# Fast Algorithms for Sequential Pattern Mining

Zhenglu Yang

Department of Information and Communication Engineering

University of Tokyo

A thesis submitted for the degree of

*Doctor of Philosophy*

February 2008

# Acknowledgements

First of all, I would like to thank my advisor, Masaru Kitsuregawa. Through his support, care, and patience, Professor Kitsuregawa has fostered a struggling graduate student into an experienced researcher. His insight and ideas have helped me to form the foundation of this dissertation, and his guidance and care helped me get over various hurdles during my graduate years.

Secondly, I would like to thank my parents for their endless love, unconditional support and deep care. They make me always feel happiness and gratitude, which help me overcome every obstacle I must face.

Special thanks to Miyuki Nakano and Masashi Toyoda for being the reviewers of my presentation and thesis. They gave me many helpful comments on various aspects of my presentation. These comments made my presentation much better, and I learned a lot from their comments. Another person to whom I want to thank is Anirban Mondal. He helped me a lot on polishing my paper and I learned much from him.

I would also like to thank other members in Kitsuregawa Lab., including Toshihiro Nemoto, Takeshi Sagara, Nobuhiro Kaji, Kazuo Goda, Shingo Otsuka, Reiko Hamada, Shinji Suzuki, Masanori Yasukawa, Saneyasu Yamaguchi, Takayuki Tamura and Takashi Hoshino. I also want to thank to Lin Li, Wenyu Qu, Kulwadee Somboonviwat, Yanhui Gu and Yongkun Wang, for their great effort on building such a nice environment that just like a family I live.

# Abstract

Sequential pattern mining, which extracts frequent subsequences from a sequence database, has attracted a great deal of interest during the recent surge in data mining research because it is the basis of many applications. Much work has been carried out on mining frequent patterns, however, their performance is still far from satisfactory because of two main challenges: large search spaces and the ineffectiveness in handling dense data sets. To offer a solution to the above challenges, we have proposed a series of novel algorithms, called the LAst Position INduction (LAPIN) sequential pattern mining, which is based on the idea that the last position of an item, $\alpha$, is the key to judging whether or not a frequent k-length sequential pattern can be extended to be a frequent (k+1)-length pattern by appending the item $\alpha$ to it. LAPIN can largely reduce the search space during the mining process, and is very effective in mining dense data sets. Our performance study demonstrates that LAPIN outperforms the existing state-of-the-art algorithms by up to orders of magnitude on pattern dense data sets. Moreover, in this thesis, we propose our effective Web log mining system based on our efficient sequential mining algorithm to extract user access patterns from traversal path in Web logs.

Recently, the skyline query has attracted considerable attention. In this thesis, we are interested in the general model of skyline query, General Dominant Relationship. We find the interrelated connection between sequential pattern mining and the general dominant relationship. Based on this discovery, we extend sequential pattern mining techniques to efficiently answer the general dominant relationship queries. Extensive experiments illustrate the effectiveness and efficiency of our methods.

# Contents


**1 Introduction** **1**

1.1 Motivation . . . . . 4

1.2 Contributions . . . . . 5

1.3 Organization of the Thesis . . . . . 6

**2 Problem Definition and Related Work** **7**

2.1 Sequential Pattern Mining Problem . . . . . 7

2.2 Existing Sequential Pattern Mining Algorithms . . . . . 8

2.2.1 AprioriALL . . . . . 8

2.2.2 GSP . . . . . 11

2.2.3 SPADE . . . . . 14

2.2.4 SPAM . . . . . 17

2.2.5 PrefixSpan . . . . . 19

**3 LAPIN: Efficient Sequential Mining Algorithms** **22**

3.1 Problem of Existing Algorithms . . . . . 22

3.1.1 Experiment of Comparing Existing Algorithms . . . . . 23

3.1.2 Analysis . . . . . 23

3.2 LAPIN Algorithms (Last Position Induction Frequent Pattern Mining) 26

3.2.1 General Idea . . . . . 26

3.2.1.1 Lexicographic Tree . . . . . 29

3.2.1.2 Formulation . . . . . 30

3.2.2 LAPIN: Design and Implementation . . . . . 33

3.2.3 LAPIN_Suffix . . . . . 36

3.2.4 LAPIN_LCI . . . . . 38

3.2.5 A Complete Example . . . . . 41
3.3 Experimental Evaluation and Performance Study . . . . . 42
3.3.1 Synthetic Data. . . . . 45
3.3.2 Real Data. . . . . 48
3.3.3 Analysis. . . . . 51

**4 Improved Efficient Sequential Mining Algorithms 53**
4.1 LAPIN_PAID Algorithm (Passed Item Deduction Frequent Pattern Mining) . . . . . 53
4.1.1 General Idea . . . . . 53
4.1.1.1 Formulation . . . . . 56
4.1.2 Design and Implementation . . . . . 59
4.2 LAPIN_SPAM Algorithm . . . . . 62
4.2.1 General Idea . . . . . 63
4.2.2 Implementation . . . . . 64
4.2.2.1 Space Optimization . . . . . 64
4.3 Experimental Evaluation and Performance Study . . . . . 65
4.3.1 Scalability test between PrefixSpan, SPADE and LAPIN algorithms . . . . . 66
4.3.2 Real Data Evaluation between PrefixSpan, SPADE and LAPIN algorithms . . . . . 67
4.3.3 Scalability Study between SPAM and LAPIN-SPAM . . . . . 69
4.3.4 Memory Usage Analysis between SPAM and LAPIN_SPAM . 69
4.3.5 Systemic Study on Different Algorithms . . . . . 72
4.3.5.1 PrefixSpan v.s. LAPIN_Suffix v.s. LAPIN_PAID . . 72
4.3.5.2 SPADE v.s. LAPIN_LCI . . . . . 74
4.3.5.3 SPAM v.s. LAPIN_SPAM . . . . . 74
4.3.5.4 PrefixSpan v.s. SPADE v.s. LAPIN_SPAM . . . . . 77
4.3.5.5 LAPIN_PAID v.s. LAPIN_LCI v.s. LAPIN_SPAM . 77
4.3.5.6 Summary . . . . . 79

**5 Applications of Sequential Pattern Mining** **82**
5.1 Introduction of Web Log Mining . . . . . 82
5.1.1 Data Preparation . . . . . 83
5.1.1.1 Data Cleaning. . . . . 83
5.1.1.2 User Identification. . . . . 83
5.1.1.3 Session Identification. . . . . 84
5.1.2 Pattern Discovery . . . . . 84
5.1.3 Pattern Analysis . . . . . 85
5.2 LAPIN_Web Algorithm . . . . . 85
5.2.1 General Idea . . . . . 85
5.2.2 Implementation . . . . . 86
5.3 Experimental Evaluation and Performance Study . . . . . 89
5.3.1 Datasets . . . . . 89
5.3.2 Experiments and Evaluations . . . . . 90
5.3.2.1 Comparing PrefixSpan with LAPIN_WEB . . . . . 90
5.3.2.2 Visualization Result . . . . . 91

**6 Extension of Sequential Pattern Mining** **95**
6.1 Introduction of Skyline Query . . . . . 95
6.2 Related Work . . . . . 97
6.2.1 Skyline Query . . . . . 97
6.2.2 Sequential Pattern Mining . . . . . 99
6.2.3 Data Cube . . . . . 100
6.2.4 Partial Order Mining . . . . . 100
6.2.5 Graph Construction . . . . . 101
6.3 General Dominance Relationship Analysis . . . . . 102
6.3.1 Preliminaries . . . . . 102
6.3.2 General Idea . . . . . 103
6.3.3 Constructing Partial Order Data Cube (ParCube) . . . . . 104
6.3.3.1 Optimization of Sequential Pattern Mining . . . . . 107
6.3.3.2 Compression of the ParCube Data Cube . . . . . 109
6.3.4 Efficient ParCube Querying . . . . . 110
6.4 Experimental Evaluation and Performance Study . . . . . 111

6.4.1 Datasets . . . . . 111
6.4.2 Skyline Query Performance . . . . . 112
6.4.3 Dominant Relationship Query Performance . . . . . 112
6.4.4 Index Data Structure Construction Performance . . . . . 112
6.4.5 Effectiveness of Compression . . . . . 115

**7 Discussion** **118**
7.1 Extension of Sequential Pattern Mining . . . . . 118
7.1.1 Constraint-based Mining of Sequential Patterns . . . . . 118
7.1.2 Mining Closed and Maximal Sequential Patterns . . . . . 119
7.1.3 Mining Approximate Sequential Patterns . . . . . 119
7.1.4 Sequential Patterns Compression . . . . . 119
7.1.5 Sequential Pattern Mining Over Data Stream . . . . . 121
7.1.6 Toward Mining Other Kinds of Structured Patterns . . . . . 121
7.2 Extension of Skyline Mining . . . . . 121
7.2.1 Ranked Skyline Queries . . . . . 121
7.2.2 Constrained Skyline Queries . . . . . 122
7.2.3 Dynamic Skyline Queries . . . . . 122
7.2.4 Enumerating and K-dominating Queries . . . . . 123
7.3 Summary . . . . . 123

**8 Conclusions** **125**
8.1 Summary of the Thesis . . . . . 125
8.2 Future Research Directions . . . . . 126

**A Publication List** **128**
A.1 Journal Papers . . . . . 128
A.2 International Conference Papers . . . . . 128
A.3 Workshop Papers . . . . . 130

**References** **142**

# Chapter 1

# Introduction

Data mining is to find valid, novel, potentially useful, and ultimately understandable patterns in data [37]. Sequential pattern mining, which extracts frequent subsequences from a sequence database, has attracted a great deal of interest during the recent surge in data mining research because it is the basis of many applications, such as customer behavior analysis, stock trend prediction, and DNA sequence analysis. The sequential mining problem was first introduced in [5]; two sequential patterns examples are: "80% of the people who buy a television also buy a video camera within a day", and "Every time Microsoft stock drops by 5%, then IBM stock will also drop by at least 4% within three days". The above patterns can be used to determine the efficient use of shelf space for customer convenience, or to properly plan the next step during an economic crisis. Sequential pattern mining is also very important for analyzing biological data [8] [36], in which a very small alphabet (i.e., 4 for DNA sequences and 20 for protein sequences) and long patterns with a typical length of few hundreds or even thousands, frequently appear.

Sequence discovery can be thought of as essentially an association discovery over a temporal database. While association rules [4, 51] discern only intra-event patterns (itemsets), sequential pattern mining discerns inter-event patterns (sequences). There are many other important tasks related to association rule mining, such as correlations [20], causality [86], episodes [67], multi-dimensional patterns [49, 60], maximal patterns [13], partial periodicity [44], and emerging patterns [35]. Elaborate exploration of sequential pattern mining issue will be beneficial to the other research problems shown above a lot. In Chapter 5, as an example, one extension of sequential pattern

mining will be introduced. Thus, effective and efficient sequential pattern mining is an important and interesting research problem.

Efficient sequential pattern mining methodologies have been studied extensively in many related problems, including the general sequential pattern mining [5, 7, 79, 88, 109], constraint-based sequential pattern mining [39], incremental sequential pattern mining [74], frequent episode mining [66], approximate sequential pattern mining [55], partial periodic pattern mining [44], temporal pattern mining in data stream [93], maximal and closed sequential pattern mining [64, 95, 105].

Although there are so many problems related to sequential pattern mining explored, we realize that the general sequential pattern mining algorithm development is the most basic one because all the others can benefit from the strategies it employs, i.e., Apriori heuristic and projection-based pattern growth. Hence we aim to develop an efficient general sequential pattern mining algorithm in this paper.

Much work has been carried out on mining frequent patterns, as for example, in [5, 7, 79, 88, 109]. However, all of these works suffer from the problems of having a large search space and the ineffectiveness in handling dense data sets, i.e., biological data. In this work, we propose new strategies to reduce the space necessary to be searched. Instead of searching the entire projected database for each item, as PrefixSpan [79] does, we only search a small portion of the database by recording the last position of each item in each sequence. Because support counting is usually the most costly step in sequential pattern mining, the LAst Position INduction (LAPIN) technique can improve the performance greatly by avoiding cost scanning and comparisons using a pre-constructed table in bit vector format.

However, we found that the improvement is at the price of much memory consuming when building the list of item's last position because LAPIN uses a bitmap strategy. We aim to obtain an efficient and balanced pattern mining algorithm with low memory consuming and thus, we proposed an improved algorithm which makes good use of not only the position of item but also the intermediate value (support value) of k-length pattern when fining (k+1)-length pattern. The experiments demonstrated that our improved algorithm performs the best in limited resource environments for dense datasets.

Ayres et al. [7] claimed that SPAM is very efficient for long pattern mining and it can outperform PrefixSpan by up to an order of magnitude. Our experiments show

that, although SPAM can handle long patterns in dense data sets, it is limited in the length of long patterns it can handle, and its high speed comes at a price of large space consumption. We proposed a new algorithm named LAPIN_SPAM, which combines the idea of LAPIN and SPAM. The experiments demonstrated that LAPIN_SPAM significantly outperforms the original SPAM, and is the best under unlimited resource assumption for dense datasets.

The WWW provides a simple yet effective media for users to search, browse, and retrieve information in the Web. Web log mining is a promising tool to study user behaviors, which could further benefit web-site designers with better organization and services. Although there are many existing systems that can be used to analyze the traversal path of web-site visitors, their performance is still far from satisfactory. In this thesis, we propose our effective Web log mining system based on our efficient sequential mining algorithm, LAPIN_WEB, an extension of LAPIN algorithm to extract user access patterns from traversal path in Web logs. Our experimental results and performance studies demonstrate that LAPIN_WEB is very efficient and outperforms well-known PrefixSpan by up to an order of magnitude on real Web log datasets. Moreover, we also implement a visualization tool to help interpret mining results as well as predict users' future requests.

Recently, the skyline query has attracted considerable attention because it is the basis of many applications, e.g., multi-criteria decision making [19], user-preference queries [50] [47] and microeconomic analysis [61]. Given an $N$-dimensional dataset D, a point p is said to dominate another point q if p is better than q in at least one dimension and equal to or better than q in the remaining dimensions. Skyline mining aims to find those non-dominated points, in a $N$-dimensional spatial dataset. This problem can be seen as a special class of pareto preference queries [50].

Efficient skyline querying methodologies have been studied extensively. However, all the papers concerned only the pure dominant relationship among a dataset, i.e., a point p is whether dominated by others or not, and got those non-dominated ones as results. In the real world, users are more interested in the detail of the dominant relationship in a dataset, i.e., a point p dominates how many other points and whom they are. This problem can be seen as a general dominant relationship analysis to the skyline query and has not been studied.

In this thesis, we find the interrelated connection between sequential pattern mining and the general dominant relationship. Based on this discovery, we propose efficient algorithms to answer the general dominant relationship queries by using efficient sequential pattern mining algorithms and several other strategies. Extensive experiments illustrate the effectiveness and efficiency of our methods.

## 1.1 Motivation

Most of the previous studies on sequential pattern mining, such as [5, 7, 88, 109], adopt an Apriori-like approach, which is based on an anti-monotone Apriori heuristic [4]: if any length k pattern is not frequent in the database, its length (k + 1) super-pattern can never be frequent. On the other hand, Pei et al. proposed a projection-based algorithm, PrefixSpan [79], which scans and counts the support of the candidates in the $projected\ databases$.

The two kinds of approaches can achieve good performance by reducing the size of candidate sets. However, in situations to mine those dense datasets with prolific sequential patterns and long patterns, the existing state-of-the-art algorithms may still suffer from the problems of having a large search space and the ineffectiveness in handling dense data sets.

As sequential pattern mining is an essential data mining task, developing efficient sequential mining techniques has been an important research direction in data mining.

There are some interesting questions that need to be answered.

- Apriori is one basic principle in frequent pattern mining. To improve the efficiency of sequential pattern mining substantially, is there any way to obtain this advantage while avoiding the costly duplicate candidate test and repeated database scan operations?

- Projection-based algorithm (i.e., PrefixSpan) shows good performance when mining sparse datasets. Could we improve the efficiency by avoiding the disadvantage of scanning large projection databases?

- Due to the trade-off between the Apriori-based algorithm and the projection-based algorithm on mining various kinds of datasets, could we find an integrated heuristic strategy to efficiently discovery the sequential patterns?
- Sequential pattern mining has many potential applications. Can we extend the effective and efficient sequential pattern mining methods to solve some other interesting data mining problems?

This thesis tries to make good progress in answering the above questions.

## 1.2 Contributions

In this thesis, we study the problem of efficient and effective sequential pattern mining, as well as some of its extensions and applications. In particular, we make the following contributions.

- We systematically develop a location aware method for sequential pattern mining. A family of novel algorithms, LAPIN, is proposed for efficiently mining sequential patterns on large dense datasets. The algorithms are developed for varies of kinds datasets. We conducted comprehensive experiments to confirm the efficiency of our algorithms.
- We apply our proposed algorithm to mine web log with modification with regard to the special property of the web log. We build an integrated system to facilitate the analysis web user behavior.
- We extend the sequential pattern mining method to allow the analysis of dominant relationship, which is an important issue in spatial data mining. The idea is based on the correlation between time and space. Our study shows that sequential pattern mining methods can be helpful in spatial data mining. Interesting techniques are developed to solve the dominant relationship analysis effectively.

## 1.3 Organization of the Thesis

The remainder of the thesis is structured as follows:

- In Chapter 2, we present the sequential pattern mining problem and an overview of related work systematically.
- In Chapter 3, a family of novel sequential pattern algorithms, LAPIN, is developed. The correctness and efficiency of LAPIN are verified by theoretical analysis and experimental tests. We also explain the reason of developing different algorithms to tackle varies kind of datasets.
- Even though LAPIN is efficient in mining large dense databases, it may have the problem while dealing with vary kinds of datasets. In Chapter 4, we propose LAPIN_PAID, which retains the advantages of LAPIN but avoids redundant support counting. Our performance study shows that LAPIN_PAID achieves good scalability in mining large databases and is also efficient in space. Moreover, we develop an improved algorithm based on SPAM [7], which is the best one in resource unlimited environments for dense datasets.
- By applying the algorithms what we develop, we build an integrated web log mining system, which will be studied in Chapter 5.
- We extend the sequential pattern mining methods to solve the general dominant relationship analysis problem in Chapter 6. The study indicates that, with some modification and customization, sequential pattern mining methods can be applied to mine patterns from various kinds of databases.
- In Chapter 7 we discuss several issues which are related and can be extended based on general sequential pattern mining techniques and skyline mining, such as constraint-based pattern mining, approximate pattern mining, pattern compression, data stream mining, Top-K query. Specially, the inter-connection between sequential pattern mining and Skyline query is emphasized and we believe that these two fields can learn from each other.
- The thesis concludes in Chapter 8. Some future directions are further presented.

# Chapter 2

# Problem Definition and Related Work

In this chapter, we first define the problem of frequent pattern mining, then we revisit the existing heuristic and algorithms.

## 2.1 Sequential Pattern Mining Problem

Let $I = \{i_1, i_2, \ldots, i_k\}$ be a set of items. A subset of $I$ is called an $itemset$ or an $element$. A $sequence$, $s$, is denoted as $\langle t_1, t_2, \ldots, t_l \rangle$, where $t_j$ is an itemset, i.e., $(t_j \subseteq I)$ for 1 $\leq$ j $\leq$ l. The $itemset$, $t_j$, is denoted as $(x_1 x_2 \ldots x_m)$, where $x_k$ is an item, i.e., $x_k \in I$ for 1 $\leq$ k $\leq$ m. For brevity, the brackets are omitted if an $itemset$ has only one item. That is, $itemset$ $(x)$ is written as $x$. The number of items in a sequence is called the $length$ of the sequence. A sequence with length $l$ is called an $l$-$sequence$. A sequence, $s_a = \langle a_1, a_2, \ldots, a_n \rangle$, is contained in another sequence, $s_b = \langle b_1, b_2, \ldots, b_m \rangle$, if there exists integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$, such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$,..., $a_n \subseteq b_{i_n}$. We denote $s_a$ a $subsequence$ of $s_b$, and $s_b$ a $supersequence$ of $s_a$. Given a sequence $s = \langle s_1, s_2, \ldots, s_l \rangle$, and an item $\alpha$, $s \diamond \alpha$ denotes that s concatenates with $\alpha$, which has two possible forms, such as $Itemset\ Extension$ ( $IE$), $s \diamond \alpha$=$\langle s_1, s_2, \ldots, s_l \cup \{\alpha\} \rangle$, or $Sequence\ Extension$ ( $SE$), $s \diamond \alpha$=$\langle s_1, s_2, \ldots, s_l, \{\alpha\} \rangle$. If $s' = p \diamond s$, then $p$ is a $prefix$ of $s'$ and $s$ is a $suffix$ of $s'$.

A $sequence\ database$, $S$, is a set of tuples $\langle sid, s \rangle$, where $sid$ is a sequence_id and $s$ is a sequence. A tuple $\langle sid, s \rangle$ is said to contain a sequence $\beta$, if $\beta$ is a $subsequence$

of $s$. The support of a sequence, $\beta$, in a sequence database, $S$, is the number of tuples in the database containing $\beta$, denoted as $support(\beta)$. Given a user specified positive integer, $\varepsilon$, a sequence, $\beta$, is called a frequent sequential pattern if $support(\beta) \geq \varepsilon$. In this work, the objective was to find the complete set of sequential patterns of database $S$ in an efficient manner.

## 2.2 Existing Sequential Pattern Mining Algorithms

Sequential pattern mining algorithms can be grouped into two categories. One category is Apriori-like algorithm, such as Apriori-all [5], GSP [88], SPADE [109], and SPAM [7], the other category is projection-based pattern growth, such as PrefixSpan [79]. Next the five algorithms are surveyed one by one.

### 2.2.1 AprioriALL

Sequential pattern mining was first introduced in [5]. The authors proposed three Apriori-based algorithms. There are five phases in the whole work flow of the algorithms. To conveniently explain them, we use a sample database hereafter, as shown in Figure 2.1.

- **Sort Phase:** At first, the original transaction database is sorted with customer-id as the major key and transaction time as the minor key. The result is the set of customer sequences. The sample database in Figure 2.1 is indeed the transaction data after sorting.

- **L-itemsets Phase:** Then we scan the sorted database obtain the large 1-itemsets based on the predefined support threshold. Suppose the minimal support is 60%, the minimal support count is thus 2. The result of large 1-itemsets is listed in Figure 2.2, which are $a$, $b$, $c$ and $d$ (mapped ID).

- **Transformation Phase:** In this phase, the customer sequences are replaced by those large itemsets they contain. All the large itemsets are mapped into a series of integers (here we use characters to convenient the explanation) to make the mining more efficient. At the end of this phase the original database is transformed into the set of customer sequences represented by those large itemsets.

| Customer ID | Transaction Time | Items Bought |
|---|---|---|
| 1 | July 3 '07 | Apple |
| 1 | July 6 '07 | Strawberry |
| 1 | July 8 '07 | Banana, Strawberry |
| 1 | July 10 '07 | Pear |
| 1 | July 12 '07 | Apple, Banana, Strawberry |
| 1 | July 16 '07 | Apple |
| 1 | July 21 '07 | Pear |
| 2 | July 4 '07 | Banana |
| 2 | July 7 '07 | Strawberry, Pear |
| 2 | July 9 '07 | Apple |
| 2 | July 10 '07 | Strawberry |
| 2 | July 15 '07 | Banana, Pear |
| 3 | July 13 '07 | Pear |
| 3 | July 15 '07 | Banana, Strawberry |
| 3 | July 21 '07 | Apple, Strawberry |
| 3 | July 24 '07 | Strawberry, Pear |

Figure 2.1: Database Sorted by Customer ID and Transaction Time

For example, transactions with customer-id 1 are transformed into customer sequence $\langle ac(bc)d(abc)ad \rangle$, with the help of the map table in Figure 2.2. Finally the result database is shown in Figure 2.3.

- **Sequence Phase:** This is an essential phase that all frequent sequential patterns are generated from the transformed sequential database.
- **Maximal Phase:** To reduce information redundant, those sequential patterns that are contained in other sequential patterns are pruned because those maximal sequential patterns are more interested by users.

It is shown that the sequence phase is the most time consuming one among the five phases [5]. Next we introduce AprioriAll because it is the best one among the three algorithms proposed in [5]. There are two steps in AprioriAll, candidate generation and test. The first step is to generate those sequences that may be frequent. Then in the second step the sequence database is scanned to check the support of each candidate

| Large Itemsets | Mapped To |
|---|---|
| Apple | a |
| Banana | b |
| Strawberry | c |
| Pear | d |

Figure 2.2: Large Itemsets

| Customer ID | Customer Sequence |
|---|---|
| 1 | < ac(bc)d(abc)ad > |
| 2 | < b(cd)ac(bd) > |
| 3 | < d(bc)(ac)(cd) > |

Figure 2.3: Transformed Database

to determine frequent sequential patterns based on the minimal support. The time cost of the two steps is mainly determined by the number of passes over the database and number of candidates.

Similar to the technique proposed in [4], AprioriAll applies Apriori heuristic to prune those candidate sequences whose subsequence is not frequent. The difference is that sequential pattern mining can be seen a general model of association rule mining and thus, more candidates are generated. For example, based on the items, $a$ and $b$, three candidates $\langle ab \rangle$, $\langle ba \rangle$ and $\langle (ab) \rangle$ can be generated. But in association rule mining only $\langle (ab) \rangle$ is generated. The reason is that in association rule mining, the time order is not taken into account. Obviously the number of candidate sequences in sequential pattern mining are much larger than the size of the candidate itemsets in association rule mining during the generation of candidate sequences. Table 2.1 shows how to generate candidate 5-sequences by joining large 4-sequences. By scanning the large 4-itemsets, it finds that the itemsets $\langle (bc)ad \rangle$ and $\langle (bc)(ac) \rangle$ share their first three items, according to the join condition of Apriori they are joined to produce the candidate sequence $\langle (bc)(ac)d \rangle$. Similarly other candidate 5-sequences are generated.

It is easily to test and count the support of candidates, by scanning the original database directly. AprioriAll was the first algorithm to mine sequential patterns. The

Table 2.1: AprioriAll Candidate Generation $L_4$ to $C_5$

| Large 4-sequences | Candidate 5-sequences |
|---|---|
| ⟨b(ac)d⟩ | ⟨(bc)(ac)d⟩ |
| ⟨bcad⟩ | ⟨d(bc)ad⟩ |
| ⟨bdad⟩ | ⟨d(bc)da⟩ |
| ⟨bdcd⟩ | ⟨d(bc)(ad)⟩ |
| ⟨(bc)ad⟩ | |
| ⟨(bc)(ac)⟩ | |
| ⟨(bc)cd⟩ | |
| ⟨c(ac)d⟩ | |
| ⟨d(ac)d⟩ | |
| ⟨dbad⟩ | |
| ⟨d(bc)a⟩ | |
| ⟨d(bc)d⟩ | |
| ⟨dcad⟩ | |

main drawback of AprioriAll is that there are many passes over the database and many candidates generated, which are time consuming. As will be introduced later, AprioriAll's key idea (i.e., Apriori heuristic) is the basis for many other efficient algorithms.

### 2.2.2 GSP

GSP [88] is proposed by the same authors of AprioriAll, which is also an Apriori based algorithm for sequential pattern mining. The difference is that GSP inserts some constraints into the mining process, i.e., time constraints, and relaxes the definition of transaction. Moreover, it takes the taxonomies into account. For time constraints, maximum gap and minimal gap are defined to specified the gap between any two adjacent transactions in the sequence. If the distance between two transactions is not in the range between the maximum gap and the minimal gap, then the two transactions can not be taken as two consecutive transactions in a sequence. This algorithm relaxes the definition of transaction by using a sliding window that, if the distance between the maximal transaction time and the minimal transaction time of those items is not bigger

than the sliding window, those items can be deemed as in the same transaction. The taxonomies is applied to generate multiple level sequential patterns. With these new parameters, sequential pattern mining can be defined as: given a sequence data D, a taxonomy T, user-defined min-gap and max-gap time constraints, a user-defined sliding window-size, to find all sequences whose support is greater than the user-defined minimum support [88].

GSP has two steps in the mining process, candidate generation and test, which is similar to other Apriori-based algorithms. In the candidate generation step, k-sequences candidates are generated based on the large (k-1)-sequences. Given a sequence $s = \langle s_1, s_2, \ldots, s_n \rangle$ and subsequence $c$, $c$ is a contiguous subsequence of $s$ if any of the following conditions hold: (1) $c$ is derived from s by dropping an item from either $s_1$ or $s_n$; (2) $c$ is derived from $s$ by dropping an item from an element $s_j$ that has at least 2 items; and (3) $c$ is a contiguous subsequence of $c'$, and $c'$ is a contiguous subsequence of $s$. The candidate sequences are generated in two phases.

- **Join Phase:** Candidate k-sequences are generated by joining two (k-1)-sequences that have the same contiguous subsequences. This is similar to AprioriAll that when joining the two sequences, the item can be inserted as a part of the element or as a separated element. For example, $\langle d(bc)a \rangle$ and $\langle d(bc)d \rangle$ have the same contiguous subsequence $\langle d(bc) \rangle$, then candidate 5-sequences $\langle d(bc)(ad) \rangle$, $\langle d(bc)ad \rangle$ and $\langle d(bc)da \rangle$ can be generated. These two steps (i.e., item is inserted as a part of the element or as a separated element), as will be shown shortly in the SPAM algorithm [7], can be called $I$-$Step$ and $S$-$Step$.

- **Prune Phase:** Those candidate sequences that have a contiguous subsequence whose support count is less than the minimal support are deleted. Furthermore, the hash-tree structure [73] is employed to fasten the prune process.

GSP algorithm is difficult to count the support of candidate sequences because of the maximal and minimal gaps incorporation. There are two phases in the contain test, forward phase and backward phase, which are repeated until all the patterns are found. Here we give an example to show how to judge a data-sequence $d$ contain a candidate sequence $s$ by two phases: (1) $Forward\ Phase$, GSP finds successive elements of $s$ in $d$ as long as the difference between the end-time of the element and

Table 2.2: GSP Candidate Generation $L_4$ to $C_5$

| Large 4-sequences | Candidate 5-sequences after joining | Candidate 5-sequences after pruning |
|---|---|---|
| ⟨b(ac)d⟩ | ⟨(bc)(ac)d⟩ | ⟨(bc)(ac)d⟩ |
| ⟨bcad⟩ | ⟨d(bc)ad⟩ | ⟨d(bc)ad⟩ |
| ⟨bdad⟩ | ⟨d(bc)da⟩ | |
| ⟨bdcd⟩ | ⟨d(bc)(ad)⟩ | |
| ⟨(bc)ad⟩ | | |
| ⟨(bc)(ac)⟩ | | |
| ⟨(bc)cd⟩ | | |
| ⟨c(ac)d⟩ | | |
| ⟨d(ac)d⟩ | | |
| ⟨dbad⟩ | | |
| ⟨d(bc)a⟩ | | |
| ⟨d(bc)d⟩ | | |
| ⟨dcad⟩ | | |

the start-time of the previous element is less than max-gap. If the difference is more than max-gap, it switches to the backward phase. If an element is not found then sequence $s$ is not contained in $d$; (2) $Backward\ Phase$, GSP tries to pull up the previous element. Suppose $s_i$ is the current element and end-time($s_i$)=$t$. GSP checks if there is existing some transactions containing $s_{i-1}$ and their transaction-times are after time of max-gap. Since after pulling up $s_{i-1}$, the difference between $s_{i-1}$ and $s_{i-2}$ may not satisfy the gap constraints, the backward pulls back until the difference of $s_{i-1}$ and $s_{i-2}$ satisfies the max-gap or the first element has been pulled up. Then the algorithm switches to the forward phase. If any element can not be pulled up the data-sequence $d$ does not contain $s$.

GSP performs better than AprioriAll because the number of candidate sequences of the former is much smaller than that of the latter, by incorporating constraint into mining process. Moreover, the discovered frequent patterns show more reasonable semantic meaning to the users.

| a | | b | | c | | d | |
|---|---|---|---|---|---|---|---|
| SID | TID | SID | TID | SID | TID | SID | TID |
| 1 | 1 | 1 | 3 | 1 | 2 | 1 | 4 |
| 1 | 5 | 1 | 5 | 1 | 3 | 1 | 7 |
| 1 | 6 | 2 | 1 | 1 | 5 | 2 | 2 |
| 2 | 3 | 2 | 5 | 2 | 2 | 2 | 5 |
| 3 | 3 | 3 | 2 | 2 | 4 | 3 | 1 |
| | | | | 3 | 2 | 3 | 4 |
| | | | | 3 | 3 | | |
| | | | | 3 | 4 | | |

Figure 2.4: Vertical Id-List

| SID | (Item, TID) pairs |
|---|---|
| 1 | (a, 1) (c, 2) (b, 2) (c, 2) (d, 4) (a, 5) (b, 5) (c, 5) (a, 6) (d, 7) |
| 2 | (b, 1) (c, 2) (d, 2) (a, 3) (c, 4) (b, 5) (d, 5) |
| 3 | (d, 1) (b, 2) (c, 2) (a, 3) (c, 3) (c, 4) (d, 4) |

Figure 2.5: Vertical to Horizontal Database

### 2.2.3 SPADE

SPADE [109] is an algorithm that is based on lattice theory and applies temporal joining operation to find sequential patterns. It is also based on Apriori heuristic and performs much better than AprioriAll and GSP.

