarray datasets, experiments are conducted in Section 6.6 on real-life biological data. This chapter will be concluded in Section 6.7.

6.2 Problem Definition and Preliminary

6.2.1 Problem definition

Let I=$\{i_1, i_2, .., i_m\}$ be a set of items. Let D be the dataset (or table) which consists of a set of rows R=$\{r_1, .., r_n\}$ with each row r_i consisting of a set of items in I, i.e $r_i \subseteq I$. Figure 6.1 shows an example dataset in which the items are represented using alphabets ‘a’ to ‘g’. There are altogether 5 rows, r_1,...,r_5 in the table. The first row r_1 contains the items ‘b’,‘d’, ‘e’ and ‘f’. To simplify notation, in the sequel, we will denote a set of row numbers like $\{r_2, r_3, r_4\}$ as ”234”. Likewise, a set of items like $\{a, c, f\}$ will also be represented as acf.

Definition 6.2.1. Item Support Set, $\mathcal{R}(I')$

Given a set of items $I' \subset I$, we use $\mathcal{R}(I') \subset R$ to denote the largest set of rows that contain I'. □

Definition 6.2.2. Row Support Set, $\mathcal{I}(R')$

Given a set of rows $R' \subset R$, we use $\mathcal{I}(R') \subset I$ to denote the largest set of items that are common among the rows in R'. □

Example 6.2.1. $\mathcal{R}(I')$ **and** $\mathcal{I}(R')$

Consider the table in Figure 6.1. Let I' be the item set $\{b, d, e\}$, then $\mathcal{R}(I') = \{r_1, r_4, r_5\}$ since these are all the rows in R that contain I'. Also let R' be the set of rows "14" (i.e. $\{r_1, r_4\}$), then $\mathcal{I}(R')$=$\{b, d, e\}$ since this is the longest pattern that occurs in both r_1 and r_4.

Definition 6.2.3. Support,$|\mathcal{R}(I')|/|D|$

Given a set of items I', the fraction of the number of rows in the dataset that contain I' is called the **support** of I'. Using earlier definition, we can denote the support of I' as $|\mathcal{R}(I')|/|D|$. □

The definitions of closed patterns and frequent closed patterns have been given in Section 2.1. Next, the two definitions will be restated with the symbols introduced above. A set of items $I' \subseteq I$ is called a **closed pattern** if there exists no I'' such that $I' \subset I''$ and $|\mathcal{R}(I'')| = |\mathcal{R}(I')|$. In other words, a pattern is considered closed when the set of rows containing the superset I'' is not exactly the same as the set of rows containing I'. A set of items $I' \subseteq I$ is called a **frequent closed pattern** if (1) $|\mathcal{R}(I')|/|D|$, the support of I' is higher than a minimum support threshold, ξ; (2) I' is a closed pattern.

Example 6.2.2. Given that $\xi = 40\%$, the itemset $\{a, c, e\}$ is a frequent closed pattern in the table of Figure 6.1 since it occurs four times in the table. $\{a, c\}$ on the other hand is not a frequent closed pattern although its support is more than the ξ threshold. This is because it has a superset $\{a, c, e\}$ such that $|R(\{a, c, e\}) = |R(\{a, c\})|$.

Problem Definition: Given a dataset D which contains records that are subset of a set of items I, the problem is to discover all frequent closed patterns with respect to a user given support threshold ξ. In addition, we assume that the database satisfies the condition $|R| << |I|$.

6.2.2 Preliminary

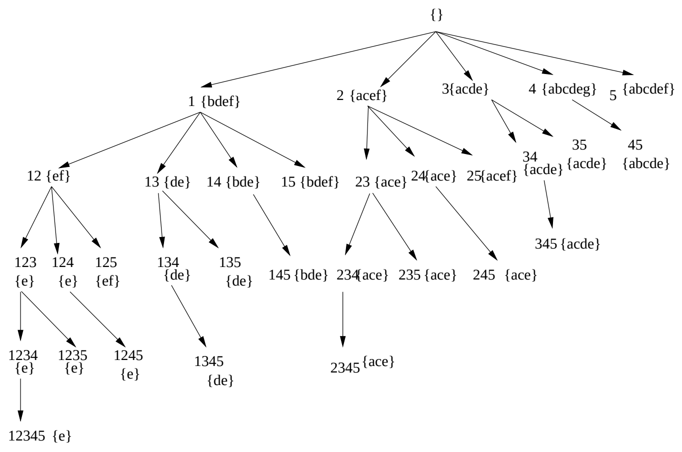


Figure 6.4: The row enumeration tree.

Table in Figure 6.2 is a transposed version of the table in Figure 6.1. In the transposed table TT, the items become the row ids while the row numbers become the items. A row number i in the original table will only appear in a row f_j in the transposed table if the item f_j occurs in row r_i of the original table. To avoid confusion, we will hereafter refer to the rows in this transposed table as **tuples** while referring to those in the original table as **rows** [6].

Let X be a subset of rows. Given the transposed table TT, a X**-projected transposed table**, denoted as $TT|_X$, is a subset of tuples from TT such that: 1) For each tuple x in TT, there exists a corresponding tuple x' in $TT|_X$. 2) x' contains all rows in x with

[6]The tuples in the transposed table actually represent the items in the original table

row ids larger than any row in X. As an example, let the transposed table in Figure 6.2 be TT and let $X = 12$. The X-projected transposed table, $TT|_X$ is as shown in Figure 6.3.

Figure 6.4 shows the complete row enumeration tree on table in Figure 6.2. The row enumeration tree represents the maximal search space of row enumeration when there is no any pruning technique to be applied. Each node of the enumeration tree represents a combination of rows which are denoted using the convention we introduced earlier. Given a node representing a subset of rows R', we also show $\mathcal{I}(R')$ at the node for use in later discussion. For example, the node "12" represents $\{r_1, r_2\}$ and "ef" indicates that $I(\{r_1, r_2\}) = \{e, f\}$.

In what follows, the thesis will give the proof that the complete set of closed patterns can be discovered by means of row enumeration.

Lemma 6.2.1. Let I be a closed pattern and $\mathcal{R}(I)$ be the subset of tuples from the original table that contains I. $\mathcal{R}(I)$ is unique. In other words, there does not exist a closed pattern I', $I' \neq I$, that satisfy $\mathcal{R}(I) = \mathcal{R}(I')$.

Proof: We will prove by contradiction. Assuming there exists a closed pattern I' that satisfies $\mathcal{R}(I) = \mathcal{R}(I')$ but $I' \neq I$. Let pattern CF = $I' \cup I$. Then $\mathcal{R}(CF)$ = $\mathcal{R}(I) = \mathcal{R}(I')$. Since $I' \subset CF$ contradicts with the definition of closed pattern. So we can say that such a F' does not exist. □

Lemma 6.2.2. Let X be a subset of rows from the original table, then $\mathcal{I}(X)$ must be a closed pattern (not necessary frequent).

Proof: We will prove by contradiction. Assuming $\mathcal{I}(X)$ is not a closed pattern, then there exists a item i_j such that $\mathcal{R}(\mathcal{I}(X)) = \mathcal{R}(\mathcal{I}(X) + i_j)$. Since X contains all items of $F(X)$, then $X \subset \mathcal{R}(\mathcal{I}(X))$. This means that i_j is also found in every row of X which contradicts the definition that $\mathcal{I}(X)$ is the largest set of items that are found in every row of X. The largest set in this case is $X + i_j$. □

The main observation used in the proof is that $\mathcal{I}(X)$ cannot be a maximal item sets that are common in all rows of X unless it is a closed pattern. Lemma 6.2.2 ensures that only closed patterns will be enumerated in the search tree. As a result, we need not perform detection of superset-subset relationship among the patterns while mining algorithms using column enumeration, such as CHARM and CLOSET, need to do that. For example, in Figure 6.4, it is not possible for the pattern $\{a, c\}$ to be enumerated although both $\{e\}$ and $\{a, c, e\}$ are closed patterns with support of 100% and 80% respectively. This is unlike CHARM and CLOSET, both of which will enumerate $\{a, c\}$ and check that it has the same support as a superset $\{a, c, e\}$ before discarding it as a non-closed pattern.

With Lemma 6.2.1, we know that each closed pattern corresponds to a unique set of rows in the original table. By enumerating all combinations of rows as shown in the enumeration tree of Figure 6.4, we can be sure that all closed patterns in the datasets are enumerated. Together with Lemma 6.2.2, we know that the complete and correct set of frequent closed patterns will be discovered by row enumeration.

It is obvious that a complete traversal of the row enumeration tree is not efficient and pruning techniques must be introduced to prune off unnecessary searches. Moreover, the efficient implementation of row enumeration is also a trick problem. Both of the two problems will be addressed in the three proposed algorithms.

6.3 CARPENTER algorithm

6.3.1 Algorithm overview

By imposing an order $\mathcal{ORD}$ based on the row number, we are able to perform a systematic search by enumerating the combinations of rows based on lexicographical order. For example, the order of search on the row enumeration tree in Figure 6.4 will be $\{$1, 12, 123, 1234, 12345, 1235,...,45, 5$\}$ (in absence of any optimization and pruning strategies).

CARPENTER performs a depth first search on the enumeration tree and recursively

Algorithm CARPENTER (D, ξ)

```
1. Scan database $D$ to find the set of frequent items $F$;
2. Remove the infrequent items in each row $r_i$ of $D$;
3. Transpose $D$ to get transposed table $TT$;
4. $FCP = \emptyset$. Let $R$ be the set of rows in the original table in the order $\mathcal{ORD}$;
5. MinePattern($TT|_\emptyset$,$R$, $FCP$);
6. Return $FCP$;

Procedure: MinePattern($TT'|_X$,$R'$, $FCP$).

7. Scan $TT'|_X$ and count the frequency of occurrences for each row, $r_i \in R'$. $Y = \emptyset$;
8. Pruning 1: Let $U \subset R'$ be the set of rows in $R'$ which occur in at least one tuple of
   $TT'|_X$. If $|U| + |X| \leq \xi$, then return; else $R' = U$;
9. Pruning 2: Let $Y$ be the set of rows which are found in every tuple of the $X$-projected
   transposed table. Let $R' = R' - Y$ and remove all rows of $Y$ from $TT'|_X$;
10. Pruning 3: If $\mathcal{I}(X) \in FCP$, then return;
11. If $|X| + |Y| \geq \xi$, add $\mathcal{I}(X)$ into $FCP$;
12. For each $r_i \in R'$, $R' = R' - \{r_i\}$ call MinePattern($TT'|_X|_{r_i}, R', FCP$);
```

Figure 6.5: The CARPENTER algorithm

generates projected transposed table. For example, at the root node that corresponds to the transposed table in Figure 6.2, CARPENTER will first project on row r_1 and build r_1-projected transposed table. Based on r_1-projected transposed table, CARPENTER will project on row r_2 to enumerate the combination r_1r_2 and so on.

6.3.2 Algorithm design

The formal algorithm is shown in Figure 6.5. There are two input parameters, the dataset D and the minimum support ξ. After doing initialization, CARPENTER calls the procedure $MinePattern$ to perform depth-first traversal of row enumeration tree. The procedure $MinePattern$ takes in three parameters $TT'|_X$, R' and FCP. TT'_X is a X-projected transposed table. R' contains the set of rows that will be used to enumerate the next level of projected transposed table while FCP contains the frequent closed patterns that have been discovered so far.

Steps 7 to 10 in the procedure perform the counting and pruning. They are extremely important to the efficiency of CARPENTER Algorithm and will be explained later. According to Lemma 6.2.2, $MinePattern$ procedure will output a pattern if and only if it is a closed pattern. Step 11 checks whether $\mathcal{I}(X)$ is a frequent closed pattern before inserting $\mathcal{I}(X)$ into FCP, and Step continues the next level of enumeration in the search tree.

Lemma 6.3.1. $TT'|_X|_{r_i} = TT'|_{X+r_i}$ $\Box$

Lemma 6.3.1 is useful for explaining Step 12. It simply states that a $X+r_i$ projected transposed table can be computed from a X projected transposed table, $TT'|_X$, by selecting those tuples that contain r_i in $TT'|_X$, i.e. $TT'|_X|_{r_i}$. This is in fact generating the $\{X + r_i\}$ projected transposed table that is needed to represent the next level of row set enumeration.

Note that Step 12 implicitly represents a form of pruning too since it is possible to have $R' = \emptyset$. It can be observed from the enumeration tree that there exist some combinations of rows, X, such that $\mathcal{I}(X) = \emptyset$. This implies that there is no item which exists in all the rows in X. When this occurs, R' will be empty and no further enumeration will be performed.

We next look at the pruning techniques that are used in CARPENTER to enhance its efficiency. The emphasis here is to show that the pruning steps do not prune off any frequent closed patterns while preventing unnecessary traversal of the enumeration tree. Combining this with the earlier explanation on how all frequent closed patterns are enumerated in CARPENTER without the pruning steps, the correctness of the algorithm will be obvious.

The first pruning step is executed at Step 8 of $MinePattern$. The pruning is essentially aimed at removing search branches which cannot yield closed patterns that satisfy the xi threshold. The following lemma is applied in the pruning.

Lemma 6.3.2. Let $TT'|_X$ be a X projected transposed table. Let U be a set of rows which occur in at least one tuple of $TT'|_X$. If $|U| + |X| < \xi$, then it is **not** possible that for any

$U' \subset U, \mathcal{I}(X + U')$ is a frequent closed pattern.

Proof: All rows that are not in the X projected transposed table will create an empty pattern if they are combined with X, thus these rows are of no interest to us. As such X can only be combined with some $U' \subset U$ in order to continue the enumeration.

The maximum support in further enumeration is however bounded by $|U| + |X|$. Since $|U| + |X| < \xi$, we can safely conclude that all the patterns in further enumeration will not be frequent. □

At Step 9 of $MinePattern$, the second pruning strategy is applied. This pruning deals with rows that occur in all tuples of the X projected transposed table. Such rows are immediately removed from $TT'|_X$ because of the following lemma:

Lemma 6.3.3. Let $TT'|_X$ be a X projected transposed table and Y be a set of rows which occur in every tuple of $TT'|_X$. Given any subset $R' \subset R$, we have $\mathcal{I}(X + R') = \mathcal{I}(X + Y + R')$.

Proof: By definition, $\mathcal{I}(X + R')$ contains a set of items which occur in every row of $X + R'$. Since the rows in Y occur in every tuple of $TT'|_X$, this means that these rows also occur in every tuples of $TT'|_{\{X+R'\}}$ (Note: $TT'|_{\{X+R'\}} \subset TT'|_X$). Thus, the set of tuples in $TT'|_{\{X+R'\}}$ is exactly the set of tuples in $TT'|_{\{X+R'+Y\}}$. From this, we can conclude that $\mathcal{I}(X + R') = \mathcal{I}(X + Y + R')$. □

The final and most complex pruning strategy is shown at Step 9 of $MinePattern$. This step will prune off any further search down the branch of node X if it is found that $\mathcal{I}(X)$ was already discovered previously in the enumeration tree. The intuitive reasoning which we will prove later is as follows: the set of closed patterns that will be enumerated from the descendants of node X must have been enumerated previously.

Another important thing to note here is that the correctness of the third pruning strategy (Step 9) **is dependent on the second pruning criteria**. This is essential because of the following lemma.

Lemma 6.3.4. Let X be the set of rows in the current search node and X' be the set of rows that result in $\mathcal{I}(X)$ (which is the same as $\mathcal{I}(X')$) being inserted into FCP in earlier enumeration. If pruning strategy 2 is applied consistently in the algorithm, then the node representing X in the enumeration tree will not be the descendent of the node representing X'in the enumeration tree.

Proof: Assume otherwise, then $X' \subset X$. Let $Z = X - X'$. Since $\mathcal{I}(X) = \mathcal{I}(X')$, all rows in Z must be contained in all tuples of the X' projected transposed table. Based on pruning strategy 2, the rows in Z would be added to X' and will be removed from subsequent transposed table down that search branch. Thus the node representing X will not be visited, which contradicts the fact that node X is currently being processed in the enumeration tree. $\Box$

Lemma 6.3.4 shows that it is NOT possible to prune off the branches of a node simply because they represent the same item sets as an ancestor node in the enumeration tree. Again, we emphasize that this come hand in hand with the second pruning strategy.

We next try to prove that all branches from a node X in the enumeration tree can be pruned off if $\mathcal{I}(X)$ is already in FCP. We have the following lemma.

Lemma 6.3.5. Let $TT'|_X$ be the projected transposed table in the current search node. Let X' be the set of rows which result in $\mathcal{I}(X)$ (which is the equals to $\mathcal{I}(X')$) being inserted into FCP in earlier enumeration. Let x_{i_j} and x'_{i_j} be the two tuples that represent item i_j in $TT'|_X$ and $TT'|_{X'}$ respectively. We will have $x_{i_j} \subset x'_{i_j}$ for all $i_j \in \mathcal{I}(X)$.

Proof: We know that $\mathcal{I}(X) = \mathcal{I}(X')$ which implies that the set of items represented by tuples in both the projected transposed tables will be the same.

Let the maximal set of rows that contains the item set $\mathcal{I}(X')$ be $R'_{max} = \{r'_1, ..., r'_n\}$ which is sorted based on the order $\mathcal{ORD}$. Let m be the minimum number such that $\mathcal{I}(\{r'_1, ..., r'_m\}) = \mathcal{I}(X')$. Denoting $\{r'_1, ..., r'_m\}$ as R'_{min}, an analysis of the enumeration tree based on $\mathcal{ORD}$ will tell us that the row set R'_{min} is the first combination of rows that cause $\mathcal{I}(X')$ to be inserted into FCP.

Based on Lemma 6.3.4, X cannot be a descendent of X' in the enumeration tree. Thus, X must be of the form $(R'_{min} - A) + B$ where $A \subset R'_{min}$ and $B \subset R'_{max} - R'_{min}$, $A \neq \emptyset$, $B \neq \emptyset$. We can conclude from here that there exists a row r'_i such that $i > m$ and $r_i \in X$.