In SPADE the original sequence database is firstly transformed into a vertical id-list database format, in which each id is associated with the corresponding customer sequence (SID) and the time stamp (TID). The vertical database of Figure 2.1 is shown in Figure 2.4. For example, item $a$ appears in (1,1), (1,5), (1,6), (2,3), (3,3) and hence, the support of item $a$ is 3. Frequent 1-sequences can be easily discovered by scanning the database once. To test the 2-sequence patterns, the original database is scanned again and the new vertical to horizontal database is created by grouping those items with the same SID and in increase order of TID. The result vertical to horizontal database is shown in Figure 2.5. By scanning the vertical to horizontal database, 2-sequences are

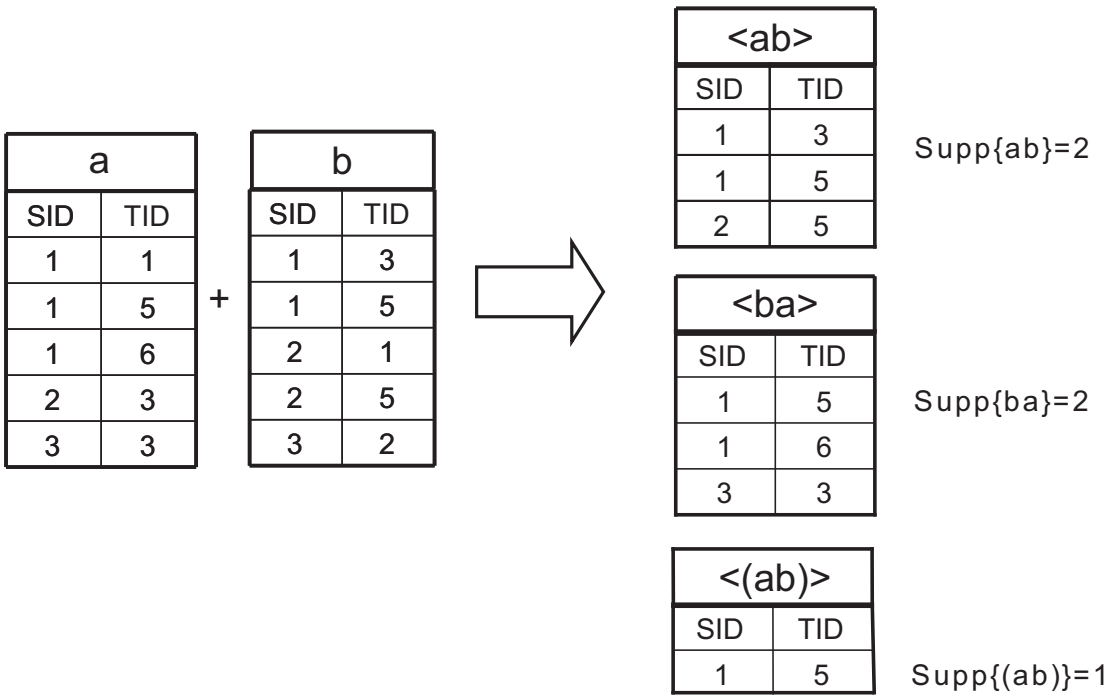



Figure 2.6: Temporal join in SPADE algorithm

generated. All the 2-length sequence found are used to construct the lattice, which is quite large to be filled in the main memory. However, the lattice can be further decomposed to different classes. Sequences that have the same prefix items belong to the same class. By decomposition, the lattice is partitioned into small partitions that can be filled in the main memory. During the third scanning of the database, all those longer sequences are enumerated by using temporal join [109].

There are two methods of enumerating frequent sequences of a class: Breadth First Search (BFS) and Depth First Search (DFS). In BFS, the classes are generated in a recursive bottom-up manner. For example to generate the 3-length sequences all the 2-length sequences have to be processed. On the contrary, in DFS only one 2-length sequence and a k-length sequence are necessary to generate (k+1)-length sequence. BFS needs much bigger main memory to store all the consecutive 2-length sequences, but DFS just needs to store the last 2-length sequence of the newly generated k-length sequences. However BFS has more information to prune the candidate k-length sequences. All the k-length patterns are discovered by temporal joining the frequent

| SID | TID | {a} | {b} | {c} | {d} |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 2 | 0 | 0 | 1 | 0 |
| 1 | 3 | 0 | 1 | 1 | 0 |
| 1 | 4 | 0 | 0 | 0 | 1 |
| 1 | 5 | 1 | 1 | 1 | 0 |
| 1 | 6 | 1 | 0 | 0 | 0 |
| 1 | 7 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 0 | 0 |
| 2 | 2 | 0 | 0 | 1 | 1 |
| 2 | 3 | 1 | 0 | 0 | 0 |
| 2 | 4 | 0 | 0 | 1 | 0 |
| 2 | 5 | 0 | 1 | 0 | 1 |
| 3 | 1 | 0 | 0 | 0 | 1 |
| 3 | 2 | 0 | 1 | 1 | 0 |
| 3 | 3 | 1 | 0 | 1 | 0 |
| 3 | 4 | 0 | 0 | 1 | 1 |

Figure 2.7: Bitmap Vertical Table

(k-1)-length patterns which have the same (k-2)-length prefix. There are three possible joining results which is the same in the candidate generation process of GSP [88].

Figure 2.6 illustrates one example of temporal join operations in SPADE. Suppose we have already got 1-length patterns, $a$ and $b$. By joining these two patterns, we can test the three candidate sequences, $\langle ab \rangle$, $\langle ba \rangle$ and $\langle (ab) \rangle$. The joining operation is indeed to compare the $\{$SID, TID$\}$ pairs of the two (k-1)-length patterns. For example, the pattern $b$ has two pairs $\{1, 3\}$, $\{1, 5\}$ which are larger than (behind) the pattern $a$'s one pair $\{1, 1\}$, in the same customer sequence. Hence, $\langle ab \rangle$ should exist in the same sequence. By the same way, other candidate sequences' support can be accumulated, as illustrated on the right part of Figure 2.6.

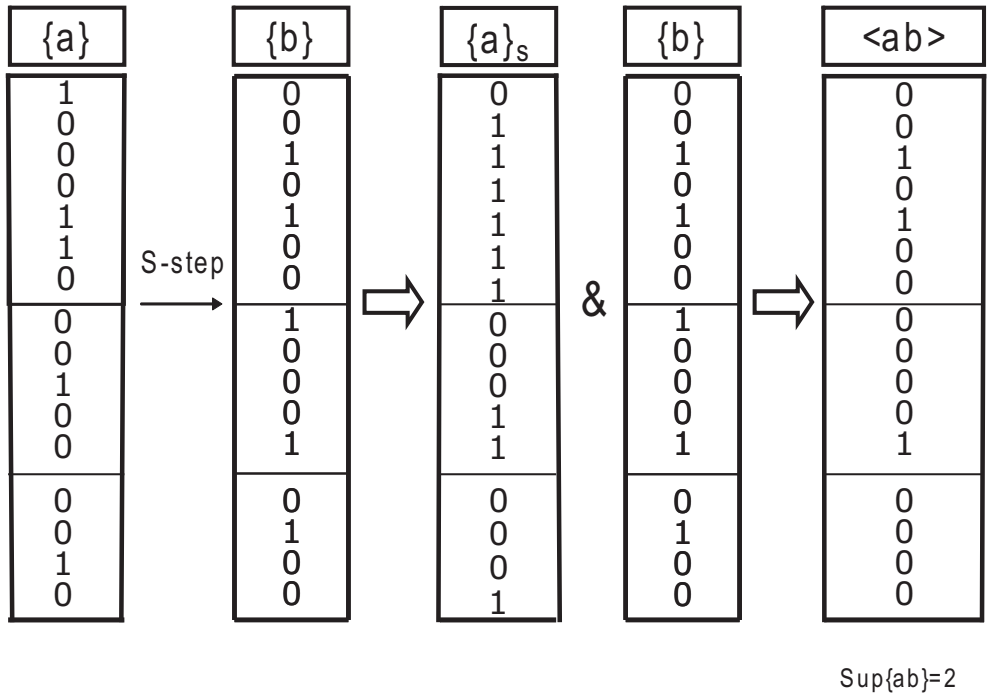



Figure 2.8: SPAM S-Step join

### 2.2.4 SPAM

Ayres et al. [7] proposed SPAM algorithm based on the key idea of SPADE. The difference is that SPAM utilizes a bitmap representation of the database instead of $\{SID, TID\}$ pairs used in the SPADE algorithm. Hence, SPAM can perform much better than SPADE and others by employing bitwise operations.

While scanning the database for the first time, a vertical bitmap is constructed for each item in the database, and each bitmap has a bit corresponding to each itemset (element) of the sequences in the database. If an item appears in an itemset, the bit corresponding to the itemset of the bitmap for the item is set to one; otherwise, the bit is set to zero. The size of a sequence is the number of itemsets contained in the sequence. Figure 2.7 shows the bitmap vertical table of that in Figure 2.3. A sequence in the database of size between $2^k$+1 and $2^{k+1}$ is considered as a $2^{k+1}$-bit sequence. The bitmap of a sequence will be constructed according to the bitmaps of items contained in it.

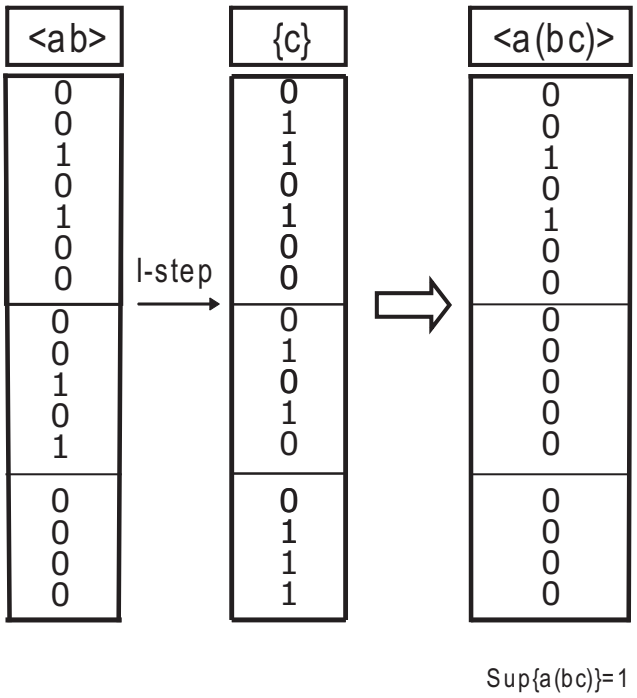



Figure 2.9: SPAM I-Step join

To generate and test the candidate sequences, SPAM uses two steps, S-step and I-step, based on the lattice concept. As a depth-first approach, the overall process starts from S-step and then I-step. To extend a sequence, the S-step appends an item to it as the new last element, and the I-step appends the item to its last element if possible. Each bitmap partition of a sequence to be extended is transformed first in the S-step, such that all bits after the first bit with value one are set to one. Then the resultant bitmap of the S-step can be obtained by doing ANDing operation for the transformed bitmap and the bitmap of the appended item. Figure 2.8 illustrates how to join two 1-length patterns, $a$ and $b$, based on the example database in Figure 2.3. On the other hand, the I-step just uses the bitmaps of the sequence and the appended item to do ANDing operation to get the resultant bitmap, as shown in Figure 2.9, which extend the pattern $\langle ab \rangle$ to the candidate $\langle a(bc) \rangle$. The support counting becomes a simple check how many bitmap partitions not containing all zeros. Yet for the inherent characteristic existed in the sequential pattern mining problem, these ANDing operations cost a lot during the whole mining process, which should be reduce for efficiency improving.

According to the two processes existed in SPAM, it uses two pruning techniques: S-step pruning and I-step pruning, based on the Apriori heuristic to minimize the size of the candidate items.

The main drawback of SPAM is the huge memory space necessary. For example, although an item, $\alpha$, does not exist in a sequence, $s$, SPAM still uses one bit to represent the existence of $\alpha$ in $s$. This disadvantage restricts SPAM as a best algorithm on mining large datasets in limit resource environments.

### 2.2.5 PrefixSpan

PrefixSpan [79] utilizes the method of database projection to make the database for next pass much smaller and consequently make the algorithm more speedy. The authors claimed that in PrefixSpan there is no need for candidates generation [79] [1]. It recursively projects the database by already found short length patterns. This pattern growth idea is similar to that in Apriori heuristic. Different projection methods were discussed for PrefixSpan: level-by-level projection, bi-level projection and pseudo projection.

In PrefixSpan, it supposes that items within elements are in alphabetical order because the item order within an element does not affect the sequential mining. The first step of PrefixSpan is to scan the sequential database to get the length-1 patterns, which is in fact the large 1-itemsets. Then the sequential database is divided into different partitions according the number of length-1 sequence. Each partition is the projection of the sequential database that take the corresponding length-1 sequences as prefix. For example, with the frequent 1-sequence as the prefix, the projected database is shown in Figure 2.10 (b). The projected databases only contain the suffix of these sequences, by scanning the projected database all the length-2 sequential patterns that have the parent length-1 sequential patterns as prefix can be generated. Then the projected database is partitioned again by those length-2 sequential patterns. The same process are executed recursively until the projected database is empty or no more frequent length-k sequential patterns can be generated.

[1] However,we find that PrefixSpan also needs to test the candidates, which are existing in the projected database. We explain this issue in Chapter 3.

| Customer ID | Customer Sequence |
|---|---|
| 1 | <ac(bc)d(abc)ad> |
| 2 | <b(cd)ac(bd)> |
| 3 | <d(bc)(ac)(cd)> |

(a) Example Database

| Large Itemsets | Projected Database |
|---|---|
| a | <c(bc)d(abc)ad><br><c(bd)><br><(_c)(cd)> |
| b | <(_c)d(abc)ad><br><(cd)ac(bd)><br><(_c)(ac)(cd)> |
| c | <c(bc)d(abc)ad><br><(_d)ac(bd)><br><(ac)(cd)> |
| d | <(abc)ad><br><ac(bd)><br><(bc)(ac)(cd)> |

(b) Projected Database

| | | | | |
|---|---|---|---|---|
| <a> | 0 | | | |
| <b> | (3 2 1) | 0 | | |
| <c> | (3 3 2) | (2 3 2) | 0 | |
| <d> | (3 3 0) | (3 3 1) | (3 3 2) | 0 |
| | <a> | <b> | <c> | <d> |

(c) The S-matrix

Figure 2.10: PrefixSpan Mining Process

The main cost of the above method is the time and space used to construct and scan the projected databases as shown in Figure 2.10 (b). This is called level-by-level projection. Another projection method is called bi-level projection, which aims to reduce the number and size of projected databases [1]. The first step is the same, by scanning the sequential database we can get the frequent 1-sequence. In the second step, instead of constructing projected database, a n×n triangle matrix M is constructed, as shown in Figure 2.10 (c). The matrix represents all the supports of length-2 sequences. For

[1]As indicated by the same authors of PrefixSpan in their extension paper [ 79], this bi-level projection is useless to improve the efficiency. We introduce it here to show the original idea of PrefixSpan.

example M[$\langle d \rangle, \langle a \rangle$]=(3, 3, 0) means supports of $\langle da \rangle, \langle ad \rangle$ and $\langle (ad) \rangle$ are 3, 3 and 0, respectively. After that the S-matrix projected databases are constructed for those frequent length-2 sequences. All the processes iterate until the projected database becomes empty or no frequent sequence can be found. By using the triangle S-matrix to represent all supports of length-2 sequences, the number of projected databases becomes smaller and hence it requires less space.

To make the projection more efficient, the authors proposed pseudo projection when the projected database can be fitted in main memory. Actually no physical projection database is constructed. Each suffix is represent by a pair of pointer and offset value. By avoiding to copy the database, pseudo projection is more efficient than the other two projection methods. However the limitation is that the size of the database must can be fitted into the main memory.

The main cost of PrefixSpan is the projected database scanning process. In order to improve the performance a bi-level projection method that uses the triangle S-Matrix is introduced. The main problem of PrefixSpan, is the time consuming on scanning the projected database, which may be very large if the original dataset is huge.

# Chapter 3

# LAPIN: Efficient Sequential Mining Algorithms

Although much work has been carried out on mining sequential patterns, as for example, in [5, 7, 79, 88, 109], all of these works suffer from the problems of having a large search space and the ineffectiveness in handling dense data sets, i.e., biological data. In this thesis, we aim to propose new strategies to reduce the space necessary to be searched, by which improving the efficiency of the sequential pattern mining . We first analyze the state-of-the-art algorithms to delve into the issue, and then new techniques will be introduced to tackle the drawbacks of the existing strategies.

## 3.1 Problem of Existing Algorithms

The issue of sequential pattern mining has been introduced more than ten years and many efficient algorithms have been proposed [5, 7, 79, 88, 109]. The state-of-the-art techniques have demonstrated that the performance has been improved by an order of magnitude or more [41, 79, 109]. However, the cost of the sequential pattern mining problem, due to its intrinsic complexity, is still unsatisfied especially for large dense datasets. We first illustrate the existing algorithms performance by conducting on varies kinds of datasets in Section 3.1.1, and then analyze the reasons in Section 3.1.2.

### 3.1.1 Experiment of Comparing Existing Algorithms

To fairly evaluate the performance of the state-of-the-art algorithms, we downloaded the program implementations from their authors' website [1]. From the comprehensive experiments what have been done, several phenomena were observed, which are different from the traditional standpoint and will be explained shortly.

At first two dataset (C10T5S5I5N100D1K and C30T20S30I20N200D20K) were used to evaluate, one is relative small (C10T5S5I5N100D1K) and the other is relative large (C30T20S30I20N200D20K). Figure 3.1 (a) shows the execution comparison result and Figure 3.1 (b) illustrates the memory usage comparison result. There are some observations:

- We find that SPADE is better than PrefixSpan on some kinds of datasets (i.e., the relative large dataset C30T20S30I20N200D20K). This is contradictory to traditional opinion that PrefixSpan is always faster than SPADE [79]. The reason will be explained shortly and more experiments will be shown in latter part of this thesis.

- SPAM is slower than the other two algorithms on some kinds of datasets (i.e., C10T5S5I5N100D1K) and fails on running some large datasets (i.e., C30T20S30 I20N200D20K). This confirms the statement of the authors in their paper [7]. To evaluate the efficiency of SPAM furthermore, we show another experiment result, as Figure 3.2 illustrates. It proves that SPAM can be faster than SPADE and PrefixSpan about an order of magnitude. However, this is an trade-off between time and space. In fact, SPAM fails on running some reasonable size of datasets.

In the next section, I will explain the reasons for the above phenomena.

### 3.1.2 Analysis

We have found that SPADE and PrefixSpan have their own advantages for different types of data sets. Suppose that we have two sequence databases, as shown in Figure

[1] PrefixSpan: http://illimine.cs.uiuc.edu/
SPADE: http://www.cs.rpi.edu/ zaki/software/
SPAM: http://himalaya-tools.sourceforge.net/Spam/

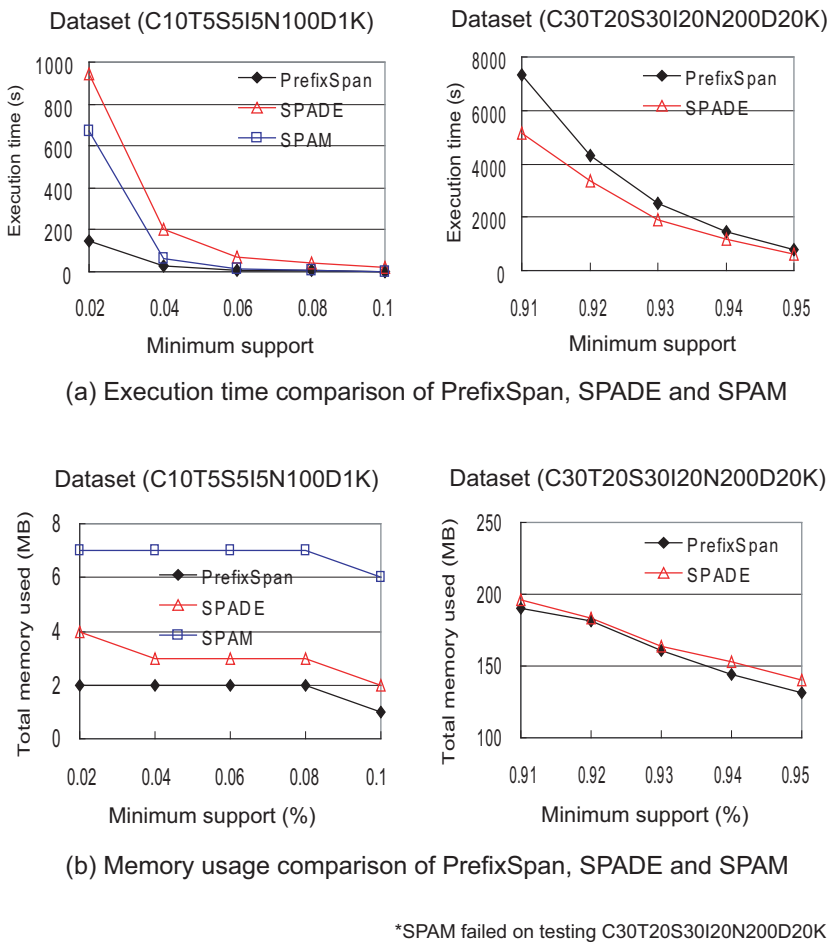



Figure 3.1: Performance comparison of existing state-of-the-art algorithms on different datasets

3.3 (a), the prefix sequence is $a$, and the min_support = 1. To test the 2-length candidate sequences, whose prefix is $a$ for DB (i), the PrefixSpan algorithm scans the projected DB, which requires a $1 \times 5 = 5$ scanning time. The SPADE algorithm scans the local candidate item list for each sequence, which requires a $5 \times 5 = 25$ scanning time. However, for DB (ii), suppose we want to grow from $\langle aa \rangle$ to longer patterns. PrefixSpan algorithm requires a 4 scanning time (because there are four items, $b$, $c$, $d$, $e$ in the projected DB of $\langle aa \rangle$), and the SPADE algorithm requires a 0 scanning time (because only one candidate item, $a$, in the local list, and no need to join). The effect of these two data sets on the two approaches is shown in Figure 3.3 (b).

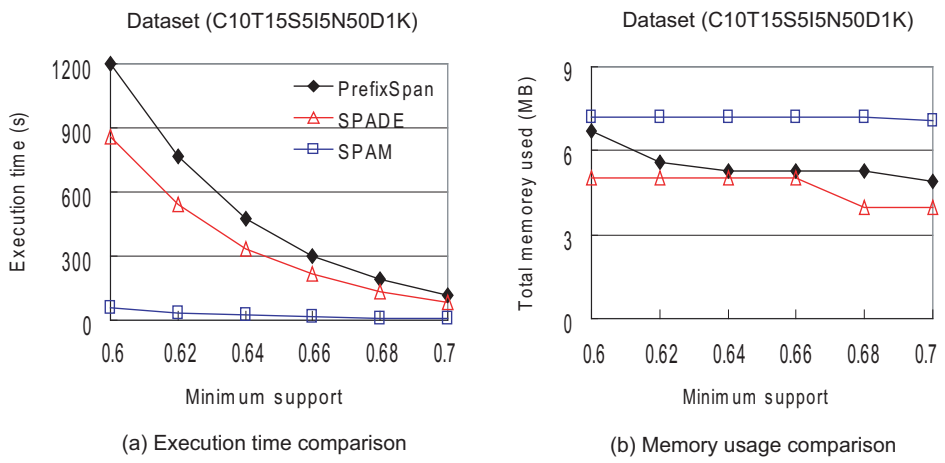



Figure 3.2: One example of SPAM outperforming other existing algorithms

SPADE belongs to $LCI$-$oriented$ category of algorithms, because the candidates are come from the local candidate item list. PrefixSpan belongs to $Suffix$-$oriented$ category of algorithms because the candidates are come from the suffix of the dataset. The above example illustrates that, if the average suffix sequence length is less than the average element length for those items in the local candidate list, as in DB (i), then $Suffix$-$oriented$ spends less time. However, if the average suffix sequence length is larger than the average element length for those items in the local candidate list, as in DB (ii), then $LCI$-$oriented$ is faster. In summary, which is different from traditional opinions that $Suffix$-$oriented$ algorithm (i.e., PrefixSpan) is always better than $LCI$-$oriented$ algorithm (i.e., SPADE), we believe that the two kinds of algorithms have their own advantages and disadvantages with regard to different datasets. The reason that PrefixSpan is worse than SPADE is due to the useless of scanning those items which are already not frequent in the projected DBs (i.e., $b$, $c$, $d$, and $e$ in DB (ii) as shown in Figure 3.3 (a)). In other words, PrefixSpan can not fully utilize the Apriori heuristic because the intrinsic difference of the two algorithms.

Based on the discovery from these experiments and evaluation, we decided to develop two kinds of algorithms based on $LCI$-$oriented$ and $Suffix$-$oriented$, respectively.

SPAM is a time efficiency improved version of SPADE, with the trade-off on the memory usage, because bitmap strategy can largely improve the efficiency by using

DB(i)

| CID | Seq. |
|---|---|
| 10 | a a |
| 20 | a b |
| 30 | a c |
| 40 | a d |
| 50 | a e |

DB(ii)

| CID | Seq. |
|---|---|
| 10 | a a b c d e |
| 20 | b c d e a a |

(a) Two special DBs

| | Avg. suffix | Avg. local cand. item list | Suffix-oriented | LCI-oriented |
|---|---|---|---|---|
| DB (i) | 1 | 5 | 5 times | 25 times |
| DB (ii) | 2 | 1 | 4 times | 0 times |

(b) Effect on different type of DBs

Figure 3.3: Performance of Suffix-oriented and LCI-oriented algorithms on different DB

bitwise operations. We believe that in an resource unlimited environment (i.e., huge memory available) SPAM can outperform others by about an order of magnitude on some kinds of datasets. Hence, we decided to develop one algorithm based on SPAM.

The comprehensive summary of existing algorithms will be introduced in Section 4.3.5.

## 3.2 LAPIN Algorithms (Last Position Induction Frequent Pattern Mining)

We have found that the last position of an item is very important to judge whether a k-length pattern could grow to a (k+1)-length pattern. With this finding, we only need to search a small portion of the database by recording the last position of each item in each sequence. Because support counting is usually the most costly step in sequential pattern mining, the LAst Position INduction (LAPIN) technique can improve the performance greatly by avoiding cost scanning and comparisons.

### 3.2.1 General Idea

**Discovering (k+1)-length frequent patterns.** For any time series database, the last position of an item is the key used to judge whether or not the item can be appended to a given prefix (k-length) sequence (assumed to be $s$). For example, in a sequence, if

Table 3.1: Sequence Database

| SID | Sequence |
|---|---|
| 10 | $ac(bc)d(abc)ad$ |
| 20 | $b(cd)ac(bd)$ |
| 30 | $d(bc)(ac)(cd)$ |

Table 3.2: $SE$ Item Last Position List

| SID | Last Position of $SE$ Item |
|---|---|
| 10 | $b_{last} \overset{\downarrow}{=} 5$ $c_{last} = 5$ $a_{last} = 6$ $d_{last} = 7$ |
| 20 | $a_{last} = 3$ $c_{last} \overset{\downarrow}{=} 4$ $b_{last} = 5$ $d_{last} = 5$ |
| 30 | $b_{last} = 2$ $a_{last} = 3$ $c_{last} \overset{\downarrow}{=} 4$ $d_{last} = 4$ |

the last position of item $\alpha$ is smaller than, or equal to, the position of the last item in $s$, then item $\alpha$ cannot be appended to $s$ as a (k+1)-length sequence extension in the same sequence. Similarly to $S$-$Step$, in $I$-$Step$, if the last position of the 2-length itemset extension sequence, $\theta$, (whose first item is the same as the last item of $s$) is smaller than the position of the last item in $s$, then $\theta$ cannot be appended to $s$ as an itemset extension in this sequence.

**Example 3.1**. Let our running database be the sequence database $S$ shown in Table 3.1 with min_support = 2. We will use this sample database hereafter in this thesis. The set of items in the database is {a,b,c,d}. The length of the second sequence is equal to 7. A 2-sequence $\langle ac \rangle$ is contained in the sequence 10, 20, and 30, respectively, and its support is equal to 3. Therefore, $\langle ac \rangle$ is a frequent pattern.

**Example 3.2**. When scanning the database in Table 3.1 for the first time, we obtain Table 3.8, which is a list of the last positions of the 1-length frequent sequences in ascending order. At the same time, we can obtain Table 3.9, which is a list of the last positions of the frequent 2-length $IE$ sequences in ascending order. Suppose that we have a prefix frequent sequence $\langle a \rangle$, and its positions in Table 3.1 are 10:1, 20:3, 30:3, where sid:eid represents the sequence ID and the element ID. Then, we check

Table 3.3: $IE$ Item Last Position List

| SID | Last Position of $IE$ Item |
|---|---|
| 10 | $(ab)_{last} \overset{\downarrow}{=} 5$ $(ac)_{last} = 5$ $(bc)_{last} = 5$ |
| 20 | $(cd)_{last} = 2$ $(bd)_{last} \overset{\downarrow}{=} 5$ |
| 30 | $(bc)_{last} = 2$ $(ac)_{last} \overset{\downarrow}{=} 3$ $(cd)_{last} = 4$ |

Table 3.8 to obtain the first indices whose positions are larger than $\langle a \rangle$'s, resulting in 10:1, 20:2, 30:3, i.e., (10:b$_{last}$ = 5, 20:c$_{last}$ = 4, and 30:c$_{last}$ = 4), symbolized as "$\downarrow$". We start from these indices to the end of each sequence, and increment the support of each passed item, resulting in $\langle a \rangle : 1, \langle b \rangle : 2, \langle c \rangle : 3,$ and $\langle d \rangle : 3$, from which, we can determine that $\langle ab \rangle$, $\langle ac \rangle$ and $\langle ad \rangle$ are the frequent patterns. In our implementation, we constructed a mapping table for a specific position to the corresponding index of the item-last-position list, thus avoiding searching in each iteration. The I-Step methodology is similar to the S-Step methodology, with the only difference being that, when constructing the mapping table, I-Step maps the specific position to the index whose position is equal to or larger than the position in Table 3.9. To determine the itemset extension pattern of the prefix sequence $\langle a \rangle$, we obtain its mapped indices in Table 3.9, which are 10:1, 20:2, and 30:2. Then, we start from these indices to the end of each sequence, and increment the support of each passed item, resulting in $\langle (ab) \rangle : 1$, and $\langle (ac) \rangle : 2$. We can also obtain the support of the 3-length sequences $\langle a(bc) \rangle : 1, \langle a(bd) \rangle : 1,$ and $\langle a(cd) \rangle : 1$, which is similar to the bi-level strategy of PrefixSpan, but we avoid scanning the entire projected database.

From the above example, we can show that the main difference between LAPIN and previous works is the scope of the search space. PrefixSpan scans the entire projected database to find the frequent pattern. SPADE temporally joins the entire ID-List of the candidates to obtain the frequent pattern of next layer. LAPIN can obtain the same result by scanning only part of the search space of PrefixSpan and SPADE, which indeed, are the last positions of the items. Table 3.4 shows the search space of LAPIN based on Table 3.1 ($S$-$Step$). We can avoid scanning the $*$ part in the projected database or in the ID-List. Let $\bar{D}$ be the average number of customers (i.e., sequences) in the projected DB, $\bar{L}$ be the average sequence length in the projected DB, $\bar{N}$ be the average total number of the distinct items in the projected DB, and $m$ be the distinct

Table 3.4: Last Position of DB (S-Step)

| SID | Sequence |
|---|---|
| 10 | $**(**)*(*bc)ad$ |
| 20 | $*(**)ac(bd)$ |
| 30 | $*(b*)(a*)(cd)$ |

item recurrence rate or density in the projected DB. Then $m$=$\bar{L}/\bar{N}$ ($m \geq 1$), and the relationship between the runtime of PrefixSpan ($T_{ps}$) and the runtime of LAPIN ($T_{lapin}$) in the support counting part is

$$T_{ps}/T_{lapin} = (\bar{D} \times \bar{L})/(\bar{D} \times \bar{N}) = m \qquad (1).$$

Because support counting is usually the most costly step in the entire mining process, Eq.(1) illustrates the main reason why LAPIN is faster than PrefixSpan for dense data sets, whose $m$ (density) can be very high. For example, suppose we have a special data set, which has only one single long sequence with one distinct item $a$ and the sequence length is 100. The total time used to scan the projected databases in PrefixSpan is $100 + 99 + 98 + 97 + \ldots + 1$=5050. However, LAPIN only needs $100 + 1 + 1 + \ldots + 1$=199 scanning time. Hence, we have $m$=5050/199≈25. From this example, we know that scanning most of the duplicate items in the projected DB is useless and time consuming.

#### 3.2.1.1 Lexicographic Tree

We used a lexicographic tree [7] [13] as the search path of our algorithm. Furthermore, we followed a lexicographic order, which adopts the Depth First Search (DFS) strategy. Figure 3.4 shows an example of the lexicographic tree used. We obeyed the following rules based on DFS:

(a) If $\gamma' = \gamma \diamond \theta$, then $\gamma < \gamma'$; (Search the prefix first, then the sequence. For example, we first search $\langle a \rangle$, then $\langle (ab) \rangle$.)