By definition of a projected transposed table, we know that all rows which occur before r'_m (based on the order $\mathcal{ORD}$) will be removed in $TT'|_{X'}$. Likewise, all rows occurring before r'_i will be removed in TT'_X. Since $i > m$, a tuple x'_{i_j} representing item i_j in $TT'|_{X'}$ will have less rows being removed than the corresponding tuple x_{i_j} representing item i_j in $TT'|_X$. Hence the proof. $\Box$

In a less formal term, Lemma 6.3.5 shows that if X' is the first combination of rows that cause $\mathcal{I}(X')$ to be inserted into FCP, then the projected transposed table $TT'|_{X'}$ will be more "general" than any other projected transposed table $TT'|_X$ in which $\mathcal{I}(X) = \mathcal{I}(X')$. "General" in this case, refers to the fact that each tuple in $TT'|_{X'}$ is in fact a superset of the corresponding tuple in $TT'|_X$. We will now formalize the third pruning strategy as a theorem.

Theorem 6.3.1. Given a node representing a set of rows X in the enumeration tree, if $\mathcal{I}(X)$ is already in FCP, then all enumeration down that node can be pruned off.

Proof Let X' be the combination of rows that first cause $\mathcal{I}(X)$ to be inserted into FCP. From Lemma 6.3.5, we know that any tuple x'_{i_j} in the X' projected table will be a superset of a corresponding tuple x_{i_j} in the X projected table. Since we know that the next level of search at node X in the enumeration tree is based on the set of rows in the X projected transposed table, it is easy to conclude that the possible enumeration at the node X is a subset of the possible enumeration at node X'. Since X' had been visited, it is thus not necessary to perform any enumeration from the node X onwards. $\Box$

The implementation of CARPENTER uses memory pointers to point at the relevant tuples in the in-memory transposed table to simulate the projected transposed table.

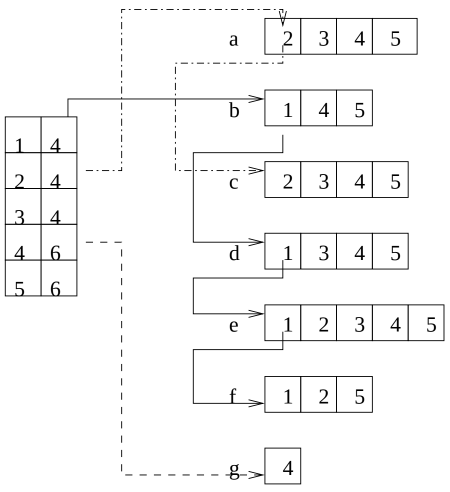


Figure 6.6: Pointer lists at node {1}.

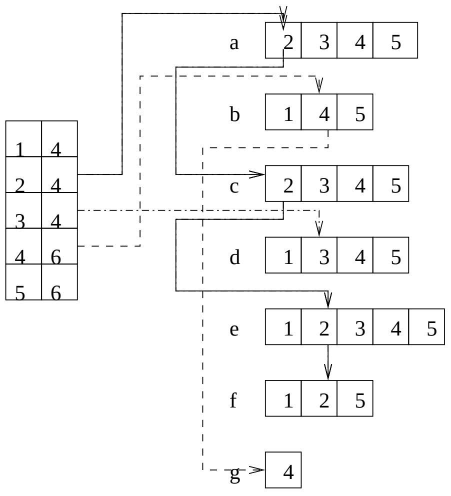


Figure 6.7: Pointer lists at node {2}.

We will illustrate the construction of memory pointers using memory pointers with an example here and interested readers are referred to [15, 72] for details.

The implementation assumes that despite the high dimensionality, the biological datasets that we are trying to handle are still sufficiently small to be loaded completely into the main memory. This is true for many gene expression datasets which have only small number of rows (usually from 100 to 300). We will discuss the case when the data cannot fit in memory in Chapter 8.

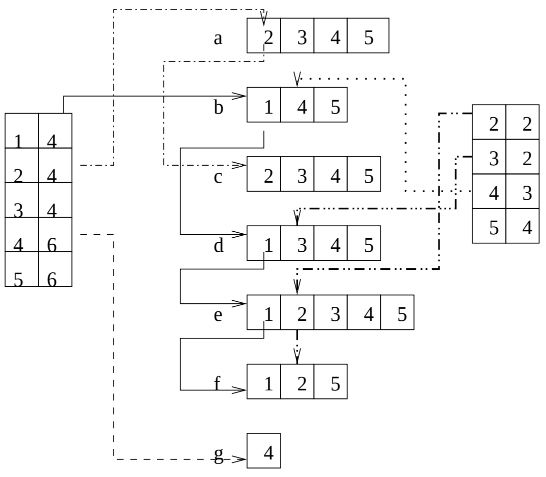


Figure 6.8: Pointer lists at node {12}.

Given the transposed table in the running example, we show the state of memory pointers when we are at node $\{1\}$ of the enumeration tree in Figure 6.6 assuming that $\xi = 1$. The in-memory transposed table is shown on the right hand side of the figure and the head table is shown in the left hand side of the figure. Each entry of the head table is composed of a row id, its frequency in the transposed table of its parent node and a pointer pointing to the transposed table corresponding to the entry.

In Figure 6.6, we can derive the 1-projected transposed table $TT'|_1$ by following the pointer list of the entry 1 in the head table. Since generating the 1-projected pointer list requires a full scan of the transposed table, CARPENTER also generates the pointer lists of r_2, r_3, r_4 and r_5 projected transposed databases on the way. However, the generation of the pointer list of entry 2 is slightly different. It now contains tuples that contain r_2 BUT NOT r_1. For example, although the tuple representing item 'e' contains row r_2, it does not appear in the the pointer list of entry 2. It will be inserted subsequently as we will see later.

A scan through the pointer list of entry 1 will allow us to generate the pointer lists of 12, 13, 14 and 15 projected transposed databases. Figure 6.8 shows the state of memory pointers when we are processing node $\{1, 2\}$. Again, note that item e is not in linked with the entry 3 (13 projected database) although the corresponding tuple does contain the set of rows $\{r_1, r_3\}$. This is because we will first process the combination $\{1, 2\}$ and move entries in pointer list of 12 to the other pointer lists (including combinations 13,14 and 15 here) later. As a result, the current pointer list of 13 only indicates tuples that have $\{1, 3\}$ BUT NOT $\{1, 2\}$.

Finally, we show the state of pointer lists after node $\{1\}$ and all its descendants have been processed in Figure 6.7. Since all enumerations involving row r_1 have been either processed or pruned off, the tuples linked by pointer list of projected database 1 are moved into the pointer lists of 2, 3, 4 and 5 projected databases. For example, tuples corresponding to items *e* and *f* are appended to the pointer list of entry 2. Subsequent enumerations can then continue in similar fashion and the items in the pointer list of entry 2 will be moved to the pointer lists of other entries after node $\{2\}$ and its descendants are processed.

Throughout all the enumerations described above, the three pruning strategies need to be implemented. The implementation of strategies 1 and 2 is straightforward. For pruning strategy 3, a trie structure [51] is used to perform the full pattern matching.

6.4 Algorithm RERII

In this section, we will first give an overview of algorithm Row Enumeration by Row ID Intersection(RERII), then describes the implementation and optimization strategies that make RERII more efficient.

6.4.1 Algorithm overview

In RERII, each node X in Figure 6.4 will be represented with a three-element group $X = \{itemlist, sup, childlist\}$, where $itemlist$ is the closed pattern corresponding to node X, sup is the number of rows at the node and $childlist$ is the list of child nodes of X. For example, the root of the tree can be represented with $\{\{\}, 0, \{1, 2, 3, 4, 5\}\}$ and the node ”12“ can be represented with $\{\{1, 2\}, 2, \{3, 4, 5\}\}$.

Given a node X in the row enumeration tree, we will perform an intersection of the $itemlist$ of node X with the $itemlist$s of all its sibling nodes after X. Each intersection will result in a new node (Note that the intersection may be pruned as discussed later) whose $itemlist$ is the intersection, whose sup is $X.sup + 1$ and whose $childlist$ will be available at next level intersection. And each new node will be intersected with its afterward siblings. In this way, the row enumeration tree will be recursively expanded in a depth-first way. For example, at node 1, we will intersect its $itemlist$ with those of nodes 2,3,4 and 5. RERII improves on the basic enumeration scheme by using a series of optimization strategies (introduced in the next subsection) to prune the search space.

RERII intersects the itemlists to obtain closed patterns. However, the algorithms of finding frequent (closed) patterns from vertical layout dataset, such as [82, 101, 102], join tidlist to obtain the support of a pattern which needs to be tested to determine whether it is closed or not in closed pattern mining algorithms. With the essential difference, the optimization techniques in these methods cannot be applied in RERII and we specially design a series of optimization strategies for RERII to improve its efficiency.

6.4.2 Algorithm design

This subsection will describe how to expand a node in the row enumeration tree and the optimization strategies to improve the efficiency of such expansion.

Given a node Xr_i and a node Xr_j, where Xr_i and Xr_j have the same parent X and $Xr_i < Xr_j$. We will perform intersections of the $itemlist$ of Xr_i with that of node Xr_j. There are several possibilities of such intersection and the following lemma explores these possibilities based on the properties of intersected $itemlist$s.

Lemma 6.4.1. Let Xr_i and Xr_j be two sibling nodes, where $Xr_i < Xr_j$. The following five properties will hold:

1) If $Xr_i.itemlist \cap Xr_j.itemlist = \emptyset$, nothing needs to be done.

2) If $Xr_i.itemlist = Xr_j.itemlist$, Xr_j will be integrated into Xr_i, i.e. $Xr_i.sup = Xr_i.sup + 1$ and any further expansion below Xr_j will be pruned.

3) If $Xr_i.itemlist \subset Xr_j.itemlist$, $Xr_i.sup = Xr_i.sup + 1$ and Xr_j will not expand Xr_i.

4) If $Xr_i.itemlist \supset Xr_j.itemlist$, any further expansion below Xr_j will be pruned and Xr_j will become a candidate extension of Xr_i. (Note that whether Xr_j will be a true extension of Xr_i is pending other checking introduced later.)

5) If $Xr_i.itemlist \neq Xr_j.itemlist$, Xr_j will become a candidate extension of Xr_i.

Proof:

Let Xr_k be an afterward sibling node of Xr_j ($Xr_k < Xr_j < Xr_i$). 1) If $Xr_i.itemlist \cap$

$Xr_j.itemlist = \emptyset$, it is obvious that there is no meaning to test such a row enumeration since it cannot generate any closed pattern. 2) If $Xr_i.itemlist = Xr_j.itemlist$, it is obvious that $Xr_k.itemlist \cap Xr_j.itemlist = Xr_k.itemlist \cap Xr_i.itemlist$. Therefore, patterns enumerated below Xr_j can be obtained below Xr_i and we can prune node Xr_j for further enumeration. 3) If $Xr_i.itemlist \subset Xr_j.itemlist$, $Xr_i.itemlist \cap Xr_j.itemlist = Xr_i.itemlist$. This means that any closed patterns discovered below Xr_i will be contained by the row r_j. Therefore there is no meaning to extend Xr_i with Xr_j and we increase $Xr_i.sup$ by 1. 4) If $Xr_i.itemlist \supset Xr_j.itemlist$, $Xr_k.itemlist \cap Xr_j.itemlist \cap Xr_i.itemlist = Xr_k.itemlist \cap Xr_j.itemlist$. So we can prune Xr_j for further enumeration. 5) If $Xr_i.itemlist \neq Xr_j.itemlist$, it is possible that Xr_j will be an extension of Xr_i.

The properties in Lemma 6.4.1 are different from the strategies used in CARPENTER to determine whether a node is expanded by a row. CARPENTER does that by checking the frequency of a row in the projected transposed table and needs to record the row combination information. On the contrary, Lemma 6.4.1 only depends on the item information but not row information.

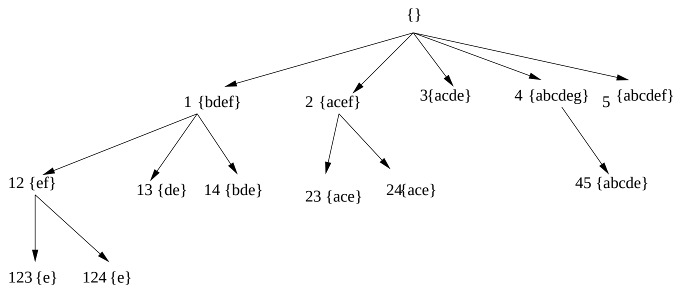


Figure 6.9: The pruned row enumeration tree.

Example 6.4.1. We now illustrate the Lemma 6.4.1 with the example table in Figure 6.1. Suppose minimum support = 1, let us look at how to apply Lemma 6.4.1 to prune the complete row enumeration tree shown in Figure 6.4. Consider node 1, its itemlist is a subset of that of node 5 (case 2) while the intersection of its itemlist with the others satisfies the case 5. As a result, we increase the support of node 1 by 1 and extend node 1 with nodes 2, 3 and 4 to get three child nodes. Next we process the node 12, the intersection of the itemlist of 12 with the itemlist of 13 and 14 satisfies the case 5 and we extend 12 with 13 and 14. At the node 123, the intersection of its itemlist with that of 124, i.e. e satisfies with case 2. In this way, we get a close pattern $\{e\}$ with support=4. Next we proceed to node 13 and the intersection of its itemlist with that of node 14, i.e. $\{de\}$, satisfies case 3. Thus we get a closed pattern de with support =3. The extension is done in a depth-first way. The nodes that are actually checked are shown in the Figure 6.9.

The first three cases in Lemma 6.4.1 are preferable over others since they will not result in deeper row enumeration tree. The first two cases are not affected by the order of $childlist$ while sorting $childlist$ of a node in the ascending order of the number of their $itemlist$s will increase the opportunity that the third case of Lemma 6.4.1 is satisfied. In Example 3, the lexicographic order of rows has already followed the ascending order of number of items in their $itemlist$s. We also notice that reordering strategy is widely used in algorithms searching frequent patterns in item enumeration space.

Before introducing other optimization techniques, we give the pseudo code of algorithm RERII in Figure 6.10. There are two input parameters of the algorithm, the dataset D and the minimum support ξ. In algorithm RERII(), infrequent items are removed and the child nodes of the root node are constructed. Then the procedure RERIIdepthfirst() is called to perform the depth-first traversal of row enumeration tree.

We further optimize the algorithm RERII using three techniques that will be explained as follows.

Single Item Pruning. RERII has already discovered the set of frequent single items by scanning the database once, we only need to discover those frequent closed patterns longer than 1. Therefore, if RERII finds that an enumeration node cannot result in pattern

```
Algorithm RERII(D, $\xi$)

1. Scan database $D$ to find the set of frequent items $F$;
2. Remove the infrequent items in each row $r_i$ of $D$;
3. Each $r_i$ forms a node in the first level of row enumeration tree and let $N$ be the set of nodes;
4. RERIIdepthfirst($N, FCP$);
5. Let $CF$ be the set of closed items in $F$, $FCP$ = $FCP \cup CF$ and return $FCP$;

Procedure: RERIIdepthfirst(N, FCP)

6.  for each node $n_i$ in $N$
7.      $N_i$ = null
8.      if the left row enumeration cannot be frequent return
9.      for each $n_j$ in $N$, where $n_j > n_i$
10.         compute the frequency of items to do support pruning
11.         d = $n_i.itemlist \cap n_j.itemlist$
12.         if $|d| > 1$
13.             if $n_i.itemlist = n_j.itemlist$
14.                 remove $n_j$ from $N$
15.                 increase $n_i.sup$ and $n'.sup$ ($n' \in N_i$) by 1
16.             if $n_i.itemlist \subset n_j.itemlist$
17.                 increase $n_i.sup$ and $n'.sup$ ($n' \in N_i$) by 1
18.             if $n_i.itemlist \supset n_j.itemlist$
19.                 remove $n_j$ from $N$
20.                 if $n_i.itemlist \cup d$ is not discovered before
21.                     add $n'$ ($n'.sup = n_i.sup + 1, n'.itemlist = d$) to $N_i$
22.             if ($n_i.itemlist \neq n_j.itemlist$)
23.                 if $n_i.itemlist \cup d$ is not discovered before
24.                     add $n'$ ($n'.sup = n_i.sup + 1, n'.itemlist = d$) to $N_i$
25.     end for
26.     if $n_i.sup \geq \xi$, add $n_i.itemset$ to $FCP$
27.     if $N_i \neq \emptyset$ and $N_i$ satisfies support pruning
28.         call RERIIdepthfirst($N_i$, $FCP$)
29. end for
```

Figure 6.10: Algorithm RERII

longer than 1, that node will be pruned. Algorithm RERII applies such an optimization at line 3 and line 12. At line 3, a row containing fewer than 2 frequent items can be pruned off since the intersection of the $itemlist$ of such a row with others cannot get patterns longer than 1. Similarly, at line 12 if the intersection of two $itemlist$s contains fewer than 2 items, there is no need to extend n_i with n_j. In addition, this pruning strategy will be applied in the next pruning strategy, called *support pruning.* One overhead of such a pruning method is that we need to check whether frequent single items are closed. RERII do this by checking whether the support of a discovered pattern P is equal to the support of its component items for each discovered frequent closed pattern. If it is the case, such items cannot be closed patterns.

Support Pruning. RERII tries to utilize support pruning by making use of both row information and item information while CARPENTER makes use of only row information. The support pruning can be done at three levels.

Level 1. This pruning is done at line 8 of RERIIdepthfirst(). Given a node X with k child nodes $Xr_1, Xr_2, ..., Xr_k$, for any child node Xr_i, if $Xr_i.sup + k - i < \xi$, there is no need to do any further enumeration below node Xr_i since any further enumeration cannot generate frequent closed patterns.

Level 2. This pruning is done at line 10 of RERIIdepthfirst(). Given a node X with k child nodes $Xr_1, Xr_2, ..., Xr_k$, for any child node Xr_i, we compute the supports for items in $X.itemlist$ (= $i_1, i_2, ...i_m$) in all nodes Xr_j such that $i \leq j \leq k$. The counter $support(i_l)$ for each item i_l in $X.itemlist$ is initialized with $X.sup$ and will be increased by 1 if the item is in $Xr_j.itemlist$. On the basis of *single item pruning*, we only need to discover patterns longer than 1. We can derive the following two pruning methods. First, if there are fewer than two items such that $i_l \in Xr_i$ and $support(i_l) \geq \xi$, there is no need to do any further enumeration below node Xr_i. Second, if there are fewer than two items such that $i_l \in X.itemset$ and $support(i_l) \geq \xi$, there is no need to do any further enumeration below nodes Xr_j ($i \leq j \leq k$).

Level 3. This pruning is done at line 27 of RERIIdepthfirst() after N_i is filled, i.e. the child nodes of n_i are obtained. We will recompute the support of items in $itemlist$ of n_i in the child nodes of n_i. If there are fewer than two items such that $i_l \in n_i.itemlist$ and $support(i_l) \geq \xi$, there is no need to do any further enumeration below node n_i and all its child nodes will be removed.

Redundant Pruning. The last pruning strategy is based on the following Lemma which is similar to the Theorem 6.3.1.

Lemma 6.4.2. On the basis of the lemma 6.4.1, at a node X, if pattern $X.itemlist$ has already been discovered in an earlier enumeration, we can prune node X and any further enumerations below X.

RERII adopts prefix tree to store the discovered patterns to save memory usage as CLOSET [73] and CLOSET+ [92] do. But we only check the equal relationship between patterns while CLOSET needs to check both equal and subsumption relationship. RERII builds a two level hash function on each pattern to speed up searching. The first hash function adopts the sum of items in the pattern and the second level hash function uses the last item in a pattern. We will store the length information of a pattern with its last item. When a pattern is hashed to a node in the prefix tree, we first check whether the length of the two patterns is the same. If it is case, we go up along the prefix tree to compare the whole patterns. Interested reader can refer to [73, 92] for details.

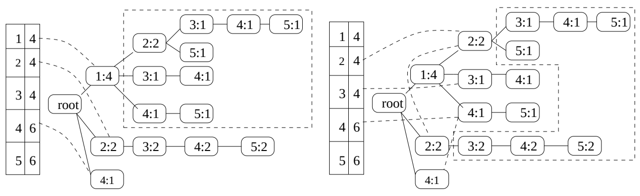
6.5 Algorithm REPT

This section will present another row-enumeration algorithm called Row Enumeration based on Prefix Tree (REPT).

6.5.1 Algorithm overview

Like CARPENTER, algorithm REPT traverses the row enumeration tree with the help of projected transposed table. Its first main difference from CARPENTER is that REPT represents (projected) transposed table with prefix trees, which can help in saving memory and saving computation in counting frequency. The second main difference of REPT from CARPENTER is that REPT designs a backward pruning method to implement Theorem 6.3.1 while CARPENTER uses a similar method as RERII. The prefix tree used to represent transposed table is similar to the FP-tree used in [43] to represent original table (horizontal layout table). In FP-tree, each node represents an item while the node of prefix tree used in REPT represents a row.

Next, we will illustrate how to represent (projected) transposed tables with prefix trees. REPT represents the transposed table in Figure 6.2 with prefix tree shown in Figure 6.11(a). The left head table in the figure records the list of rows in the transposed table and their frequencies. At each node of the prefix tree, REPT records row number and the count of the row in a prefix path (separated by":" in Figure 6.11(a)). Another information recorded at each node but not shown in the figure is the set of items represented at the node, such as items b, d, e and f at node 1:4. Such information will help REPT to know quickly the pattern corresponding to a projected transposed table.



(a) $PT|_1$ (b) $PT|_2$

Figure 6.11: The Projected Prefix Tree.

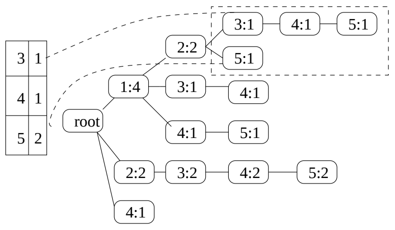


Figure 6.12: The 12-projected prefix tree $PT|_{12}$.

Example 6.5.1. Projected Prefix Tree

The parts of nodes enclosed by dotted line in Figure 6.11(a) is the 1-projected prefix tree, $PT|_1$. Note that there are pointers linking the child nodes of root with corresponding row in head table. By following the pointer starting from the row 1 of header table, we can get the $PT|_1$. After mining the $PT|_1$, the child paths of node with label 1 will be assigned to other rows of header table after row 1 and we get the 2-projected prefix tree, $PT|_2$. In Figure 6.11(b), the part enclosed by dotted line is $PT|_2$. By following the pointer from the row 2 in header table, we can get the $PT|_2$.

In algorithm REPT, a node X will be associated with X-projected transposed table (represented with prefix tree $PT|_X$) and a $rowlist$ that is the list of rows in X-projected transposed table. Continue with the above example, 1-projected prefix tree $PT|_1$ can be located by following the pointers of row 1 in head table. By assembling the items of nodes in the pointer link, the closed pattern corresponding to $PT|_1$ can be obtained (b, d, e, f). REPT will scan the $PT|_1$ to build head table for row set 12 and obtain the $PT|_{12}$ as shown in Figure 6.12. After the $PT|_1$ has been mined recursively, the paths in $PT|_1$ will be assigned to the projected prefix tree of rows after row 1(i.e. rows 2, 3, 4 and 5) and the pointer status of head table is shown in Figure 6.11(b).

6.5.2 Algorithm design

Figure 6.13 gives the pseudo code of algorithm REPT. There are two input parameters of the algorithm, the dataset D and the minimum support ξ. In algorithm REPT(), the dataset D is transposed. Then the procedure REPTdepthfirst() is called to perform the depth-first traversal of row enumeration tree.

```
**Algorithm REPT(D, $\xi$)**

1. Scan database $D$ to find the set of frequent items $F$
2. Remove the infrequent items in each row $r_i$ of $D$ and let $R$ be the set of rows in $D$
3. Transpose table $D$ and build prefix tree $PT|_\emptyset$
4. Call REPTdepthfirst($PT|_\emptyset$,$R$, $FCP$)
5. Let $CF$ be the set of closed items in $F$, $FCP$ = $FCP \cup CF$ and return $FCP$

**Procedure:** REPTdepthfirst($PT'|_X$,$R'$, $FCP$).
6. Scan $PT'|_X$ and compute the frequency $support(r_i)$ for each row, $r_i \in R'$
7. **Support Pruning:** Let $U \subset R'$ be the set of rows in $R'$ such that $support(u) > 1$ and $u \in R'$.
   If $|U| + |X| \leq \xi$, then return; else $R' = U$
8. **Row Merging Pruning:** Let $Y$ be the set of rows such that $support(y) = |\mathcal{I}(X)|$ and $y \in R'$.
   Let $R' = R' - Y$ and $X = X \cup Y$ and remove all rows of $Y$ from $PT'|_X$;
9. **Backward Pruning:** If there is a row $r'$ that appears in each prefix path w.r.t $\mathcal{I}(X)$ and does
   not belong to $X$, then return
10. **if** $|X| \geq \xi$, add $\mathcal{I}(X)$ into $FCP$
11. **for** each $r_i \in R'$
12.     $R' = R' - \{r_i\}$ and generate $PT'|_{Xr_i}$
13.     REPTdepthfirst($PT'|_{Xr_i}, R', FCP$)
```

Figure 6.13: The REPT algorithm

REPT applies three pruning methods. The first is *support pruning* implemented at line 7. This strategy is essentially the same as the case 1 of support pruning in algorithm RERII and will not be explained again here. The strategy implicitly employs the *single item pruning* by ignoring such a row r_i, $support(r_i) = 1$, since enumerating such rows generates only single item. The second pruning strategy is *row merging pruning* implemented at line 8. At an enumeration node X, enumerating rows in $PT|_X$ whose supports are the same as $|\mathcal{I}(X)|$ will only contribute to the support but not to new patterns. Therefore, such rows can be pruned off. Such a strategy is also used in CARPENTER. The last pruning method is *backward pruning* implemented at line 9. The pruning method essentially does the same pruning as the Lemma 6.4.2 although the implementation is completely different. If there exists a row r' that appears in each prefix path w.r.t the set of nodes contributing to $\mathcal{I}(X)$ and does not belong to row set X, the pattern $\mathcal{I}(X)$ and all patterns below X must have already been discovered below some enumeration node containing r'. REPT implements this strategy by means of row information while RERII does that using pattern information.

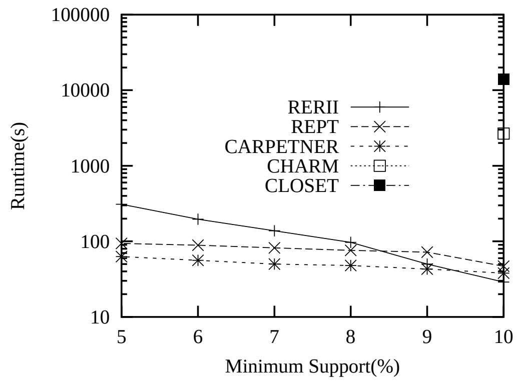
6.6 Performance studies

This section will study and compare the performance of three row enumeration methods with closed pattern discovery algorithms CLOSET [73], CLOSET+ [92][7] and CHARM [101][8]. Experiments in [73, 101] have shown that depth-first mining algorithms like CHARM and CLOSET are substantially better than level-wise mining algorithms like Close[70] and Pascal [11]. All the experiments were performed on a PC with a Pentium 2.4 Ghz CPU, 1GB RAM running Linux and a PC with Pentium IV 2.6, 1 G RAM running Windows XP. Algorithms were coded in Standard C.

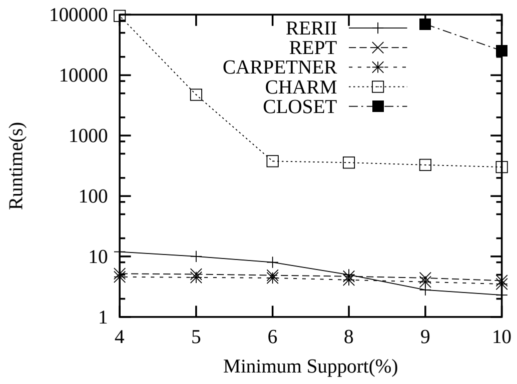
Datasets: We choose 2 real-life datasets for performance studies. The 2 datasets are

[7] I am grateful to Dr. Jiawei Han and Dr. Jianyong Wang for making the executable code of CLOSET+ running on Windows avialble

[8] I am grateful to Dr. Mohammed Zaki for making us the Linux version source code of CHARM available



(a) Breast Cancer



(b) ALL-AML leukemia

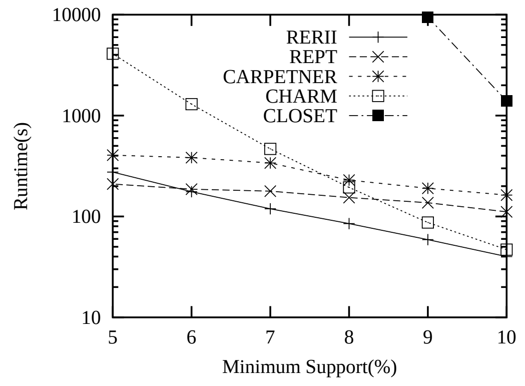
Figure 6.14: Equal-depth Partitioned Datasets

clinical data on Breast Cancer (BC) [9] and ALL-AML leukemia (ALL) [10]. In such datasets, the rows represent clinical samples while the columns represent the activity level of genes/protein presence in the sample. There are 97 samples in BC and 72 samples in ALL. Each sample in BC is described by the activity level of 24481 genes while each sample in ALL is described by the activity level of 7129 genes.

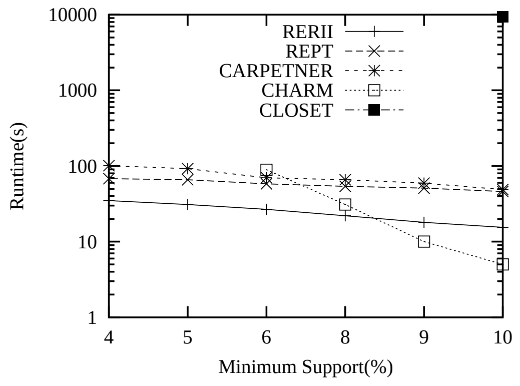
The datasets are discretized by doing a equal-depth partition for each column with 10 buckets and a equal-width partition for each column with 50 buckets (Note that many buckets in equal-width partition are empty). This studies show that having fewer buckets will result in extremely high running time (up to a few days) for CHARM and CLOSET. A setting of 10 and 50 buckets reduces this time sufficiently for performing the experiments with reasonable efficiency. Equal-depth partition will result in evenly distributed data while equal-width partition will result in data with some dense items.

Figure 6.14 shows the experimental results on the two equal-depth partitioned datasets and Figure 6.15 shows the experimental results on the two equal-width partitioned datasets. Note that the y-axes of these graphs are in logarithmic scale. The highest *minimum support* is set at 10% for all datasets. Then we try to reduce the *minimum support* by 1% each time to plot a new point if the reduced *minimum support* makes a difference.

[9] http://www.rii.com/publications/default.htm
[10] http://www-genome.wi.mit.edu/cgi-bin/cancer

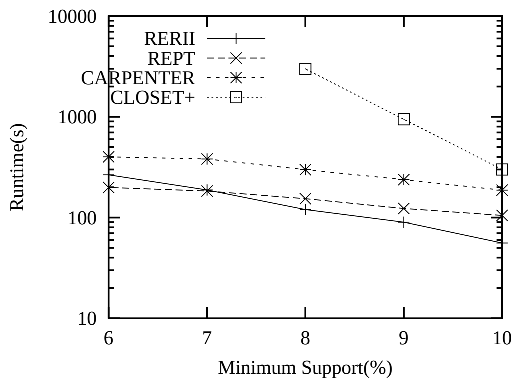


(a) Breast Cancer

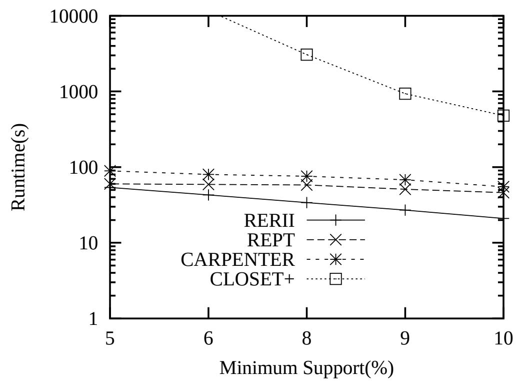


(b) ALL-AML leukemia

Figure 6.15: Equal-width Partitioned Datasets



(a) Breast Cancer



(b) ALL-AML leukemia

Figure 6.16: Comparison with CLOSET+

Sometimes, 1% decrease in *minimum support* does not make a difference because the total number of samples is very small. For example, on dataset ALL, 8% actually represents the same level as the *minimum support* at 7%. This also explains why the values on the x-axis of some graphs are not continuous. At some points in Figure 6.14 and 6.15, the runtime of CHARM is not shown because CHARM cannot finish by reporting error after using up all available memory. We do not give the runtime of CLOSET on all points because it is too slow and showing them will make the differences in runtime of other algorithms unclear in these graphs.

We first observe that there is a large variation in the running time for both CHARM and CLOSET even though the variation in minimum support ξ is small. This is because the average length of each row after removing the infrequent features can increase (decrease) substantially due to a small decrease (increase) in ξ value. This increases (decreases) the search space of both CHARM and CLOSET substantially, resulting in large differences in running time

The three algorithms using row enumeration strategy usually perform much better than the two column enumeration algorithms, CHARM and CLOSET on both equal-depth partitioned and equal-width partitioned datasets. Although CHARM sometimes outperforms RERII and REPT at high support level in Figure 6.15, the absolute difference between them is negligible compared to the difference in running time at low minimum support.

As shown in Figure 6.14 on equal-depth partitioned dataset, CARPENTER is usually a bit better than RERII and REPT. However, as shown in Figure 6.15 on equal-width partitioned dataset, RERII is usually 2-4 times faster and REPT is usually 1-2 times faster than CARPENTER. The possible reason for the difference is that the data structures used in RERII and REPT are more effective for equal-width partitioned datasets in which some items are dense. This is similar to the column enumeration algorithms adopting FP-tree structure (similar to REPT) usually work well on dense datasets while column enumeration algorithms adopting in-memory pointer (similar to CARPENTER) usually work well on sparse datasets.

We report the comparison results with CLOSET+ on the two equal-depth partitioned datasets BC and ALL in Figure 6.16. The CLOSET+ reports errors when running on equal-depth partitioned datasets. CLOSET+ cannot finish by reporting error after using up all available memory at $\xi = 7\%$ on BC. Figure 6.16 shows that both RERII and REPT are usually 1 order of magnitude faster than CLOSET+.

As can be seen, in all the experiments we conducted, the three row enumeration algorithms outperform CHARM and CLOSET in most cases. This result demonstrates that row enumeration strategy is extremely efficient in finding frequent closed patterns on datasets with a small number of rows and a large number of columns.

6.7 Conclusion

In this chapter, we described three algorithms, CARPENTER , RERII and REPT, for finding frequent closed patterns in microarray datasets. The three algorithms explore various implementation of row enumeration which overcomes the extremely high dimensionality of microarray datasets. Experiments showed that these algorithms based on row enumeration outperforms existing closed pattern discovery algorithms like CHARM, CLOSET and CLOSET+ by a large order of magnitude when they are running on microarray datasets. Moreover, the experiments also showed that the algorithm CARPENTER usually perform well on sparse data while RERII and REPT usually perform well on data containing dense items.

Chapter 7

Mining Interesting Rule Groups from Microarray Datasets

On the basis of CARPENTER algorithm, this chapter extends the row enumeration techniques to mine interesting rule group with a given consequent. This chapter also handles the problem of large number of rules by means of the concept of interesting rule group.

7.1 Introduction

One special association rule takes the form of $LHS \rightarrow C$, where LHS is a set of items and C is a class label. The term "support of A" is used to refer to the number of rows containing A in the database and denote this number as $sup(A)$. The probability of the rule being true is referred to as "the confidence of the rule" and is computed as $sup(LHS \cup C)/sup(LHS)$. The number of rows in the database that match the rule is defined as "the support of the rule". User-specified constraints such as minimum support (a statement of generality) and minimum confidence (a statement of predictive ability) are often imposed on mining such association rules.

Recent studies have shown that such kinds of association rules themselves are very useful in the analysis of microarray data. Due to their relative simplicity, they are more easily interpreted by biologists, providing great help in the search for gene predictors (especially those still unknown to biologists) of the sample categories.

Moreover, such association rules can be applied in the scenario of classification: it is shown in [33, 53] that classifiers built from association rules are rather accurate in identifying cancerous cell since such association rules can relate gene expressions to their cellular environments or categories. It can discover the relationship between different genes, so that the functions of an individual gene can be inferred based on its relationship with others [29].

Microarray datasets pose great challenges for existing rule mining algorithms in both runtime and the number of discovered rules. The high dimension and huge number of rules also render existing rule mining algorithms impractical for microarray data. Existing association rule mining algorithms all perform column enumeration, and thus usually take long time to run on microarray datasets.