(b) If $\gamma = \alpha \diamond_s \theta$ and $\gamma' = \alpha \diamond_i \theta$, then $\gamma < \gamma'$; (Search the sequence-extension first, then the itemset-extension. For example, we first search $\langle ab \rangle$, then $\langle (ab) \rangle$.)

(c) If $\gamma = \alpha \diamond \theta$ and $\gamma' = \alpha \diamond \theta', \theta < \theta'$ indicates $\gamma < \gamma'$. (For two sequences

that have the same prefix, search them based on the alphabetic order of the suffix. For example, we first search $\langle aa \rangle$, then $\langle ab \rangle$.)

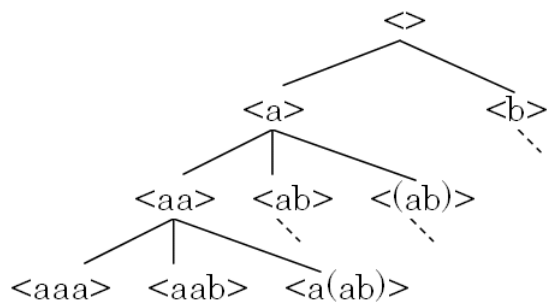



Figure 3.4: Lexicographic Tree

#### 3.2.1.2 Formulation

LAPIN follows the pattern growth strategy. To grow to (k+1)-length sequence, LAPIN first gets the knowledge of k-length prefix frequent pattern. For the sake of simplicity, the definitions and lemmas involved with $S$-$Step$ are introduced in this section.

**Definition 1** (Prefix border position set)**.** *Given two sequences, A=$\langle A_1 A_2 \ldots A_m \rangle$ and B=$\langle B_1 B_2 \ldots B_n \rangle$, suppose that there exists C=$\langle C_1 C_2 \ldots C_l \rangle$ for $l \leq m$ and $l \leq n$, and that C is a common prefix for A and B. We record both positions of the last item $C_l$ in A and B, respectively, e.g., $C_l$=$A_i$ and $C_l$=$B_j$. The position set, (i, j), is called the $prefix\ border\ position\ set$ of the common prefix C, denoted as $\mathcal{S}_c$. Furthermore, we denote $\mathcal{S}_{c,i}$ as the prefix border position of the sequence, i.*

For instance, if A=$\langle abc \rangle$ and B=$\langle acde \rangle$, then we can deduce that one common prefix of these two sequences is $\langle ac \rangle$, whose prefix border position set is (3,2), which is the last item c's positions in A and B. To get the prefix border position, PrefixSpan needs $O(\bar{D} \times \bar{L})$ time (see Section 1.3), because it uses pseudo projection in sequential search order, while LAPIN applies the binary search for vertical representation of database, as will be explained in Section 2.2, whose time complexity is $O(\bar{D} \times log(\bar{L}))$.

To test (k+1)-length candidate sequences, PrefixSpan scans in the prefix k-length frequent pattern's projected database. In contrast, LAPIN searches in the prefix k-length frequent pattern's projected item-last-position list.

**Definition 2** (Item-last-position list)**.** *Given two sequences, A=$\langle A_1 A_2 \ldots A_m \rangle$ and B=$\langle B_1 B_2 \ldots B_n \rangle$, the list of the last positions of the different frequent 1-length items in ascending order (or if the same, based on alphabetic order) for these two sequences is called the item-last-position list, denoted as $L$. Furthermore, we denote $L_n$ as the $item$-$last$-$position\ list$ of the sequence, n. Each node of $L_n$ is associated with two values, i.e., an item and an element number (denoted as $D_n.item$ and $D_n.num$ for $D_n \in L_n$)*

**Definition 3** (Candidate border index set)**.** *Given two sequences, A=$\langle A_1 A_2 \ldots A_m \rangle$ and B=$\langle B_1 B_2 \ldots B_n \rangle$, suppose that there exists C=$\langle C_1 C_2 \ldots C_l \rangle$ for $l \leq m$ and $l \leq n$, and that C is a common prefix for A and B. Then we have the prefix border position set $S_{C,i}$ and the item-last-position list $L_i$ for customer sequence i. We denote the index $CanI_{C,i}$, which points to the node $D_i$, as the candidate border index of customer sequence i, and the index set $CanI_C$ as the candidate border index set of the common prefix C, if the following conditions hold:*

*(a)* $D_i \in L_i$

*(b)* $D_i.num > S_{C,i}$

*(c)* $\forall E_i \in L_i$ *before* $D_i, E_i.num <= S_{C,i}$

For instance, for the example DB in Table 3.1, if we have the prefix sequence which is $\langle a \rangle$, then we can get its $candidate\ border\ index\ set$, 10:1, 20:2, 30:3, symbolized as "↓" in Table 3.8.

**Definition 4** (Projected item-last-position list)**.** *Let C be a sequential pattern in a sequence database S. The C-projected item-last-position list, denoted as $P|_C$, is the collection of suffixes in the item-last-position list with regards to prefix C.*

For instance, in Table 3.8, the part behind the candidate border index (↓) (including the element to which the index points) of each user sequence, is the projected item-last-position list of the corresponding prefix $\langle a \rangle$.

**Definition 5** (Support counting)**.** *Let C be a sequential pattern in sequence database S, and A be a sequence with prefix C. The support counting of A in C-projected item-last-position list $P|_C$, denoted as $support_{P|_C}(A)$, is the number of sequences $\alpha$ in $P|_C$ such that A $\sqsubseteq$ C $\cdot\alpha$.*

The relationship between the runtime of PrefixSpan and the runtime of LAPIN has been described as Eq.(1) in Section 1.3, which is decided by the density of the

projected DB, $m$. The worst case of LAPIN is that when there is no duplicate item existing in the database (i.e., association rule mining), which means that the time used in searching the projected item-last-position list is the same as that used in searching the projected DB. However, for common data sets, especially for dense database such as DNA sequence, $m$ is probably very large. Hence, searching in projected item-last-position list is much more efficient than searching in projected database. Nevertheless, here we have a question that, is the result got by searching in the projected DB the same as searching in the projected item-last-position list? The answer can be got from the following two Lemmas.

**Lemma 1** (Projected item-last-position list)**.** *Let A and C be two sequential patterns in a sequence database S such that C is a prefix of A.*

1. $P|_A = (P|_C)|_A$, *and*
2. *for any sequence B with prefix C,* $support_{P(B)}$ = $support_{P|_C}(B)$.

*Proof.* The proof of the Lemma 1 is similar to the proof in [79] (Lemma 3.2). The first part of the lemma follows the fact that, for a sequence $B$, the suffix of $B$ with regards to $A$, $B/A$, equals to the sequence resulted from first doing projection of $B$ with regards to $C$, i.e., $B/C$, and then doing projection $B/C$ with regards to A. That is $B/A = (B/C)/A$. The second part of the lemma states that to collect support count of a sequence B, only the sequences in the database sharing the same prefix should be considered. Furthermore, only those suffixes with the prefix being a super-sequence of B should be counted. □

**Lemma 2** (Support counting equivalency)**.** *Let C be a sequential pattern in sequence database S, then support counting in C-projected item-last-position list gets the same result as support counting in C-projected database.*

*Proof.* The only difference between C-projected item-last-position list (Definition 4) and C-projected database is that the former records the last position of different items, and the latter records all positions list of different items. From the support definition, duplicate appearance in the same customer sequence does not contribute to the support counting, which means that the additional information stored in C-projected database is useless. Hence, support counting in C-projected item-last-position list gets the same result as support counting in C-projected database. □

Note that the definitions and lemmas involved with $I$-$Step$ can be presented in a similar way, as done in this section.

### 3.2.2 LAPIN: Design and Implementation

In this section, we describe the LAPIN algorithms in detail. We use a lexicographic tree [7] as the search path of LAPIN and adopt a lexicographic order [7], which employs the Depth First Search (DFS) strategy. The pseudo code of LAPIN is shown in Table 3.5.

As in other algorithms, certain key strategies were adopted, i.e., candidate sequence pruning, database partitioning, and customer sequence reducing. Combined with the LAPIN strategy, our algorithms can efficiently find the complete set of frequent patterns.

In Step 1, by scanning the DB once, we obtain the $SE$ position list table, as in Table 3.6 and all the 1-length frequent patterns. Based on the last element in each position list, we sort and construct the $SE$ $item$-$last$-$position$ $list$ in ascending order, as shown in Table 3.8. To find the frequent 2-length $IE$ sequences, during the first scan, we construct a 2-dimensional array indexed by the items' ID and update the counts for the corresponding 2-length $IE$ sequences by using similar methods to those used in [109]. Then, we merge the $SE$ position lists of the two items, which compose the frequent 2-length $IE$ sequence, to obtain the 2-length $IE$ sequence position list. Finally, we sort and construct the $IE$ $item$-$last$-$position$ $list$ of each frequent 2-length $IE$ sequence in ascending order, as shown in Table 3.9. As Example 3.2 in Section 3.2.1 shows, the I-Step methodology is similar to the S-Step methodology in LAPIN. We will first describe the S-Step process, and the I-Step process will be explained in Section 2.2.3.

In function $Gen\_Pattern$, to find the prefix border position set of k-length $\alpha$ (Step 4), we first obtain the position list of the last item of $\alpha$, and then perform a binary search in the list for the (k-1)-length prefix border position. For $S$-$Step$, we look for the first position that is larger than the (k-1)-length prefix border position.

Step 5, shown in Figure 3.5, is used to find the frequent $SE$ (k+1)-length pattern based on the frequent k-length pattern and the 1-length candidate items. Step 5 can be justified based on Lemma 1 and Lemma 2 in Section 2.1. Commonly, support counting is the most time consuming part in the entire mining process. Here, we face a

Table 3.5: LAPIN Algorithm pseudo code

| | |
|---|---|
| INPUT: | A sequence database, and the minimum support threshold, $\varepsilon$ |
| OUTPUT: | The complete set of sequential patterns |
| Function: | Gen_Pattern($\alpha$, $S$, $CanI_s$, $CanI_i$ ) |
| Parameters: | $\alpha$ = length k frequent sequential pattern; $S$ = prefix border position set of (k-1)-length sequential pattern; $CanI_s$ = candidate sequence extension item list of (k+1)-length sequential pattern; $CanI_i$ = candidate itemset extension item list of (k+1)-length sequential pattern |
| Goal: | Generate (k+1)-length frequent sequential pattern |
| | |
| Main(): | |
| 1. | Scan DB once to do: |
| 1.1 | $P_s \leftarrow$ Create the position list representation of the 1-length $SE$ sequences |
| 1.2 | $B_s \leftarrow$ Find the frequent 1-length $SE$ sequences |
| 1.3 | $L_s \leftarrow$ Obtain the item-last-position list of the 1-length $SE$ sequences |
| 1.4 | $B_i \leftarrow$ Find the frequent 2-length $IE$ sequences |
| 1.5 | $P_i \leftarrow$ Construct the position lists of the frequent 2-length $IE$ sequences |
| 1.6 | $L_i \leftarrow$ Obtain the item-last-position list of the frequent 2-length $IE$ sequences |
| 2. | For each frequent $SE$ sequence $\alpha_s$ in $B_s$ |
| 2.1 | Call Gen_Pattern ($\alpha_s$, 0, $B_s$, $B_i$) |
| 3. | For each frequent $IE$ sequence $\alpha_i$ in $B_i$ |
| 3.1 | Call Gen_Pattern ($\alpha_i$, 0, $B_s$, $B_i$) |
| | |
| Function: | $Gen\_Pattern(\alpha, S, CanI_s, CanI_i)$ |
| 4. | $S_\alpha \leftarrow$ Find the prefix border position set of $\alpha$ based on $S$ |
| 5. | $FreItem_{s,\alpha} \leftarrow$ Obtain the $SE$ item list of $\alpha$ based on $CanI_s$ and $S_\alpha$ |
| 6. | $FreItem_{i,\alpha} \leftarrow$ Obtain the $IE$ item list of $\alpha$ based on $CanI_i$ and $S_\alpha$ |
| 7. | For each item $\gamma_s$ in $FreItem_{s,\alpha}$ |
| 7.1 | Combine $\alpha$ and $\gamma_s$ as $SE$, results in $\theta$ and output |
| 7.2 | Call Gen_Pattern ($\theta$, $S_\alpha$, $FreItem_{s,\alpha}$, $FreItem_{i,\alpha}$) |
| 8. | For each item $\gamma_i$ in $FreItem_{i,\alpha}$ |
| 8.1 | Combine $\alpha$ and $\gamma_i$ as $IE$, results in $\eta$ and output |
| 8.2 | Call Gen_Pattern ($\eta$, $S_\alpha$, $FreItem_{s,\alpha}$, $FreItem_{i,\alpha}$) |

Table 3.6: $SE$ Position List of DB

| SID | $Item\ \ Positions$ |
|---|---|
| 10 | $a : 1 \rightarrow 5 \rightarrow 6 \rightarrow null$ |
| | $b : 3 \rightarrow 5 \rightarrow null$ |
| | $c : 2 \rightarrow 3 \rightarrow 5 \rightarrow null$ |
| | $d : 4 \rightarrow 7 \rightarrow null$ |
| 20 | $a : 3 \rightarrow null$ |
| | $b : 1 \rightarrow 5 \rightarrow null$ |
| | $c : 2 \rightarrow 4 \rightarrow null$ |
| | $d : 2 \rightarrow 5 \rightarrow null$ |
| 30 | $a : 3 \rightarrow null$ |
| | $b : 2 \rightarrow null$ |
| | $c : 2 \rightarrow 3 \rightarrow 4 \rightarrow null$ |
| | $d : 1 \rightarrow 4 \rightarrow null$ |

problem. ”Where do the appended 1-length candidate items come from?” We can test each candidate item in the local candidate item list ($LCI$-$oriented$), which is similar to the method used in SPADE [109]. Another choice is to test the candidate item in the projected DB, just as PrefixSpan [79] does ($Suffix$-$oriented$). The correctness of these methods was discussed in [109] and [79], respectively.

We have found that $LCI$-$oriented$ and $Suffix$-$oriented$ have their own advantages for different types of data sets, as described in Section 3.1. Based on this discovery, we formed a series of algorithms categorized into two classes. One class was $LCI$-$oriented$, LAPIN_LCI, and the other class was $Suffix$-$oriented$, LAPIN _Suffix. We can dynamically compare the suffix sequence length with the local candidate item list size and select the appropriate search space to build a single general framework. However, because we used a space consuming bitmap strategy in LAPIN_LCI, which will be explained in Section 2.2.1, in order to save memory space and clarify the advantages and disadvantages of each method, we deconstructed the general framework into two approaches. Nevertheless, it is easy to combine these two approaches, and evaluate the efficiency of the entire general framework, whose runtime, $T$, and maximum memory space required, $M$, are

Table 3.7: Finding the SE frequent patterns using LAPIN_Suffix

| | |
|---|---|
| INPUT: | $S_\alpha$ = prefix border position set of length k frequent sequential pattern $\alpha$; |
| | $L_s$ = $SE$ item-last-position list; $\varepsilon$ = user specified minimum support |
| OUTPUT: | $FreItem_s$ = local frequent $SE$ item list |
| 1. | For each sequence, F |
| 2. | $S_{\alpha,F} \leftarrow$ obtain prefix border position of F in $S_\alpha$ |
| 3. | $L_{s,F} \leftarrow$ obtain $SE$ item-last-position list of F in $L_s$ |
| 4. | M = Find the corresponding index for $S_{\alpha,F}$ |
| 5. | while ( M $<$ L$_{s,F}$.size) |
| 6. | Suplist[M.item]++; |
| 7. | M++; |
| 8. | For each item $\beta$ in Suplist |
| 9. | If ( Suplist[$\beta$] $\geq \varepsilon$) |
| 10. | $FreItem_s$.insert($\beta$); |

$$T \approx \{T_{LAPIN\_LCI}, T_{LAPIN\_Suffix}\}_{min}$$
$$M \approx \{M_{LAPIN\_LCI}, M_{LAPIN\_Suffix}\}_{max}.$$

### 3.2.3 LAPIN_Suffix

When the average size of the candidate item list is larger than the average size of the suffix, then scanning in the suffix to count the support of the (k+1)-length sequences is better than scanning in the local candidate item list, such as for DB (i) in Figure 3.3. Therefore, we proposed a new algorithm, LAPIN_Suffix. In the *item-last-position list*, i.e., Table 3.8, we look for the first element whose last position is larger than the prefix border position. Then, we go to the end of this list and increment each passed item's support. Obviously, we only pass and count once for each different item in the suffix (projected database). In contrast, PrefixSpan needs to pass every item in the projected database regardless of whether or not they are the same as before. The pseudo code of LAPIN_Suffix is shown in Table 3.7.

**Example 3.3**. When scanning the database in Table 3.1 for the first time, we obtain Table 3.8, which is a list of the last positions of the 1-length frequent sequences in ascending order. At the same time, we can obtain Table 3.9, which is a list of the

Table 3.8: $SE$ Item Last Position List

| SID | Last Position of $SE$ Item |
|---|---|
| 10 | $b_{last} \overset{\downarrow}{=} 5$ $c_{last} = 5$ $a_{last} = 6$ $d_{last} = 7$ |
| 20 | $a_{last} = 3$ $c_{last} \overset{\downarrow}{=} 4$ $b_{last} = 5$ $d_{last} = 5$ |
| 30 | $b_{last} = 2$ $a_{last} = 3$ $c_{last} \overset{\downarrow}{=} 4$ $d_{last} = 4$ |

Table 3.9: $IE$ Item Last Position List

| SID | Last Position of $IE$ Item |
|---|---|
| 10 | $(ab)_{last} \overset{\downarrow}{=} 5$ $(ac)_{last} = 5$ $(bc)_{last} = 5$ |
| 20 | $(cd)_{last} = 2$ $(bd)_{last} \overset{\downarrow}{=} 5$ |
| 30 | $(bc)_{last} = 2$ $(ac)_{last} \overset{\downarrow}{=} 3$ $(cd)_{last} = 4$ |

last positions of the frequent 2-length $IE$ sequences in ascending order. Suppose that we have a prefix frequent sequence $\langle a \rangle$, and its positions in Table 3.1 are 10:1, 20:3, 30:3, where sid:eid represents the sequence ID and the element ID. Then, we check Table 3.8 to obtain the first indices whose positions are larger than $\langle a \rangle$'s, resulting in 10:1, 20:2, 30:3, i.e., (10:b$_{last}$ = 5, 20:c$_{last}$ = 4, and 30:c$_{last}$ = 4), symbolized as "$\downarrow$". We start from these indices to the end of each sequence, and increment the support of each passed item, resulting in $\langle a \rangle : 1, \langle b \rangle : 2, \langle c \rangle : 3,$ and $\langle d \rangle : 3$, from which, we can determine that $\langle ab \rangle$, $\langle ac \rangle$ and $\langle ad \rangle$ are the frequent patterns. In our implementation, we constructed a mapping table for a specific position to the corresponding index of the item-last-position list, thus avoiding searching in each iteration. The I-Step methodology is similar to the S-Step methodology, with the only difference being that, when constructing the mapping table, I-Step maps the specific position to the index whose position is equal to or larger than the position in Table 3.9. To determine the itemset extension pattern of the prefix sequence $\langle a \rangle$, we obtain its mapped indices in Table 3.9, which are 10:1, 20:2, and 30:2. Then, we start from these indices to the end of each sequence, and increment the support of each passed item, resulting in $\langle (ab) \rangle : 1$, and $\langle (ac) \rangle : 2$. We can also obtain the support of the 3-length sequences $\langle a(bc) \rangle : 1, \langle a(bd) \rangle : 1,$ and $\langle a(cd) \rangle : 1$, which is similar to the bi-level strategy of PrefixSpan, but we avoid scanning the entire projected database.

### 3.2.4 LAPIN_LCI

LAPIN_LCI tests each item which is in the local candidate item list. In each customer sequence, it directly judges whether an item can be appended to the prefix sequence or not by comparing this item's last position with the prefix border position. Increment the support value of the candidate item by 1 if the candidate item's last position is larger than the prefix border position. As an optimization, we use bitmap strategy to avoid such comparison process. A pre-constructed table, named ITEM_IS_EXIST_TABLE is constructed while first scanning to record the last position information. For example, Figure 3.5 (a), which is based on the example database shown in Table 1, shows one part of the ITEM_IS_EXIST_TABLE for the first sequence. The left-hand column denotes the position number and the top row is the item ID. In the table, we use a bit vector to represent all the 1-length frequent items existing for a specific position. If the bit value is unity, then it indicates that the corresponding item exists. Otherwise, the item does not exist. To accumulate the candidate sequence's support, we only need to check this table, and add the corresponding item's vector value, thus avoiding the comparison process.

This strategy is especially useful for those dense data sets with long patterns and a small number of items, such as DB (ii) in Figure 3.3. However, the comparison precess will consume a lot of time because the recursive property. Can we avoid such comparison and directly accumulate the candidates support?

We can avoid the comparison operations of LAPIN_LCI by using a pre-constructed table, named ITEM_IS_EXIST_TABLE. The last position information is recorded in a bit vector for each specific position. For example, Figure 3.5 (a), which is based on the example database shown in Table 1, shows one part of the ITEM_IS_EXIST_TABLE for the first sequence. The left-hand column denotes the position number and the top row is the item ID. In the table, we use a bit vector to represent all the 1-length frequent items existing for a specific position. If the bit value is unity, then it indicates that the corresponding item exists. Otherwise, the item does not exist. The bit vector size is equal to the size of the 1-length frequent items list. For example, when the current position is 5, we obtain the bit vector 1001, indicating that only items $a$ and $d$ exist in the same sequence after the current prefix. To accumulate the candidate sequence's support, we only need to check this table, and add the corresponding item's vector value, thus avoiding the comparison process.

| Pos \ Item | a | b | c | d |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 |
| 3 | 1 | 1 | 1 | 1 |
| 4 | 1 | 1 | 1 | 1 |
| 5 | 1 | 0 | 0 | 1 |
| 6 | 0 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 |

(a) ITEM_IS_EXIST_TABLE

| Pos \ Item | a | b | c | d |
|---|---|---|---|---|
| 5 | 1 | 0 | 0 | 1 |
| 6 | 0 | 0 | 0 | 1 |

(b) Optimized ITEM_IS_EXIST_TABLE

Figure 3.5: Bitmap representation table

It can be easily shown that the main memory used in LAPIN_LCI is no more than that used in SPAM [7], because we use each bit to represent each item's existence for each specific position in each sequence. However, this invokes a significantly high space cost if the original database is large. After consideration, we found that only part of the table was useful, and that most was not.

For example, in Figure 3.5 (a), Positions 5 and 6 are *key positions*, whereas the others are not. (Position 7 is not a *key position* because its bit vector is equal to zero).

**Space Optimization of LAPIN_LCI.** We found that only part of the table was useful, and that most was not. For example, in Figure 3.5 (a), when the prefix border position was smaller than five, then all the items exist, and when the position was larger than six, no items existed. Therefore, the useful information is stored in some key positions' lines, which indicate the last positions of the 1-length frequent items (except the last one). The optimized ITEM_IS_ EXIST_TABLE is shown in Figure 3.5 (b), which stores only two bit vectors instead of the seven shown in Figure 3.5 (a). For a dense data set, this space saving strategy proved more efficient. The pseudo code of LAPIN_LCI is shown in Table 3.10.

**Example 3.4.** Let us assume that we have obtained the prefix border position set of the pattern $\langle a \rangle$ in Table 3.1, i.e., (1,3,3). We also know that the *local candidate item list* is (a, b, c, d). Then, instead of comparing each last position of the candidate item with the prefix border position, we obtain the bit vector mapped from the specific position. Here, we obtain the bit vectors 1111, 0111, and 0011 with respect to

Table 3.10: Finding the SE frequent patterns using LAPIN_LCI

| | |
|---|---|
| INPUT: | $S_\alpha$ = prefix border position set of length k frequent sequential pattern $\alpha$; BV$_s$ = bit vectors of the ITEM_IS_EXIST_TABLE; $CanI_s$ = candidate sequence extension items; $\varepsilon$ = user specified minimum support |
| OUTPUT: | $FreItem_s$ = local frequent $SE$ item list |
| 1. | For each sequence, F |
| 2. | $S_{\alpha,F} \leftarrow$ obtain prefix border position of F in $S_\alpha$ |
| 3. | bitV $\leftarrow$ obtain the bit vector of the $S_{\alpha,F}$ indexed from BV$_s$ |
| 4. | For each item $\beta$ in $CanI_s$ |
| 5. | Suplist[$\beta$] = Suplist[$\beta$] + bitV[$\beta$]; |
| 6. | For each item $\gamma$ in Suplist |
| 7. | if (Suplist[$\gamma$] $\geq \varepsilon$) |
| 8. | $FreItem_s$.insert($\gamma$); |

the pattern $\langle a \rangle$'s prefix border position set, (1,3,3), and accumulate them, resulting in $\langle a \rangle : 1, \langle b \rangle : 2, \langle c \rangle : 3,$ and $\langle d \rangle : 3$. From here, we can deduce that $\langle ab \rangle$, $\langle ac \rangle$, and $\langle ad \rangle$ are frequent patterns.

**I-Step of LAPIN.** As Ayres et al. did in [7], the whole process of sequential pattern mining should includes two steps: a $sequence\text{-}extension\ step$ ($S\text{-}Step$) and a $itemset\text{-}extension\ step$ ($I\text{-}Step$).

In LAPIN, the $I\text{-}Step$ is similar to the $S\text{-}Step$. One difference is that in $I\text{-}Step$, the basic unit is 2-length itemset extension sequence, i.e., (ab), (bc), instead of 1-length sequence, i.e., a, b, in S-Step. From the step 1 of Figure 3.5, we can get the frequent 2-length $IE$ sequence position list as shown in Table 3.11 and the $I\text{-}Step\ item\text{-}last\text{-}position\ list$, as shown in Table 3.9. In the step 4 of Figure 3.5, we first get the position list of the last 2-length $IE$ item of $\alpha$, then do a binary search in Table 3.11. Here we look for the first position which is equal to or larger than the (k-1)-length prefix border position (in STL, the function is named $lower\_bound$). Note that when patterns grow, $I\text{-}Step$ should guarantee that the prefix sequences of the tested candidates are the same. We deal with it by joining the position list of each candidate $IE$ item after current prefix position, which is similar to the method used in SPADE [109]. This issue, however, has not been mentioned in the implementation of PrefixSpan algorithm [79].

Table 3.11: $IE$ Position List of DB

| SID | $Item\ Positions$ |
|---|---|
| 10 | $(ab) : 5 \rightarrow null$<br>$(ac) : 5 \rightarrow null$<br>$(bc) : 3 \rightarrow 5 \rightarrow null$ |
| 20 | $(bd) : 5 \rightarrow null$<br>$(cd) : 2 \rightarrow null$ |
| 30 | $(ac) : 3 \rightarrow null$<br>$(bc) : 2 \rightarrow null$<br>$(cd) : 4 \rightarrow null$ |

We think that the remedy should be the same as SPADE and LAPIN algorithms, which should pay for some time efficiency. To find the frequent (k+1)-length $IE$ sequences in the step 6 of Figure 3.5, similar to S-Step, we have two classes of algorithms, one is local-candidate-items oriented which directly compares the last positions of 2-length $IE$ sequence with the prefix border positions to judge whether the frequent k-length sequence could be appended with the 2-length $IE$ sequence, to be a (k+1)-length $IE$ sequence. The first item of the 2-length $IE$ sequence should be the same as the last item of the k-length prefix sequence. The other is postfix oriented which uses the Table 3.9 to facilitate the $I$-$Step$ support counting.

### 3.2.5 A Complete Example

In this section, we show a complete example to illustrate the work flow of the LAPIN algorithm. For simplicity and without loss of generality, we focus on S-Step of the mining process.

By scanning the original dataset shown in Table 3.1, we can know the 1-length frequent patterns are $\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$, and $\langle d \rangle$. The by far traversed lexicographic tree is shown in Figure 3.6. Based on DFS order, we next need to get those 2-length frequent patterns whose prefix is $\langle a \rangle$. We first get the prefix $\langle a \rangle$'s positions by binary searching in the $SE\ Position\ List\ of DB$, as shown in Figure 3.7 (a). The blue arrow indicates the index whose value is the position. Then we look for the indices in the Item_Last_Pos_Table, as shown in Figure 3.7 (b), whose values are just larger the prefix

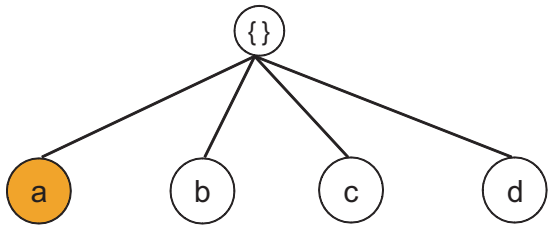



Figure 3.6: Lexicographic tree after 1-length frequent patterns discovered

$\langle a \rangle$'s positions. The discovered indices are those point to the items 10:$b$, 20:$c$, 30:$c$, as indicated by the blue arrows in Figure 3.7 (b). Start from these indices, we scan to the end of each sequence in the $Item\_Last\_Pos\_Table$, accumulate the support of each item passed. In this example, we get the support of each item as $\langle a \rangle$:1, $\langle b \rangle$:2, $\langle c \rangle$:3, and $\langle d \rangle$:3, from where we know that the 2-length frequent patterns are $\langle ab \rangle$, $\langle ac \rangle$, and $\langle ad \rangle$ (with min_support=2). Figure 3.8 shows the lexicographic tree traversed so far. Based on the DFS order, next we need to test those candidates whose prefix is $\langle ab \rangle$. To find the position of the prefix, we use binary search in the $SE\ Position\ List\ of\ DB$, as shown in Figure 3.9 (a). The purple indices indicates the result positions, whose values are 10:3, 20:5, 30:null. Next we look for the indices in the $Item\_Last\_Pos\_Talbe$, as shown in Figure 3.9 (b), whose values are just larger than the prefix $\langle ab \rangle$'s positions. The result indices are illustrated as blue color in the table. Start from these indices, we scan to the end of each sequence to accumulate the support of each passed item, resulting in $\langle a \rangle$:1, $\langle b \rangle$:1, $\langle c \rangle$:1, and $\langle d \rangle$:1, from where we know that no frequent pattern with prefix $\langle ab \rangle$ exists. The by far traversed lexicographic tree is shown in Figure 3.10. The next pattern we need to test are those ones whose prefix is $\langle ac \rangle$. The later steps and methods are similar to the ones described as above. By this recursive process, we finally find all the sequential patterns.