Due to the high dimensions and the combinatorial explosion of frequent itemsets [14], enormous association rules may be generated even for given consequent, many of which are redundant. To address the problem of huge number of redundant rules, we propose the concept of interesting rule group (IRG) that can greatly reduce the number of rules by grouping similar rules and removing uninteresting groups.

The concept of IRG is illustrated with a simple example. Given a one row dataset with five features and one class label: $\{a, b, c, d, e, Cancer\}$, we could have 31 rules of the form $LHS \rightarrow Cancer$ since any combination of a, b, c, d, e could be the LHS for the rule. These 31 rules are all covered by the same row and have the same confidence (100%). Such a large set of rules contains a lot of redundancy and is difficult to interpret. Instead of generating all 31 rules, we propose to discover these rules as a rule group whose consequent is Cancer, which can be identified by a unique **upper bound** and a set of **lower bounds**. The upper bound of a rule group is the rule with the most specific LHS among the rules. In this case, the upper bound rule is $abcde \rightarrow Cancer$. The lower bounds of the rule group are the rules with the most general LHS in the rule group. For the example, the rule group has 5 lower bounds ($a \rightarrow Cancer$, $b \rightarrow Cancer$,

$c \rightarrow Cancer$, $d \rightarrow Cancer$, and $e \rightarrow Cancer$). Given the upper bound and the lower bounds of the rule group, other rules within the group can be easily derived.

The number of rules can be further reduced by finding interesting rule groups only. Consider two rules $abcd \rightarrow Cancer$ with confidence 90% and $ab \rightarrow Cancer$ with confidence 95%, it is obvious that ab is a better indicator of *Cancer* since $ab \rightarrow Cancer$ has a higher confidence and all rows covering $abcd \rightarrow Cancer$ must cover $ab \rightarrow Cancer$. With $ab \rightarrow Cancer$, rule $abcd \rightarrow Cancer$ is not interesting [1].

The interesting rule discussed in [13] is quite similar to the interesting rule group. However, [13] randomly discovers one rule for each rule group while FARMER discovers the upper bound and lower bounds for each rule group. Moreover, [13] search interesting rules in column enumeration space and usually cannot work on microarray datasets as shown in experimental study in Section 7.4.

This chapter describes a novel algorithm called FARMER [2], that is specially designed to mine interesting rule groups whose consequent is a specified class label. The target datasets are microarray data which have large number of columns and relatively small number of rows. FARMER discovers upper bounds of interesting rule groups with given consequent from datasets by performing depth-first row enumeration instead of the usual column enumeration approach taken by existing rule mining algorithms. On the basis of row enumeration, efficient search pruning strategies are designed based on user-specified thresholds (minimum support, minimum confidence and minimum chi square value) and the fixed consequent information. This chapter also describes an efficient algorithm for computing the lower bounds. Experimental results show that FARMERsubstantially outperforms other rule mining algorithms described in [13], [101](CHARM) and [92](CLOSET+) on Microarray datasets and the pruning strategies are effective.

[1] Rules like $abcd \rightarrow Cancer$ are also not useful for method like CBA [55] of building classifier using association rules.

[2] FARMER stands for <u>F</u>inding Interesting <u>A</u>ssociation <u>R</u>ule by Enu<u>m</u>eration of <u>R</u>ows.

To further illustrate the usefulness of the discovered interesting rule groups in biology, a simple classifier is built based on these interesting rule groups, which outperforms several well-known existing classification methods, e.g., CBA [55] and SVM[47] on several real life datasets.

The rest of this chapter is organized as follows: More formal definition of interesting rule group is given in Section 7.2. The FARMER algorithm is described in Section 7.3. To illustrate the performance of FARMER and the usefulness of discovered interesting rule groups in classification, the experimental results are described in Section 7.4. Finally, Section 7.5 concludes this chapter.

7.2 Preliminary

This section will introduce some basic notations and concepts that are useful for further discussion and give a formal definition for an **interesting rule group**.

7.2.1 Basics

Dataset: the dataset (or table) D consists of a set of rows, R=$\{r_1, ..., r_n\}$. Let I=$\{i_1, i_2, ..., i_m\}$ be the complete set of items of D, and $C = \{C_1, C_2, ..., C_k\}$ be the complete set of class labels of D, then each row $r_i \in R$ consists of one or more items from I and a class label from C.

As an example, Figure 7.1(a) shows a dataset whose items are represented with alphabets from ‘a’ to ‘t’. There are altogether 5 rows, r_1,...,r_5, in the table, the first three of which are labeled C while the other two are labeled $\neg C$. To simplify the notation, the *row id set* is used to represent a set of rows and the *item id set* is used to represent a set of items. For instance, “234” denotes the row set $\{r_2, r_3, r_4\}$, and “acf” denotes the itemset $\{a, c, f\}$.

This chapter will also use the **item support set** $\mathcal{R}(I')$ and **row support set** $\mathcal{I}(R')$ which are defined in Chapter 4.

Association Rule: an **association rule** γ, or just **rule** for short, from dataset D takes the form of $A \rightarrow C$, where $A \subseteq I$ is the antecedent and C is the consequent (here, it is a class label). The **support** of γ is defined as the $|\mathcal{R}(A \cup C)|/|D|$, and its **confidence** is $|\mathcal{R}(A \cup C)|/|\mathcal{R}(A)|$. The antecedent of γ is denoted as $\gamma.A$, the consequent as $\gamma.C$, the support as $\gamma.sup$, the confidence as $\gamma.conf$ and the chi square value as $\gamma.chi$.

As discussed in the introduction, in real biological applications, people are often interested in rules with a specified consequent C that meet specified thresholds, like minimum support and minimum confidence.

7.2.2 Interesting rule groups (IRGs)

The interesting rule group is a concept which helps to reduce the number of rules discovered by identifying rules that come from the same set of rows and clustering them conceptually into interesting groups.

Definition 7.2.1. Rule Group

Let D be the dataset with itemset I and C be the specified class label. $G = \{A_i \rightarrow C | A_i \subseteq I\}$ is a rule group with antecedent support set R and consequent C, iff (1) $\forall A_i \rightarrow C \in G$, $\mathcal{R}(A_i) = R$, and (2) $\forall \mathcal{R}(A_i) = R$, $A_i \rightarrow C \in G$.$\Box$

Definition 7.2.2. the Upper Bound of a Rule Group

Let $G = \{A_i \rightarrow C\}$ be a rule group of dataset D. Rule $\gamma_u \in G$ (γ_u: $A_u \rightarrow C$) is an **upper bound** of G iff there exists no $\gamma' \in G$ (γ':$A' \rightarrow C$) such that $A' \supset A_u$.$\Box$

Definition 7.2.3. the Lower Bound of a Rule Group

Let $G = \{A_i \rightarrow C\}$ be a rule group of dataset D. Rule $\gamma_l \in G$ (γ_l: $A_l \rightarrow C$) is a **lower bound** of G iff there exists no $\gamma' \in G$ (γ': $A' \rightarrow C$) such that $A' \subset A_l$.$\Box$

Lemma 7.2.1. Given a rule group G with the consequent C and the antecedent support set R, it has a unique upper bound γ (γ: $A \rightarrow C$).

Proof: Assume there exists another upper bound $\gamma'(A' \rightarrow C) \in G$ such that $A' \neq A$

and $A' \not\subseteq A$. Let $A'' = A \cup A'$. Because of $\mathcal{R}(A') = \mathcal{R}(A) = R$, we get $\mathcal{R}(A'') = R$, and then $A'' \to C \in G$ and $A'' \supset A$. Therefore, $\gamma(A \to C)$ can not be an upper bound of G. So the upper bound of a rule group must be unique.$\Box$

Based on lemma 7.2.1, a rule group G can be represented with its unique upper bound γ_u.

Example 7.2.1. Rule Group

A running example is shown in Figure 7.2. $\mathcal{R}(\{e\}) = \mathcal{R}(\{h\}) = \mathcal{R}(\{ae\}) = \mathcal{R}(\{ah\}) = \mathcal{R}(\{eh\}) = \mathcal{R}(\{aeh\}) = \{r_2, r_3, r_4\}$. They make up a rule group $\{e \to C, h \to C, ..., aeh \to C\}$ of consequent C, with the upper bound $aeh \to C$ and the lower bounds $e \to C$ and $h \to C$.$\Box$

It is obvious that all rules in the same rule group have the same support, confidence and chi square value since they are essentially derived from the same subset of rows. Based on the upper bound and all the lower bounds of a rule group, its remaining members can be identified according to the lemma below.

Lemma 7.2.2. Suppose rule group G with the consequent C and antecedent support set R has an upper bound $A_u \to C$ and a lower bound $A_l \to C$. Rule $\gamma(A \to C)$, where $A \subset A_u$ and $A \supset A_l$, must be a member of G.

Proof: Since $A \subset A_u$, $\mathcal{R}(A) \supseteq \mathcal{R}(A_u)$. Likewise, $\mathcal{R}(A) \subseteq \mathcal{R}(A_l)$. Since $\mathcal{R}(A_l) = \mathcal{R}(A_u) = R$, $\mathcal{R}(A) = R$. So $\gamma(A \to C)$ belongs to G. $\Box$

Definition 7.2.4. Interesting Rule Group (IRG)

A rule group G with upper bound γ_u is an interesting rule group iff for any rule group with upper bound $\gamma'_u \subset \gamma_u$, $\gamma'_u.conf < \gamma_u.conf$. For brevity, the abbreviation IRG will be used to refer to interesting rule group. $\Box$

The algorithm FARMER is designed for finding IRGs that satisfy user-specified constraints including *minimum support*, *minimum confidence* and *minimum chi square value*

i	r_i	class
1	a,b,c,l,o,s	C
2	a,d,e,h,p,l,r	C
3	a,c,e,h,o,q,t	C
4	a,e,f,h,p,r	¬C
5	b,d,f,g,l,q,s,t	¬C

(a) Example Table

i_j	$R(i_j)$	
	C	$\neg C$
a	1,2,3	4
b	1	5
c	1,3	
d	2	5
e	2,3	4
f		4,5
g		5
h	2,3	4
l	1,2	5
o	1,3	
p	2	4
q	3	5
r	2	4
s	1	5
t	3	5

(b) Transposed Table, TT

Figure 7.1: Running example

i_j	$R(i_j)$	
	C	$\neg C$
a	1,2,3	4
e	2,3	4
h	2,3	4

Figure 7.2: $TT|_{\{2,3\}}$

[3]. These constraint parameters are deliberately left out of the definition of IRG to avoid restricting the definition of interestingness to these measures. FARMER finds the upper bounds of all IRGs first, and then gathers their lower bounds. This makes it possible for users to recognize all the rule group members as and when they want to.

7.3 The FARMER algorithm

This section first gives a running example (Figure 7.1) to illustrate the algorithm. Table TT (Figure 7.1(b)) is a transposed version of the example table (Figure 7.1(a)). In TT, the items become the row ids while the row ids become the items. A row number r_m in the original table will appear in tuple i_n of TT if and only if the item i_n occurs in the row r_m of the original table. For instance, since item d occurs in row r_2 and r_5 of the original table, row ids "2" and "5" occur in tuple d of TT. To avoid confusion, this chapter hereafter refers to the rows in the transposed table as **tuples** while referring to

[3]Other constraints such as lift, conviction, entropy gain, gini and correlation coefficient can be handled similarly

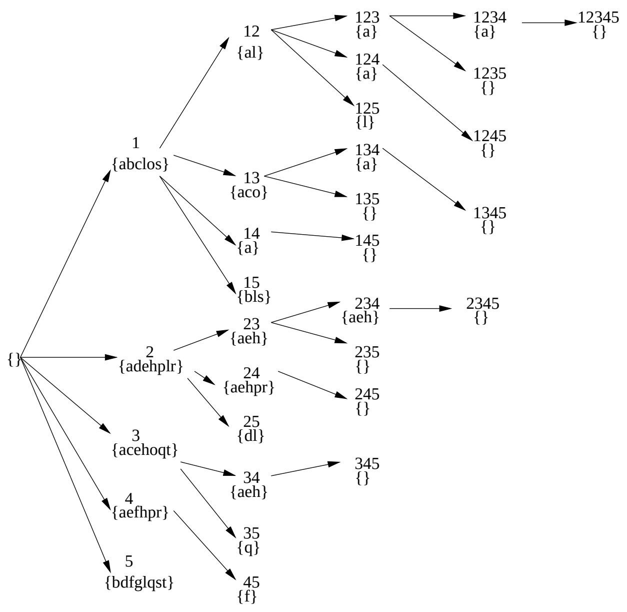


Figure 7.3: The row enumeration tree.

those in the original table as **rows** [4].

A conceptual explanation of algorithm FARMER to discover upper bounds of interesting rule groups is given in Section 7.3.1, the pruning strategies are given in Section 7.3.2, and the implementation details are given in Section 7.3.3. Section 7.3.4 describe subroutine MineLB of FARMER to discover the lower bounds of interesting rule groups.

7.3.1 Enumeration

FARMER performs search by enumeration of row sets to find interesting rule groups with consequent C. Figure 7.3 illustrates the row enumeration tree on the table in Figure 7.1. Each node X of the enumeration tree corresponds to a combination of rows R' and is labeled with $\mathcal{I}(R')$ that is the antecedent of the upper bound of a rule group identified at this node. For example, node "12" corresponds to the row combination $\{r_1, r_2\}$ and "al" indicates that $\mathcal{I}(\{r_1, r_2\}) = \{a, l\}$. An upper bound $al \rightarrow C$ can be discovered at node "12". This is correct because of the following lemma.

Lemma 7.3.1. Let X be a subset of rows from the original table, then $\mathcal{I}(X) \rightarrow C$ must

[4]The tuples in the transposed table actually represent the items in the original table

be the upper bound of the rule group G whose antecedent support set is $\mathcal{R}(\mathcal{I}(X))$ and consequent is C.

Proof: First, according to definition 7.2.1, $\mathcal{I}(X) \rightarrow C$ belongs to rule group G with antecedent support set $\mathcal{R}(\mathcal{I}(X))$ and consequent C. Second, assume that $\mathcal{I}(X) \rightarrow C$ is not the upper bound of G, then there must exist an item i such that $i \notin \mathcal{I}(X)$, and $\mathcal{I}(X) \cup \{i\} \rightarrow C$ belongs to G. So we get $\mathcal{R}(\mathcal{I}(X)) = \mathcal{R}(\mathcal{I}(X) \cup \{i\})$. Since rows in X contain all items of $\mathcal{I}(X)$, we get $X \subseteq \mathcal{R}(\mathcal{I}(X))$, and then $X \subseteq \mathcal{R}(\mathcal{I}(X) \cup \{i\})$. This means that i is also found in every row of X, which contradicts the definition that $\mathcal{I}(X)$ is the largest set of items that are found in every row of X. So $\mathcal{I}(X) \rightarrow C$ is the upper bound of the rule group with antecedent support set $\mathcal{R}(\mathcal{I}(X))$.□

FARMER performs a depth first search on the enumeration tree by moving along the edges of the tree. By imposing an order $\mathcal{ORD}$, in which the rows with consequent C are ordered BEFORE the rows without consequent C, on the set of row numbers, we are able to perform a systematic search by enumerating the combinations of rows based on the order $\mathcal{ORD}$. Note that the order also serves for confidence pruning purpose (explained in Section 7.3.2).

Next, this section proves that the complete rule groups can be discovered by a complete enumeration of row combinations. Following is the lemma.

Lemma 7.3.2. By enumerating all possible row combinations in the enumeration tree in Figure 7.3, the complete set of upper bounds and the corresponding complete set of rule groups in the dataset can be obtained.

Proof: With Lemma 7.2.1, we know that each rule group can be represented by a unique upper bound. Based on the definition of rule group (Definition 7.2.1), all possible antecedent support sets of rule groups can be obtained by enumerating all possible row combinations. Each antecedent support set X corresponds to a rule group with upper bound "$\mathcal{I}(X) \rightarrow C$". So we get the proof.□

It is obvious that a complete traversal of the row enumeration tree is not efficient. Various pruning techniques will be introduced to prune off unnecessary searches in the next section. This section will next introduce the framework of the algorithm FARMER for discovering the upper bounds of rule groups. Two concepts are given as follows.

Definition 7.3.1. Conditional Transposed Table ($TT|_X$**)**
Given the transposed table TT (used at the root of the enumeration tree), a X-conditional transposed table ($TT|_X$) at node X (X is the row combination at this node) is a subset of tuples from TT such that for each tuple t of TT that $t \supseteq X$, there exists a tuple $t' = t$ in $TT|_X$.□

Example 7.3.1. Let TT be the transposed table in Figure 7.1(b) and let $X = \{2, 3\}$. The X-conditional transposed table, $TT|_X$ is shown in Figure 7.2.□

Definition 7.3.2. Enumeration Candidate List ($TT|_X.E$**)**
Let $TT|_X$ be the X-conditional transposed table and $r_{min} \in X$ be the row id with the lowest $\mathcal{ORD}$ order in row combination X. Let $E_P = \{r | r \succ_{ORD} r_{min} \wedge r \in \mathcal{R}(C)\}$ (all rows of consequent C ordered after r_{min}), and $E_N = \{r | r \succ_{ORD} r_{min} \ \wedge \ r \in \mathcal{R}(\neg C)\}$ (all rows with class C ordered after r_{min}). The enumeration candidate list for $TT|_X$, denoted as $TT|_X.E$, is defined to be $E_P \cup E_N$.□

Note that the definition of Conditional Transposed Table is different from that in Chapter 6. The rows that are not in enumeration candidate list are not removed from transposed table since the information of rows that are not in enumeration candidate list may be useful in the pruning techniques(Section 7.3.2).

In the remaining chapter, the following notations are used to represent the attributes of $TT|_X$:

$TT|_X.E$: the enumeration candidates;

$TT|_X.E_P$: the enumeration candidates with label C;

$TT|_X.E_N$: the enumeration candidates without label C;

$TT|_X.Y$: the enumeration candidates that occur in each tuple of $TT|_X$.

The formal algorithm is shown in Figure 7.4. FARMER involves recursive computations of conditional transposed tables by performing a depth-first traversal of the row enumeration tree. (Section 7.3.3 will show that memory pointers are used to generate conditional transposed tables instead of constructing conditional transposed tables physically). Each computed conditional table represents a node in the enumeration tree of Figure 6.4. For example, the $\{2,3\}$-conditional table is computed at node "23". After initialization, FARMER calls the subroutine $MineIRGs$ to recursively generate X-conditional tables.