## 3.3 Experimental Evaluation and Performance Study

In this section, we will describe our experiments and evaluations conducted on both synthetic and real data, and compare LAPIN with PrefixSpan and SPAM to demonstrate the efficiency of the proposed algorithms. We performed the experiments using a 1.6 GHz Intel Pentium(R)M PC machine with a 1 G memory, running Microsoft

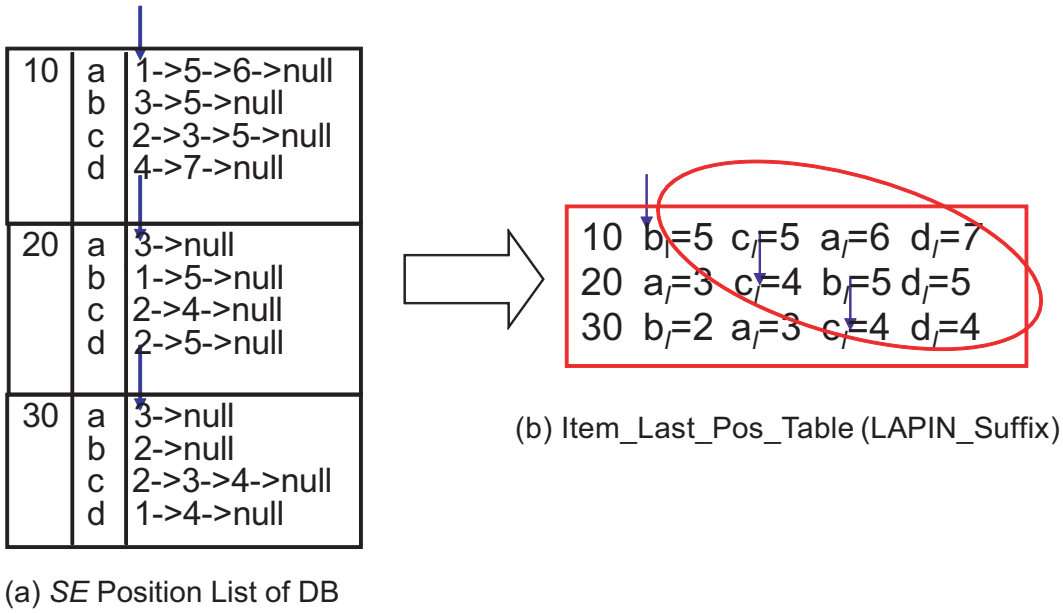



Figure 3.7: Mining process when testing the 2-length candidate sequences whose prefix is $a$

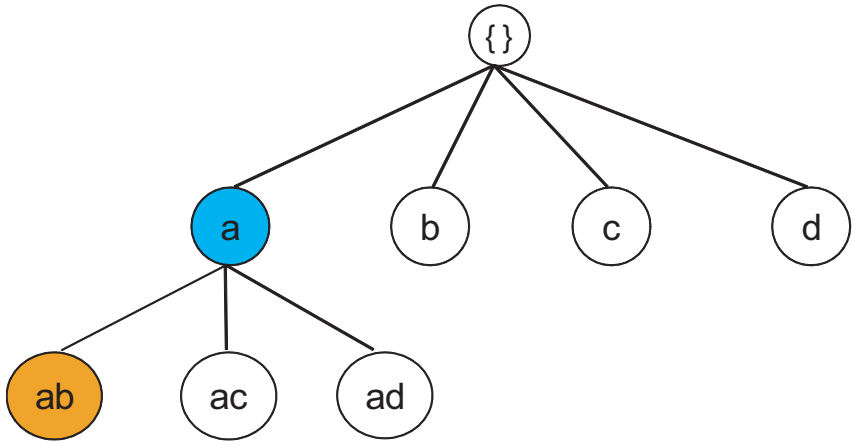



Figure 3.8: Lexicographic tree after 2-length frequent patterns (with prefix $a$) discovered

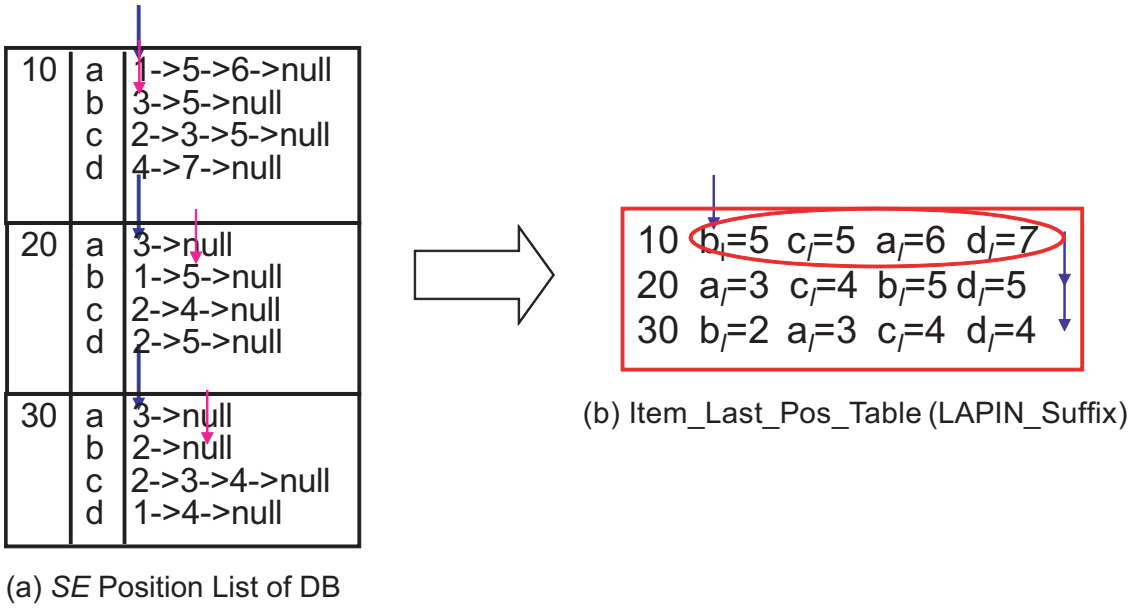



Figure 3.9: Mining process when testing the 3-length candidate sequences whose prefix is $ab$

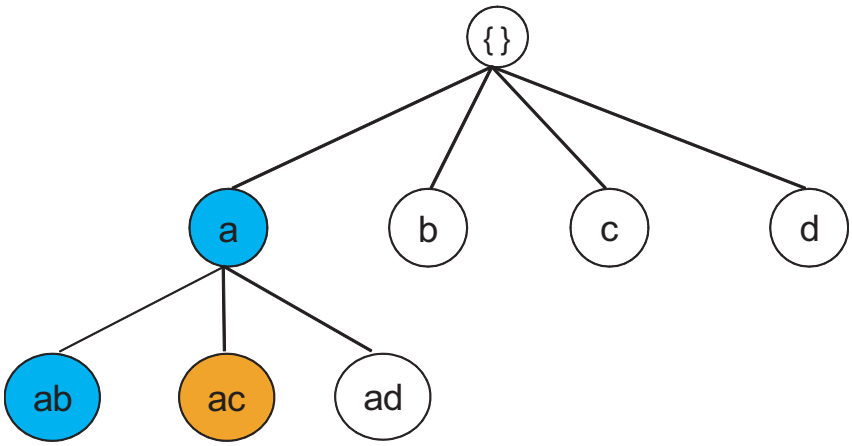



Figure 3.10: Lexicographic tree after 3-length frequent patterns (with prefix $ab$) discovered

Table 3.12: Parameters used in data set generation

| Symb. | Meaning |
|---|---|
| D | Number of customers in the data set |
| C | Average number of transactions per customer |
| T | Average number of items per transaction |
| S | Average length of maximum sequences |
| I | Average length of transactions within maximum sequences |
| N | Number of different items in the data set |

Windows XP. All three algorithms are written in C++ software, and were compiled in an MS Visual C++ environment. The output of the programs was turned off to make the comparison equitable.

We first compared PrefixSpan and our algorithms using synthetic and real data sets, and showed that LAPIN outperformed PrefixSpan by up to an order of magnitude on dense data sets with long patterns and low minimum support.

### 3.3.1 Synthetic Data.

The synthetic data sets were generated by an IBM data generator, as described in [4]. The meaning of the different parameters used to generate the data sets is shown in Table 3.12. In the first experiment, we compared PrefixSpan, SPADE and our algorithms using several low-, medium-, and high- density data sets for various minimum supports. The statistics of these data sets is shown in Figure 3.11 (a).

We defined $search\ space$ as in PrefixSpan, to be the size of the projected DB, denoted as $S_{ps}$, and in LAPIN the sum of the number of different items for each sequences in the suffix (LAPIN_Suffix) or in the local candidate item list (LAPIN_LCI), denoted as $S_{lapin}$. Figure 3.11 (b) and Figure 3.11 (c) show the execution times and the searched space comparison between PrefixSpan and LAPIN and clearly illustrate that PrefixSpan is slower than LAPIN using the medium density data set (C30T20S30I20N200D20K) and the high density data set (C50T20S50I20N300D100K). This is because the searched spaces of the two data sets in PrefixSpan were much larger than that in LAPIN. For the low density data set (C10T5S5I5N100D1K), the ineffectiveness of searched space saving and the initial overhead needed to set up meant that

LAPIN was slower than PrefixSpan. Overall, our runtime tests showed that LAPIN excelled at finding the frequent sequences for many different types of large data sets.

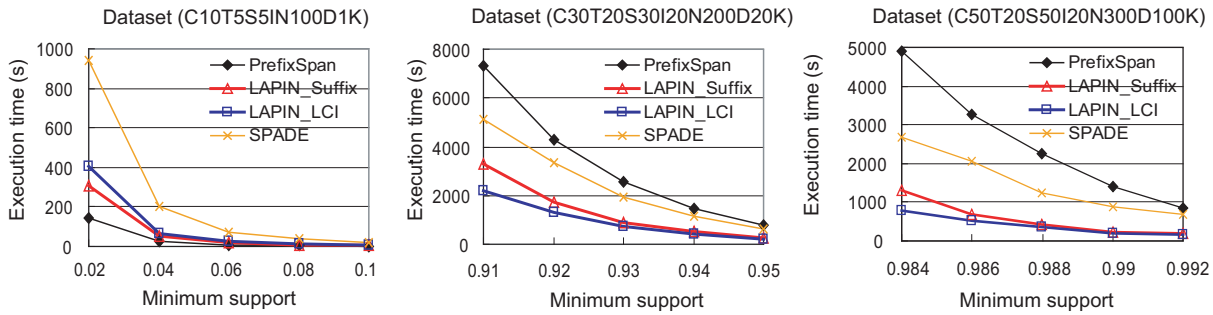
Eq.(1) in Section 1.3 illustrates the relationship between the runtime of PrefixSpan and the runtime of LAPIN in the support counting part. However, for the entire mining time, we also need to consider the initialization part and the implementation detail, which are very difficult to evaluate because of the complexity of the sequential pattern mining problem. Commonly, support counting is usually the most costly step in the entire mining process. Hence, we can approximately express the relationship between the entire mining time of PrefixSpan and that of LAPIN based on Eq.(1), where we generalize the meaning of $\bar{N}$ to denote the average total number of the distinct items in either the projected DB (LAPIN_Suffix) or in the local candidate item list (LAPIN_LCI), and the meaning of $m$ to denote either the distinct item recurrence rate of the projected DB (LAPIN_Suffix) or the local candidate list (LAPIN_LCI). Eq.(1) illustrates that, the higher the value of $m$ is, then the faster LAPIN becomes compared to PrefixSpan. However, the entire mining time of LAPIN is not faster than that of PrefixSpan $m$ times because of the initialization overhead, but near to $m$ times because of the importance of the support counting in the entire mining process. The experimental data shown in Figure 3.11 (b) and Figure 3.11 (c) is in accordance with our theoretical analysis, where the $searched\ space$ comparison determines the value of $m$, $m = S_{ps}/S_{lapin}$.

**LAPIN_Suffix vs. LAPIN_LCI:** Because LAPIN_Suffix and LAPIN_LCI are implemented in the same framework, in addition to the small difference in the initial phase, the only implementation difference is in the support counting phase: LAPIN_Suffix searches in the suffix, whereas LAPIN_LCI searches in the local candidate item list. Let $\bar{N}_{Suffix}$ be the average total number of the distinct items in the projected DB, $\bar{N}_{LCI}$ be the average total number of the distinct items in the local candidate item list, $m_{Suffix}$ be the distinct item recurrence rate of the projected DB, $m_{LCI}$ be the distinct item recurrence rate of the local candidate item list. We can express the relationship between the entire mining time of LAPIN_Suffix ($T_{Suffix}$) and that of LAPIN_LCI ($T_{LCI}$) as

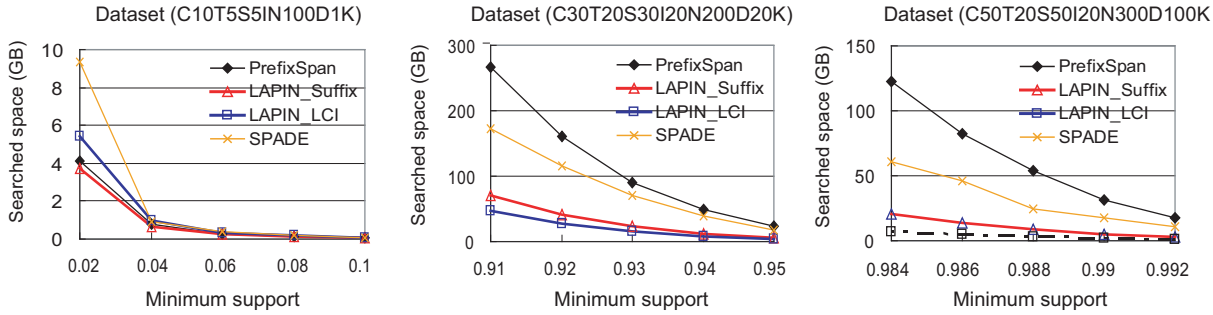
$$T_{Suffix}/T_{LCI} \approx S_{Suffix}/S_{LCI} = m_{LCI}/m_{Suffix} \qquad (2).$$

| Dataset | # sequences | Avg. length | Total size |
|---|---|---|---|
| C10T5S5I5N100D1K (small) | 1000 | 46 | 270K |
| C30T20S30I20N200D20K (medium) | 20000 | 518 | 46M |
| C50T20S50I20N300D100K (large) | 100000 | 903 | 401M |

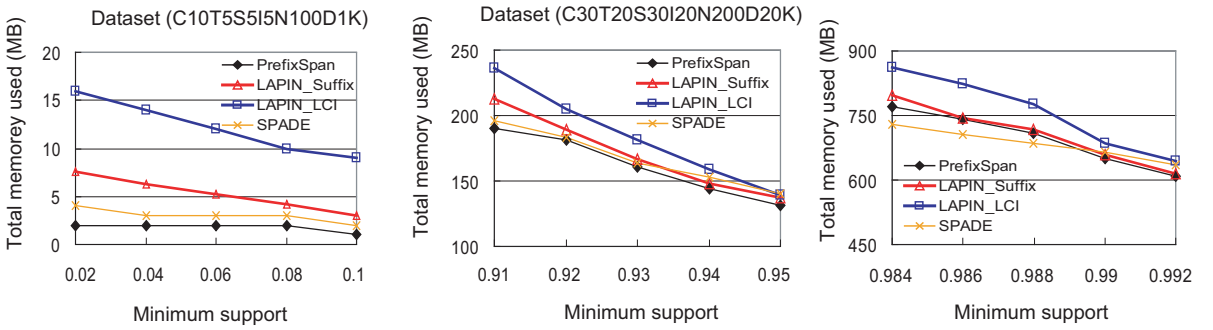
(a) Dataset characteristics




(b) Execution time comparison




(c) Searched space comparison




(d) Memory usage comparison

Figure 3.11: Performance comparison on different size of data sets

where we have the searched space of LAPIN_Suffix, $S_{Suffix} = \bar{D} \times \bar{N}_{Suffix} = \bar{D} \times \bar{L}/m_{Suffix}$, and the searched space of LAPIN_Suffix, $S_{LCI} = \bar{D} \times \bar{N}_{LCI} = \bar{D} \times \bar{L}/m_{LCI}$. Eq.(2) is in accordance with the experimental data shown in Figure 3.11 (b) and Figure 3.11 (c). LAPIN_Suffix is faster than LAPIN_LCI for low-density data sets because the former one searches smaller spaces than the latter one does. However, for medium and high-density data sets, which have many long patterns, LAPIN_LCI is faster than LAPIN_Suffix because the situation is reversed.

**Different parameters analysis:** In the second experiment, we compared the performance of the algorithms as several parameters in the data set generation were varied. The meaning of these parameters are shown in Table 3.12. As Figure 3.12 shows, when $C$ increases, $T$ increases, and $N$ decreases, then the performance of LAPIN improves even more relative to PrefixSpan, by up to an order of magnitude. Let us consider Eq.(1), $m$=$\bar{L}/\bar{N}$=$\bar{C} \times \bar{T}/\bar{N}$, where $\bar{C}$ is the average number of transactions per customer in the projected DB, and $\bar{T}$ is the average number of items per transaction in the projected DB. On keeping the other parameters constant, increasing $C$, $T$ and decreasing $N$, respectively, will result in an increase in the distinct item recurrence rate, $m$, which is in accordance with the experimental data shown in Figure 3.12. This confirms the correctness of Eq.(1).

With regards to the other three parameters, as $S$, $I$ and $D$ varies, the discrepancy between the execution times does not change significantly because under uniform distribution assumption, these parameters do not apparently contribute to the variance of the distinct item recurrence rate, $m$, which means that the discrepancy between the searched space does not change much as these three parameters are varied. Between the two LAPIN algorithms, LAPIN_LCI and LAPIN_Suffix, the former one is always the fastest because its searched space is less than that of the latter one.

### 3.3.2 Real Data.

We consider that results from real data will be more convincing in demonstrating the efficiency of our proposed algorithm. In this section, we discuss tests on two real data sets, Gazelle and Protein. A portion of Gazelle was used in KDD-Cup 2000. More details on the information in this data set can be found in [52]. The second real data set used, Protein, was extracted from the web site of the National Center for Biotech-

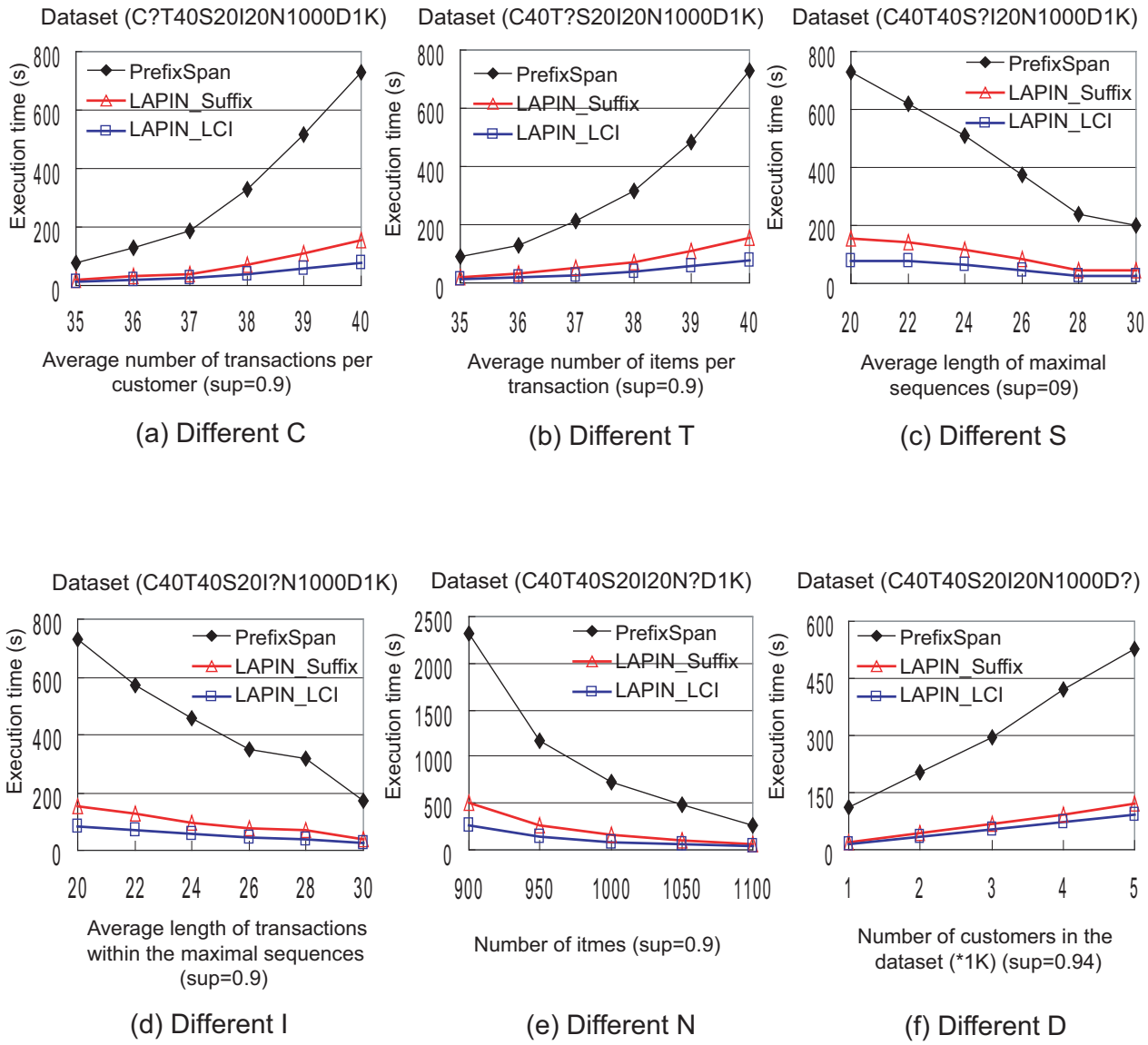
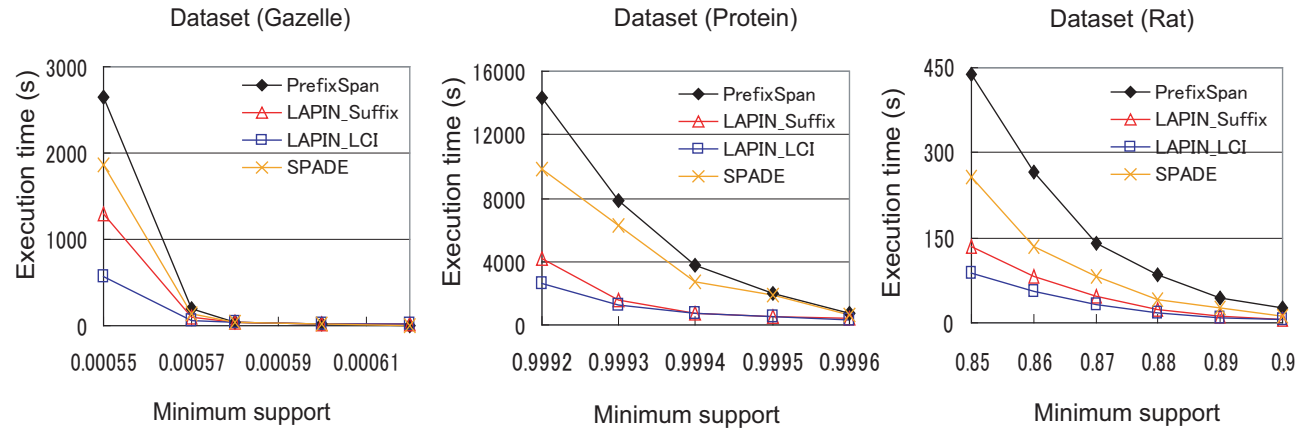


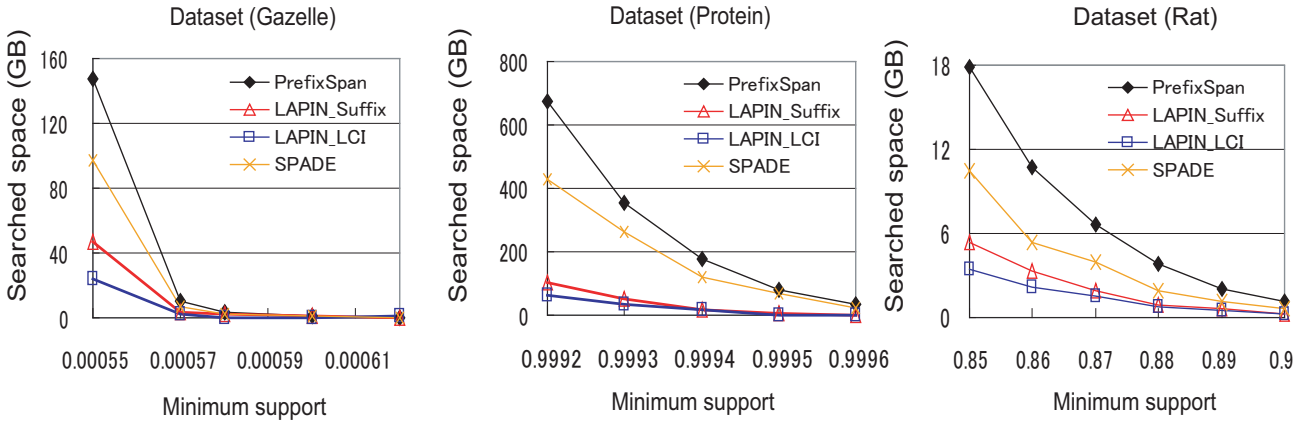

Figure 3.12: Varying the parameters of the data sets

| Dataset | # sequences | # items | Min len. | Max len. | Avg. len. | Total size |
|---|---|---|---|---|---|---|
| Gazelle | 59602 | 497 | 1 | 267 | 2.5 | 1.4M |
| Protein | 116142 | 24 | 400 | 600 | 482 | 438M |
| Rat | 451 | 21 | 1 | 2505 | 312 | 2.2M |

(a) Dataset characteristics




(b) Execution time comparison




(c) Searched space comparison

Figure 3.13: Real data sets

nology Information (USA) [1]. This was extracted using a conjunction of: (1) search category = "Protein", (2) sequence length range = [400:600], and (3) data submission period = [2004/7/1, 2004/12/31]. The third real data set, Rat (Rattus norvegicus)[2] is constructed from gene data contained in GenBank [15] release 117. The statistics of these data sets is shown in Figure 3.13 (a). Note that in this thesis we mine the sequence datasets in basic model of sequential pattern mining, the same as [5] [109] [79] [7]. The gap between two consecutive elements in a sequence is unimportant in our model of study. Therefore, LAPIN can not directly mine particular applications, such as DNA sequence with semantic meaning, which requires gap-sensitive and approximate mining. However, LAPIN can be easily extended in similar ways with that in [94] [46] to deal with biological datasets. Another issue is that we set the support value very high in testing the Protein data. The reason is that we can not terminate if setting the value to small one. This seems to be unpractical for real DNA sequence mining. As explained in the former part in this paragraph, DNA sequence has its own special property. With the constraint of these property, we believe that the mining efficiency will improve much and the support value could be set lower. However, in this thesis, we only consider on how to mine general sequences.

As shown in Figure 3.13 (b), LAPIN outperformed PrefixSpan for all the three real data sets. The reason why LAPIN performed so well was similar to that for the synthetic data sets in Section 3.1.1, and was based on the searched space saving, as shown in Figure 3.13 (c). This experiment confirmed the superiority of the proposed method using real-life data.

### 3.3.3 Analysis.

With the above thorough performance study, we are convinced that LAPIN is much more effective than PrefixSpan, and SPADE on dense datasets. There are several reasons:

- The Last Position Induction strategy (LAPIN) scans only a small part of projected database or local candidate list. However, the other algorithms need to scan the whole (projected) database, or join all the pairs of candidate items in the local list to count the candidate support.

[1] http://www.ncbi.nlm.nih.gov

[2] Rat is available at: http://bit.uq.edu.au/altExtron/gb147/ae_gb147_gene_data.html.

- By using a vertical representation format of the original database, LAPIN can apply binary search in each iteration to find the corresponding frequent prefix sequence position. In contrast, the other algorithms apply sequential search, which is obviously much slower than LAPIN for long pattern large datasets.

However, as experiments illustrated, different algorithm has its own advantage and disadvantage. Given a sequence dataset, how to select a proper efficient algorithm is an important issue. We will systemically study this problem in the next section. With new improved algorithms introduced, the selecting process will becomes simpled.

# Chapter 4

# Improved Efficient Sequential Mining Algorithms

## 4.1 LAPIN_PAID Algorithm (Passed Item Deduction Frequent Pattern Mining)

The general LAPIN strategy is very efficient with regard to execution time. However, this improvement is at the price of much memory consuming when building the list of item's last position because LAPIN uses a bitmap strategy, as introduced in the last section. This problem motivates our further work. We aim to obtain an efficient and balanced pattern mining algorithm with low memory consuming.

### 4.1.1 General Idea

**Discovering (k+1)-length frequent patterns.** For any sequence database, the last position of an item is the key used to judge whether or not the item can be appended to a given prefix (k-length) sequence (assumed to be $s$). For example, in a sequence, if the last position of item $\alpha$ is smaller than, or equal to, the position of the last item in $s$, then item $\alpha$ cannot be appended to $s$ as a (k+1)-length sequence extension in the same sequence. This strategy is named LAPIN, as described in the former part of this thesis. Moreover, we can deduce which items "disappear" after growing k-length sequence to (k+1)-length sequence, based on their last positions are larger than the (k+1)-length sequence border position or not. Then, we can reduce the supports of these "disap-

Table 4.1: Sequence Database

| SID | Sequence |
|---|---|
| 10 | $bdbcbdabad$ |
| 20 | $dcaabcbdab$ |
| 30 | $cadadcadca$ |

peared" items from already found results (support value) of k-length frequent patterns to get (k+1)-length candidate sequence' supports, avoiding to scan k-length frequent pattern's projected database, which is probably very large. Because the reuse strategy in this algorithm is based on passed (disappeared) items, we named this algorithm as LAPIN_PAID (PAssed Item Deduction based on LAPIN).

**Example 4.1**. When scanning the database in Table 4.1 for the first time, we obtain Figure 4.1 (a), which is a list of the last positions of the 1-length frequent sequences in ascending order. "↓" means the first index whose position is larger than the position of the last item in the corresponding prefix sequence. The initialization value for these indices is "1", whose corresponding prefix is $\langle\ \rangle$. We can also get 1-length frequent sequences with their support, $\langle a\rangle : 3$, $\langle b\rangle : 2$, $\langle c\rangle : 3$, and $\langle d\rangle : 3$, as shown in Figure 4.1 (b), which is indeed the support count of each item in the $\langle\ \rangle$-projected DB.

Following the Depth First Search (DFS) path, to find the 2-length frequent patterns whose common prefix is item $a$, we first get item $a$'s positions in Table 4.1 are 10:7, 20:3, 30:2, where sid:eid represents the sequence ID and the element ID. Then, we check Figure 4.1 (a) to obtain the first indices whose positions are larger than $\langle a\rangle$'s, resulting in 10:2, 20:1, 30:1, i.e. (10:$b_{last}$ = 8, 20:$c_{last}$ = 6, and 30:$d_{last}$ = 8), as shown in Figure 4.2 (a). From Figure 4.1 (a) and Figure 4.2 (a), we can know that only the index for the first customer sequence changed, which indicates item c "disappeared" by extending from $\langle\rangle$ to $\langle a\rangle$ for the customer whose SID is 10. So we reduce the support value of item $c$ by 1 from Figure 4.1 (b) to get each candidate item's support, resulting in $\langle$a$\rangle$ : 3, $\langle$b$\rangle$ : 2, $\langle$c$\rangle$ : 2, and $\langle$d$\rangle$ : 3, as shown in Figure 4.2 (b). They are indeed the supports of 2-length sequences, i.e. $\langle$aa$\rangle$ : 3, $\langle$ab$\rangle$ : 2, $\langle$ac$\rangle$ : 2, $\langle$ad$\rangle$ : 3. Thus we can determine that $\langle aa\rangle$, $\langle ab\rangle$, $\langle ac\rangle$ and $\langle ad\rangle$ are the 2-length frequent patterns whose

| SID | Last Position of Item |
|---|---|
| 10 | $c_{last}$=4 $b_{last}$=8 $a_{last}$=9 $d_{last}$=10 |
| 20 | $c_{last}$=6 $d_{last}$=8 $a_{last}$=9 $b_{last}$=10 |
| 30 | $d_{last}$=8 $c_{last}$=9 $a_{last}$=10 |

(a) Indices in Last Position of Item Table

| | a | b | c | d |
|---|---|---|---|---|
| Support in projected DB | 3 | 2 | 3 | 3 |

(b) Support in < >-projected DB

Figure 4.1: Prefix sequence is ⟨ ⟩

| SID | Last Position of Item |
|---|---|
| 10 | $c_{last}$=4 $b_{last}$=8 $a_{last}$=9 $d_{last}$=10 |
| 20 | $c_{last}$=6 $d_{last}$=8 $a_{last}$=9 $b_{last}$=10 |
| 30 | $d_{last}$=8 $c_{last}$=9 $a_{last}$=10 |

(a) Indices in Last Position of Item Table

| | a | b | c | d |
|---|---|---|---|---|
| Support in Projected DB | 3 | 2 | 2 | 3 |

(b) Support in <a>-projected DB

Figure 4.2: Prefix sequence is ⟨ a ⟩

common prefix is $a$.

From the above example, we can show that the main difference between LAPIN_PAID and previous works is to make use of the intermediate result (support value) or not. PrefixSpan, SPADE and LAPIN accumulate the support of each candidate item from scratch. However, LAPIN_PAID can obtain the same result by reducing the support of "disappeared" candidate item from previously found prefix pattern's support. In LAPIN_PAID, it judges an item's disappearance by checking its last position. Obviously, by making good use of the intermediate result, LAPIN_PAID can scan much smaller space than the other algorithms do. For the above example, to find the 2-length frequent patterns whose common prefix is $a$, PrefixSpan needs 18 scanning times in the projected DB and LAPIN needs 11 scanning times. However, LAPIN_PAID only needs 4 comparison times to discover the same result.

#### 4.1.1.1 Formulation

**Definition 1** (Prefix border position set)**.** *Given two sequences, A=$\langle A_1 A_2 \ldots A_m \rangle$ and B=$\langle B_1 B_2 \ldots B_n \rangle$, suppose that there exists C=$\langle C_1 C_2 \ldots C_l \rangle$ for $l \leq m$ and $l \leq n$, and that C is a common prefix for A and B. We record both positions of the last item $C_l$ in A and B, respectively, e.g., $C_l$=$A_i$ and $C_l$=$B_j$. The position set, (i, j), is called the prefix border position set of the common prefix C, denoted as $\mathcal{S}_c$. Furthermore, we denote $\mathcal{S}_{c,i}$ as the prefix border position of the sequence, i.*

For instance, if A=$\langle abc \rangle$ and B=$\langle acde \rangle$, then we can deduce that one common prefix of these two sequences is $\langle ac \rangle$, whose prefix border position set is (3,2), which is the last item c's positions in A and B. To get the prefix border position, PrefixSpan needs $O(\bar{D} \times \bar{L})$ time, where $\bar{D}$ is the average number of customers and $\bar{L}$ is the average sequence length in the projected database, because it uses pseudo projection in sequential search order, while LAPIN_PAID applies the binary search, whose time complexity is $O(\bar{D} \times log(\bar{L}))$.

To test (k+1)-length candidate sequences, PrefixSpan searches in the prefix k-length frequent pattern's projected database. In contrast, LAPIN_PAID scans in the prefix k-length frequent pattern's projected item-last-position list.