The subroutine $MineIRGs$ takes in four parameters at node X: $TT'|_X$, sup_p, sup_n and IRG. $TT'|_X$ is the X-conditional transposed table at node X with enumeration candidates $TT'|_X.E_P$ and $TT'|_X.E_N$. sup_p is the number of identified rows that contain $\mathcal{I}(X) \cup C$ while sup_n is the number of identified rows that contain $\mathcal{I}(X) \cup \neg C$ before scanning $TT'|_X$. IRG stores the upper bounds of interesting rule groups discovered so far.

Steps 1, 2, 4 and 5 in the subroutine $MineIRGs$ perform the pruning. They are extremely important for the efficiency of FARMER algorithm and will be explained in the next subsection. Step 3 scans the table $TT'|_X$. Step 6 moves on into the next level enumerations in the search tree. Step 7 checks whether $\mathcal{I}(X) \rightarrow C$ is the upper bound of an IRG that satisfies the user-specified constraints before inserting it into IRG. Note that step 7 must be performed after step 6 (the reason will be clear later). This section first proves the correctness of the two steps by two lemmas as follows:

Lemma 7.3.3. $TT|_X|_{r_i} = TT|_{X+r_i}$, $r_i \in TT|_X.E$. $\Box$

Lemma 7.3.3 is useful for explaining Step 6. It simply states that a $X+r_i$ conditional transposed table can be computed from a X conditional transposed table $TT|_X$ in the next level search after node X.

```
Algorithm FARMER
Input: table $D$, specified consequent $C$, $\xi$, $minconf$, and $minchi$.
Output: interesting rule groups with consequent $C$ satisfying minimum measure thresholds.
Method:
  1. Initialization: Let $TT$ be the transposed table of ORD ordered $D$; $IRG = \emptyset$.
  2. Mine Interesting Rule Groups: MineIRGs($TT|_{\emptyset}$, 0, 0, $IRG$).
  3. Mine Lower Bounds of Interesting Rule Groups: Optional.

Subroutine: MineIRGs($TT'|_X$, $sup_p$, $sup_n$, $IRG$).
Parameters:
  • $TT'|_X$: a $X$-conditional transposed table;
  • $sup_p$ and $sup_n$: support parameters;
  • $IRG$: the set of discovered interesting rule groups;
Method:
  1. Apply Pruning 2: If $\mathcal{I}(X) \rightarrow C$ is already identified, then return.
  2. Apply Pruning 3: If prunable with the loose upper bounds of support or confidence, then return.
  3. Scan $TT'|_X$ and count the frequency of occurrences for each enumeration candidate, $r_i \in TT'|_X.E$,
     Let $U_p \subseteq TT'|_X.E_P$ be the set of rows from $TT'|_X.E_P$ which occur in at least one tuple of $TT'|_X$;
     Let $U_n \subseteq TT'|_X.E_N$ be the set of rows from $TT'|_X.E_N$ which occur in at least one tuple $TT'|_X$;
     Let $Y_p \subset TT'|_X.E_P$ be the set of rows from $TT'|_X.E_P$ found in every tuple of $TT'|_X$;
     Let $Y_n \subset TT'|_X.E_N$ be the set of rows from $TT'|_X.E_N$ found in every tuple of $TT'|_X$;
     $sup_p = sup_p + |Y_P|$ ($|\mathcal{R}(\mathcal{I}(X) \cup C)|$);
     $sup_n = sup_n + |Y_N|$ ($|\mathcal{R}(\mathcal{I}(X) \cup \neg C)|$);
  4. Apply Pruning 3: If prunable with one of the three tight upper bounds, then return.
  5. Apply Pruning 1: Update enumeration candidate list, $TT'|_X.E_P = U_P - Y_P,\ TT'|_X.E_N = U_N - Y_N$.
  6. for each $r_i \in TT'|_X.E$ do
         if $r_i \in \mathcal{R}(C)$ then
             $TT'|_X|_{r_i}.E_P = \{r_j | r_j \in TT'|_X.E_P \wedge r_j \succ_{ORD} r_i\}$; $TT'|_X|_{r_i}.E_N = TT'|_X.E_N$;
             $a = sup_p + 1$; $b = sup_n$;
         else
             $TT'|_X|_{r_i}.E_P = \emptyset$; $TT'|_X|_{r_i}.E_N = \{r_j | r_j \in TT'|_X.E_N \wedge r_j \succ_{ORD} r_i\}$;
             $a = sup_p$; $b = sup_n + 1$;
         $MineIRGs(TT'|_X|_{r_i}, a, b, IRG)$;
  7. Let $conf$ = $(sup_p)/(sup_p + sup_n)$;
     If ($sup_p \geq \xi$) $\wedge$ ($conf \geq minconf$)$\wedge$ ($chi$($sup_p$, $sup_p$+ $sup_n$) $\geq minchi$) then
         if $\forall\gamma$, ($\gamma \in IRG$) $\wedge$ ($\gamma.A \subset \mathcal{I}(X)$) $\Rightarrow$ ($conf > \gamma.conf$)
         then add upper bound rule $\mathcal{I}(X) \rightarrow C$ into $IRG$.
```

Figure 7.4: The FARMER algorithm

Lemma 7.3.1 ensures that at Step 7 only upper bounds of rule groups are possibly inserted into IRG. To determine whether an upper bound γ discovered at node X represents an interesting rule group satisfying user-specified constraints, FARMER needs to compare $\gamma.conf$ with all $\gamma'.conf$, where $\gamma'.A \subset \gamma.A$ and γ' satisfies user specified constraints. FARMER ensures that all such γ' have already been discovered and kept in IRG at Step 7 by lemma 7.3.4 below.

Lemma 7.3.4. Let $\gamma : \mathcal{I}(X) \rightarrow C$ be the upper bound rule discovered at node X. The rule group with upper bound $\gamma' : A' \rightarrow C$ such that $A' \subset \mathcal{I}(X)$ can always be discovered at the descendent nodes of node X or in an earlier enumeration.

proof: Since $A' \subset \mathcal{I}(X)$, and γ' and γ are the upper bounds of two different rule groups, we see $\mathcal{R}(A') \supset \mathcal{R}(\mathcal{I}(X)) \supseteq X$. Let $RS = \{r | r \in \mathcal{R}(A') \ \wedge \ r \notin X\}$ and $r_{min} \in X$ be the row with the lowest ORD rank in row set X. If $\exists r' \in RS$ such that $r' \prec r_{min}$, then node $\mathcal{R}(A')$ is traversed before node X; otherwise node $\mathcal{R}(A')$ is traversed at a descendent node of node X. $\Box$

Step 7 is implemented after Step 6 to ensure all descendant nodes down X are explored before determining whether the upper bound γ at X is an IRG. Together with Lemma 7.3.2, it is known that the complete and correct set of interesting rule groups will be in IRG.

Note that Step 6 implicitly does the pruning since it is possible that the enumeration candidate list is empty, i.e. $TT'|_X.E = \emptyset$. It can be observed from the enumeration tree that there exist some combinations of rows, X, such that $\mathcal{I}(X) = \emptyset$ (an example is node "134"). This implies that there is no item existing in all the rows in X. When this happens, $TT'|_X.E$ is empty and no further enumeration will be performed.

7.3.2 Pruning strategy

This section presents the pruning techniques that are used in FARMER, which are essential for the efficiency. The emphasis here is to show that the pruning steps do not prune off any interesting rule groups while preventing unnecessary traversals of the enumeration tree. Combining this with the earlier explanations on how all interesting rule groups are enumerated in FARMER without the pruning steps, the correctness of the algorithm FARMER will be obvious.

Pruning Strategy 1

Pruning strategy 1 is implemented at Step 5 of MineIRGs by pruning $TT|_X.Y$, the set of enumeration candidate rows that occur in all tuples of the $TT|_X$. $TT|_X.Y$ is partitioned to two subsets: Y_p with consequent C and Y_n without. The intuitive reason for the pruning is that we obtain the same set of upper bound rules along the branch X WITHOUT such rows. The correctness of such a pruning strategy is because of the following lemma.

Lemma 7.3.5. Let $TT'|_X$ be a X-conditional transposed table. Given any subset R', $R' \subset TT'|_X.E$, we have $\mathcal{I}(X \cup R') = \mathcal{I}(X \cup TT'|_X.Y \cup R')$.

Proof: By definition, $\mathcal{I}(X \cup R')$ contains a set of items which occur in every row of $(X \cup R')$. Suppose candidate $y \in TT'|_X.Y$ (y occurs in every tuple of $TT'|_X$), then either $y \in X \cup R'$ (if $y \in R'$) or y occurs in every tuple of the $TT'|_{X \cup R'}$ (if $y \notin R'$). In either case, $\mathcal{I}(X \cup R') = \mathcal{I}(X \cup R' \cup \{y\})$. Thus, $\mathcal{I}(X \cup R') = \mathcal{I}(X \cup TT'|_X.Y \cup R')$.$\Box$

With Lemma 7.3.5, FARMER can safely delete the rows in $TT'|_X.Y$ from the enumeration candidate list $TT'|_X.E$.

Example 7.3.2. Consider $TT|_{\{2,3\}}$, the conditional transposed table in Figure 7.2. Since enumeration candidate row 4 occurs in every tuples of $TT|_{\{2,3\}}$, it can be concluded that $\mathcal{I}(\{2,3\}) = \mathcal{I}(\{2,3,4\}) = \{a, e, h\}$. Thus, there is no need to traverse node “234” and to create $TT|_{\{2,3,4\}}$. Row 4 can be safely deleted from $TT|_{\{2,3\}}.E$.$\Box$

Since $\mathcal{I}(\{2,3,4\}) = \mathcal{I}(\{2,3\})$, the upper bound rule is identified at node “23” and node “234” is redundant. We say that node “234” is compressed to node “23”.

We argue here that Lemma 7.3.4 still holds after applying pruning strategy 1. Without applying pruning strategy 1, for each node X, $A' \rightarrow C$, where $A' \subset \mathcal{I}(X)$, is identified at a node X', which is traversed before node X or is a descendent node of node X. With pruning strategy 1, X' might be compressed to a node X'' ($X'' \subset X'\ and\, \mathcal{I}(X'') = \mathcal{I}(X') = A'$), and it can be seen that node X'' is either traversed before the subtree rooted at node X, or inside this subtree.

Pruning Strategy 2

This pruning strategy is implemented at Step 1 of MineIRGs. It will stop searching the subtree rooted at node X if the upper bound rule $\mathcal{I}(X) \rightarrow C$ was already discovered previously in the enumeration tree because this implies that any upper bounds to be discovered at the descendants of node X have been discovered too.

Lemma 7.3.6. Suppose pruning strategy 1 is utilized in the enumeration tree. Let $TT'|_X$ be the conditional transposed table of the current node X. All upper bounds to be discovered in the subtree rooted at node X must have already been discovered if there exists such a row r' that satisfies the following conditions: (1)$r' \notin X$; (2)$r' \notin TT'|_X.E$; (3)for any ancestor node X_i of node X, $r' \notin TT|_{X_i}.Y$(pruned by strategy 1); and (4)r' occurs in each tuple of $TT'|_X$.

Proof: Let $X = \{r_1, r_2, ..., r_m\}$, where $r_1 \prec_{ORD} r_2 \prec_{ORD} ... \prec_{ORD} r_m$. Suppose that there is a node X''($X'' = X \cup \{r'\}$), we can have the following properties: (1) $\mathcal{I}(X) = \mathcal{I}(X'')$; (2) $r' \prec_{ORD} r_m$, since $r' \notin TT'|_X.E$ and $r' \notin X$; (3) $TT'|_X.E = TT'|_{X''}.E$.

X'' is either enumerated or compressed to a node X_C ($X_C \subset X''$), where $\mathcal{I}(X_C) = \mathcal{I}(X'')$ and $TT'|_{X''}.E \subseteq TT'|_{X_C}.E$. It can be proved that either node X'' or node X_C is traversed before node X by considering the following two cases: (1) If $r' \prec_{ORD} r_1$, node X'' or node X_C falls in the subtree rooted at node $\{r'\}$, which is traversed before node X. (2) If row ids in X'' follow the order $r_1 \prec_{ORD} r_2 \prec_{ORD} ... \prec_{ORD} r_t \prec_{ORD} r' \prec_{ORD} r_{t+1} \prec_{ORD} ... \prec_{ORD} r_m$, node X'' or node X_C falls in the subtree rooted at node

$X' = \{r_1, r_2, ..., r_t, r'\}$, which is also traversed before node X. Because of $TT'|_X.E = TT'|_{X''}.E$ and $TT'|_{X''}.E \subseteq TT'|_{X_C}.E$, it can be concluded that all upper bounds to be discovered in the subtree rooted at node X must have already been discovered earlier in the subtree rooted at node X'' or node X_C. $\square$

In the implementation of pruning strategy 2, the existence of such a r' can be efficiently detected by a process call **back counting** without scanning the whole of $TT'|_X$. Details are explained in Section 7.3.3.

Example 7.3.3. Consider node "23" in Figure 6.4 where the upper bound rule $\{a, e, h\} \rightarrow C$ is identified for the first time. When it comes to node "34", it is noticed that row "2" occurs in every tuple of $TT|_{\{3,4\}}$, '2' $\notin TT|_{\{3,4\}}.E$, and '2' $\notin TT|_{\{3\}}.Y$. So it is concluded that all upper bounds to be discovered down node "34" have already been discovered before ($\mathcal{I}(\{3,4\}) = \mathcal{I}(\{2,3\}) = \{a, e, h\}$. $\mathcal{I}(\{3,4,5\}) = \emptyset$). The search down node "34" can be pruned.$\square$

Pruning Strategy 3

Pruning strategy 3 performs pruning by utilizing the user-specified thresholds, ξ, $minconf$ and $minchi$. It estimates the upper bounds of the measures for the subtree rooted at the current node X. If the estimated upper bound at X is below the user-specified threshold, FARMER stops searching down node X.

Pruning strategy 3 consists of 3 parts: pruning using confidence upper bound, pruning using support upper bound and pruning using chi square upper bound. This strategy is executed separately at Step 2 and Step 4 (Figure 7.4). Step 2 performs pruning using the two loose upper bounds of support and confidence that can be calculated BEFORE scanning $TT'|_X$. Step 4 calculates the three tight upper bounds of support, confidence and chi square value AFTER scanning $TT'|_X$.

For clarity, the notations to be used in the lemmas in this subsection are listed as follows.

X: the current enumeration node;

γ: the upper bound $\mathcal{I}(X) \rightarrow C$ at node X;

X': the immediate parent node of X;

γ': the upper bound rule $\mathcal{I}(X') \rightarrow C$ at node X';

r_m: a row id that $TT|_X = TT|_{X'}|_{r_m}$;

Pruning Using Support Upper Bound

Two support upper bounds for the rule groups can be identified at the subtree rooted at node X: the tight support upper bound U_{s1} (after scanning $TT'|_X$) and the loose support upper bound U_{s2} (before scanning $TT'|_X$). If the estimated upper bound is less than minimum support ξ, the subtree can be pruned.

If r_m has consequent C:

$U_{s1} = \gamma'.sup + 1 + MAX(|TT'|_X.E_P \cap t|), t \in TT'|_X$;

$U_{s2} = \gamma'.sup + 1 + |TT'|_X.E_P|$;

If r_m has no consequent C:

$U_{s1} = U_{s2} = \gamma'.sup$;

Lemma 7.3.7. U_{s1} and U_{s2} are the support upper bounds for the upper bound rules discovered in subtree rooted at node X.

proof: Because of the ORD order (definition 7.3.2), if r_m has no consequent C, the enumeration candidates of nodes down node X will not have consequent C, either. The support can not increase down node X, so the support of upper bounds discovered at the subtree rooted at node X is less than $\gamma'.sup$. If r_m has consequent C, for node X and its descendent nodes, the maximum increase of support from $\gamma'.sup$ must come from the number of enumeration candidates with consequent C ($|TT'|_X.E_P|$) at node X plus 1 (1 for r_m)(U_{s2}), or more strictly, from the maximum number of enumeration candidates with consequent C within a tuple of $TT'|_X$ ($MAX(|TT'|_X.E_P \cap t|), t \in TT'|_X$) plus 1

(U_{s1}).$\square$

Note that it is needed to scan $TT'|_X$ to get U_{s1} while U_{s2} can be obtained directly from the parameters sup_p and X passed by its parent node without scanning $TT'|_X$. If r_m has consequent C, we get $\gamma'.sup+1 = sup_p$; otherwise we get $\gamma'.sup = sup_p$, where sup_p is the input parameter of current MineIRGs subroutine (Figure 7.4).

Pruning Using Confidence Upper Bound

Similarly, FARMER estimates two confidence upper bounds for the subtree rooted at node X, the tight confidence upper bound U_{c1} and the loose confidence upper bound U_{c2}. If the estimated upper bound is less than minimum confidence $minconf$, the subtree can be pruned.

Given U_{s1} and U_{s2}, the two confidence upper bounds of subtree rooted at node X, U_{c1}(tight) and U_{c2}(loose), are:

$U_{c1} = U_{s1}/(U_{s1} + |\mathcal{R}(\gamma.A \cup \neg C)|)$;

$U_{c2} = U_{s2}/(U_{s2} + |\mathcal{R}(\gamma'.A \cup \neg C)|)$ (r_m has consequent C);

$U_{c2} = U_{s2}/(U_{s2} + |\mathcal{R}(\gamma'.A \cup \neg C)| + 1)$ (r_m has no consequent C).

Lemma 7.3.8. U_{c1} and U_{c2} are the confidence upper bounds for the upper bound rules discovered in the subtree rooted at node X.

Proof: For a rule γ'' discovered in subtree rooted at node X, its confidence is computed as $|\mathcal{R}(\gamma''.A \cup C)|/(|\mathcal{R}(\gamma''.A \cup C)| + |\mathcal{R}(\gamma''.A \cup \neg C)|)$. This expression can be simplified as $x/(x+y)$, where $x = |\mathcal{R}(\gamma''.A \cup C)|$ and $y = |\mathcal{R}(\gamma''.A \cup \neg C)|$. This value is maximized with the largest x (U_{s1} and U_{s2}) and smallest y. Suppose rule γ is discovered at node X. For any upper bound rule γ'' discovered down node X, $\gamma''.A \subset \gamma.A$ because of pruning strategy 1, so we can see $|\mathcal{R}(\gamma''.A \cup \neg C)| \geq |\mathcal{R}(\gamma.A \cup \neg C)|$. Thus y is minimized at value $|\mathcal{R}(\gamma.A \cup \neg C)|$ or loosely at $|\mathcal{R}(\gamma'.A \cup \neg C)| + 1$(if r_m has no consequent C) and $|\mathcal{R}(\gamma'.A \cup \neg C)|$ (if r_m has consequent C). $\square$

Note that if r_m has consequent C, we get $|\mathcal{R}(\gamma'.A \cup \neg C)| = sup_n$; otherwise we get

$|\mathcal{R}(\gamma'.A \cup \neg C)| + 1 = sup_n$, where sup_n is the input parameter of current subroutine MineIRGs (figure 7.4).