**Definition 2** (Item-last-position list)**.** *The list of the last positions of the different frequent 1-length items in ascending order (or if the same, based on alphabetic order) for a sequences is called the item-last-position list, denoted as $L_s$. Furthermore, we denote $L_{s,n}$ as the $item\text{-}last\text{-}position\ list$ of the sequence, n. Each node of $L_{s,n}$ is associated with two values, i.e., an item and an element number (denoted as $D_{s,n}.item$ and $D_{s,n}.num$ for $D_{s,n} \in L_{s,n}$)*

**Definition 3** (Candidate border index set)**.** *Given two sequences, A=$\langle A_1 A_2 \ldots A_m \rangle$ and B=$\langle B_1 B_2 \ldots B_n \rangle$, suppose that there exists C=$\langle C_1 C_2 \ldots C_l \rangle$ for $l \leq m$ and $l \leq n$, and that C is a common prefix for A and B. Then we have the prefix border position set $S_{C,i}$ and the item-last-position list $L_{s,i}$ for customer sequence i. We denote the index $CanI_{C,i}$, which points to the node $D_i$, as the candidate border index of customer sequence i, and the index set $CanI_C$ as the candidate border index set of the common prefix C, if the following conditions hold:*

*(a)* $D_i \in L_{s,i}$

*(b)* $D_i.num > S_{C,i}$

*(c)* $\forall E_i \in L_{s,i}$ *before* $D_i, E_i.num <= S_{C,i}$

For instance, for the example DB in Table 4.1, if we have the prefix sequence which is $\langle a \rangle$, then we can get its $candidate\ border\ index\ set$, 10:2, 20:1, 30:1, i.e. (10:b$_{last}$ = 8, 20:c$_{last}$ = 6, and 30:d$_{last}$ = 8), symbolized as "↓", as shown in Figure 4.2 (a).

**Definition 4** (Projected item-last-position list)**.** *Let C be a sequential pattern in a sequence database S. The C-projected item-last-position list, denoted as $P|_C$, is the collection of suffixes in the item-last-position list with regards to prefix C.*

**Definition 5** (Support counting in projected item-last-position list)**.** *Let C be a sequential pattern in sequence database S, and A be a sequence with prefix C. The support counting of A in C-projected item-last-position list $P|_C$, denoted as $support_{P|_C}(A)$, is the number of sequences $\alpha$ in $P|_C$ such that A $\sqsubseteq$ C $\cdot\alpha$.*

The time complexity of searching in the projected database is $O(\bar{D} \times \bar{L})$, while searching in the projected item-last-position list is $O(\bar{D} \times \bar{N})$, where $\bar{N}$ is the average total number of the distinct items in the projected DB. Here we have $\bar{L}/\bar{N} = m$ $(m \geq 1)$, where $m$ denotes the distinct item recurrence rate of the projected DB, or the density of the projected DB. The worst case of LAPIN_PAID is that when there is no duplicate item existing in the database (i.e. association rule mining), $m$ is equal to 1, which means that the time used in searching the projected item-last-position list is the same as that used in searching the projected DB. However, for common datasets, especially for dense database such as DNA sequence, $m$ is probably very large. Here we have a question that, is the result got by searching in the projected DB the same as searching in the projected item-last-position list? The answer can be got from the following two Lemmas.

**Lemma 3** (Projected item-last-position list)**.** *Let A and C be two sequential patterns in a sequence database S such that C is a prefix of A.*

1. $P|_A = (P|_C)|_A$, *and*
2. *for any sequence B with prefix C,* $support_{P(B)} = support_{P|_C}(B)$.

*Proof.* The proof of the Lemma 1 is similar to the proof in [79] (Lemma 3.2). The first part of the lemma follows the fact that, for a sequence $B$, the suffix of $B$ with regards to $A$, $B/A$, equals to the sequence resulted from first doing projection of $B$ with regards to $C$, i.e., $B/C$, and then doing projection $B/C$ with regards to A. That

is $B/A = (B/C)/A$. The second part of the lemma states that to collect support count of a sequence B, only the sequences in the database sharing the same prefix should be considered. Furthermore, only those suffixes with the prefix being a super-sequence of B should be counted. □

**Lemma 4** (Support counting equivalency)**.** *Let C be a sequential pattern in sequence database S, then support counting in C-projected item-last-position list gets the same result as support counting in C-projected database.*

*Proof.* From Definition 4, we know that the only difference between C-projected item-last-position list and C-projected database is that the former records the last position of different items, and the latter records all positions list of different items. From the support definition, we know that duplicate appearance in the same customer sequence does not contribute to the support counting, which means that the additional information stored in C-projected database is useless. Hence, support counting in C-projected item-last-position list gets the same result as support counting in C-projected database. □

From Lemma 1 and Lemma 2, we can know that if we want to get the support of (k+1)-length sequence, we can count the support of its k-length prefix's projected item-last-position list. However, for large datasets, this naive support counting in projected item-last-position list is not efficient than making use of already found results (support value) of k-length frequent pattern strategy. Hence, we have the following Lemma.

**Lemma 5** ((k+1)-length sequence support counting)**.** *Let C be a k-length sequential pattern and A be a (k+1)-length sequential pattern in sequence database S, such that C is a prefix of A. Then we have* $support_{P|_A} = support_{P|_C} - support_{P|_C - P|_A}$.

*Proof.* From Definition 4, we know that the relationship between C-projected item-last-position list and A-projected item-last-position list is $P|_A = P|_C - (P|_C - P|_A)$. Because item-last-position list only records each distinct item once in every customer sequence, the relationship of support counting between the two projected item-last-position lists $P|_A$ and $P|_C$ is $support_{P|_A} = support_{P|_C} - support_{P|_C - P|_A}$. □

Lemma 3 illustrates the core idea of LAPIN_PAID, which makes good use of the intermediate result (support value) of frequent k-length sequences, to get the supports of (k+1)-length candidate sequences by reducing the supports of those "disappeared"

items. Based on the above discussion, the pseudo code of LAPIN_PAID is presented in Table 4.2.

### 4.1.2 Design and Implementation

We used a lexicographic tree [7] as the search path of our algorithm and adopted a lexicographic order [7]. This used the Depth First Search (DFS) strategy. The pseudo code of LAPIN_PAID is presented in Table 4.2.

In Step 1, by scanning the DB once, we can obtain the position list table, as in Table 4.3 and all the 1-length frequent patterns. Based on the last element in each position list, we can sort and construct the $item\text{-}last\text{-}position\ list$ in ascending order, as shown in Figure 4.1 (a). After scanning the DB once, we can also get the support count of $\langle\rangle$-projected item-last-position list, which is indeed the 1-length frequent sequences support.

In function $Gen\_Pattern$, to find the prefix border position set of k-length $\alpha$ (Step 3), we first obtain the position list of the last item of $\alpha$, and then perform a binary search in the list for the (k-1)-length prefix border position. (We can do this because the position list is in ascending order.) We look for the first position that is larger than the (k-1)-length prefix border position.

Step 4 and Step 5, shown in Table 4.2, are used to find the (k+1)-length frequent pattern based on the frequent k-length pattern and the 1-length candidate items in the projected DB (projected item-last-position list). These two steps are justified by Lemma 1, Lemma 2 and Lemma 3. Commonly, support counting is the most time consuming part in the entire mining process. Here, we test the candidate item in the projected DB (projected item-last-position list), just as PrefixSpan [79] does. The correctness of the strategy was discussed in [79]. The pseudo code of finding (k+1)-length sequential patterns is shown in Table 4.4.

**Finding (k+1)-length frequent pattern.** In the $item\text{-}last\text{-}position\ list$, i.e., Fig 4.1 (a), we look for the first element whose last position is larger than the prefix border position, as Step 4 and Step 5 in Table 4.4 do. Then, we discovery those items, which "disappear" after growing from (k-1)-length prefix frequent sequence to k-length prefix frequent sequence, and decrement these "disappeared" items support from the support count of the projected item-last-position list, as shown in Step 6 to Step 8. Finally,

Table 4.2: LAPIN_PAID algorithm pseudo code

| | |
|---|---|
| INPUT: | A sequence database, and the minimum support threshold, $\varepsilon$ |
| OUTPUT: | The complete set of sequential patterns |
| Function: | Gen_Pattern($\alpha$, $S$, $Support_P$) |
| Parameters: | $\alpha$ = length k frequent sequential pattern; $S$ = prefix border position set of (k-1)-length sequential pattern; $Support_P$ = support count in (k-1)-length sequential pattern's projected item-last-position list |
| Goal: | Generate (k+1)-length frequent sequential pattern |
| | |
| Main(): | |
| 1. | Scan DB once to do: |
| 1.1. | $P_s \leftarrow$ Create the position list representation of 1-length sequences |
| 1.2. | $B_s \leftarrow$ Find the frequent 1-length sequences |
| 1.3. | $L_s \leftarrow$ Obtain the item-last-position list of the 1-length sequences |
| 1.4. | $Support_P\|_{\langle\rangle} \leftarrow$ Find the support count of $\langle\rangle$-projected item-last-position list |
| 2. | For each frequent sequence $\alpha_s$ in $B_s$ |
| 2.1. | Call Gen_Pattern ($\alpha_s$, 0, $Support_P\|_{\langle\rangle}$) |
| | |
| Function: | $Gen\_Pattern(\alpha, S, Support_P)$ |
| 3. | $S_\alpha \leftarrow$ Find the prefix border position set of $\alpha$ based on $S$ |
| 4. | $Support_P\|_\alpha \leftarrow$ Obtain the support count of $\alpha$-projected item-last-position list, based on $S$, $S_\alpha$ and $Support_P$ |
| 5. | $FreItem_{s,\alpha} \leftarrow$ Obtain the item list of $\alpha$ based on $Support_P\|_\alpha$ |
| 6. | For each item $\gamma_s$ in $FreItem_{s,\alpha}$ |
| 6.1. | Combine $\alpha$ and $\gamma_s$, results in $\theta$ and output |
| 6.2. | Call Gen_Pattern ($\theta$, $S_\alpha$, $Support_P\|_\alpha$) |

Table 4.3: Position List of DB

| SID | $Item\ \ Positions$ |
|---|---|
| 10 | $a : 7 \rightarrow 9 \rightarrow null$<br>$b : 1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow null$<br>$c : 4 \rightarrow null$<br>$d : 2 \rightarrow 6 \rightarrow 10 \rightarrow null$ |
| 20 | $a : 3 \rightarrow 4 \rightarrow 9 \rightarrow null$<br>$b : 5 \rightarrow 7 \rightarrow 10 \rightarrow null$<br>$c : 2 \rightarrow 6 \rightarrow null$<br>$d : 1 \rightarrow 8 \rightarrow null$ |
| 30 | $a : 2 \rightarrow 4 \rightarrow 7 \rightarrow 10 \rightarrow null$<br>$b : null$<br>$c : 1 \rightarrow 6 \rightarrow 9 \rightarrow null$<br>$d : 3 \rightarrow 5 \rightarrow 8 \rightarrow null$ |

we can get the frequent items in the projected item-last-position list (projected DB), as shown in Step 9 to Step 11. Obviously, in dense datasets, the size of "disappeared" projected item-last-position list should be much smaller than the projected DB, which is scanned by PrefixSpan algorithm. Moreover, we only pass and count once for each different item in the "disappeared" projected item-last-position list because, in $item$-$last$-$position\ list$, we record the last position of each item for a specific sequence. In contrast, PrefixSpan needs to pass every item in the projected database regardless of whether or not they are the same as before. Therefore, LAPIN_PAID will save much time because our search space is much smaller than the one used in PrefixSpan by making good use of the intermediate result (support value). The example has been described in Section 4.1.1.

**I-Step of LAPIN_PAID.** As Ayres et al. did in [7], the whole process of sequential pattern mining should includes two steps: a $sequence$-$extension\ step$ ($S$-$Step$) and a $itemset$-$extension\ step$ ($I$-$Step$). We have already described the $S$-$Step$ of LAPIN_PAID. The $I$-$Step$ is similar to the $S$-$Step$. One difference is that in $I$-$Step$, the basic unit is 2-length itemset extension sequence, i.e., (ab), (bc), instead of 1-length sequence, i.e., a, b, in S-Step. Another difference is that when patterns grow, $I$-$Step$

Table 4.4: Finding (k+1)-length frequent patterns

| | |
|---|---|
| INPUT: | $S$ = prefix border position set of (k-1)-length frequent sequential pattern; $S_\alpha$ = prefix border position set of k-length frequent sequential pattern $\alpha$; $Support_P$ = support count in (k-1)-length sequential pattern's projected item-last-position list; $\varepsilon$ = user specified minimum support |
| OUTPUT: | $FreItem_s$ = local frequent item list |
| 1. | For each sequence, F |
| 1.1. | $S_F \leftarrow$ obtain prefix border position of F in $S$ |
| 1.2. | $S_{\alpha,F} \leftarrow$ obtain prefix border position of F in $S_\alpha$ |
| 1.3. | M = Find the corresponding index for $S_F$ |
| 1.4. | N = Find the corresponding index for $S_{\alpha,F}$ |
| 1.5. | while ( M != Null && M $<$ N ) |
| 1.5.1. | $Support_P$[M.item] - -; |
| 1.5.2. | M++; |
| 2. | For each item $\beta$ in Suplist |
| 2.1. | If ( $Support_P[\beta] \geq \varepsilon$) |
| 2.1.1. | $FreItem_s$.insert($\beta$); |

should guarantee that the prefix sequences of the tested candidates are the same. We deal with it by joining the position list of each candidate $IE$ item after current prefix position, which is similar to the method used in SPADE [109].

## 4.2 LAPIN_SPAM Algorithm

As introduced in Section 3.1, we have found that SPAM is very efficient in resource unlimited environments (i.e., huge memory available). Hence, we aim to develop one algorithm based on SPAM. We have found more efficiency improving space in the support counting process of SPAM based on the basic idea of LAPIN.

In SPAM, to judge a candidate is a pattern or not, it does as many ANDing operation as to the number of customers involved. For example, if there are 10000 customers in certain dataset, it will cost 10000 ANDing operation time for each candidate item testing. Consider the recursive characteristic in the implementation, this cost is too big. So how to avoid this ANDing operation becomes essential step.

| Pos \ Item | a | b | c | d |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 |
| 3 | 1 | 1 | 1 | 1 |
| 4 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 |
| 6 | 1 | 1 | 0 | 1 |
| 7 | 1 | 1 | 0 | 1 |
| 8 | 1 | 0 | 0 | 1 |
| 9 | 0 | 0 | 0 | 1 |
| 10 | 0 | 0 | 0 | 0 |

Figure 4.3: Bitmap representation table

| Pos \ Item | a | b | c | d |
|---|---|---|---|---|
| 4 | 1 | 1 | 0 | 1 |
| 8 | 1 | 0 | 0 | 1 |
| 9 | 0 | 0 | 0 | 1 |

Figure 4.4: Optimized ITEM_IS_EXIST_TABLE

### 4.2.1 General Idea

As mentioned earlier, if given a current position in certain customer, we can know which items are behind current position and which are not based on the last position of them. So a naive method to judge a candidate is to compare the last position of it with the current position. This is in fact the same cost as ANDing operation in SPAM. To avoid this comparison or ANDing operation, we can construct a ITEM_IS_EXIST_TABLE when scanning the database for the first time. In each iteration, we only need to check this table to get information that a candidate is behind current position or not. By this way, we can save much time by avoiding ANDing operation or comparison.

Figure 4.3, which is built based on the example database in Table 1, shows one part of the ITEM_IS_EXIST_TABLE for the first customer. The left column is the position number and the top row is the item ID. In the table, we use bit vector to represent candidates existence for respective position. Bit value is 1 indicates the item existing, otherwise the item does not exist. The bit vector size is equal to the total number of the

```
1. For each customer sequence F
2.    for each item α in local candidate item list
3.       result ← bitmap_prefix & bitmap_α;
4.       if (result != 0)
5.          Suplist[α]++;
```

Figure 4.5: S-Step support counting of SPAM

candidate items. For example, if the current position is 2, we can get its corresponding bit vector as 1111, which means that all candidates can be appear behind current prefix. When the current position is 8, we can get the bit vector as 1001, indicates that only item a and d exist in the same customer sequence after the current prefix. To accumulate the candidate sequence's support, we only need to check this table and add the corresponding item's vector value, avoiding comparison, ANDing operation or constructing S-Matrix in each recursive step, which largely improve efficiency during mining. Attention that here we only discuss the S-Step process, the reader can easily extend it to the I-Step process based on the same strategy.

### 4.2.2 Implementation

The pseudo code of SPAM and LAPIN-SPAM are shown as Figure 4.5 and Figure 4.6, which present the main difference between of them on support counting part.

The Figure 4.5 and Figure4.6 show that LAPIN-SPAM avoids the step 3 of SPAM. The experimental result in Section 4.3 shows that SPAM always does three times ANDings operations more than LAPIN-SPAM does.

#### 4.2.2.1 Space Optimization

SPAM assumes that the whole vector representation of the database should be filled in the main memory, yet the space necessary is always a key factor of an algorithm. As Figure 4.3 shows, we can easily know that the main memory used in LAPIN-SPAM is no more than twice of that used in SPAM, because each item needs one bit for every transaction no matter it exists or not in the ITEM_IS_EXIST_TABLE.

```
1. For each customer sequence F
2.     bitV ← get the bit vector indexed by the
                 prefix border position
3.     for each item α in local candidate item list
4.       Suplist[α]=Suplist[α]+bitV[α];
```

Figure 4.6: S-Step support counting of LAPIN_SPAM

After consideration, we find that only part of the table is useful and most are not. For example in Figure 4.3, when the current position is smaller than 4, all items exist and when the position is larger than 9, there is no item existing. So the useful information is store in some key positions' lines. We define *key position* as follows: Given a position, if its corresponding bit vector is different from that of the position one smaller than it (except the one whose bit vector is equal to 0), this position is called *key position*. For example, in Figure 4.3, the position 4, 8 and 9 are key positions and others are not (position 10 is not because its bit vector is equal to 0). We can find that these key positions are indeed the last positions of the candidates items (except the last one). The optimized ITEM_IS_EXIST_TABLE is shown in Figure 4.4, which stores only two bit vectors instead of eight ones shown in Figure 4.3. For long pattern dataset, this space saving strategy is more efficient. Through thorough experiments what we will mention in Section 4.3.2, the memory used to store the ITEM_IS_EXIST_TABLE is less than 10 percent of the one used in SPAM, which can be neglected when comparing LAPIN-SPAM and SPAM efficiency.

## 4.3 Experimental Evaluation and Performance Study

In this section, we will describe our experiments and evaluations conducted on both synthetic and real data, and compare LAPIN_PAID and LAPIN_SPAM with LAPIN_Suffix, LAPIN_LCI, PrefixSpan and SPAM to demonstrate the efficiency of the proposed algorithms. Moreover, we evaluated all the algorithms on judging which situation they prefer and give a summary at the end. We first performed the experiments using a 1.6 GHz Intel Pentium(R)M PC machine with a 1 G memory, running Microsoft Windows XP. All these algorithms are written in C++ software, and were compiled in an

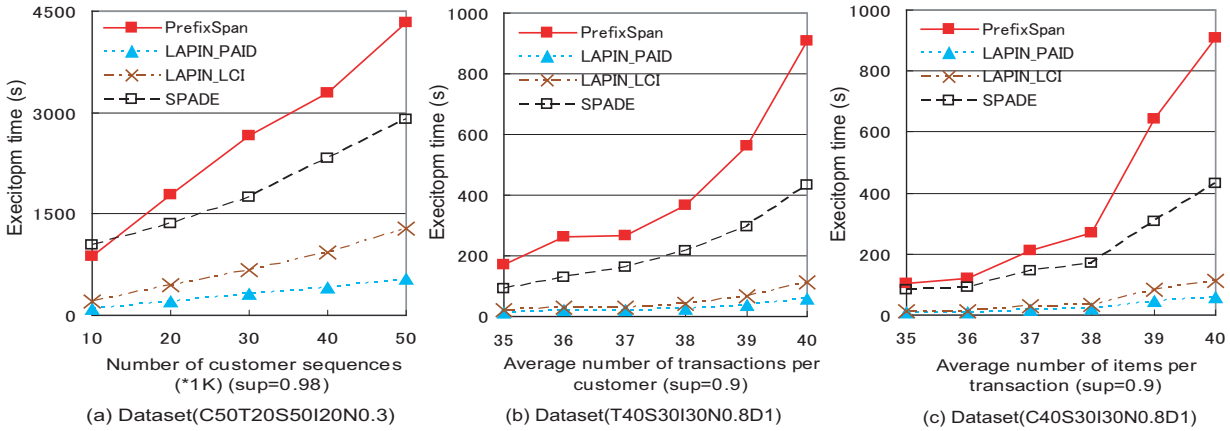


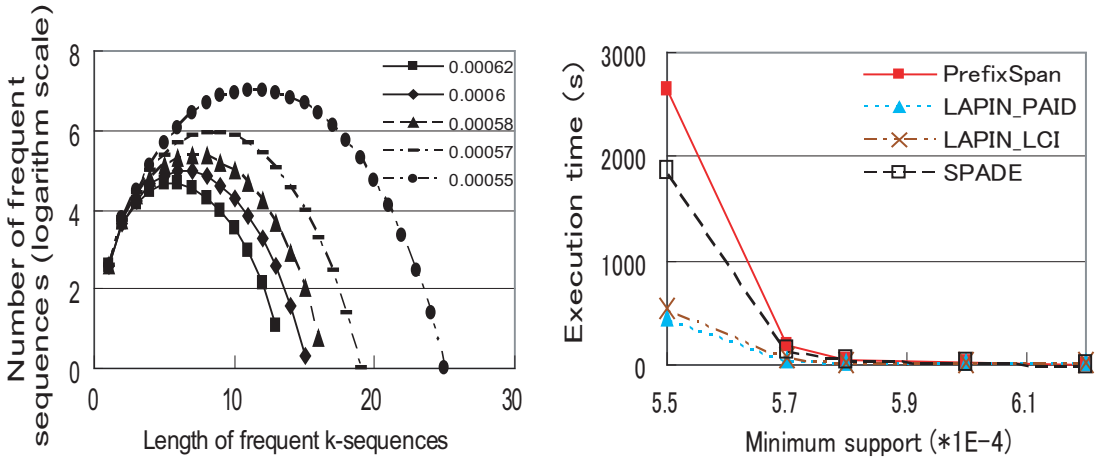

Figure 4.7: Scalability performance

MS Visual C++ environment. The output of the programs was turned off to make the comparison equitable.

### 4.3.1 Scalability test between PrefixSpan, SPADE and LAPIN algorithms

**Scalability Test.** Because only some parameters are important for performance, as illustrated and explained in Chapter 3. We study how LAPIN_PAID performs with increasing number of customer sequences (D), average number of events per customer sequence ($C$) and average number of items per event ($T$), respectively. We first only compare the execution time and shortly, we will systemically study the performance of all the algorithms.

Figure 4.7 (a) shows how LAPIN_PAID scales up as the number of customer sequences is increased, from 10K to 50K, whose corresponding database size ranging from 40M to 200M. The experiment was performed on the C50T20S50I20N0.3 with minimum support is 0.98. It can be observed that LAPIN_PAID scales almost linearly and is much faster than the other three algorithms. For example, in Figure 4.7




(a) Distribution of frequent sequences (b) Performance comparison

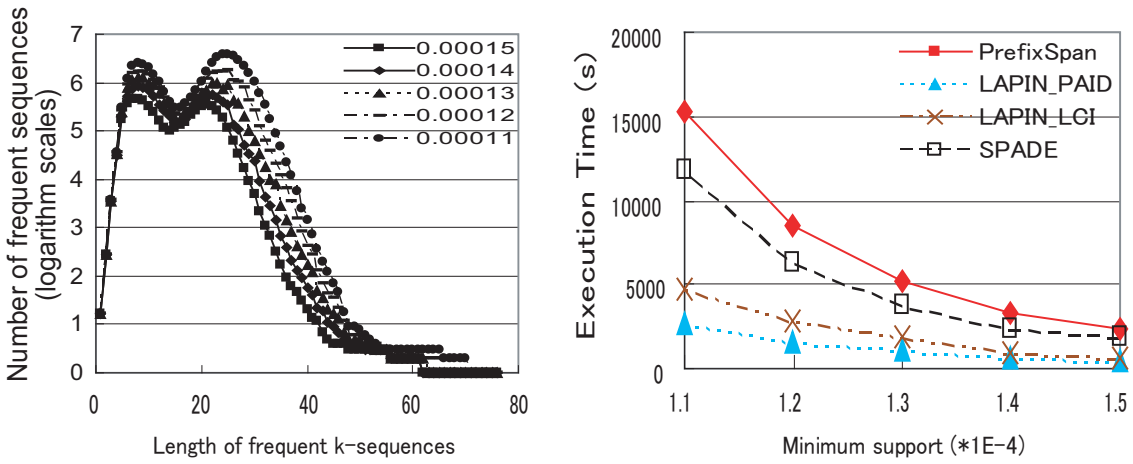
Figure 4.8: Dataset (Gazelle)

(a), when $D$ = 50, LAPIN_PAID (runtime = 544 seconds) is about 8 times faster than PrefixSpan (runtime = 4329 seconds), five times faster than SPADE (runtime = 2903 seconds) and two times faster than LAPIN (runtime = 1292 seconds). Figure 4.7 (b) and 4.7 (c) show the other two scalability experimental results. For both the graphs, we used S30I30N0.8D1. In Figure 4.7 (b) we set $T$ to 40, and varied $C$ from 35 to 40, and Figure 4.7 (c) we set $C$ to 40, varied $T$ from 35 to 40. It can be easily observed LAPIN_PAID scales linearly with the two varying parameters and is always faster than the other three algorithms.

### 4.3.2 Real Data Evaluation between PrefixSpan, SPADE and LAPIN algorithms

We consider that results from real data will be more convincing in demonstrating the efficiency of our proposed algorithm. In this section, we discuss tests on two real datasets.

The first real dataset, Gazelle, was obtained from Blue Martini company, which was also used in KDD-Cup 2000. This dataset contains 59602 sequences (i.e., customers), 149639 sessions, and 497 distinct page views. The average sequence length is 2.5 and the maximum sequence length is 267. More detailed information about




(a) Distribution of frequent sequences (b) Performance comparison

Figure 4.9: Dataset (MSNBC)

this dataset can be found in [52]. Figure 4.8 (a) shows the distribution of frequent sequences of Gazelle dataset for different support thresholds. We can see that this dataset is a sparse dataset compared with Rat because, only when the support threshold is very low are there some long frequent sequences (i.e., when min_sup=0.00062, total frequent sequences number = 207168). Figure 4.8 (b) shows the performance comparison between PrefixSpan, SPADE and LAPIN_PAID for Gazelle dataset. We can see that LAPIN_PAID is more efficient than PrefixSpan and SPADE. For example, at support 0.00055, LAPIN_PAID (runtime = 583 seconds) is near five times faster than PrefixSpan (runtime = 2642 seconds) and four times faster than SPADE (runtime = 1857 seconds). The running time of LAPIN is 677 seconds.

We have obtained the second dataset, MSNBC, from the UCI KDD Archive[1]. This dataset comes from Web server logs for msnbc.com and news-related portions of msn.com on Sep. 28, 1999. There are 989,818 users and only 17 distinct items, because these items are recorded at the level of URL category, not at page level, which greatly reduces the dimensionality. The average sequence length is 6 and the maximum sequence length is 14,795. Figure 4.9 (a) shows the distribution of frequent sequences of Rat dataset for different support thresholds, from 0.00011 to 0.00015. When the support threshold is 0.00011, there are many frequent sequences (total num-

[1]http://kdd.ics.uci.edu/databases/msnbc/msnbc.html

ber = 45,541,248). Figure 4.9 (b) shows the performance comparison among PrefixSpan, SPADE and LAPIN_PAID for MSNBC dataset. The result mirrors that of the Gazelle dataset closely.

### 4.3.3 Scalability Study between SPAM and LAPIN-SPAM

We tested both SPAM and LAPIN-SPAM using the synthetic data, since SPAM limits the sequence length to 64. We consider the different parameters used to generate the datasets on testing the performance. Figure 4.10(a) shows the result when changing the number of the customers. Figure 4.10(b) presents the effect when varying average number of transactions per customer. Figure 4.10(c) shows the result when changing the average number of items per transaction parameter. Figure 4.10(d) modifies the average length of maximal sequences and the variable in Figure 4.10(e) is the number of different items in the datasets. We can see that no matter which parameter changes, LAPIN-SPAM is always faster than SPAM about 2 to 3 times. The primary reason that LAPIN-SPAM performs so well for all datasets is due to avoiding ANDing operation or comparison of the bitmap for efficient counting. This process is critical because it is performed many times at each recursive step, and LAPIN-SPAM can save much time compared with SPAM.

### 4.3.4 Memory Usage Analysis between SPAM and LAPIN_SPAM

Using bit vector to represent items' existence is the most space consuming part in our LAPIN_SPAM algorithms. Let $D$ be the number of customers in the database, $C$ the average number of transactions per customer, and $N$ the total number of items across all of the transactions. As Ayres et al. pointed in [7], SPAM requires $(D \times C \times N)/8$ bytes to store all of the data. In LAPIN, let $C'$ be the average number of the $key\ positions$ per customer, then our LAPIN_SPAM requires $(D \times C' \times N)/8$ bytes to store the last position information for all the items. For those datasets with long pattern and small number of items, the $C'/C$ should be very small, which means that our LAPIN_SPAM algorithm get high speed performance at the price of relevant small space consuming compared with SPAM.

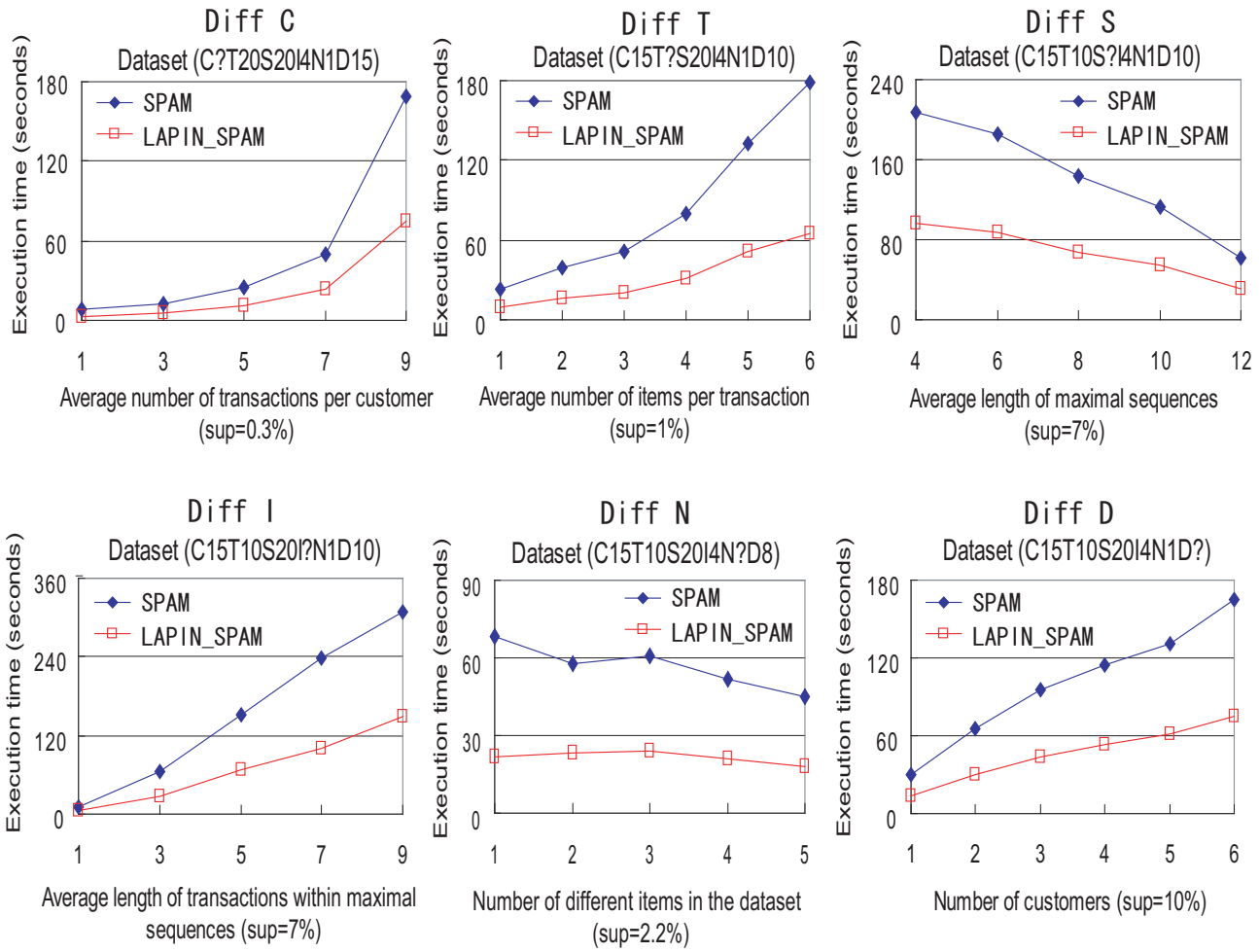



Figure 4.10: Execution time comparison when varying parameters of the datasets (SPAM vs LAPIN_SPAM)

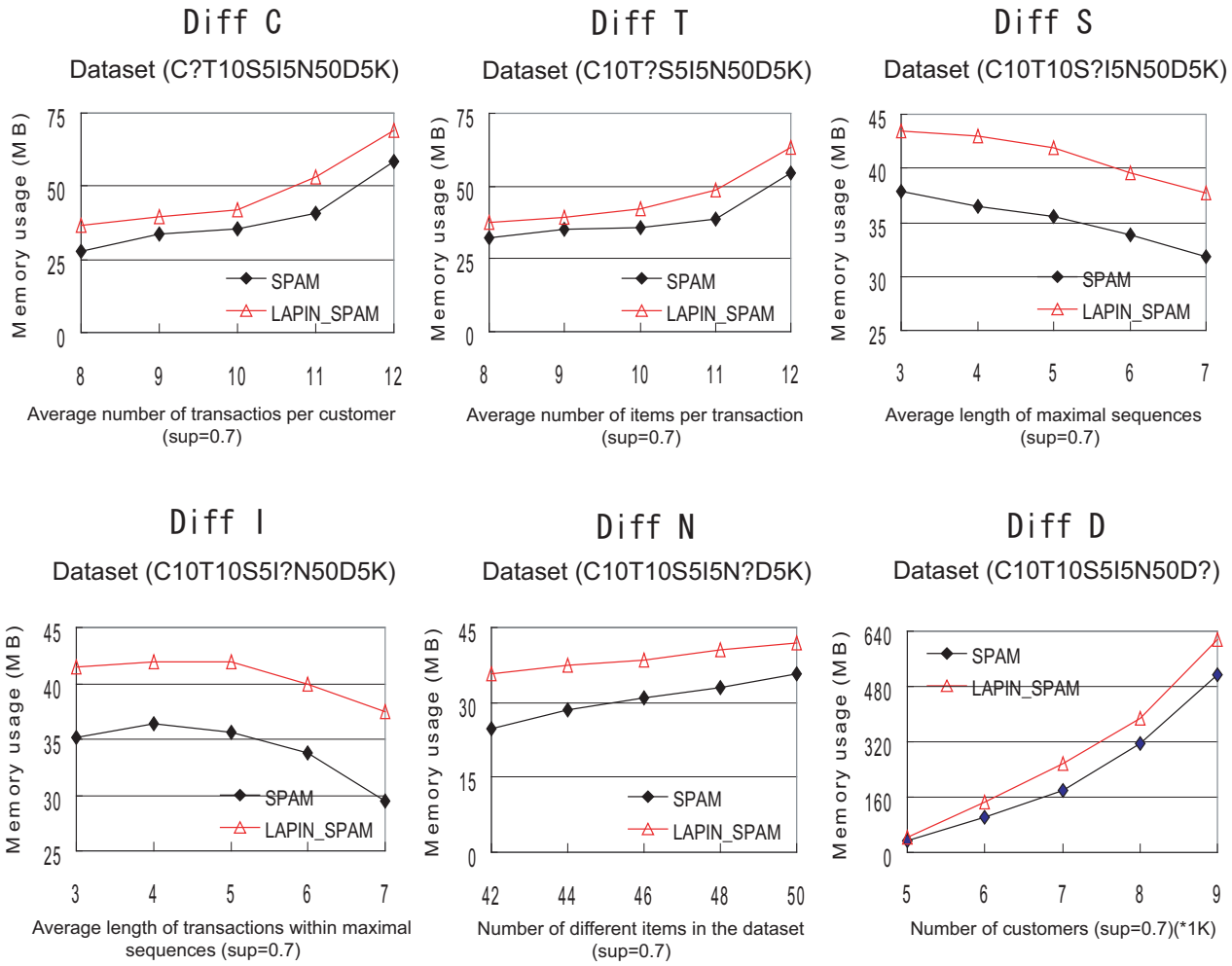



Figure 4.11: Memory usage comparison when varying parameters of the datasets (SPAM vs LAPIN_SPAM)

| Existing Algorithm | LAPIN Family Algorithms |
|---|---|
| PrefixSpan | LAPIN_Suffix, LAPIN_PAID |
| SPADE | LAPIN_LCI |
| SPAM | LAPIN_SPAM |

Figure 4.12: Classified algorithms to be evaluated

### 4.3.5 Systemic Study on Different Algorithms

As mentioned above and in Chapter 3, we have found the effect of the three parameters, $C$, $T$ and $N$ on the performance of different algorithms. The reason is that under uniform distribution assumption [1], these three parameters contribute to the density $m$, where we have $m$=$C \times T/N$. Hereafter, we decide to use the density value $m$, as the basic parameter to analyze the performance of the algorithms. Another decision is that, as shown in Figure 4.12, we will compare algorithms in their own category (i.e., the algorithms in the same row, as for example, SPAM and LAPIN_SPAM) to convenient our evaluation. Later we will compare the best ones in each category to make a comprehensive conclusion.

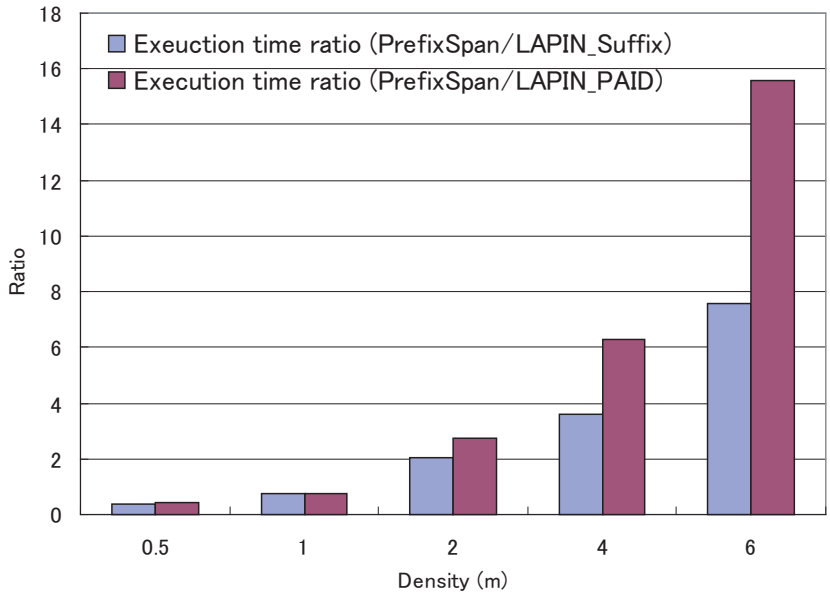
#### 4.3.5.1 PrefixSpan v.s. LAPIN_Suffix v.s. LAPIN_PAID

In this section, we compare the algorithms of PrefixSpan, LAPIN_Suffix and LAPIN_PAID on varying the density $m$. The characteristic of the tested datasets is shown in Figure 4.13 (a). We change the density from 0.5 to 6. Figure 4.13 illustrates the result. Specifically, Figure 4.13 (b) shows that LAPIN_Suffix and LAPIN_PAID will become much faster than PrefixSpan, when $m$ becomes larger than 1. The execution time performance becomes reverse if m is small than 1. For the memory usage, as illustrated in Figure 4.13 (c), LAPIN_Suffix and LAPIN_PAID always consumes the same memory because they use the same data structure. As $m$ increases, the difference between LAPIN algorithms with PrefixSpan will get smaller. Specifically, when $m$ is 2, LAPIN algorithms consume about 2 times memory of PrefixSpan. In summary, LAPIN_PAID is always better than LAPIN_Suffix on considering both execution time and memory usage. $m = 2$ can be seen as a criterion to judge whether use LAPIN_PAID or PrefixSpan that, when $m \leq 2$, LAPIN_PAID is preferred and otherwise, select PrefixSpan.

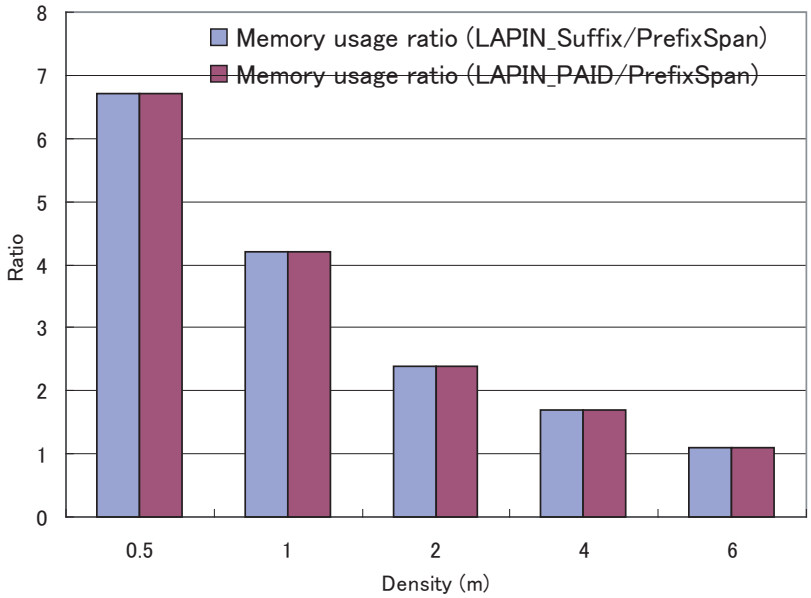
[1] The IBM data generator what we used is supposed to generate uniform distribution data.

| Density (m) | Dataset |
|---|---|
| 0.5 | C10T5I5S5N100D1K |
| 1 | C10T10I5S5N100D1K |
| 2 | C20T10I5S5N100D1K |
| 4 | C20T20I5S5N100D1K |
| 6 | C30T20I5S5N100D1K |

(a) Datasets used to test




(b) Execution time comparison




(c) Memory usage comparison

Figure 4.13: Performance results of comparing PrefixSpan, LAPIN_Suffix and LAPIN_PAID

#### 4.3.5.2 SPADE v.s. LAPIN_LCI

In this section, we compare the algorithms of SPADE and LAPIN_LCI on varying the density $m$. The characteristic of the tested datasets is shown in Figure 4.14 (a). We change the density from 0.5 to 6. Figure 4.14 illustrates the result. Specifically, Figure 4.14 (b) shows that LAPIN_LCI will become much faster than SPADE, when $m$ becomes larger than 1. We can also mention that although $m$ is smaller than 1, LAPIN_LCI is still faster than SPADE to some extent. For the memory usage, as illustrated in Figure 4.14 (c), as $m$ increases, the difference between LAPIN_LCI and SPADE will get smaller. Similar to the experiment in last section, in summary, $m = 2$ can be seen as a criterion to judge whether use LAPIN_LCI or SPADE that, when $m \leq$2, LAPIN_LCI is preferred and otherwise, we use SPADE.

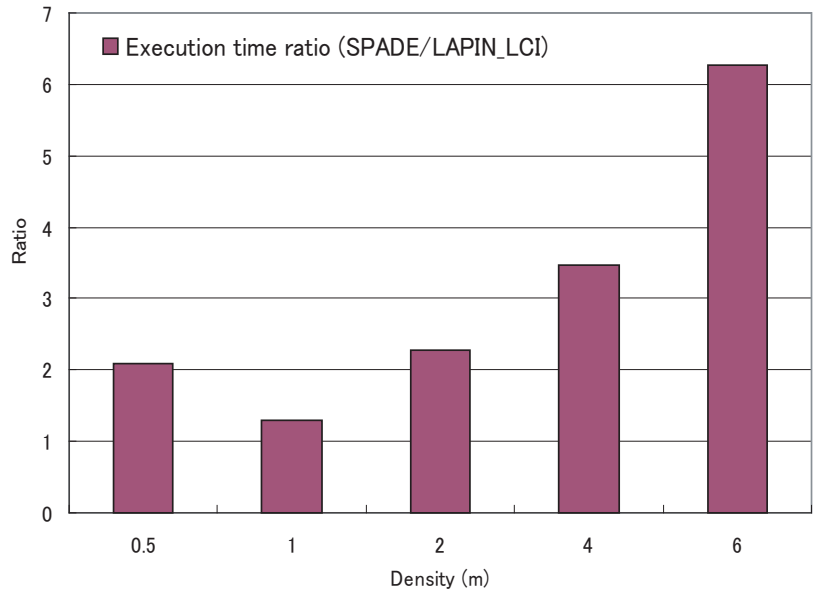
#### 4.3.5.3 SPAM v.s. LAPIN_SPAM

In this section, we compare the algorithms of SPAM and LAPIN_SPAM on varying the density $m$. The characteristic of the tested datasets is shown in Figure 4.15 (a). We change the density from 0.1 to 6. Figure 4.15 illustrates the result. Specifically, Figure 4.15 (b) shows that LAPIN_SPAM is always faster than SPAM, by about two to three times, no matter what the value of $m$. The reason is that the ANDing operations in SPAM is about three times more than that of LAPIN_SPAM. For the memory usage, as illustrated in Figure 4.15 (c), the difference between LAPIN_SPAM and SPAM is between 1 and 1.5, which means LAPIN_SPAM at most consume about 150% memory of that used in SPAM. In summary, the overall performance of LAPIN_SPAM is better than SPAM no matter with the value of $m$..

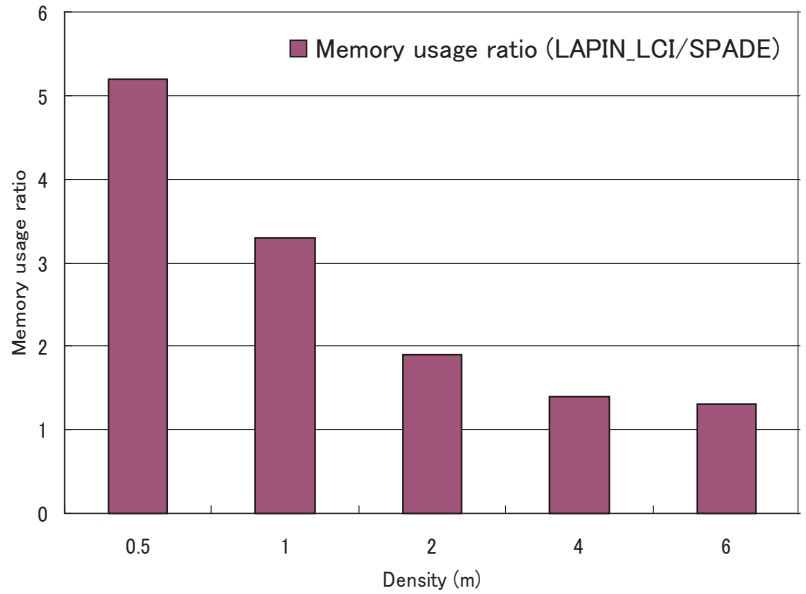
Through the three sections, Section 4.3.5.1, Section 4.3.5.2 and Section 4.3.5.3, we have found that the best algorithm in each category. Moreover, we discovered the trade off between these algorithms and found a criterion value of $m$, which is 2, to judge the best algorithm. In the next sections, we further compare the best one in each category to give a overall summary of existing algorithms. When the value of $m$ is smaller than 2, the best three algorithms in the three categories are PrefixSpan, SPADE and LAPIN_SPAM. When the value of $m$ is larger than 2, the best three algorithms in the three categories are LAPIN_PAID, LAPIN_LCi and LAPIN_SPAM. We will compare them accordingly.

| Density (m) | Dataset |
|---|---|
| 0.5 | C10T5I5S5N100D1K |
| 1 | C10T10I5S5N100D1K |
| 2 | C20T10I5S5N100D1K |
| 4 | C20T20I5S5N100D1K |
| 6 | C30T20I5S5N100D1K |

(a) Datasets used to test




(b) Execution time comparison

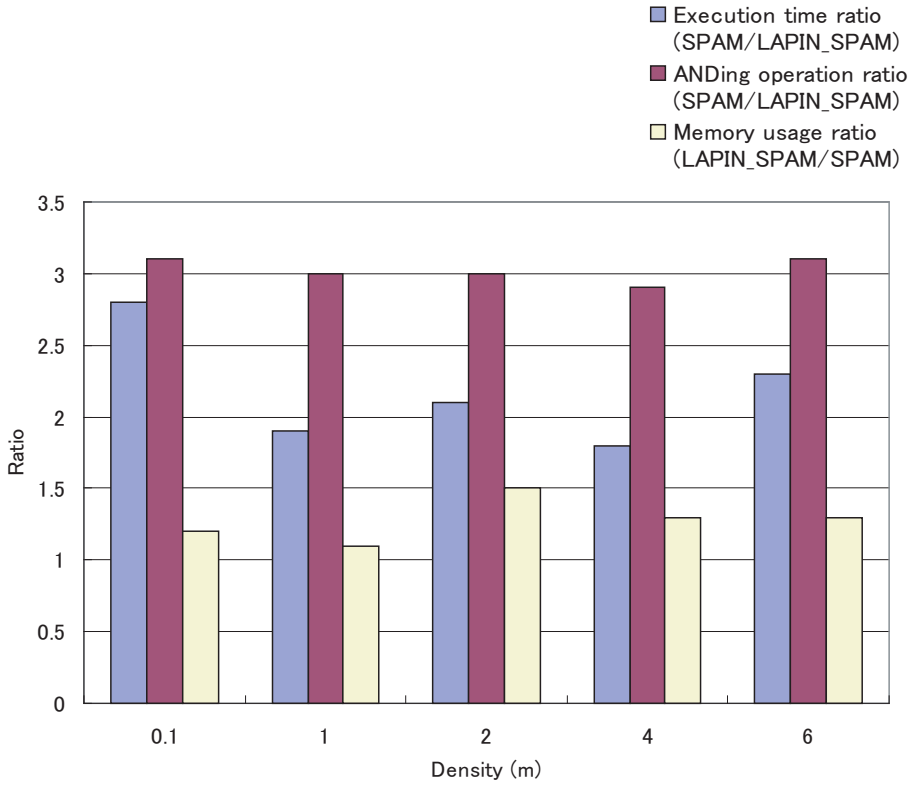



(c) Memory usage comparison

Figure 4.14: Performance results of comparing SPADE and LAPIN_LCI

| Density (m) | Dataset |
|---|---|
| 0.1 | C10T10I5S5N1KD1K |
| 1 | C20T5I5S5N100D1K |
| 2 | C20T10I5S5N100D1K |
| 4 | C20T20I5S5N100D1K |
| 6 | C30T20I5S5N100D1K |

(a) Datasets used to test




(b) Performance comparison between SPAM and LAPIN_SPAM

Figure 4.15: Performance results of comparing SPAM and LAPIN_SPAM

#### 4.3.5.4 PrefixSpan v.s. SPADE v.s. LAPIN_SPAM

In this section, we first compared the algorithms of SPADE and PrefixSpan when the density $m$ is smaller than 2. The characteristic of the tested datasets is shown in Figure 4.16 (a). We change the density from 0.1 to 1.5. Figure 4.16 (b) illustrates the result. Specifically, PrefixSpan is always faster than SPADE, by up to orders of magnitude on small value of $m$. As $m$ increases, the difference of the two algorithms becomes smaller. For the memory usage, as illustrated in Figure 4.16 (b), PrefixSpan always consumes smaller memory than SPADE, and the difference between the two algorithms becomes smaller as $m$ increases. In summary, the overall performance of PrefixSpan is better than SPADE when the value of $m$ is smaller than 2.

Next we compare the algorithms of PrefixSpan and LAPIN_SPAM when the density $m$ is smaller than 2. The characteristic of the tested datasets is shown in Figure 4.16 (a). We change the density from 0.1 to 1.5. Figure 4.16 (c) illustrates the result. Specifically, when $m$ is small (i.e., 0.1), PrefixSpan is much faster than LAPIN_SPAM,. However, when $m$ is 1.5, LAPIN_SPAM is much faster than PrefixSpan. Note that when $m$ is 1, LAPIN_SPAM is about 2 times faster than PrefixSpan. For the memory usage, as illustrated in Figure 4.16 (c), PrefixSpan always consumes smaller memory than LAPIN_SPAM. However, the difference becomes smaller when $m$ increases and note that when $m$ is 1, LAPIN_SPAM consumes about two times memory of that used in PrefixSpan. In summary, the overall performance considering on execution and memory is that, when $m$ is smaller than 1, PrefixSpan is better than LAPIN_SPAM, otherwise (when $1< m <2$), LAPIN_SPAM is better.

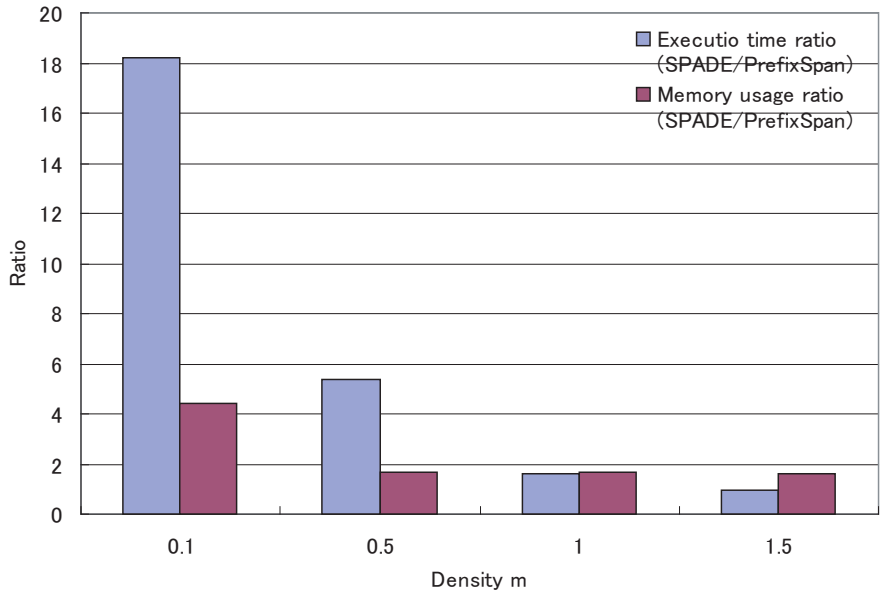
#### 4.3.5.5 LAPIN_PAID v.s. LAPIN_LCI v.s. LAPIN_SPAM

In this section, we first compared the algorithms of LAPIN_LCI and LAPIN_PAID when the density $m$ is larger than 2. The characteristic of the tested datasets is shown in Figure 4.17 (a). We change the density from 2 to 6. Figure 4.17 (b) illustrates the result. Specifically, LAPIN_PAID is always faster than LAPIN_LCI, by up to two times. For the memory usage, as illustrated in Figure 4.17 (b), LAPIN_PAID always consumes smaller memory than LAPIN_LCI. In summary, the overall performance of LAPIN_PAID is better than LAPIN_LCI though the difference is not so big, when the value of $m$ is larger than 2.

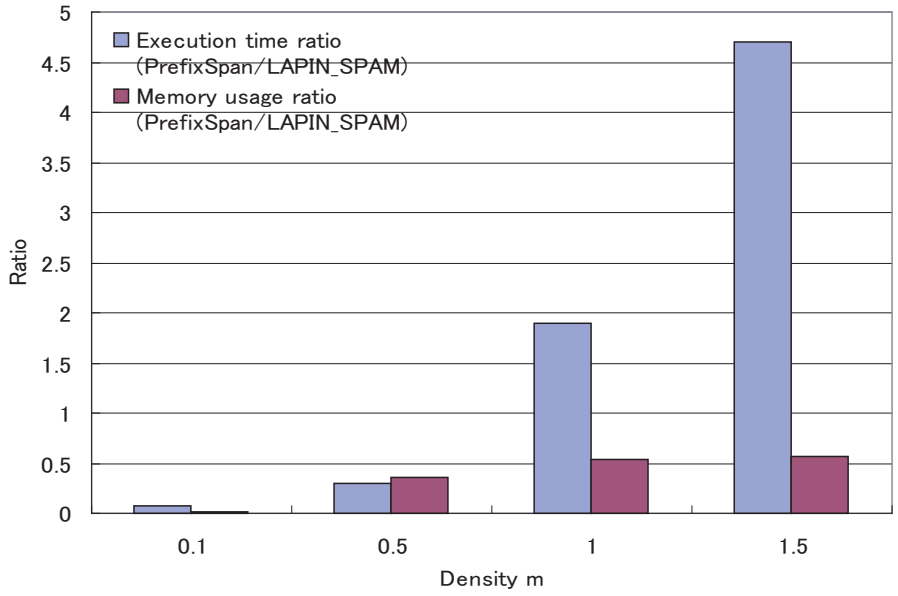
Next we compare the algorithms of LAPIN_PAID and LAPIN_SPAM when the

| Density (m) | Dataset |
|---|---|
| 0.1 | C10T10I5S5N1KD1K |
| 0.5 | C10T5I5S5N100D1K |
| 1 | C10T10I5S5N100D1K |
| 1.5 | C15T10I5S5N100D1K |

(a) Datasets used to test




(b) Performance comparison between SPADE and PrefixSpan




(c) Performance comparison between PrefixSpan and LAPIN_SPAM

Figure 4.16: Performance results of comparing PrefixSpan, SPADE and LAPIN_SPAM

density $m$ is larger than 2. The characteristic of the tested datasets is shown in Figure 4.17 (a). We change the density from 2 to 6. Figure 4.17 (c) illustrates the result. Specifically, when $m$ increases, LAPIN_SPAM is much faster than LAPIN_PAID by up to an order of magnitude. For the memory usage, as illustrated in Figure 4.17 (c), LAPIN_PAID always consumes smaller memory than LAPIN_SPAM. However, the difference is very small. In summary, the overall performance is that LAPIN_SPAM is better than LAPIN_PAID and LAPIN_LCI.

#### 4.3.5.6 Summary

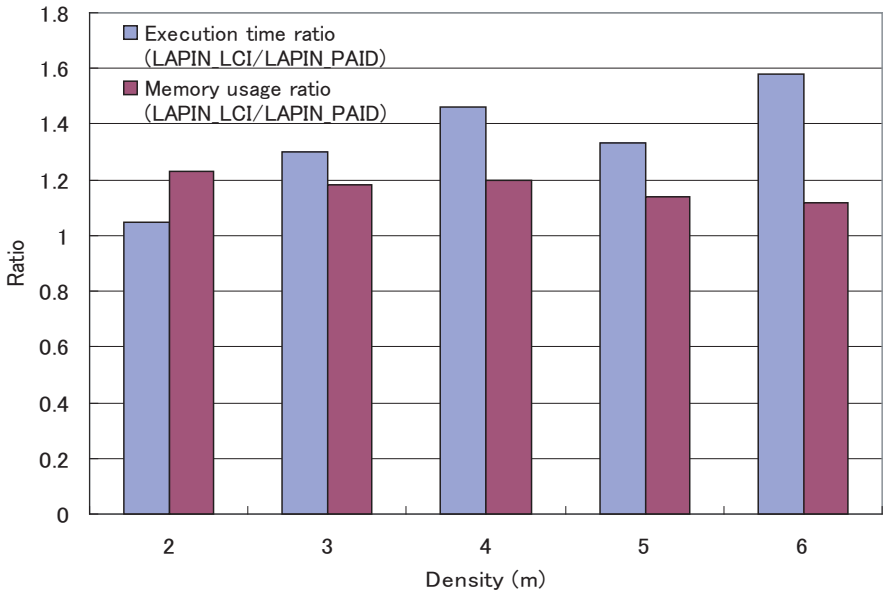
The above sections systemically studied the performance of all the algorithms. It illustrates that our LAPIN strategy is very efficient, if mining on dense datasets with long patterns. Furthermore, there are two issues should be mentioned:

- In resource limited environments, we observed that the criterion value of $m$ is around 2, which is used to judge whether use PrefixSpan or LAPIN_PAID. However, here we only give a rough criterion because of the complexity of the issue. It may be that when $m$ is smaller than 2, LAPIN_PAID is better than PrefixSpan. In other words, once the value of $m$ is large enough to eliminate the side-effect of LAPIN strategy (i.e., building the item_last_position_table), then LAPIN_SPADE will be better than PrefixSpan. From theoretical analysis, if $m$ is smaller than 1, PrefixSpan will definitely faster than LAPIN_PAID. Hence, the real criterion is indeed should be a range (i.e., $m$ is from 1 to 2). If $m$ is larger than this range, then LAPIN_PAID is better. Otherwise if $m$ falls into the criterion range, it is difficult to judge which algorithm is better unless to do the experiments. The reason is due to the different programming skill used in algorithm implementation.

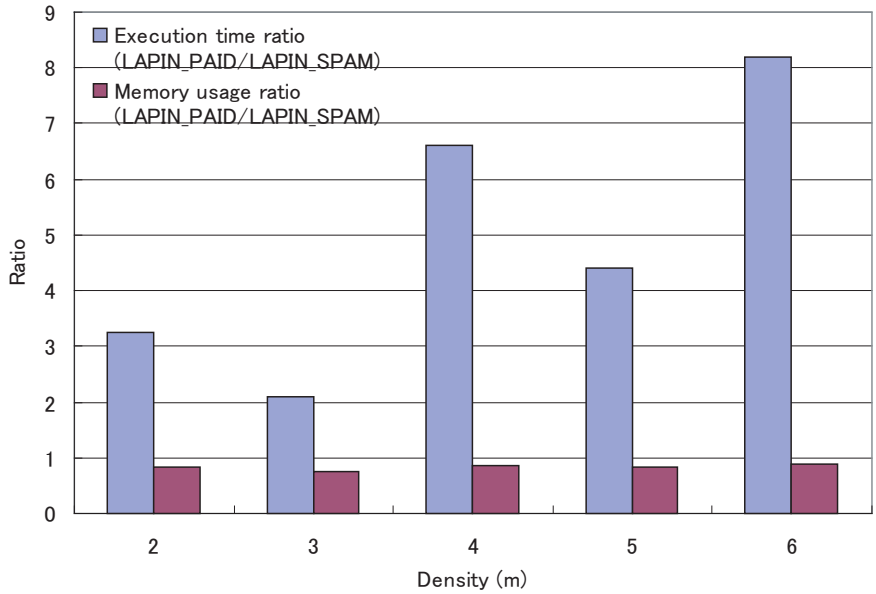
- The value of $m$ is computed as $C \times T/N$, as explained in Chapter 3, is under the assumption that the data distribution is uniform. We can make this assumption for synthetic datasets because the data generator creates only uniform distribution data. However, for real datasets, this assumption does not hold. That is the reason why for some real datasets with the density $m$ smaller than 1, our algorithms are still much faster than PrefixSpan and SPADE. The readers can see the experiments in Section 3.3.2 and Section 4.3.2 (i.e., Gazelle dataset). For these kinds of datasets, it is very difficult to judge the criterion. Fortunately, our

| Density (m) | Dataset |
|---|---|
| 2 | C20T10I5S5N100D1K |
| 3 | C30T10I5S5N100D1K |
| 4 | C20T20I5S5N100D1K |
| 5 | C50T10I5S5N100D1K |
| 6 | C30T20I5S5N100D1K |

(a) Datasets used to test




(b) Performance comparison between LAPIN_LCI and LAPIN_PAID




(c) Performance comparison between LAPIN_PAID and LAPIN_SPAM

Figure 4.17: Performance results of comparing LAPIN_PAID, LAPIN_LCI and LAPIN_SPAM

LAPIN family algorithms always show better performance on these real datasets, which confirm the efficiency of LAPIN strategy what we proposed.

# Chapter 5

# Applications of Sequential Pattern Mining

There are many applications that are based on the sequential pattern mining techniques, i.e., DNA sequence discovery, customer behavior analysis, stock trend prediction, etc. In this chapter, we study one typical application that how Web log systems can be beneficial from the improvement of basic sequential pattern mining algorithms.

## 5.1 Introduction of Web Log Mining

The World Wide Web has become one of the most important media to store, share and distribute information. At present, Google is indexing more than 8 billion Web pages [40]. The rapid expansion of the Web has provided a great opportunity to study user and system behavior by exploring Web access logs. Web mining that discovers and extracts interesting knowledge/patterns from Web could be classified into three types based on different data that mining is executed: Web Structure Mining that focuses on hyperlink structure, Web Contents Mining that focuses on page contents as well as Web Usage Mining that focuses on Web logs. In this thesis, we are concerned about Web Usage Mining (WUM), which also named Web log mining.

The process of WUM includes three phases: data preprocessing, pattern discovery, and pattern analysis [31].

### 5.1.1 Data Preparation

During preprocessing phase, raw Web logs need to be cleaned, analyzed and converted before further pattern mining. The data recorded in server logs, such as the user IP address, browser, viewing time, etc, are available to identify users and sessions. However, because some page views may be cached by the user browser or by a proxy server, we should know that the data collected by server logs are not entirely reliable. This problem can be partly solved by using some other kinds of usage information such as cookies. After each user has been identified, the entry for each user must be divided into sessions. A timeout is often used to break the entry into sessions. The following are some preprocessing tasks [31]: (a) Data Cleaning: The server log is examined to remove irrelevant items. (b) User Identification: To identify different users by overcoming the difficulty produced by the presence of proxy servers and cache. (c) Session Identification: The page accesses must be divided into individual sessions according to different Web users.

The raw Web log data is usually diverse and incomplete and difficult to be used directly for further pattern mining. In order to process it, we need to:

#### 5.1.1.1 Data Cleaning.

In our system, we use server logs in Common Log Format. We examine Web logs and remove irrelevant or redundant items like image, sound, video files which could be downloaded without an explicit user request. Other removal items include HTTP errors, records created by crawlers, etc., which can not truly reflect users' behavior.