Example 7.3.4. Suppose minimum confidence $minconf = 95\%$. At node "134", the discovered upper bound rule is "$a \rightarrow C$" with confidence $0.75 < 0.95$. Since row 4 has no consequent C, any descendent enumeration will only reduce the confidence. Thus next level searching can be stopped.

Pruning Using Chi Square Upper Bound

The chi square value of an association rule is the normalized deviation of the observed values from the expectation.

Let γ be a rule in the form of $A \rightarrow C$ of dataset D, n be the number of rows of D, and m be the number of instance with consequent C of D. The four observed values for chi square value computation are listed in the following table. For example, $O_{A\neg C}$ represents the number of rows that contain A but do not contain C. Let $x = O_A$ and $y = O_{AC}$. Since m and n are constants, the chi square value is determined by x and y only and we get chi square function $chi(x, y)$.

	C	$\neg C$	Total
A	$O_{AC} = y$	$O_{A\neg C}$	$O_A = x$
$\neg A$	$O_{\neg AC}$	$O_{\neg A\neg C}$	$O_{\neg A} = n - x$
Total	$O_C = m$	$O_{\neg C} = n - m$	n

The following lemma gives an estimation of upper bound of chi square value for rules down the node X.

Lemma 7.3.9. Suppose rule γ is discovered at enumeration node X. The chi square upper bound for the upper bound rules discovered at the subtree rooted as node X is: $max\{chi(x(\gamma) - y(\gamma) + m, m), chi(y(\gamma) + n - m, y(\gamma))$.

Proof: Suppose rule γ' ($A' \rightarrow C$) is identified in the subtree rooted at node X, $x' = O_{A'}$ and $y' = O_{A'C}$. Since $O(A) = |\mathcal{R}(A)|$ and $A' \subset A$. The followings are satisfied.

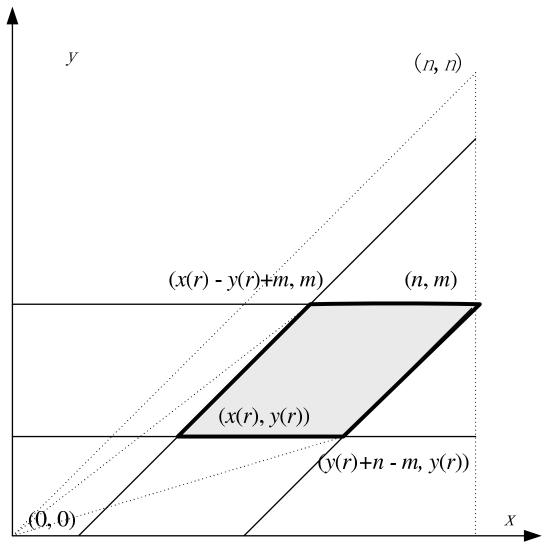


Figure 7.5: The possible Chi-square variables

1) $x \le x' \le n$ ($|\mathcal{R}(A)| \le |\mathcal{R}(A')|$)

2) $y \le y' \le m$ ($|\mathcal{R}(A \cup C)| \le |\mathcal{R}(A' \cup C)|$)

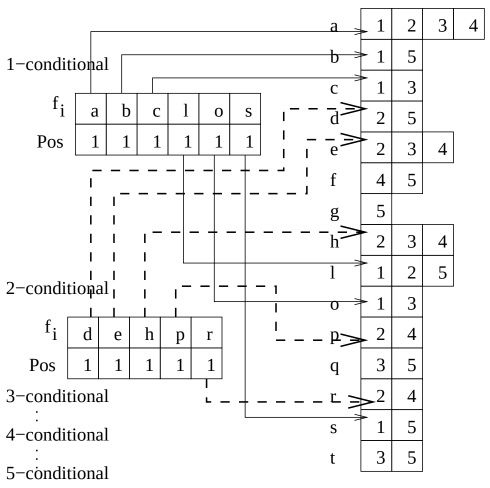
3) $y' \le x'$ ($|\mathcal{R}(A' \cup C)| \le |\mathcal{R}(A')|$)

4) $n - m \ge x' - y' \ge x - y$ ($|\mathcal{R}(A' \cup \neg C)| \ge |\mathcal{R}(A \cup \neg C)|$)

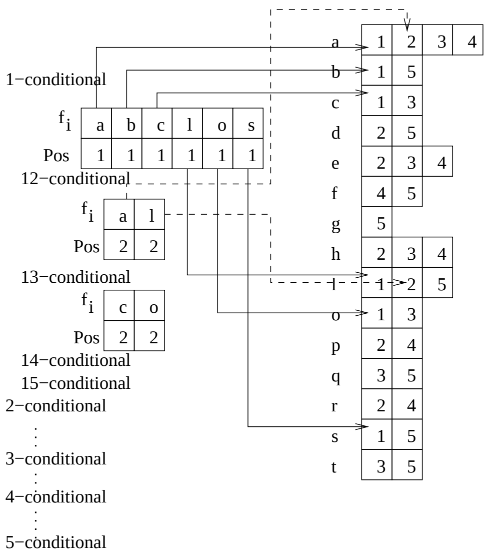
The value pair $(x'(\gamma'), y'(\gamma'))$ falls in the gray parallelogram $(x(\gamma), y(\gamma))$, $(x(\gamma)-y(\gamma)+m, m)$, (n, m), $(y(\gamma)+n-m, y(\gamma)$ (Figure 7.5). Since $x(\gamma) > 0$ and $y(\gamma) > 0$, all value pairs (x', y') in the gray parallelogram can be matched to the quadrangle (0, 0), $max\{chi(x(\gamma)-y(\gamma)+m, m)$, $chi(y(\gamma)+n-m, y(\gamma))$, and (m, n). Since the chi square function $chi(x, y)$ is a convex function [62], which is maximized at one of its vertexes, and $chi(0, 0) = chi(n, m) = 0$ (please refer to [62]), only the remaining two vertexes are needed to be considered. □

7.3.3 Implementation

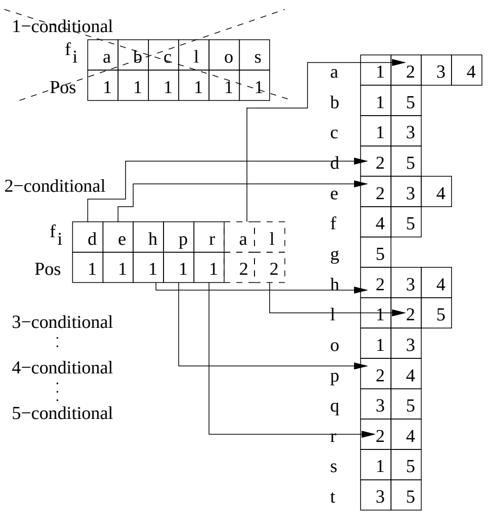
In the implementation of CARPENTER , we use memory pointers [15] to point at the relevant tuples in the in-memory transposed table to simulate the conditional transposed table.



(a) Node {1}



(b) Node {1,2}



(c) Node {2}

Figure 7.6: Conditional Pointer Lists

Following is the running example. Suppose the current node is node "1" (Figure 7.6(a)), and $\xi = 1$. The in-memory transposed table is shown on the right hand side of the figure. Memory pointers are organized into **conditional pointer lists**.

In Figure 7.6(a), the "1"-conditional pointer list (at the top left corner of the figure) has 6 entries in the form of $< f_i, Pos >$ which indicates the tuple (f_i) that contains r_1 and the position of r_1 within the tuple (Pos). For example, the entry $< a, 1 >$ indicates that row r_1 is contained in the tuple 'a' at position 1. We can extend the "1"-conditional transposed table $TT'|_{\{1\}}$ by following the Pos. During one full scan of the transposed table, CARPENTER also generates the conditional pointer lists for other rows (i.e. r_2, r_3, r_4 and r_5). However, the generated "2"-conditional pointer list is slightly different in that it contains an entry for each tuple that contains r_2 BUT NOT r_1. For example, although the tuple 'a' contains r_2, it does not appear in the "2"-conditional pointer list. It will be inserted subsequently as we will see later.

A further scan through the "1"-conditional pointer list will allow us to generate the "12", "13", "14" and "15" conditional pointer lists. Figure 7.6(b) shows the state of memory pointers when we are processing node $\{1, 2\}$.

Finally, we show the state of conditional pointer lists after node $\{1\}$ and all its descendants have been processed (Figure 7.6(c)). Since all enumerations involving row r_1 have been either processed or pruned off, the entries in the "1"-conditional pointer list are moved into the remaining conditional pointer lists. The entries in the "2"-conditional pointer list will be moved to the other conditional pointer lists after node $\{2\}$ and its descendants are processed, and so on.

Throughout all the enumerations described above, we need to implement our three pruning strategies. The implementation of strategies 1 and 3 is straightforward. For pruning strategy 2, we do a back scan through the conditional list to see whether there exists some row that satisfies the condition of Lemma 7.3.6. For example at node "2" in Figure 7.6(c), we scan from the position of each pointer to the head of each tuple, instead of scanning the transposed table from the position of each pointer to the end of each tuple. In this example, there is no row that satisfies the pruning condition of Lemma 7.3.6. Such an implementation is proven to be efficient for our purpose as shown in our experiments.

Moreover, two optimization strategies are utilized in the implementation.

Optimization 1. Order ORD. The rows of consequent C will be ordered before those rows that do not contain C. For the rows of the same class, the less frequent the row, the higher ORD order it gets. This will reduce the running time greatly because it reduces the entry moving processes.

Optimization 2. Enumeration Candidate Constraint. At step 3 of subroutine $MineIRGs$, if a row only occurs in one tuple of $TT'|_X$, the descendent enumeration with the row only generate a length-1 rule, while all length 1 rules can be easily discovered by scanning the transposed table TT' once. It is required that the enumeration candidate must occur in at least two tuples of $TT'|_X$.

7.3.4 Finding lower bounds

After discovering the upper bounds of interesting rule groups, users might be interested to have a closer look at the members of the rule group. The algorithm, MineLB, is designed for the purpose by finding the lower bounds of a rule group [5]. Since a rule group has a unique upper bound and the consequent of a rule group is fixed, the problem can be regarded as generating the lower bounds for the antecedent of the upper bound rule. This antecedent could be regarded as a closed set (definition 7.3.3) and the problem can be solved as long as the lower bounds of a closed set (definition 7.3.4) are generated.

Definition 7.3.3. Closed Set
Let D be the dataset with itemset I and row set R. A ($A \subseteq I$) is a closed set of dataset D, iff there is no $A' \supset A$ such that $\mathcal{R}(A) = \mathcal{R}(A')$.$\Box$

Definition 7.3.4. Lower Bound of a Closed Set
Suppose A is a closed set of dataset D. A_l, $A_l \subseteq A$, is a lower bound of closed set A, iff $\mathcal{R}(A_l) = \mathcal{R}(A)$ and there is no $A' \subset A_l$ such that $\mathcal{R}(A') = \mathcal{R}(A)$.$\Box$

MineLB is an incremental algorithm that is initialized with one closed set A, which is the antecedent of an upper bound $A \rightarrow C$ of a rule group. It then updates the lower bounds of A incrementally whenever a new closed set A' is added, where $A' \subset A$ and A'

[5]Note that the lower bounds can not be directly derived from upper bounds.

is the antecedent of the newly added upper bound $A' \rightarrow C$. In this way, MineLB keeps track of the latest lower bounds of A. MineLB is based on the following lemma.

Lemma 7.3.10. Let A be the closed set whose lower bounds will be updated recursively and $\digamma$ be the set of closed sets that are already added. Let $A.\Gamma$ be the current collection of lower bounds for A. When a new closed set $A' \subset A$ is added, $A.\Gamma$ is divided into two groups, $A.\Gamma 1$ and $A.\Gamma 2$, where $A.\Gamma 1 = \{l_i | l_i \in A.\Gamma \wedge l_i \subseteq A'\}$, $A.\Gamma 2 = A.\Gamma - A.\Gamma 1$. Then the newly generated lower bounds of A must be in the form of $l_1 \cup \{i\}$, where $l_1 \in A.\Gamma 1$, $i \in A - A'$.

Proof: Suppose l is a newly generated lower bound of A.

(1) we prove $l \supset l_1$. Since $\mathcal{R}(l) = \mathcal{R}(A)$ (Definition 6.2.2) before A' is added, there must exist a $l_i \in A.\Gamma$ such that $l_i \subset l \subset A$. If $l_i \in A.\Gamma 2$, l can not be a new lower bound, since $l_i \in A.\Gamma 2$ is still a lower bound of A after A' is added. So $l \supset l_1$, $l_1 \in A.\Gamma 1$.

(2) Obviously, the newly generated lower bound must contain at least one item from the set $(A - A')$.

(3) $l' = l_1 \cup \{i\}$ is a bound for A after adding A', where $l_1 \in A.\Gamma 1$, $i \in A$ and $i \notin A'$. Before A' is added, $l' = l_1 \cup \{i\}$ is a bound, so for any $X \in \digamma$, $l' \nsubseteq X$. After A' is added, $l' \nsubseteq A'$ because $i \notin A'$. So, $l' = l_1 \cup \{i\}$ is a new bound for A after adding A'.

Based on (1), (2) and (3), we come to the conclusion that the newly generated lower bound for A after inserting A' takes the form of $l_1 \cup \{i\}$, where $l_1 \in A.\Gamma 1$ and $i \in (A - A')$. $\square$

Itemset $l_1 \cup \{i\}$ described in Lemma 7.3.10 is a candidate lower bound of A after A' is added. If $l_1 \cup \{i\}$ does not cover any $l_2 \in A.\Gamma 2$ and other candidates, $l_1 \cup \{i\}$ is a new lower bound of A after A' is added. MineLB adopts bit vector for the above computation. Thus $A.\Gamma$ can be updated efficiently. The detailed algorithm is illustrated in Figure 7.7.

The Lemma 7.3.11 can ensure that the closed sets (those that cover all the longest closed set $A' \subset A$) obtained at Step 2 are sufficient for the correctness of MineLB.

Lemma 7.3.11. If closed set $A1 \subset A$ is already added and the collection of A's lower bounds $A.\Gamma$ is already updated, $A.\Gamma$ will not change after adding closed set $A2$, $A2 \subset A1$.

proof: After $A1 \subset A$ is added, $A.\Gamma$ is updated so that no $l_i \in A.\Gamma$ can satisfy $l_i \subseteq A1$. So no $l_i \in A.\Gamma$ can satisfy $l_i \subseteq A2$, $A2 \subset A1$. Since A2 will not cover any $l_i \in A.\Gamma$, $A.\Gamma$ will not change, according to Lemma 7.3.10. $\Box$

Example 7.3.5. Find Lower Bound

Given an upper bound rule with antecedent $A = abcde$ and two rows, $r_1 : abcf$ and $r_2 : cdeg$, the lower bounds $A.\Gamma$ of A can be determined as follows:

1)Initialize the set of lower bounds $A.\Gamma = \{a, b, c, d, e\}$;

2)add "abc" (= $\mathcal{I}(r_1) \cap A$): We get $A.\Gamma 1 = \{a, b, c\}$ and $A.\Gamma 2 = \{d, e\}$. Since all the candidate lower bounds, "ad", "ae", "bd", "be", "cd", "ce" cover a lower bound from $A.\Gamma 2$, no new lower bounds are generated. So $A.\Gamma = \{d, e\}$;

3)add "cde" (= $\mathcal{I}(r_2) \cap A$): We get $A.\Gamma 1 = \{d, e\}$ and $A.\Gamma 2 = \emptyset$. The candidate lower bounds are "ad", "bd", "ae" and "be". Because none of them is covered by another candidate and $A.\Gamma 2 = \emptyset$, $A.\Gamma = \{ad, bd, ae, be\}$.$\Box$

7.4 Performance studies

This section will study both the efficiency of FARMER and the usefulness of the discovered IRGs. All the experiments were performed on a PC with a Pentium IV 2.4 Ghz CPU, 1GB RAM and a 80GB hard disk. Algorithms were coded in Standard C. Following are the 5 datasets used in the experiments.

Datasets: The 5 datasets are the clinical data on lung cancer (LC)[6], breast cancer (BC) [7], prostate cancer (PC) [8], ALL-AML leukemia (ALL) [9], and colon tumor (CT) [10]. In such

[6]http://www.chestsurg.org

[7]http://www.rii.com/publications/default.htm

[8]http://www-genome.wi.mit.edu/mpr/prostate

[9]http://www-genome.wi.mit.edu/cgi-bin/cancer

[10]http://microarray.princetion.edu/oncology/affydata/index.html

```
Subroutine: MineLB(Table:$D$, upper bound rule: $\gamma$).

1. $A = \gamma.A$; $A.\Gamma = \{i | i \in A\}$; $\Sigma = \emptyset$;

2. for each row $r_{id}$ of $D$ that $r_{id} \notin \mathcal{R}(A)$: if $(\mathcal{I}(r_{id}) \cap A) \subset A$ then add $(\mathcal{I}(r_{id}) \cap A)$ to $\Sigma$;

3. for each closed set $A' \in \Sigma$: {
     $A.\Gamma1 = A.\Gamma2 = \emptyset$;
     for each lower bound $l_i \in A.\Gamma$: if $l_i \subseteq A'$ then add $l_i$ to $A.\Gamma1$; else add $l_i$ to $A.\Gamma2$;
     $CandiSet = \emptyset$;
     for each $l_i \in A.\Gamma1$ and each $i \in A$ && $i \notin A'$: add candidate $l_i \cup \{i\}$ to $CandiSet$;
     $A.\Gamma = A.\Gamma2$;
     for each candidate $c_i \in CandiSet$
         if $c_i$ does not cover any $l_i \in A.\Gamma2$ and $c_i$ does not cover any other $c_j \in CandiSet$
         then add $c_i$ to $A.\Gamma$
   }

4. Output $A.\Gamma$.
```

Figure 7.7: MineLB

datasets, the rows represent clinical samples while the columns represent the activity levels of genes/proteins in the samples. There are two categories of samples in these datasets.

dataset	# row	# col	class 1	class 0	#row of class 1
BC	97	24481	relapse	nonrelapse	46
LC	181	12533	MPM	ADCA	31
CT	62	2000	negative	positive	40
PC	136	12600	tumor	normal	52
ALL	72	7129	ALL	AML	47

Table 7.1: Microarray datasets

Table 7.1 shows the characteristics of these 5 datasets: the number of rows (# row), the number of columns (# col), the two class labels (class 1 and class 0), and the number of rows for class 1 (# class 1). All experiments presented here use the class 1 as the consequent; it has been found that using the other consequent consistently yields qualitatively similar results.