#### 5.1.1.2 User Identification.

To identify the users, one simple method is requiring the users to identify themselves, by logging in before using the web-site or system. Another approach is to use cookies for identifying the visitors of a web-site by storing an unique ID. However, these two methods are not general enough because they depend on the application domain and the quality of the source data, thus in our system we only set them as an option. More detail should be implemented according to different application domains.

We have implemented a more general method to identify user based on [31]. We have three criteria:

(1) A new IP indicates a new user.

(2) The same IP but different Web browsers, or different operating systems, in terms of type and version, means a new user.

(3) Suppose the topology of a site is available, if a request for a page originates from the same IP address as other already visited pages, and no direct hyperlink exists between these pages, it indicates a new user. (option)

#### 5.1.1.3 Session Identification.

To identify the user sessions is also very important because it will largely affects the quality of pattern discovery result. A user session can be defined as a set of pages visited by the same user within the duration of one particular visit to a web-site.

According to [99] [80], a set of pages visited by a specific user is considered as a single user session if the pages are requested at a time interval not larger than a specified time period. In our system, we set this period to 30 minutes.

### 5.1.2 Pattern Discovery

The second phase of WUM is pattern mining and researches in data mining, machine learning as well as statistics are mainly focused on this phase. As for pattern mining, it could be: (a) statistical analysis, used to obtain useful statistical information such as the most frequently accessed pages; (b) association rule mining [4], used to find references to a set of pages that are accessed together with a support value exceeding some specified threshold; (c) sequential pattern mining [5], used to discover frequent sequential patterns which are lists of Web pages ordered by viewing time for predicting visit patterns; (d) clustering, used to group together users with similar characteristics; (e) classification, used to group together users into predefined classes based on their characteristics. In this thesis, we focus on sequential pattern mining for finding interesting patterns based on Web logs.

Sequential pattern mining, which extracts frequent subsequences from a sequence database, has attracted a great deal of interest during the recent surge in data mining research because it is the basis of many applications, such as Web user analysis, stock trend prediction, and DNA sequence analysis. Much work has been carried out on mining frequent patterns, as for example, in [5] [88] [109] [79] [7]. However, all of these works suffer from the problems of having a large search space and the ineffectiveness in handling long patterns. In Section 3 and Section 4, we proposed a family of novel

algorithms to reduce searching space greatly. Instead of searching the entire projected database for each item, as PrefixSpan [79] and SPADE [109] does, we only search a small portion of the database by recording the last position of item in each sequence (LAPIN: LAst Position INduction). While support counting usually is the most costly step in sequential pattern mining, the proposed LAPIN could improve the performance significantly by avoiding cost scanning and comparisons. In order to meet special features of Web data and Web log, we propose LAPIN_WEB by extending our previous work.

### 5.1.3 Pattern Analysis

In pattern analysis phase, which mainly filter out uninteresting rules obtained, we implement a visualization tool to help interpret mined patterns and predict users' future request.

We could see from pattern mining process that given a support, usually there are great number of patterns produced and effective method to filter out and visualize mined pattern is necessary. In addition, web-site developers, designers, and maintainers also need to understand their efficiency as what kind of visitors are trying to do and how they are doing it. Towards this end, we developed a navigational behavior visualization tool based on Graphviz [1].

At present, our prototype system has only implemented the basic sequential pattern discovery as the main mining task, which requires relevant simple user-computer interface and visualization. As more functions are added and experiment done, we will make the tool more convenient to the users.

## 5.2 LAPIN_Web Algorithm

### 5.2.1 General Idea

LAPIN algorithm, as introduced in the former chapters of this thesis, can be employed to mine Web log sequences. However, we can not get the best performance by directly applying LAPIN (LAPIN_PAID) to Web log ming because of the different properties between general sequence datasets and Web log datasets. Comparing with general

[1] http://www.research.att.com/sw/tools/graphviz

transaction data sequences that are commonly used, Web logs have following characteristics:

(a) no two items/pages are accessed at the same time by the same user.

(b) very sparse, which means that there are huge unique items and few item repetition in one user sequence.

(c) user preference should be considered during mining process.

Based on above points, we extended LAPIN (i.e., LAPIN_PAID) to LAPIN_WEB with:

(1) dealing with only Sequence Extension ($SE$) case, no Itemset Extension ($IE$) case.

(2) using sequential search instead of binary search. In more detail, LAPIN_WEB does not use binary search in the item position list, but use pointer+offset sequential search strategy, which is similar to that used in PrefixSpan.

(3) incorporating user preference into mining process to make the final extracted pattern more reasonable.

### 5.2.2 Implementation

We used a lexicographic tree [7] as the search path of our algorithm. Furthermore, we adopted a lexicographic order, which was defined in the same way as in [105]. This used the Depth First Search (DFS) strategy.

For Web log, because it is impossible that a user clicks two pages at the same time, Itemset Extension ($IE$) case in common sequential pattern mining does not exist in Web log mining. Hence, we only deal with Sequence Extension ($SE$) case. The pseudo code of LAPIN_WEB is shown in Figure 5.1. In Step 1, by scanning the DB once, we can obtain all the 1-length frequent patterns. Then we sort and construct the $SE$ item-last-position list in ascending order based on each 1-length frequent pattern' last position, as shown in Table 3.8.

**Definition 6** (Prefix border position set)**.** *Given two sequences, A=$\langle A_1 A_2 \ldots A_m \rangle$ and B=$\langle B_1 B_2 \ldots B_n \rangle$, suppose that there exists C=$\langle C_1 C_2 \ldots C_l \rangle$ for $l \leq m$ and $l \leq n$, and that C is a common prefix for A and B. We record both positions of the last item $C_l$ in A and B, respectively, e.g., $C_l$=$A_i$ and $C_l$=$B_j$. The position set, (i, j), is called the* prefix border position set *of the common prefix C, denoted as $\mathcal{S}_c$. Furthermore, we denote $\mathcal{S}_{c,i}$ as the prefix border position of the sequence, i. For example, if A=$\langle abc \rangle$*

Table 5.1: LAPIN_WEB Algorithm pseudo code

| | |
|---|---|
| INPUT: | A sequence database, and the minimum support threshold, $\varepsilon$ |
| OUTPUT: | The complete set of sequential patterns |
| Function: | Gen_Pattern($\alpha$, $S$, $CanI_s$) |
| Parameters: | $\alpha$ = length k frequent sequential pattern; $S$ = prefix border position set of (k-1)-length sequential pattern; $CanI_s$ = candidate sequence extension item list of length k+1 sequential pattern |
| Goal: | Generate (k+1)-length frequent sequential pattern |
| | |
| Main(): | |
| 1. | Scan DB once to do: |
| 1.1. | $B_s \leftarrow$ Find the frequent 1-length $SE$ sequences |
| 1.2. | $L_s \leftarrow$ Obtain the item-last-position list of the 1-length $SE$ sequences |
| 2. | For each frequent $SE$ sequence $\alpha_s$ in $B_s$ |
| 2.1. | Call Gen_Pattern ($\alpha_s$, 0, $B_s$) |
| | |
| Function: | $Gen\_Pattern(\alpha, S, CanI_s)$ |
| 3. | $S_\alpha \leftarrow$ Find the prefix border position set of $\alpha$ based on $S$ |
| 4. | $FreItem_{s,\alpha} \leftarrow$ Obtain the $SE$ item list of $\alpha$ based on $CanI_s$ and $S_\alpha$ |
| 5. | For each item $\gamma_s$ in $FreItem_{s,\alpha}$ |
| 5.1. | Combine $\alpha$ and $\gamma_s$ as $SE$, results in $\theta$ and output |
| 5.2. | Call Gen_Pattern ($\theta$, $S_\alpha$, $FreItem_{s,\alpha}$) |

*and B=$\langle acde \rangle$, then we can deduce that one common prefix of these two sequences is $\langle ac \rangle$, whose prefix border position set is (3,2), which is the last item C's positions in A and B.*

In function $Gen\_Pattern$, to find the prefix border position set of k-length $\alpha$ (Step 3), we first obtain the sequence pointer and offset of the last item of $\alpha$, and then perform a sequential search in the corresponding sequence for the (k-1)-length prefix border position. This method is similar to pseudo-projection in PrefixSpan, which is efficient for sparse datasets.

**Definition 7** (Local candidate item list)**.** *Given two sequences, A=$\langle A_1 A_2 \ldots A_m \rangle$ and B=$\langle B_1 B_2 \ldots B_n \rangle$, suppose that there exists C=$\langle C_1 C_2 \ldots C_l \rangle$ for $l \leq m$ and $l \leq n$,*

Table 5.2: Finding the SE frequent patterns

| | |
|---|---|
| INPUT: | $S_\alpha$ = prefix border position set of length k frequent sequential pattern $\alpha$; BV$_s$ = bit vectors of the ITEM_IS_EXIST_TABLE; $L_s$ = $SE$ item-last-position list; $CanI_s$ = candidate sequence extension items; $\varepsilon$ = user specified minimum support |
| OUTPUT: | $FreItem_s$ = local frequent $SE$ item list |
| 1. | For each sequence, F, according to its priority (descending) |
| 2. | $S_{\alpha,F} \leftarrow$ obtain prefix border position of F in $S_\alpha$ |
| 3. | if ($Size_{local\_cand\_item\_list} > Size_{suffix\_sequence}$) |
| 4. | bitV $\leftarrow$ obtain the bit vector of the $S_{\alpha,F}$ indexed from BV$_s$ |
| 5. | For each item $\beta$ in $CanI_s$ |
| 6. | Suplist[$\beta$] = Suplist[$\beta$] + bitV[$\beta$]; |
| 7. | $CanI_{s,p} \leftarrow$ obtain the candidate items based on prior sequence |
| 8. | else |
| 9. | $L_{s,F} \leftarrow$ obtain $SE$ item-last-position list of F in $L_s$ |
| 10. | M = Find the corresponding index for $S_{\alpha,F}$ |
| 11. | while ( M $<$ L$_{s,F}$.size) |
| 12. | Suplist[M.item]++; |
| 13. | M++; |
| 14. | $CanI_{s,p} \leftarrow$ obtain the candidate items based on prior sequence |
| 15. | For each item $\gamma$ in $CanI_{s,p}$ |
| 16. | if (Suplist[$\gamma$] $\geq \varepsilon$) |
| 17. | $FreItem_s$.insert($\gamma$); |

*and that C is a common prefix for A and B. Let $D = (D_1 D_2 \ldots D_k)$ be a list of items, such as those appended to C, and $C' = C \diamond D_j$ $(1 \leq j \leq k)$ is the common sequence for A and B. The list D is called the $local\ candidate\ item\ list$ of the prefix C'. For example, if A=$\langle abce \rangle$ and B=$\langle abcde \rangle$, we can deduce that one common prefix of these two sequences is $\langle ab \rangle$, and $\langle abc \rangle$, $\langle abe \rangle$ are the common sequences for A and B. Therefore, the item list (c,e) is called the $local\ candidate\ item\ list$ of the prefixes $\langle abc \rangle$ and $\langle abe \rangle$.*

Step 4, shown in Figure 5.1, is used to find the frequent $SE$ (k+1)-length pattern based on the frequent k-length pattern and the 1-length candidate items. Commonly,

support counting is the most time consuming part in the entire mining process. In Section 3, we have found that $LCI$-$oriented$ and $Suffix$-$oriented$ have their own advantages for different types of datasets. Based on this discovery, in this thesis, during the mining process, we dynamically compare the suffix sequence length with the local candidate item list size and select the appropriate search space to build a single general framework. In other words, we combine the two approaches, LAPIN_LCI and LAPIN_Suffix, together to improve efficiency at the price of low memory consuming. The pseudo code of the frequent pattern finding process is shown in Figure 5.2.

From a system administrator's view, the logs of special users (i.e. domain experts) are more important than other logs and thus, should be always considered more prior, as shown in Figure 5.2 (Step 1). The appended candidate items are also judged based on this criteria (Step 7 and Step 14).

## 5.3 Experimental Evaluation and Performance Study

In this section, we will describe our experiments and evaluations conducted on the real-world datasets. We performed the experiments using a 1.6 GHz Intel Pentium(R)M PC machine with a 1 G memory, running Microsoft Windows XP. The core of LAPIN_WEB algorithm is written in C++ software. When comparing the efficiency between LAPIN_WEB and PrefixSpan, we turned off the output of the programs to make the comparison equitable.

### 5.3.1 Datasets

We consider that results from real data will be more convincing in demonstrating the efficiency of our Web log mining system. There are two datasets used in our experiments, DMResearch and MSNBC. DMResearch was collected from the web-site of China Data Mining Research Institute [1], from Oct. 17, 2004 to Dec. 12, 2004. The log is large, about 56.9M, which includes 342,592 entries and 8,846 distinct pages. After applying data preprocessing described in Section 2.1, we identified 12,193 unique users and average length of the sessions for each user is 28. The second dataset, MSNBC, was obtained from the UCI KDD Archive [2]. This dataset comes from Web server logs

[1] http://www.dmresearch.net
[2] http://kdd.ics.uci.edu/databases/msnbc/msnbc.html

Table 5.3: Real Dataset Characteristics

| Dataset | # Users | # Items | Min. len. | Max. len. | Avg. len. | Total size |
|---|---|---|---|---|---|---|
| DMResearch | 12193 | 8846 | 1 | 10745 | 28 | 56.9M |
| MSNBC | 989818 | 17 | 1 | 14795 | 5.7 | 12.3M |

for msnbc.com and news-related portions of msn.com on Sep. 28, 1999. There are 989,818 users and only 17 distinct items, because these items are recorded at the level of URL category, not at page level, which greatly reduces the dimensionality. The 17 categories are ”frontpage”, ”news”, ”tech”, ”local”, ”opinion”, ”on-air”, ”misc”, ”weather”, ”health”, ”living”, ”business”, ”sports”, ”summary”, ”bbs”, ”travel”, ”msn-news”, and ”msn-sports”. Each category is associated with a category number using an integer starting from ”1”. The statistics of these datasets is given in Table 5.3.

## 5.3.2 Experiments and Evaluations

### 5.3.2.1 Comparing PrefixSpan with LAPIN_WEB

Figure 5.1 shows the running time and the searched space comparison between PrefixSpan and LAPIN_WEB. Figure 5.1 (a) shows the performance comparison between PrefixSpan and LAPIN_WEB for DMResearch data set. From Figure 5.1 (a), we can see that LAPIN_WEB is much more efficient than PrefixSpan. For example, at support 1.3%, LAPIN_WEB (runtime = 47 seconds) is more than an order of magnitude faster than PrefixSpan (runtime = 501 seconds). This is because the searched space of Prefixspan (space = 5,707M) was much larger than that in LAPIN_WEB (space = 214M), as shown in Figure 5.1 (c).

Figure 5.1 (b) shows the performance comparison between PrefixSpan and LAPIN_WEB for MSNBC data set. From Figure 5.1 (b), we can see that LAPIN_WEB is much more efficient than PrefixSpan. For example, at support 0.011%, LAPIN_WEB (runtime = 3,215 seconds) is about five times faster than PrefixSpan (runtime = 15,322 seconds). This is because the searched space of Prefixspan (space = 701,781M) was much larger than that in LAPIN_WEB (space = 49,883M), as shown in Figure 5.1 (d).

We have not compared PrefixSpan and LAPIN_WEB on user’s preference, because the former one has no such function.

#### 5.3.2.2 Visualization Result

To help web-site developers, and Web administrators analyze the efficiency of their web-site by understanding what and how visitors are doing on a web-site, we developed a navigational behavior visualization tool. Figure 5.2 and Figure 5.3 show the visualization result of traversal pathes for the two real datasets, respectively. Here, we set minimum support to 9% for DMResearch and 4% for MSNBC. The thickness of edge represents the support value of the corresponding traversal path. The number value, which is right of the traversal path, is the support value of the corresponding path. The ”start” and ”end” are not actual pages belong to the site, they are actually another sites placed somewhere on the internet, and indicate the entry and exit door to and from the site.

From the figures, We can easily know that the most traversed edges, the thick ones, are connecting pages ”start” → ”\loginout.jsp” → ”end” in Figure 5.2, and ”start” → ”frontpage” → ”end” in Figure 5.3. Similar interesting traversal path can also be understood, and used by web-site designers to make improvement on link structure as well as document content to maximize efficiency of visitor path.

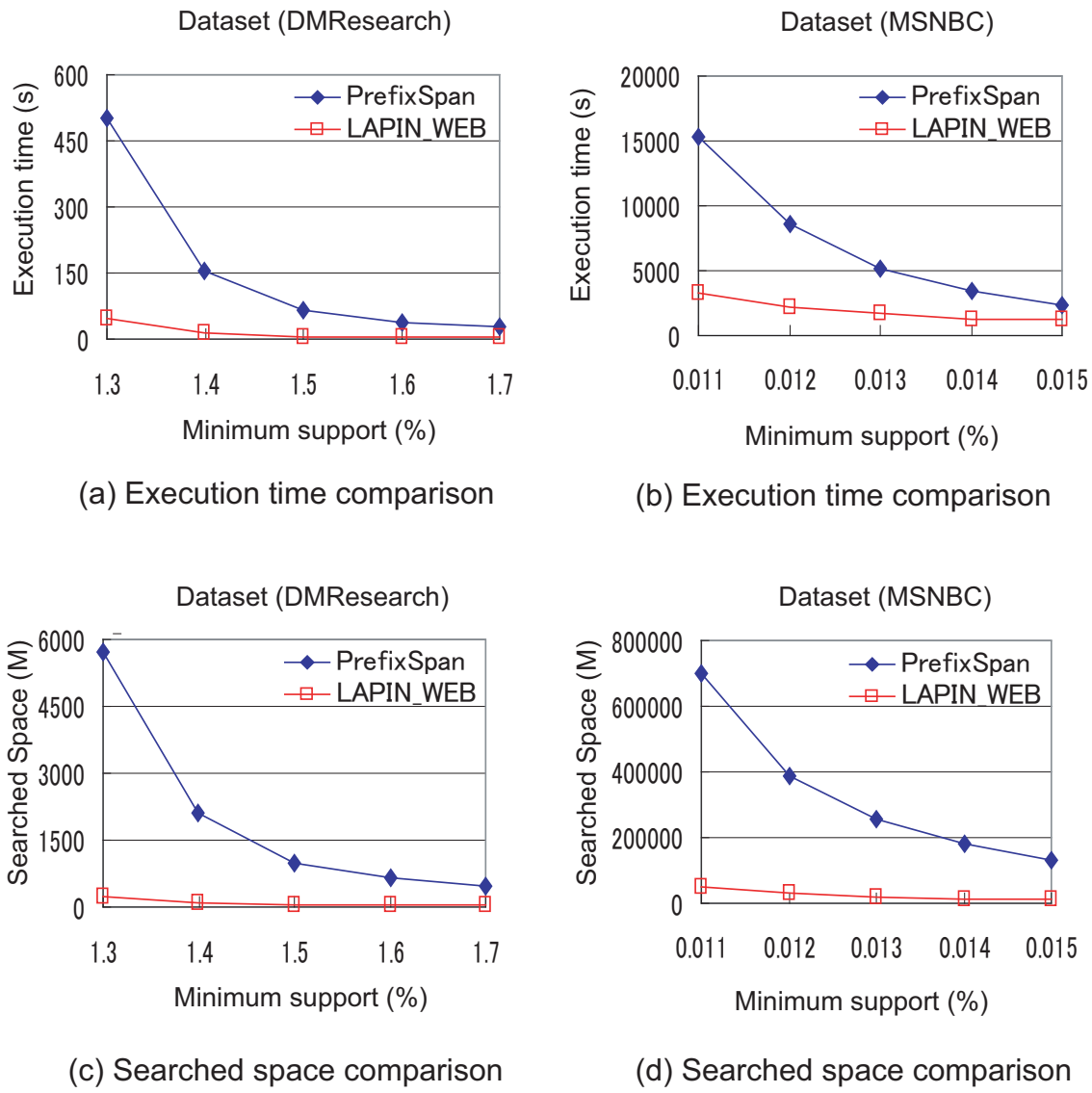



Figure 5.1: Real datasets comparison

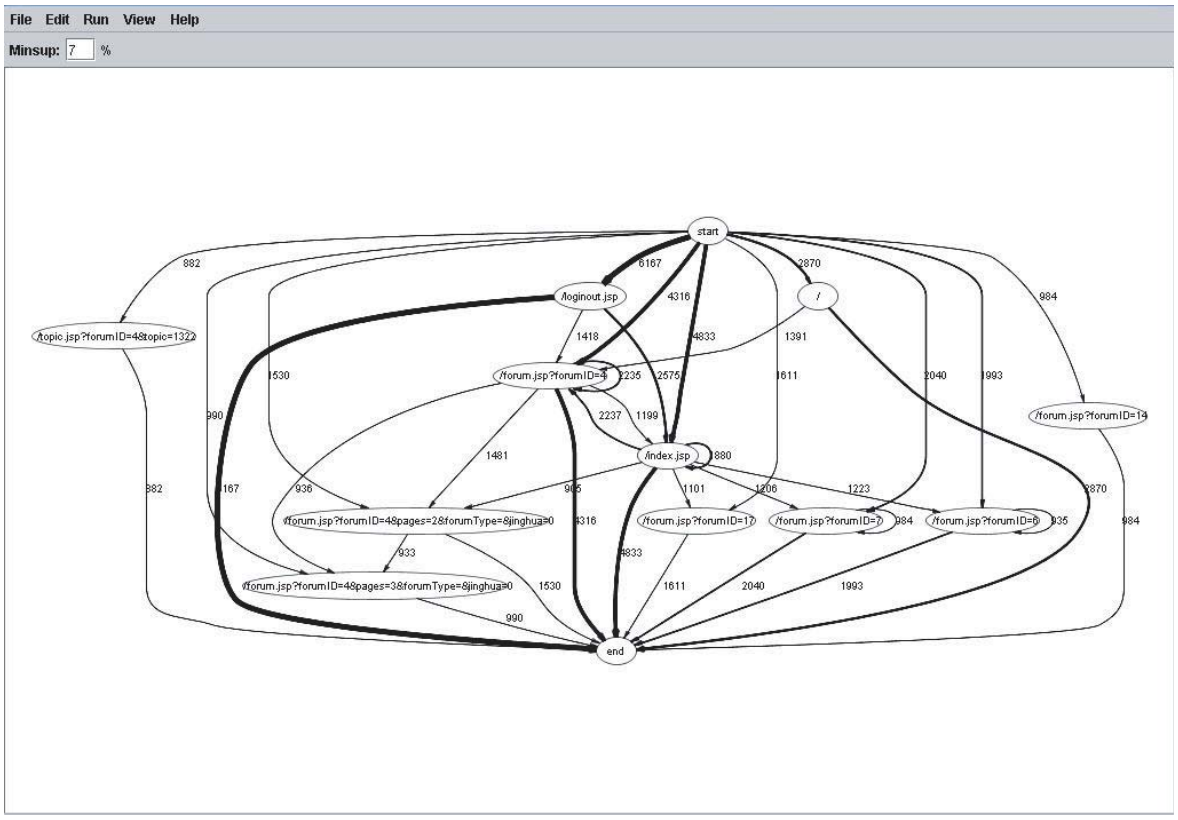



Figure 5.2: DMResearch visualization result

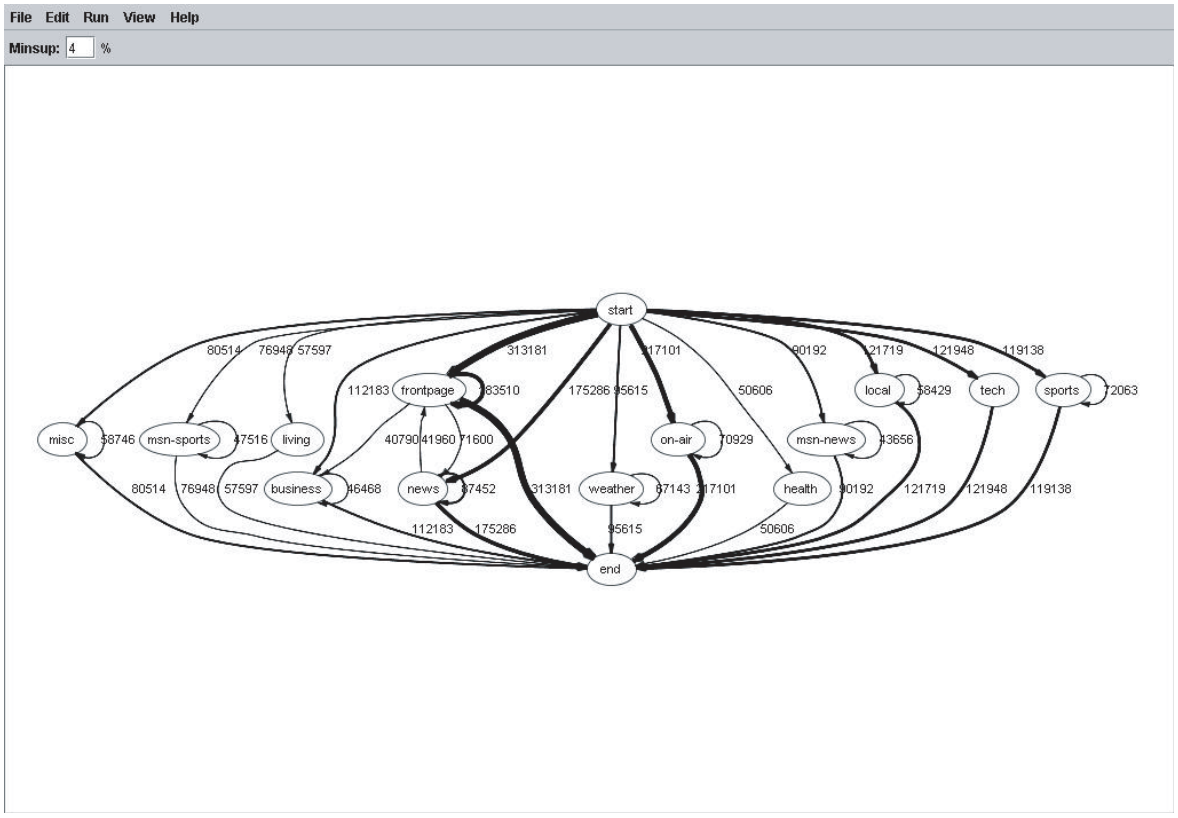



Figure 5.3: MSNBC visualization result

# Chapter 6

# Extension of Sequential Pattern Mining

## 6.1 Introduction of Skyline Query

Recently, the skyline query has attracted considerable attention because it is the basis of many applications, e.g., multi-criteria decision making [19], user-preference queries [50] [47] and microeconomic analysis [61]. Skyline mining [19] aims to find those points, which are not dominated by others, in a d-dimensional spatial dataset. This problem can be seen as a special class of pareto preference queries [50]. Figure 6.1 shows one classic example of skyline query that customers are always interested in those "best" hotels that are better than others at least at one of the two criteria, the distance and the price, with smaller values. The skyline of the example dataset in Figure 6.1 consists of $a$ and $c$.

Efficient skyline querying methodologies have been studied extensively, including the general full-space skyline points querying [19] [29] [54] [72], subspace skyline points mining [108] [92] [101], skyline points extracting in stream [62] [91] [69], Top-k and high-dimensional skyline points extracting [25] [24], mining skyline in distributed environments [10] [48] [100], approximate skyline querying [53].

All the above papers concerned only the pure dominant relationship among a dataset, i.e., a point $p$ is whether dominated by others or not, and got those non-dominated ones as results. However, in some real applications, users are more interested in the detail of the general dominant relationship in a business model, i.e., a point $p$ dominates how

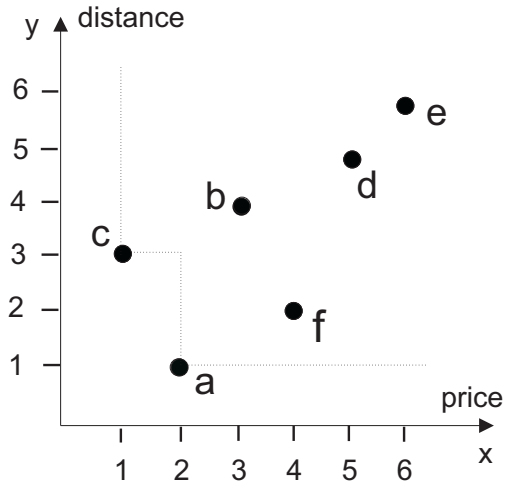



Figure 6.1: Example of the skyline query to find “best” hotels

many other points and is dominated by how many others. In Figure 6.1, although the hotel $b$ is not a skyline, its manager also wants to know how many hotels $b$ dominates (i.e. 2), how many hotels dominates $b$ (i.e. 2) and whom they are, from where the manager can know the business position of $b$ in the local area. Obviously, this kind of dominant relationship analysis requires more information explored than the original one of skyline query [19].

To illustrate our proposed core idea, here we show a simple example. Figure 6.2 is the corresponding partial order (encoded as DAG format) of Figure 1. We can know that item $b$ dominates items $d$ and $e$ and is dominated by items $a$ and $c$, by checking the $out$-$link$ and $in$-$link$ of $b$, respectively. Moreover, no $in$-$link$ nodes such as $a$ and $c$ are candidate items ($skyline$). From this example, we know that the general dominant relationships of a dataset can be represented into their corresponding partial order representation (i.e., DAGs). In this chapter we formally justify this discovery and moreover, explore how to efficiently find such succinct representative partial orders based on traditional sequential pattern mining techniques.

Different users may have different preferences (i.e., a user may issue either weight or sensor resolution combined with price as his preference). To cope with this problem, we propose a compressed data cube to encode all the possibilities in a concise format and devise efficient strategies to extract the information. There are some other issues (i.e., top-k query, querying on non-numerical attributes, etc) to be further discussed in

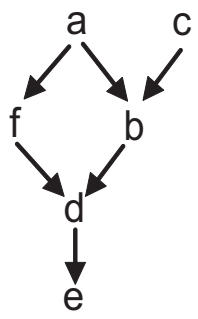



Figure 6.2: DAG representation in 2-d space

this thesis.

In this thesis, we aim at proposing efficient and effective methods to answer the general dominance relationship queries. Because of the interrelated connection between the partial order and the dominant relationship, we propose a new data structure called $ParCube$, which concisely represents the complete information of the general dominant relationship based on the partial order analysis. Specifically, we record the partial order as a Directed Acyclic Graph (DAG) for each cuboid in $ParCube$ and propose efficient data structures and strategies to answer the general dominant relationship queries. Furthermore, we introduce efficient strategies to construct and maintain $ParCube$. The experimental results and performance study confirms the efficiency and effectiveness of our strategies.

## 6.2 Related Work

### 6.2.1 Skyline Query

Skyline query was first introduced in [19]. The problem comes from some old classic topics, such as convex hull [83] and maximum vectors [56].

Skyline query algorithms can be classified into two categories. The first one is non-index based method, i.e., BNL [19], SFS [29], DC [19]. The second category is index based method, i.e., NN [54], BBS [72], SUBSKY [92]. As expected, the index-based methods have been shown to be superior over the non-index-based ones and furthermore, the index-based strategies can progressively return answers without

having to scan the entire data input.

BNL [19] uses a straightforward approach that compare each point $p$ with every other point to judge whether $p$ is skyline. It uses some optimizations, i.e., those points which are found to dominate multiple other points are likely to be checked first. SFS [29] uses pre-sorting technique to improve the efficiency of BNL. DC [19] divides the dataset into several partitions and mines in each one. Finally it gets the whole skyline by merging the partial results. Bitmap [90] applies bit-wise operations by encoding the dataset into bitmaps to improve efficiency.

Index [90] uses B-tree to organize the data into $d$ lists, where the $i$-th ($1 \leq i \leq d$) list contains points whose coordinates on the i-th axis are the smallest among all the dimensions ($minC$). It gets all skyline by scanning B-tree according to ascending order of $minC$. NN [54] combines nearest neighbor search and DC techniques together, and constructs a R-tree to index all the data. It also applies several methods to eliminate duplicate result. BBS [72] is the best algorithm for full space skyline discoverying. It only performs a single traversal of R-tree, which is different from NN's multiple queries. BBS processes the leaf nodes of their $mindist$ (minimum distance) in ascending order. Recently, SUBSKY [92] was proposed to compute low-dimensional Skylines. Based on the data distribution, SUBSKY creates an anchor point for each cluster, and builds a B+-tree on the $L_\infty$ distance between each object to its corresponding anchor. Then, SUBSKY scans the tree leaf nodes according to the ascending order of the points's smallest value of d-dimension to get Skylines.

From the view point of dimension concerned, the existing algorithms can be also classified into two categories, i.e., full space based method [54] [72], and subspace based method [92] [108] [101]. Other related work on skyline mining includes mining skyline in distributed environments [10] [48] [100], skyline query in data stream [62] [91] [69], approximate skyline query [53], interesting skyline points in high-dimensional space [25] [24].

All the above works concerned only the pure dominant relationship and, outputted those points which are not "dominated" by others. Note that in addition to the original meaning in [19], "dominated" here can be a variant, i.e., k-dominant [24].

In contrast, Li et al. proposed to analyze a more general dominant relationship from a microeconomic aspect [61]. The users are always interested in not only the binary dominant relation between the points in a dataset, but also the statistical information, i.e., how many other points are dominating/dominated by a specific point.