Two methods are used to discretize the datasets. One is the entropy-minimized partition (for CBA and IRG classifier)[11] and the other is the equal-depth partition with 10

[11] the code is available at http://www.sgi.com/tech/mlc/

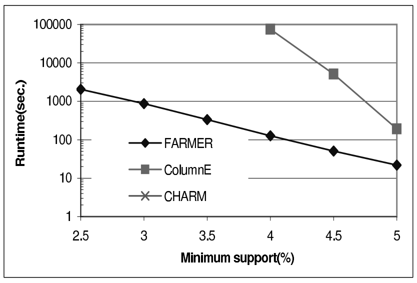
buckets. Ideally, we would like to use only the entropy discretized datasets for all experiments since we want to look at the classification performance of IRGs. Unfortunately, the two rule mining algorithms that we want to compare against are unable to run to completion within reasonable time (they usually run out of memory after several hours or longer) on the entropy discretized datasets, although FARMER is still efficient. As a result, the efficiency results will be reported based on the equal-depth partitioned data while the classifiers are built using the entropy-discretized datasets.

7.4.1 Efficiency of FARMER

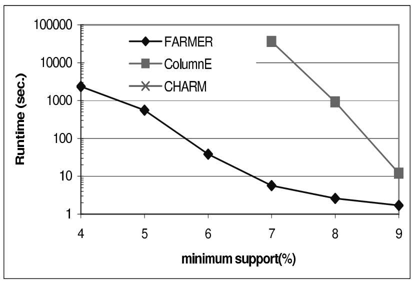
The efficiency of FARMER will first be evaluated. We compare FARMER with the interesting rule mining algorithm in [13] [12]. The algorithm in [13] is the one most related to FARMER in terms of interesting rule definition. But the interesting rule discussed in [13] does not contain the complete information of a rule group as FARMER does by discovering the upper bound and lower bounds for each rule group. However, [13] only randomly discovers one rule for each rule group. To the best knowledge, the algorithm in [13] is also the most efficient algorithm that exists with the purpose of mining interesting rules of the kind that FARMER discovers. We denote this algorithm as *ColumnE* since it also adopts column enumeration like most existing rule mining algorithms.

We also compare FARMER with the closed set discovery algorithms CHARM [101] and CLOSET+ [92], which are shown to be more efficient than other association rule mining algorithms in many cases. We found that CHARM is always orders of magnitude faster than CLOSET+ on the microarray datasets and thus we do not report the CLOSET+ results here. Note that the runtime of FARMER includes the time for computing both the upper bound and lower bounds of each interesting rule group. Compared with CHARM, FARMER does extra work in: 1)computing the lower bounds of Interesting Rule Groups and 2) identifying the IRGs from all rule groups. Unlike FARMER

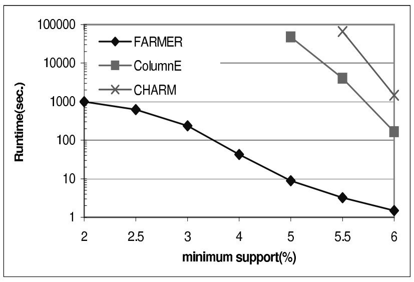
[12]My own implementation is used in the experimental study since the code from the authors is not available.



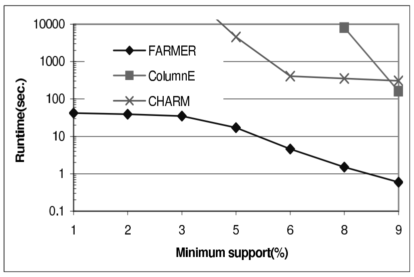
(a) Lung Cancer



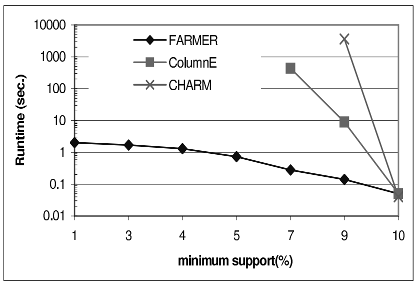
(b) Breast Cancer



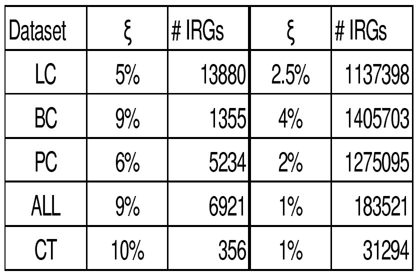
(c) Prostate Cancer



(d) ALL-AML leukemia



(e) Colon Tumor



Dataset	ξ	# IRGs	ξ	# IRGs
LC	5%	13880	2.5%	1137398
BC	9%	1355	4%	1405703
PC	6%	5234	2%	1275095
ALL	9%	6921	1%	183521
CT	10%	356	1%	31294

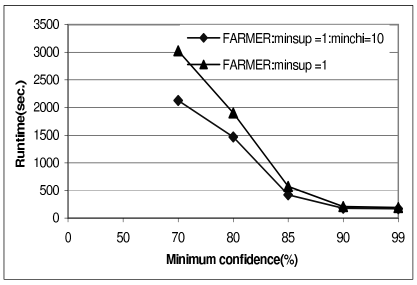
(f) The Number of IRGs(minchi=0)

Figure 7.8: Varying minimum support

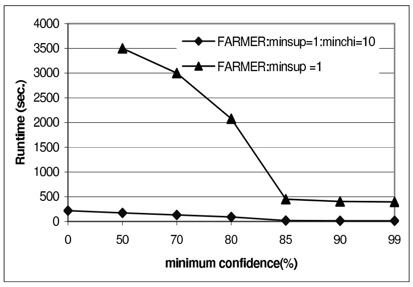
that discovers both upper bound and lower bounds for each IRG, ColumnE only gets one rule for each IRG.

Varying Minimum Support

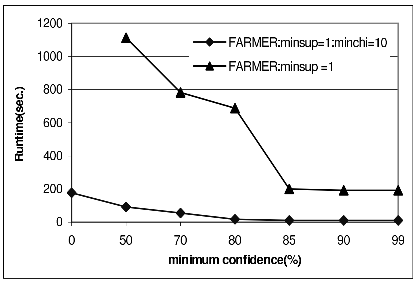
The first set of experiments (Figure 7.8) shows the effect of varying minimum support threshold ξ. The graphs plot runtime for the three algorithms at various settings of minimum support. Note that the y-axes in Figure 7.8 are in logarithmic scale. Both $minconf$ and $minchi$ are set as ZERO, which disables the pruning with confidence



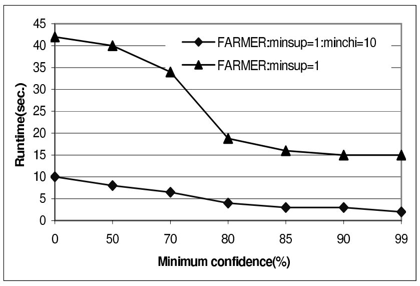
(a) Lung Cancer



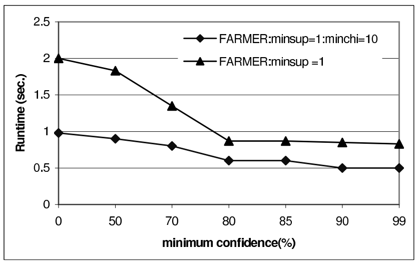
(b) Breast Cancer



(c) Prostate Cancer



(d) ALL-AML leukemia



(e) Colon Tumor

Dataset	minconf	# IRGs	minconf	#IRGs
LC	0.99	37316	0.7	1173020
BC	0.99	214640	0.5	2154705
PC	0.99	184310	0.5	1233525
ALL	0.99	38333	0	183521
CT	0.99	5101	0	31294

(f) The Number of IRGs (minchi=0)

Figure 7.9: Varying $minconf$

upper bound and the pruning with the chi square upper bound of FARMER.

For CHARM, xi represents the least number of rows that the closed sets must match. The runtime of CHARM is not shown in Figures 7.8(a) and 7.8(b) because CHARM runs out of memory even at the highest support in Figure 7.8 on datasets BC and LC.

Figure 7.8 shows that FARMER is usually 2 to 3 orders of magnitude faster than ColumnE and CHARM (if it can be run). Especially at low minimum support, FARMER outperforms ColumnE and CHARM greatly. This is because the candidate search space for ColumnE and CHARM, dependent on the possible number of column combinations after removing the infrequent items, is orders of magnitude greater than the search space of FARMER, dependent on the possible number of row combinations, on microarray datasets.

As shown in Figure 7.8(f), the number of interesting rule groups discovered at a low ξ is much larger than that at a high ξ. Besides the size of row enumeration space, the number of IRGs also affects the efficiency of FARMER. First, FARMER discovers IRGs by comparison (see algorithm section, step 7). The more IRGs discovered, the more time is consumed for the comparison. The time can take 20% of the runtime of FARMER at low ξ. Second, the time complexity of computing lower bounds in FARMER is $O(n)$, where n is the number of IRGs. It is observed that at high ξ, the time used to compute lower bounds takes 5% to 10% of the runtime of FARMER while the time can take up to 20% at low ξ. ColumnE also does the comparison to get interesting rules while all the runtime of CHARM is used to discover closed sets.

Readers may wonder why different minimum support thresholds are chose for different datasets. The principle here is to make the runtime of FARMER about 10 seconds. It can seen that ColumnE and CHARM might catch up with FARMER if the ξ is further increased. However, the absolute time difference must be less than 10 seconds and is not interesting for comparison. This is negligible compared to the difference in running time at low ξ. To make figures clear, they are not shown here.

Varying Minimum Confidence

The next set of experiments (Figure 7.9) shows the effect of varying $minconf$ when minimum support is fixed. In this subsection, the minimum support is represented with $minsup$ which is absolute number of rows. The $minchi$ pruning is still disabled by setting it as ZERO. For all the parameter settings in Figure 7.9, CHARM can not finish because it always runs out of memory within several hours while ColumnE always needs more than 1 day to finish. This is because we adopt a relative low $minsup$ to study the effectiveness of confidence pruning in the experiment. To show the effect of various $minconf$ clearly, we do not give the runtime of ColumnE since it is too slow to compare.

Let's first ignore the lines marked with "minchi =10" here. We set $minsup = 1$, which means that minimum support pruning is almost disabled.

Figure 7.9 shows that the runtime of FARMER decreases when increasing $minconf$ on all the 5 datasets (Figure 7.9(f) lists the number of IRGs). This shows that it is effective to exploit the confidence constraint for pruning. There is only a slight decrease in runtime of FARMER when the $minconf$ increases from 85% to 99%. The reason behind this is that there are few upper bound rules whose confidences fall between 85% and 99%. It is observed that nearly all IRGs discovered at confidence 85% on these 5 datasets have a 100% confidence. As a result, FARMER does no additional pruning when $minconf$ increases from 85% to 99%.

The result that many discovered IRGs have a 100% confidence is interesting and promising. It means that the IRGs are decisive and have good predicability.

Varying Minimum Chi Square Value

The last set of experiments was preformed to study the effectiveness of the chi square pruning. Minimum chi square constraint is usually treated as a supplementary constraint of minimum support and minimum confidence. We set $minchi = 10$ and draw the runtime vs various $minconf$ in Figure 7.9 due to the space limitation, where $minconf$ is set the same as in Section 7.4.1.

The pruning exploited by constraint $minchi = 10$ is shown to be very effective on datasets BC, PC, CT and ALL. In some cases, the saving can be more than an order of magnitude. The pruning effect is not so obvious on dataset LC. By checking the identified IRGs, we found that discovered IRGs from LC usually have higher chi square value. If a tighter chi square constraint is imposed by increasing $minchi$, the $minchi$ pruning will be more obvious as found in experimental study.

As can be seen, in all the experiments conducted, FARMER outperforms ColumnE and CHARM. Moreover, the pruning based on minimum support, confidence and chi-square are effective. In general, the runtime of FARMER correlates strongly with the number of interesting rule groups that satisfy all of the specified constraints. The experimental results demonstrate that FARMER is extremely efficient in finding IRGs on datasets with small number of rows and large number of columns.

7.4.2 Usefulness of IRGs

One avenue to assess the usefulness of IRGs is to pass them to the biologists for interpretation since the number of IRGs is much smaller than the number of rules and the IRGs themselves are intuitive. The other evaluation method is to build classifier to show the usefulness of discovered IRGs. This thesis will reports the classification results in [27] [13] to provide some evidence that the discovery of IRGs is at least useful for such purpose. The classifier in [27] is called **IRG classifier**. Interested readers can also refer to the extension work [26] of this thesis for another classification method for microarray datasets using rule groups.

The IRG classifier is compared with two well-known classifiers CBA [55] and SVM [47]. The code of these two classifiers is open-source and is available through the internet. The IRG classifier building shares the similar three steps with CBA as follows.

1. Ranking of Rules

   Here, the rules are first sorted by confidence (descending), second by the support of rules (descending), and last by the length of the rules (ascending). For two rules with the same confidence and support, the rule with fewer items will be placed first.

2. Pruning of Rules

   Having sorted the rules, CBA will try to find a cover for each training data instance. Given two rules that match a training instance, the rule with the higher sorted order is deemed to cover the instance. Rules that do not cover any instance will be removed.

[13] Xin Xu played primary role in the study of building classifiers using IRGs

3. Test Data Prediction

   After the ranking and pruning, the test data is predicted by the highest ordered rule that matches the test instance. A default class is set in case no rules are matched by the test instance.

While the general approach is the same, the IRG classifer is different from CBA in that the upper bounds of the IRGs are used INSTEAD of the frequent rules. This reduces a significant amount of noisy rules in the classifier which will boost the performance of the IRG classifier substantially.

As the entropy based discretization algorithm also performs feature selection as part of its process, the number of columns in the datasets is reduced after the discretization. We indicate the new dimensionality of the datasets in Table 7.2.

Despite the reduction in the number of columns, the open-source CBA algorithm (and all competitors we look at in the earlier section) failed to finish running in one week. However, based on the upper bounds and lower bounds generated by FARMER, we can obtain the frequent rules [14] to build the CBA classifier (only the SHORTEST members of IRGs are possible to be selected by CBA). We compared the performance of the IRG classifier with CBA and SVM on the 5 microarray datasets we used earlier. For CBA and the IRG classifier, we use the entropy discretized datasets; for SVM, we use the corresponding dataset with the *same* set of features output by the discretization algorithm but the actual numeric values before discretization are used.

For SVM, we always use the default setting of SVM^{light}[47]. For CBA, we set the minimum support threshold as $0.5*$number of training data of class C for each class C and set the minimum confidence threshold as 0.8 (According to the experiments, if we further lower the minimum confidence threshold, the final CBA classifier is the same). For IRG classifier, we set the same minimum support and minimum confidence thresholds, and set $minchi$ as 0 (To keep the comparisons fair) when running FARMER.

[14]We note that this itself is a contribution of FARMER, since conventional rule mining algorithms fail to generate rules in reasonable time.

Dataset	# features	# training	# test	IRG Classifier	CBA	SVM
BC	619	78	19	**78.95%**	57.89%	36.84%
LC	2173	32	149	89.93%	81.88%	**96.64%**
CT	135	47	15	**93.33%**	73.33%	73.33%
PC	1554	102	34	**88.24%**	82.35%	79.41%
ALL	866	38	34	64.71%	91.18%	**97.06%**
Average Accuracy				**83.03%**	77.33%	76.66%

Table 7.2: Classification results

Table 7.2 illustrates the new characteristic of the 5 microarray datasets, together with the percentages of correctly predicted test data for the IRG classifier, CBA and SVM. We can see that the IRG classifier has the highest average accuracy. Although SVM performs very well on LC and ALL, it fails on BC. No classifier outperforms the others on all datasets, but they can be supplementary to each other. The IRG classifier is more understandable than SVM which employs complicated kernel models and distance model making it difficult to derive understandable explanation of diagnostic decision made by SVM method, Therefore, IRG classifier could be a good reference tool for the biological research.

7.5 Conclusion

In this chapter, we proposed an algorithm called FARMER for finding the interesting rule groups in microarray datasets. FARMER makes use of the consequent and the special characteristic of microarray datasets to enhance its efficiency. It adopts the novel approach of performing row enumeration instead of the conventional column enumeration so as to overcome the extremely high dimensionality of microarray datasets. Experiments show that FARMER outperforms existing algorithms like CHARM and ColumnE by several orders of magnitude on microarray datasets. The discovered IRGs were also shown to be useful for the classification of microarray data.

Chapter 8

Conclusions

This thesis described a framework for mining, recycling and reusing frequent patterns in association rule mining. Within this framework, recycling previous mining results and mining microarray data are addressed.

Practical data mining is often a highly interactive and iterative process. Users change constraints and run the mining algorithm many times before they are satisfied with the final results. The frequent patterns discovered in the early round of mining encapsulate the complete or part of computation required by subsequent mining processes. This thesis has shown that recycling previous mining results from the same user or different users in the mining system is valuable and useful for subsequent mining.

Chapter 4 proposed the concept of tree boundary to summarize and reorganize the previous mining results (including frequent patterns and intermediate results) and a recycling technique to extending the tree boundary to discover the set of frequent itemsets that were not counted in previous mining. Based on the tree boundary and recycling technique, two existing algorithms were adapted to re-mine these itemsets under relaxed constraints. Experimental results demonstrated that the proposed technique was highly effective in improving the efficiency of subsequent mining with relaxed minimum support. The application of such technique to more complicated constraints was also discussed in Chapter 4.

Chapter 5 presented a different technique to recycle frequent patterns discovered in the early round of mining (by the same user or different users) to enhance subsequent mining. Chapter 5 proposed a two phase strategy that first compressed the database based on frequent patterns from an early round of mining and then mined the compressed database. Two compression strategies were designed for this technique and three existing mining algorithms were adapted to work on compressed databases. The experimental results showed that the proposed strategy was effective, and the proposed recycling algorithms outperformed their non-recycling counterparts significantly. The results also showed that a cost-based compression strategy was preferred over a storage-based strategy. In Chapter 5, the recycling technique described in Chapter 4 was compared qualitatively with that described in Chapter 5.

Chapter 6 described three algorithms using row enumeration for finding frequent closed patterns in microarray datasets with long columns and a small number of rows. The three algorithm explored various implementations of row enumeration strategy. Experiments showed that the three algorithms usually outperformed existing closed pattern discovery algorithms like CHARM and CLOSET by a large order of magnitude when they were running on microarray datasets.

Association rules discovered from microarray data are often too large in number to handle and people may often be interested in rules with given right hand side. However, the number of rules with given consequent is also huge in number and many of them are redundant. Chapter 7 addressed these problems by proposing the concept of interesting rule groups and presenting FARMER algorithms to find the interesting rule groups. FARMER also performed the novel approach of performing row enumeration and pruned search space using the minimum support, minimum confidence and minimum chi square constraints. Experiments showed that FARMER outperformed the best known existing algorithms by a large order of magnitude on microarray datasets. The IRG classifier built on interesting rule groups discovered by FARMER not only demonstrated the usefulness of discovered IRGs, but also extended the associative classification to microarray datasets.

8.1 Discussion and future work

Along the direction of recycling data mining results, one immediate future work is to combine the two recycling methods in Chapters 4 and 5. More specifically, the tree boundary in Chapter 4 is used to summarize the previous results and the technique in Chapter 5 is used to extend tree boundary.

One interesting question here is whether the recycling technique can be applied to microarray data when the constraints are changed. Unfortunately, the recycling techniques in Chapters 4 and 5 usually cannot help the frequent pattern mining from microarray data. This is because there are usually a large number of frequent patterns mined from microarray and the patterns to be recycled will be numerous if the infrequent intermediate results are also considered. Processing these patterns to generate tree boundary (required by the technique in Chapter 4) or select a subset of patterns to compress database (required by the technique in Chapter 5) will be a time consuming process. In addition, the average computation of discovering each pattern is not very expensive considering there is only a few rows in microarray data. This means that the potential saving of using such a pattern to compress database in the technique in Chapter 5 may not be large. All these make the techniques in Chapters 4 and 5 usually cannot work well on microarray data. How to recycle patterns for microarray data will be a future work.

The other open problems that can be investigated in future are listed as follows: (1) to investigate the possibility of applying the proposed recycling technique in Chapter 5 to other frequent pattern algorithms, such as Apriori-like algorithms and algorithms mining frequent patterns from vertical layout datasets. In addition, it would be also interesting to study the recycling techniques for the proposed algorithms described in this thesis; (2) to examine the possibilities of compressing databases using frequent patterns from other sources, such as the branches of decision trees and the frequent patterns discovered from a sample of dataset for the recycling technique in Chapter 5; (3) to investigate how frequent patterns can be recycled for decision tree construction since there are many aggregation operations in classification that could be sped up with the proposed technique in Chapter 5; and (4) to investigate the application of recycling in other data mining tasks, such as clustering.

Both frequent closed pattern mining algorithms in Chapter 6 and interesting rule groups mining algorithm in Chapter 7 are designed for biological data with a small number of rows and a large number of columns, especially the emerging microarray data. They usually work well for datasets with less than 300 rows. Although it is true that current microarray datasets usually have small number of rows, these proposed algorithms could be extended to other large datasets, such as the Thrombin Data in KDD cup 2001, that are characteristic of both long columns and large number of rows by using a combination of column and row enumerations. More specifically, column enumerations can be first applied to discover short frequent patterns, and then row enumerations can be applied to extend these short patterns (or rules) to get longer ones. In [67], we made some attempt to combine the two enumeration strategy but the combination is still naive. More work is needed to make the combination more effective and efficient.

This method can also help the three row enumeration algorithms in Chapter 6 to deal with those datasets too large to fit in memory, as it is well known that some column-wise mining algorithms have linear scalability with dataset size. The other method for the three row enumeration algorithms to deal with the memory limitation problem is to utilize the database projection (disk-based) techniques as suggested in [42, 43]. This technique was also used in the recycling algorithms in Chapter 5.

Another problem that deserves further investigation is to optimize the IRG classifier for microarray data classification. The IRG classifier described in this thesis was built on discovered IRGs to illustrate the usefulness of discovered IRGs. The IRG classifier is adapted from CBA [55] method. Therefore, a natural extension is to investigate whether the improved classification techniques over CBA, such as CMAR [54] and [10], could be used to improve IRG classifier. In [26], we tries to discover Top-k covering rule groups for each sample and build a refined classifier which is shown to be more accurate than IRG classifier.

Finally, an interesting problem that can be addressed in future is how to implement the framework given in Chapter 3. Chapter 3 described the components for the framework that was presented as a vision. In order to implement such a framework, more detailed problems need to be addressed, such as how to represent the recycling technique with rules that can be automatically understood by the data mining system. Moreover, although this thesis presented some qualitative heuristics of choosing appropriate algorithms according to dataset property, they are still not enough and it is an interesting topic to study more operative and subtle rules.

Bibliography

[1] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Proc. of the ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, 2000.

[2] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. In *Journal of Parallel and Distributed Computing (Special Issue on High Performance Data Mining)*, 2000.

[3] C. Aggarwal and P. Yu. Online generation of association rules. In *Proc. Int'l Conf. Data Engineering (ICDE)*, 1998.

[4] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 94–105, June 1998.

[5] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 207–216, May 1993.

[6] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pages 487–499, Sept. 1994.

[8] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. Int'l Conf. Data Engineering (ICDE)*, pages 3–14, Mar. 1995.

[9] S. Babu, M. N. Garofalakis, and R. Rastogi. Spartan: A model-based semantic compression system for massive data tables. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data(SIGMOD)*, Santa Barbara, California, USA, May 2001.

[10] E. Baralis and P. Garza. A lazy approach to pruning classification rules. In *Proc. Int'l Conf. on Data Mining (ICDM)*, 2002.

[11] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent closed itemsets with counting inference. In *SIGKDD Explorations, 2(2)*, Dec. 2000.

[12] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 85–93, June 1998.

[13] R. J. Bayardo and R. Agrawal. Mining the most interesting rules. In *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, 1999.

[14] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining on large, dense data sets. In *Proc. Int'l Conf. Data Engineering (ICDE)*, 1999.

[15] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 1999.

[16] C. Borgelt and R. Kruse. Induction of association rules: Apriori implementations. In *Proc. of 15th Conf. on Computational Statistics*, 2002.

[17] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 265–276, May 1997.

[18] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket analysis. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 255–264, May 1997.

[19] C. Bucila, J. E. Gehrke, D. Kifer, and W. White. DualMiner: A dual-pruning algorithm for itemsets with constraints. In *Proc. of the Eighth ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, 2002.

[20] D. Burdick, M.Calimlim, and J. E. Gehrke. MAFIA: a maximal frequent itemset algorithm for transactional databases. In *Proc. of Int'l Conf. on Data Engineering*, 2001.

[21] Y. Cheng and G. M. Church. Biclustering of expression data. In *Proc of the 8th Int'l. Conf. on intelligent Systems for Molecular Biology*, 2000.

[22] D.W. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. Int'l Conf. Data Engineering (ICDE)*, pages 106–114, Feb. 1996.

[23] G. Cong, W. Lee, H. Wu, and B. Liu. Semi-supervised text classification using partitioned EM. In *Proc. Int'l Conf. on Database Systems for Advanced Applications (DASFAA)*, 2004.

[24] G. Cong and B. Liu. Speed-up iterative frequent itemset mining with constraint changes. In *Proc. Int'l Conf. on Data Mining (ICDM)*, 2002.

[25] G. Cong, B.C. Ooi, K.-L Tan, and A. K. H. Tung. Go green: Recycle and reuse frequent patterns. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 2004.

[26] G. Cong, K.-L. Tan, A.K.H. Tung, and X. Xu. Mining top-k covering rule groups for gene expression data. In *Submitted for publication*, 2004.

[27] G. Cong, A.K.H. Tung, X. Xu, F. Pan, and J. Yang. FARMER: Fining interesting association rule groups by row enumeration in biological datasets. In *Proc. of ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2004.

[28] G. Cong, L. Yi, B. Liu, and K. Wang. Discovering frequent substructures from hierarchical semi-structured data. In *Proc. SIAM Int'l Conf. on Data Mining (SDM)*, 2002.

[29] C. Creighton and S. Hanash. Mining gene expression databases for association rules. *Bioinformatics*, 19, 2003.

[30] V. Dhar and A. Tuzhilin. Abstract-driven pattern discovery in databases. *IEEE Transactions on Knowledge and Data Engineering*, 5, 1993.

[31] S. Doddi, A. Marathe, S.S.Ravi, and D.C.Torney. Cluster analysis and display of genome-wide expression patterns. *Proc. Natl. Acad. Sci.*, 95:14863–14868, 1998.

[32] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proc. of the ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, pages 43–52, San Diego, CA, Aug. 1999.

[33] G. Dong, X. Zhang, L. Wong, and J. Li. CAEP: Classification by aggregating emerging patterns. In *Proc. 2nd Int'l Conf. Discovery Science (DS)*.

[34] B. Dunkel and N. Soparkar. Data organization and acess for efficient data mining. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 1999.

[35] R. Feldman, Y. Aumann, A. Amir, and H. Manila. Efficient algorithm for discovering frequent sets in incremental databases. In *Proc. ACM-SIGMOD Int'l Workshop Data Mining and Knowledge Discovery (DMKD)*, 1997.

[36] R. Feldman and H. Hirsh. Mining associations in text in the presence of background knowledge. In *Proc. of the ACM SIGKDD Int'l Conf. on Knowledge Discovery (KDD)*, 1996.

[37] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *Proc. 1996 ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 13–23, Montreal, Canada, June 1996.

[38] B. Goethals and M. J. Zaki. FIMI'03: Workshop on frequent itemset mining implementations. In *Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementation*, 2003.

[39] E. Han, G. Karypis, V. Kumar, and B. Mobasher. Clustering based on association rule hypergraphs. In *Proc. of the SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 1997.

[40] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pages 420–431, Zurich, Switzerland, Sept. 1995.

[41] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[42] J. Han and J. Pei. Mining frequent patterns by pattern-growth:methodology and implications. *KDD Exploration*, 2, 2000.

[43] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2000.

[44] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.

[45] M. Houtsma and A. Swami. Set-oriented mining for association rules in relational databases. In *Proc. Int'l Conf. Data Engineering (ICDE)*, pages 25–34, Taipei, Taiwan, Mar. 1995.

[46] J.Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining(KDD)*, Alberta, Canada, July 2002.

[47] T. Joachims. Making large-scale svm learning practical. 1999. svm-light.joachims.org.

[48] T. Johnson, L. V. S. Lakshmanan, and R. T. Ng. The 3W model and algebra for unified data mining. In *Proc. Int'l Conf. Very Large Data Bases(VLDB)*, Cairo, Egypt, 2000.

[49] D. Kifer, J. E. Gehrke, C. Bucila, and W. White. How to quickly find a witness. In *Proc. of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 2003.

[50] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A.I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proc. 3rd Int'l Conf. Information and Knowledge Management (CIKM)*, pages 401–408, Gaithersburg, Maryland, Nov. 1994.

[51] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, second edition, 1998.

[52] L. V. S. Lakshmanan, R. T. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 157–168, Philadelphia, PA, June 1999.

[53] J. Li and L. Wong. Identifying good diagnostic genes or genes groups from gene expression data by using the concept of emerging patterns. *Bioinformatics,* 18:725–734, 2002.

[54] W. Li, J. Han, and J. Pei. CMAR: Accurate and efficient classification based on multiple class-association rules. In *Proc. Int'l Conf. on Data Mining (ICDM),* 2001.

[55] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proc. of the ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD).*

[56] B. Liu, W. Hsu, and Y. Ma. Pruning and summarizing the discovered associations. In *Proc. of the ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, 1999.

[57] G. Liu, H. Lu, Y. Xu, and J. X. Yu. Ascending frequency ordered prefix-tree: Efficient mining of frequent patterns. In *Proc. Int'l Conf. on Database Systems for Advanced Applications (DASFAA)*, 2003.

[58] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *Proc. AAAI'94 Workshop Knowledge Discovery in Databases (KDD).*

[59] D. Margaritis, C. Faloutsos, and S. Thrun. NetCube: A scalable tool for fast data mining and compression. In *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2001.

[60] K. Mok, W. Lee, and S. Stolfo. Mining audit data to build intrusion models. In *Proc. of the ACM SIGKDD Intl. Conf. Knowledge Discovery and Data Mining (KDD)*, 1998.

[61] Y. Morimoto, T. Fukuda, H. Matsuzawa, and T. Tokuyama. Algorithms for mining association rules for binary segmentations of huge categorical databases. In *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, New York, NY, Aug. 1998.

[62] S. Morishita and J. Sese. Traversing itemset lattices with statistical metric pruning. In *Proc. of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 2002.

[63] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 1998.

[64] R. T. Ng, L. V. S. Lakshmanan, J. Han, and T. Mah. Exploratory mining via constrained frequent set queries (demo). In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 556–558, June 1999.

[65] C. Ordonez and P. Cereghini. SQLEM: Fast clustering in SQL using the EM algorithm. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, Dallas, Texas, USA, May 2000.

[66] F. Pan, G. Cong, Anthony K. H. Tung, J. Yang, and M. J. Zaki. CARPENTER: Finding closed patterns in long biological datasets. In *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining(KDD)*, 2003.

[67] F. Pan, A. K. H. Tung, G. Cong, and X. Xu. COBBLER: Combining column and row enumeration for closed pattern discovery. In *Proc. 16th Int'l Conf. on Scientific and Statistical Database Management*, 2004.

[68] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 175–186, San Jose, CA, May 1995.

[69] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *Proc. of the 8th Int'l Conf. Information and Knowledge Management (CIKM)*, Kansas City, MO, USA,, November 1999.

[70] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. 7th Int'l Conf. Database Theory (ICDT)*, 1999.

[71] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 433–332, Heidelberg, Germany, April 2001.

[72] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proc. Int'l Conf. Data Mining (ICDM)*, November 2001.

[73] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proc. ACM-SIGMOD Int'l Workshop Data Mining and Knowledge Discovery (DMKD)*, 2000.

[74] V. Pudi and J. Haritsa. ARMOR: Association rule mining based on oracle. In *Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementation*, 2003.

[75] V. Pudi and J. R. Haritsa. Quantifying the utility of the past in mining large databases. *Information System*, 25, 2000.

[76] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[77] R. Rastogi and K. Shim. Mining optimized association rules with categorical and numeric attributes. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 1998.

[78] P. Resnick and H.R. Varian. CACM special issue on recommender systems. *Communications of the ACM*, 40:56–58, 1997.

[79] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 343–354, Seattle, WA, June 1998.

[80] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pages 432–443, Sept. 1995.

[81] Ron Shamir and Roded Sharan. Algorithmic approaches to clustering gene expression data. *Current Topics in Computational Biology*, 2002.

[82] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 22–23, Dallas, TX, May 2000.

[83] A. Silberschatz and A. Tuzhilin. What makes patterns interesting in knowledge discovery systems. *IEEE Trans. on Knowledge and Data Engineering*, 8:970–974, Dec. 1996.

[84] L. Singh, Peter Scheuermann, and Bin Chen. Generating association rules from semi-structured documents using an extended concept hierarchy. In *Proc. of the Sixth Int'l Conf. on Information and Knowledge Management (CIKM)*, 1997.

[85] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 1–12, Montreal, Canada, June 1996.

[86] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, 1997.

[87] S.Tavazoie, J.D.Hughes, M.J.Campbell, R.J.Cho, and G.M.Church. Systematic determination of genetic network architecture. *Nature Genet.*, 22:281–285, 1999.

[88] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining(KDD)*, 1997.

[89] S. Thomas and S. Chakravarthy. Incremental mining of constrained associations. In *Proc. of HiPC'2000*, 2000.

[90] H. Toivonen. Sampling large databases for association rules. In *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pages 134–145, Bombay, India, Sept. 1996.

[91] D. Tsur, J. D. Ullman, S. Abitboul, C. Clifton, R. Motwani, and S. Nestorov. Query flocks: A generalization of association-rule mining. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 1–12, Seattle, WA, June 1998.

[92] J. Wang, J. Han, and J. Pei. CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, 2003.

[93] K. Wang and H.Q. Liu. Discovering association of structure from semistructured objects. *IEEE Transactions on Knowledge and Data Engineering*, 12:353–371, 2000.

[94] K. Wang, S. Zhou, and S. C. Liew. Building hierarchical classifiers using class proximity. In *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pages 363–374, Edinburgh, UK, Sept. 1999.

[95] G. I. Webb. Efficient search for association rules. In *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 99–107, 2000.

[96] G.I. Webb. OPUS: An efficient admissibe algorithm for unordered search. *Journal of Artificial Intelligence Research*, 3:431–465, 1995.

[97] Y. Xu, J. Yu, G. Liu, and H. Lu. From path tree to frequent patterns: A framework for mining frequent patterns. In *Proc. Int'l Conf. on Data Mining (ICDM)*, 2002.

[98] M. J. Zaki. Efficient enumeration of frequent sequences. In *Proc. 7th Int'l Conf. Information and Knowledge Management (CIKM)*, pages 68–75, Washington DC, Nov. 1998.

[99] M. J. Zaki. Generating non-redundant association rules. In *Proc. of the ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, 2000.

[100] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. of the ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, 2003.

[101] M. J. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed association rule mining. In *Proc. SIAM Int'l Conf. on Data Mining (SDM)*, 2002.

[102] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, pages 283–286, Newport Beach, CA, Aug. 1997.

[103] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithm for discovery of association rules. *Data Mining and Knowledge Discovery*, 1:343–374, 1997.

[104] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD)*, pages 103–114, Montreal, Canada, June 1996.

[105] Z. Zhang, A. Teo, B.C. Ooi, and K.-L. Tan. Mining deterministic biclusters in gene expression data. In *4th Symposium on Bioinformatics and Bioengineering*, 2004.

[106] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining(KDD)*, Sep 2001.