In [61], the authors proposed three basic $Dominant\ Relationship\ Queries$ ($DRQs$) and constructed a data cube, DADA, to efficiently organize the information necessary to $DRQs$. Moreover, a novel data structure, D*-tree, was proposed to fulfill efficient computation for $DRQs$.

However, in the real world, users are always interested in not only "how many" objects are dominating/dominated by a specific object, but also "whom" they are, which was however, not mentioned in [61]. Moreover, the real world is dynamic instead of static. These problems cannot be easily solved by using the methodologies proposed in [61] because of the large duplicate storage cost in DADA and naive updating scheme. In this chapter, we propose efficient data structure and strategies to solve such kind of general dominant relationship queries in dynamic environments based on our discovery that GDRQ has interrelated connection with partial order.

### 6.2.2 Sequential Pattern Mining

Sequential pattern mining has been attracted much attention after it was first introduced in [5] due to the broad applications it involves. Efficient mining methodologies have been studied extensively, including the general sequential pattern mining [7, 68, 79, 88, 109], constraint-based sequential pattern mining [39, 81], frequent episode mining [66], cyclic association rule mining [71], temporal relation mining [16], partial periodic pattern mining [44], and long sequential pattern mining in noisy environment [107].

In recent years many studies have presented convincing arguments that for mining frequent patterns (for both itemsets and sequences), one should not mine all frequent patterns but the closed ones because the latter leads to not only more compact yet complete result set but also better efficiency. Mining closed patterns has a direct connection with the elegant mathematical framework of formal concept analysis (FCA) [38]. Initial use of closed itemsets for association rules was studied in [75, 112]. Since then many algorithms for mining all the closed sets have been proposed [12, 21, 32, 78, 97, 111]. The idea of mining just closed sequential patterns instead of all frequent patterns stems from the parallel case of mining closed itemsets. There are two famous algorithms in closed sequential pattern mining, Clospan [105] and BIDE [96]. Both algorithms adopt the framework of PrefixSpan [79], which grows patterns by itemset extension and sequence extension. Clospan prunes the search space by sub/super sequences with equivalent projected databases, while BIDE prunes the

search space if the current pattern can be enumerated by other pattern.

In this chapter, we direct apply these existing efficient algorithms with several modification with regard to the special property of the skyline context.

### 6.2.3 Data Cube

The skyline query research is also related to the data cube problem [42] and recently, several scholars mentioned this [108] [101] [61]. Interestingly, as claimed in [61], "the dominant relationship between cells in the attribute space can be organized as a lattice structure as well", informally confirmed the possibility of combining the skyline dominant relationship analysis and closed sequential pattern mining together, because the formalization of closed sequential pattern mining is just based on Galois lattice [9] [22] [33]. This thesis can be seen as a generalization of [61], yet we use a total different strategy.

For the pure data cube computation, the concept of data cube was first proposed in [42]. Data cube computation has been an active research topic. A number of techniques have been introduced [2, 17, 42, 102, 113]. Specifically, several heuristics for computing multiple group-bys (i.e., cuboids) efficiently have been identified, such as smallest-parent, cache-results, amortizedscans, share-sorts, and share-partitions [84].

### 6.2.4 Partial Order Mining

Partial order has appeared, sometimes a little coyly, in many computational models. There are a lot of applications involves with partial order issues, such as concurrent models [58], optimistic rollback recovery [89], biology [59], security [87] and preference query [50].

In this thesis, we mainly consider the problem that how to convert the spatial dataset into partial order representation, which are then queried to get the general dominant relationship efficiently. As far as we know, there is no work on this problem. An interesting study investigated the problem of mining a small set of partial orders globally fitting data best [65]. Particularly, [65] addressed sequence data. Very different from the problem studied here, [65] tried to find one or a (small) set of partial orders that fit the whole dataset as well as possible, which is an optimization problem. An implicit assumption is that the whole dataset somehow follows a global order. More recently, [23] were intended for discovering several small partial orders from a set of sequences

instead of only one that describes all or most of the set. They proposed to use closed partial orders to summarize sequential data in a concise manner. Yet different from this thesis, they did not further explore the partial orders for a specific purpose (i.e., dominant relationship extraction).

### 6.2.5 Graph Construction

There has been a lot of research on inducing a graph structure based on the contents of tuples of a database. In [18], the authors proposed to treat the tuples as nodes, which are connected by edges induced by foreign key and other relationships. Answers to keyword queries are ranked using a notion of proximity coupled with the prestige of nodes based on incoming edges. There are some other related work in this direction, including [11] and [43]. Although very interesting, this line of work further reinforces the notion that while in the case of the web the structure of the web graph is apparent, there is no work done to convert the online relational data into partial order representation (DAGs).

In this chapter, we need to determinate the partial orders given a spatial dataset. We propose a simple method of converting the spatial dataset to the corresponding sequence dataset and then, apply existing strategies such as that used in [23] with modification by considering skyline property to generate the partial orders. Furthermore, we propose several optimization strategies when querying on the partial order representation models (DAGs).

Another related direction is the work on preference query systems. A framework for quantitative preference queries that rank answers based on scoring functions has been proposed in [6], and performance issues have been addressed in work such as [47]. Both [28] and [50] have separately proposed frameworks for qualitative preference queries that deal with binary preference relations between tuples. While there are several operators designed to evaluate preference queries (e.g., winnow operator [27] and Best Matches Only [50]), these schemes are designed for more general preference queries. Moreover, they require at least one scan through the dataset, making it unattractive for producing fast initial response time.

## 6.3 General Dominance Relationship Analysis

### 6.3.1 Preliminaries

Given a $d$-dimension space $S$=$\{s_1, s_2, \ldots, s_d\}$, a set of points $D$=$\{p_1, p_2, \ldots, p_n\}$ is said to be a dataset on $S$ if every $p_i \in D$ is a $d$-dimensional data point on $S$. We use $p_i.s_j$ to denote the $j^{th}$ dimension value of point $p_i$. For each dimension $s_i$, we assume that there exists a total order relationship. For simplicity and without loss of generality, we assume smaller values are preferred [19] (i.e., MIN operation) in this chapter.

**Definition 8** (dominate)**.** *A point $p$ is said to dominate another point $q$ on $S$ if and only if $\forall s_k \in S$, $p.s_k \leq q.s_k$ and $\exists s_t \in S$, $p.s_t < q.s_t$.*

A partial order on $D$ is a binary relation $\preceq$ on $D$ such that, for all $x$,$y$,$z \in D$, (i) $x \preceq x$ (reflexivity), (ii) $x \preceq y$ and $y \preceq x$ imply $x$=$y$ (antisymmetry), (iii) $x \preceq y$ and $y \preceq z$ imply $x \preceq z$ (transitivity). We use ($D$,$\preceq$) to denote the partial order set (or poset) of D. We denote by $\prec$ the strict partial order on $D$, i.e., $x \prec y$ if $x \preceq y$ and $x \neq y$. Given $x$,$y \in D$, $x$ and $y$ are said to be comparable if either $x \prec y$ or $y \prec x$; otherwise, they are said to be incomparable.

The Definition 1 can be translated into the ordering context as follows:

**Definition 9** (dominate in ordering context)**.** *A point $p$ is said to dominate another point $q$ on $S$ if and only if $\forall s_k \in S$, $p.s_k \preceq q.s_k$ and $\exists s_t \in S$, $p.s_t \prec q.s_t$.*

The partial order ($D$,$\preceq$) can be represented by a DAG $G = (D, E)$, where $(\upsilon, \omega) \in E$ if $\omega \preceq \upsilon$ and there does not exist another value $x \in D$ such that $\omega \preceq x \preceq \upsilon$. For simplicity and without loss of generality, we assume that $G$ is a single connected component.

**Definition 10** (dominating set, DGS(p, D, S'))**.** *Given a point p, we use DGS(p, D, S') to denote the set of points from D which are dominated by p in the subspace S' of S.*

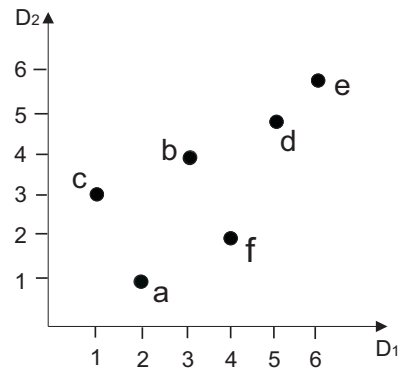
**Definition 11** (dominated set, DDS(p, D, S'))**.** *Given a point p, we use DDS(p, D, S') to denote the set of points from D which dominate p in the subspace S' of S.*

The problems that we want to solve are as follows:

**Problem 1** (General Point Query(GPQ))**.** *Given a dataset D, dimension space S' and a point p, find DGS(p, D, S') and DDS(p, D, S').*

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| $D_1$ | 2 | 3 | 1 | 5 | 6 | 4 |
| $D_2$ | 1 | 4 | 3 | 5 | 6 | 2 |
| $D_3$ | 3 | 1 | 6 | 2 | 5 | 4 |

(a) Example spatial dataset




(b) Representation in 2-d space {$D_1$,$D_2$}

Figure 6.3: Example spatial dataset

Note that GPQ is the generalized model of subspace analysis queries (SAQ)[61].

**Example 1.** *Consider the 3-dimensional dataset D = {a, b, c, d, e, f} in Figure 6.3 (a). Given a query point $b$, dimension space $S'$={$D_1$, $D_2$}, the dominating set DGS($b$, D, $S'$) = {d, e} and the dominated set DDS($b$, D, $S'$) = {a, c}. We will use this dataset as a running example in the rest of this chapter.*

### 6.3.2 General Idea

We find that the dominance relationship between two items in a $d$-dimensional dataset (i.e., $a$ dominants $b$) can be represented as a frequent sequence pattern in the corresponding $d$-customer sequence dataset. Because the small-large pair (dominant) relationship in the spatial dataset is equivalent to the early-late pair (dominant) relationship in the converted sequence dataset. For example, the example spatial dataset in Figure 6.3 (a) can be converted to a sequence dataset, by considering each dimension as a customer in the sequence dataset. The result dataset is D1: ⟨cabfde⟩, D2: ⟨afcbde⟩, D3: ⟨bdafec⟩. By employing sequential pattern mining algorithms, we can get the frequent patterns, i.e., (afde), which indicates that $a$ dominates $f$ and dominates $d$ and dominates $e$. The order in the pattern describes the dominance relationship between items. After we get the frequent sequence pattern, we merge them into partial orders, which is the concise model of the dominance relationship representation.

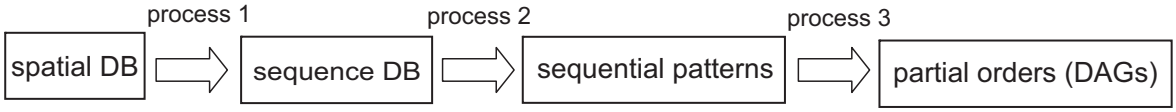



Figure 6.4: The work flow of ParCube constructing

### 6.3.3 Constructing Partial Order Data Cube (ParCube)

In this section, we explain how to construct the partial order data cube (ParCube) with a spatial dataset input. As far as we know, there is no work on this problem. In this thesis, we propose to apply strategies from another research context, sequential pattern mining [5], to get the partial order representation from a spatial dataset. The whole work flow is shown in Figure 6.4. We propose a simple method of converting the spatial dataset to the corresponding sequence dataset in the first process and then, apply existing strategies such as that used in [23] with little modification in the second and third processes to generate DAGs from the transformed sequence dataset. Note that we mainly illustrate how to compute the cube for a dominating set since computation of a dominated set can be done in a similar fashion.

The first process in Figure 6.4 is to convert the original spatial dataset to the sequence dataset. With a $k$-dimensional dataset, we simply get a $k$-customer sequence dataset, by sorting the objects in each customer (dimension) according to their value in ascending order. For example, Figure 6.5 (b) shows the converted sequence dataset of the example spatial dataset in Figure 6.5 (a). Note that the specific values of these points on k-dimension are resident on the disk. Efficient management methods (i.e., D*-tree [61]) are employed, as will be explained in later sections.

**Theorem 1.** *The converted sequence dataset records all the dominant relationship of the points in the spatial dataset.*

*Proof.* Trivial because the small-large pair (dominant) relationship in the spatial dataset is equivalent to the early-late pair (dominant) relationship in the converted sequence dataset. □

The second and the third processes in Figure 6.4 aim to determine a partial order that describes the point set in the subspace $S'$ of data space $S$ in $D'$. The related problem is addressed in [65] and more recently in [23]. In this chapter, we simply apply the approach in [23] with a minor modification that, instead of mining closed sequential

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| $D_1$ | 2 | 3 | 1 | 5 | 6 | 4 |
| $D_2$ | 1 | 4 | 3 | 5 | 6 | 2 |
| $D_3$ | 3 | 1 | 6 | 2 | 5 | 4 |

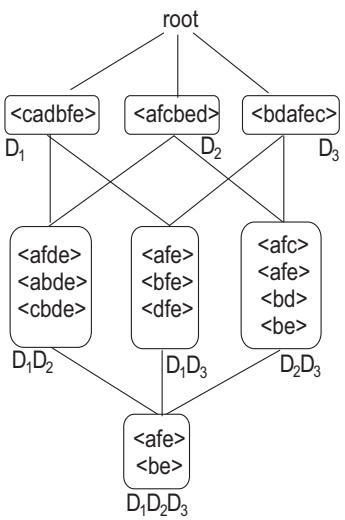
(a) Example spatial dataset

process 1

| Dim. | Sequence |
|---|---|
| $D_1$ | c a b f d e |
| $D_2$ | a f c b d e |
| $D_3$ | b d a f e c |

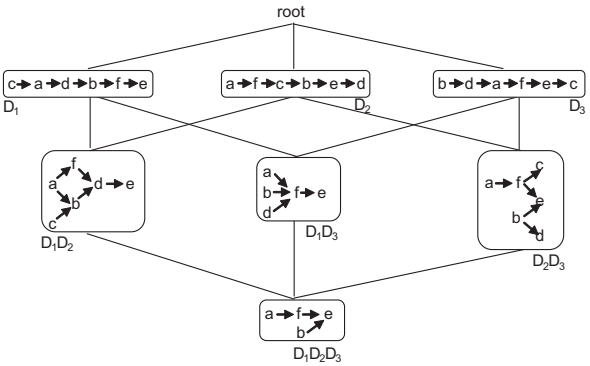
(b) Transformed sequence dataset

process 2




(c) The data cube (lattice) whose cuboid consists of the local maximal common sequential patterns

process 3




(d) DAG representation of partial order in data cube ParCube

Figure 6.5: The result representation of each process for the example spatial dataset

patterns [105], we mine general sequential patterns [5]. In process 2 as shown in Figure 6.4, we discovery the sequential patterns from the transformed sequence dataset by applying PrefixSpan algorithm [79]. Note that we can use any one of the state-of-the-art sequential pattern mining algorithms to do the task. To save space, we merge these sequential patterns as $local\ maximal\ sequential\ sequences$, which are not the subsequence of other sequential sequences in the same subspace. For example, in subspace $\{D_1, D_2\}$, although $\langle afd \rangle$, $\langle afe \rangle$, $\langle ade \rangle$ and $\langle fde \rangle$ are length-3 sequential patterns, we merge them as length-4 $local\ maximal\ sequential\ sequences$, as $\langle afde \rangle$. The result data cube ($SeqCube$) got from process 2 for the example dataset is shown in 6.5 (c).

**Theorem 2.** *SeqCube records all the dominant relationship of the points in the sequence dataset D.*

*Proof.* (Proof by Contradiction.) For simplicity, we only prove for a specific subspace of $SeqCube$. Assume to the contrary that there is a dominant relationship between two points, $a$ dominates $b$ in a subspace $S'$, is not represented in the cuboid $S'$ of $SeqCube$. This means that the sequential pattern $\langle ab \rangle$ is not listed in $S'$ of $SeqCube$, which contradicts our assumption that the sequential pattern mining process can find all the sequential patterns. □

In process 3, the combinations of the $local\ maximal\ sequential\ sequences$ are enumerated to generate partial orders with DAGs representation, by applying the method proposed in [23]. The result data cube ($ParCube$) got from process 3 for the example dataset is shown in 6.5 (d).

**Theorem 3.** *ParCube records all the dominant relationship of the points in the spatial dataset D.*

*Proof.* Proof can be deduced based on Theorem 1, Theorem 2 in this thesis and [23]. □

The pseudo code of constructing $ParCube$ is shown in Figure 6.1, where the three lines describe the three processes and can be justified based on Theorem 1, 2 and 3, respectively.

Table 6.1: Constructing ParCube

| | |
|---|---|
| INPUT: | The spatial dataset $D$ |
| OUTPUT: | The data cube $ParCube$ |
| 1. | Convert the spatial dataset $D$ to the corresponding sequence dataset $D'$ |
| 2. | Apply PrefixSpan [79] to get all the sequential patterns from $D'$, merge them to the $maximal\ sequential\ sequences$ as $SeqCube$ |
| 3. | Apply the algorithm proposed in [23] to get the partial order representation (DAGs) as $ParCube$ |

**Complexity analysis:** It is well known that to compute the (maximal) sequential pattern in process 2 is a #P-complete problem [106], which indicates no polynomial-time algorithms exist. Fortunately, we only execute $ParCube$ construction once as off-line preprocessing.

#### 6.3.3.1 Optimization of Sequential Pattern Mining

Among the three processes of partial orders finding as illustrated in Figure 6.5, the second one, sequential pattern mining, is the slowest process although the state-of-the-art algorithm is used. To improve the efficiency of the whole system, we aim to develop a new algorithm to fasten the mining process by considering the special property of the converted sequence datasets.

We find that the converted sequence dataset has one important characteristic: for each customer sequence (dimension), one item appears and only appears once. In other words, there is no two same items existing in the same customer sequence (dimension). This is very different from general sequence, i.e., Web log sequence, customer shopping history or DNA sequence. Based on this discovery, we have the following two lemmas:

**Lemma 6** (Transitivity)**.** *Let AB and BC be two sequential patterns in k-customer sequences with support 1, then AC should also be a frequent sequence with support 1 in the k-customer sequences.*

**Lemma 7** (Pattern Growth)**.** *Let AB and BC be two sequential patterns in k-customer sequences with support 1, then ABC should also be a frequent sequence with support 1 in the k-customer sequences.*

Table 6.2: Finding (k+1)-length frequent patterns with optimization

| | |
|---|---|
| INPUT: | $DB$ = the converted sequence DB |
| OUTPUT: | $FreMaxPatterns$ = frequent maximal sequential patterns |
| Function: | $Gen\_Pattern(S)$ |
| Parameters: | $S$ = Set of k-length frequent patterns |
| Goal: | Generate (k+1)-length frequent sequential pattern |
| | |
| Main(): | |
| 1. | $F_2$ = Scan $DB$ to find 2-length sequential patterns; |
| 2. | Call $Gen\_Pattern$ ($F_2$); |
| 3. | $FreMaxPatterns$ = Merge all the atoms in $F_i$; |
| | |
| Function: | $Gen\_Pattern(S)$ |
| 4. | For all atoms $A_i \in F_2$ |
| 5. | $T_i$ = $\emptyset$; |
| 6. | For all atoms $A_j \in F_2$, with $j \geq i$ |
| 7. | $R$ = $A_i \vee A_j$; |
| 8. | $T_i$ = $T_i \cup R$; |
| 9. | $F_{\|R\|}$ = $F_{\|R\|} \cup R$; |
| 10. | For all $T_i \neq \emptyset$ |
| 11. | Call $Gen\_Pattern$ ($T_i$); |

Based on Lemma 7, we can develop a much more efficient and simple algorithm to find the sequential patterns. The pseudo code of the algorithm is shown in Table 6.2.

Because every item (point) must exist in each customer sequence (dimension), we do not need to find 1-length patterns. In line 1, we thus directly find the 2-length sequential patterns. We scan each item's suffix database to accumulate the support of 2-length candidate sequences. Note that here we need to combine S-Step and I-Step together to fulfill the special property of dominant relation semantic meaning. Then in line 2, we recursively call the function Gen_Pattern to get those patterns whose length are larger than 2. We just merge two atoms together based on their prefix sequences. For example, when merging two patterns, i.e., $ab$ and $ac$, we need to check the existence of $bc$ or $cb$ in the frequent pattern list. The pattern $abc$ could be claimed if $bc$

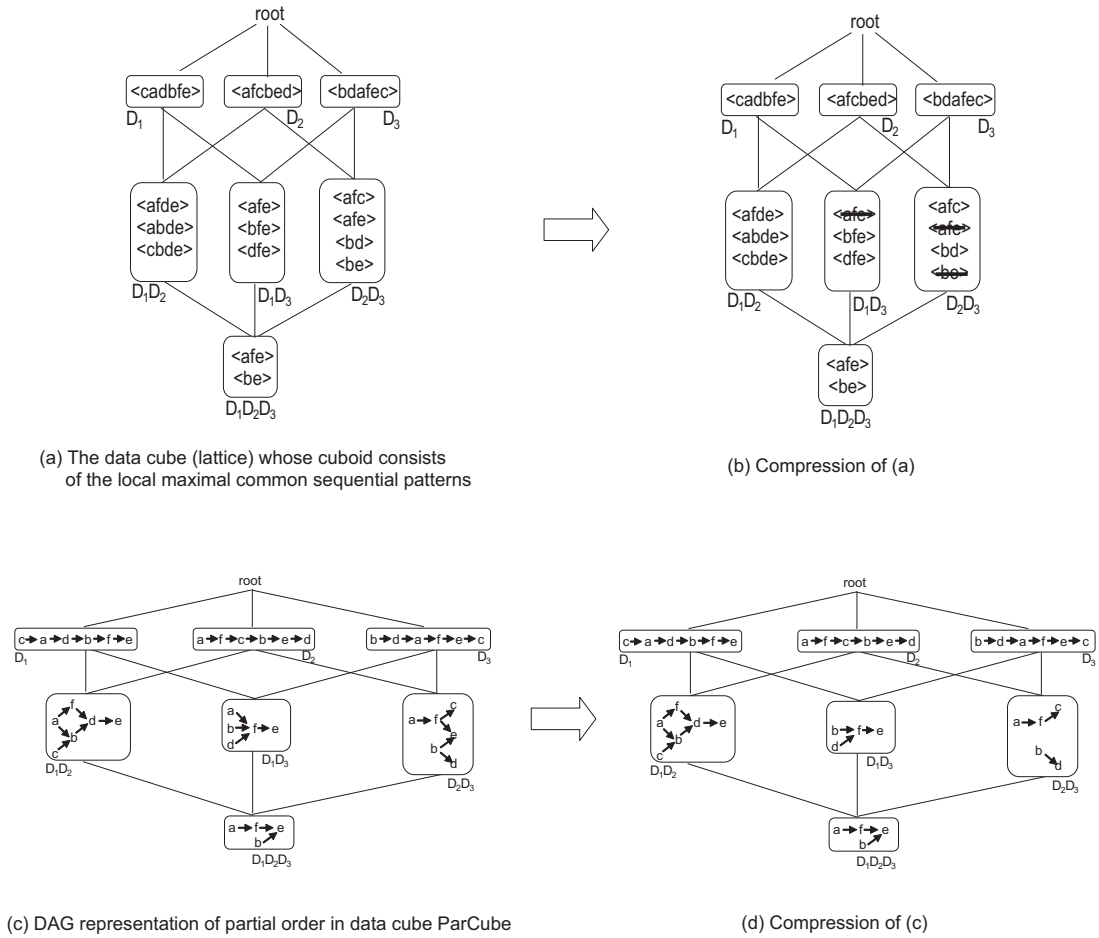



Figure 6.6: Compression of ParCube data cube

is found. By this way, we do not need to do candidate-generation-test operation in SPADE algorithm, or the Db projecting and scanning operation in PrefixSpan, which largely reduce the computation cost. The experiments in Section 6.4 illustrates the improvement of this strategy. In line 3, we merge these sequential patterns to get maximal ones.

#### 6.3.3.2 Compression of the ParCube Data Cube

The local maximal sequential sequences compress the data to some extent, we can further improve the compression by employing the technique of closed sequence [105] [22]. If a local maximal sequence $l$ exists in two subspaces, $S_1$ and $S_2$ where $S_1 \subset S_2$,

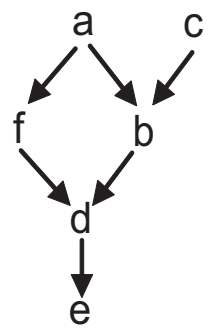



Figure 6.7: DAG representation of the example dataset in 2-dimensional space {$D_1$, $D_2$}

then $l$ is only recorded in $S_2$. The method used in Section 6.3.3 is unchanged and the result data cube which records the partial order of each cuboid is shown in Figure 6.6. For instance, although a sequence, $a \rightarrow f \rightarrow e$, exists in two subspaces $\{D_1,D_3\}$ and $\{D_1,D_2,D_3\}$, we only record it in the super-subspace, i.e., $\{D_1,D_2,D_3\}$. However, this kind of compression is obtained while paying for query time. The reason is that instead of getting all the dominant relationship in the local subspace, it should also check the super-subspace because some local sequences are absorbed by their super-subspace ones. Because the purpose of our strategy is to get a best query performance, in this thesis, we do not apply the technique of closed sequences.

### 6.3.4 Efficient ParCube Querying

In this section, we introduce the strategy to efficiently answer the general dominant relationship query. The semantic meaning kept in the $ParCube$ data cube is the key used to extract the general dominant relationship.

Given a dataset $D$, a query point $P_{query}$, and a subspace $S'$, the most basic GPQ is to compute the points that dominate or are dominated by $P_{query}$. We focus the case when the point $P_{query}$ is in the dataset $D$, $P_{query} \in D$.

- $P_{query} \in D$

An important observation in this case is that, if $P_{query}$ is in $D$, all the general dominant relationship related to $P_{query}$ can be easily discovered by traversing the DAG

in a specific subspace. No disk access is performed because we do not need to check their individual value of each dimension.

As an example, Fig. 6.7 shows the DAG representation in subspace $\{D_1, D_2\}$. To facilitate the counting process, the numbers of points dominating/dominated by current nodes (points) are inserted into each node. This process is executed in the precomputed-mode. Suppose the query point is $b$, we can get the points dominated by $b$ immediately, which is 2. For users who are interested in knowing where these two points are, they go downward following the out-link of $b$, and gets the dominating set of $b$ as $\{d, e\}$.

## 6.4 Experimental Evaluation and Performance Study

To evaluate the efficiency and effectiveness of our strategies, we conducted extensive experiments. We performed the experiments using a Intel(R) Core(TM) 2 Dual CPU PC (3GHz) with a 3G memory, running Microsoft Windows XP. All the algorithms were written in C++, and compiled in an MS Visual C++ environment. We conducted experiments on both synthetic and real life datasets.

Detailed implementation of the algorithms used to compare is described as follows:

1. *SUBSKY. SUBSKY* was tested with the algorithm developed in [92], which is the state-of-the-art algorithm for subspace skyline query.

2. *BBS*$^+$. *BBS*$^+$ was tested with the modification of BBS algorithm [72] which takes the characteristic of dominance relationship query into account.

3. *ParCube. ParCube* was implemented as described in this thesis. The optimization strategy has been introduced in Section 6.3.3.1.

### 6.4.1 Datasets

We employ the synthetic data generator [19] to create our synthetic datasets. They have $independent$ distribution, with dimensionality $d$ in the range [3, 6] and data size in the range [10k, 50k]. The default values of dimensionality were 5. The default value of cardinality for each dimension was 50k.

### 6.4.2 Skyline Query Performance

In this section, we evaluated the query answering performance of $ParCube$ compared with the state-of-the-art algorithm, $SUBSKY$ [92].

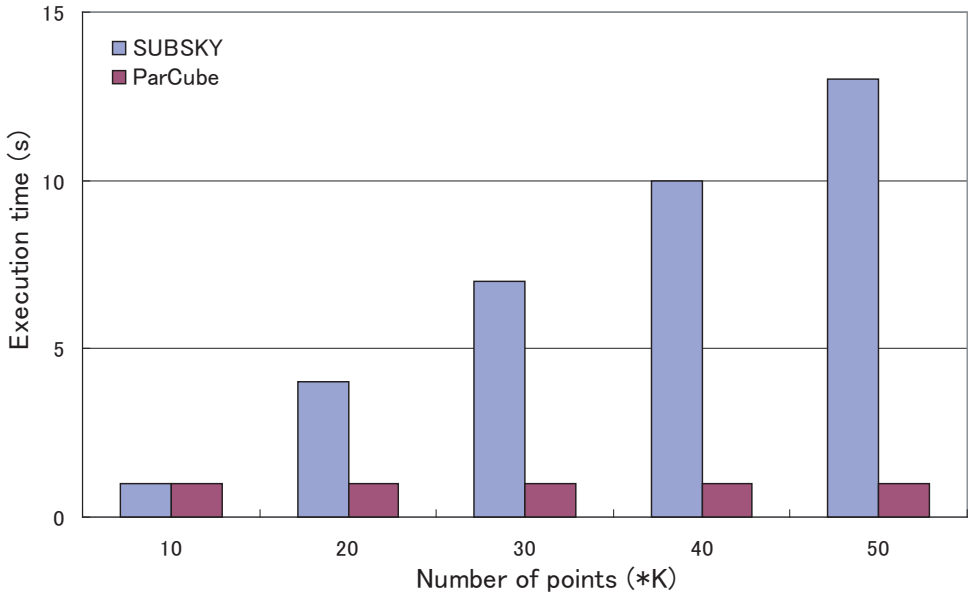
Figure 6.8(a) and Figure 6.8 (b) show the skyline query time against number of points in the datasets and dimensionality, respectively. We can see that the $ParCube$ algorithm outperforms the $SUBSKY$ in both cases by up to an order of magnitude. This is because the $SUBSKY$ algorithm needs to traverse the tree data structure (i.e., B-tree) to extract the skyline on the fly. On contrary, $ParCube$ pre-computes and stores the skyline points into partial order data structure, which can be easily extracted out because they exist in the first layer of DAG graph. Moreover, from the figures we can know that dimensionality has more effect on query performance compared with the number of points in the datasets.

### 6.4.3 Dominant Relationship Query Performance

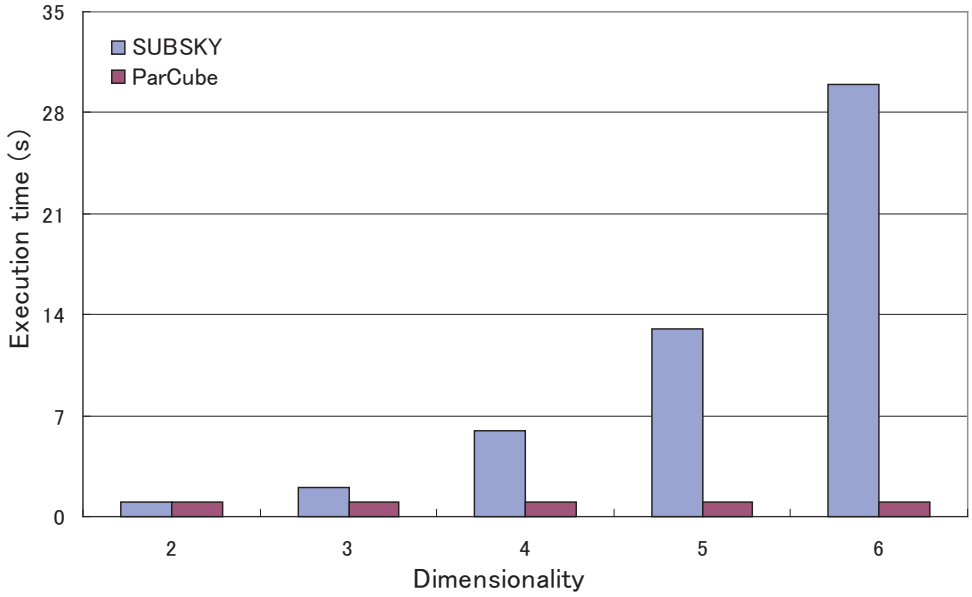
To test the effect of General Dominant Relationship query ($GDRQ$), we randomly generated 100 different points based on the synthetic dataset and get the final execution time as the average time of the 100 points. Figure 6.9 (a) and 6.9 (b) show the query time against number of points in the datasets and dimensionality, respectively. We can see that the $ParCube$ approach is better than $BBS^+$. The performance of $BBS^+$ becomes worse as number of points or dimensionality is larger, while $ParCube$ remains almost the same. The reason is similar to that explained in Section 6.4.2. $BBS^+$ needs to traverse the index data structure (i.e., R-Tree) to compare and extract all the required points. In contrast, $ParCube$ only traverse the DAG graph to direct extract every node it passed and no comparison is necessary.

### 6.4.4 Index Data Structure Construction Performance

The efficiency of $ParCube$ is rooted in the compressed data structure it discoveries, partial order data cube ($ParCube$). In this section, we show the construction time for $ParCube$ with optimization (which is introduced in Section 6.3.3.1.) compared with cost of building other index data structure (i.e., R-Tree) in the $BBS^+$ algorithm. Figure 6.10 (a) and 6.10 (b) show the execution time for index building against number of points in the datasets and dimensionality, respectively. We can see that the $ParCube$ is

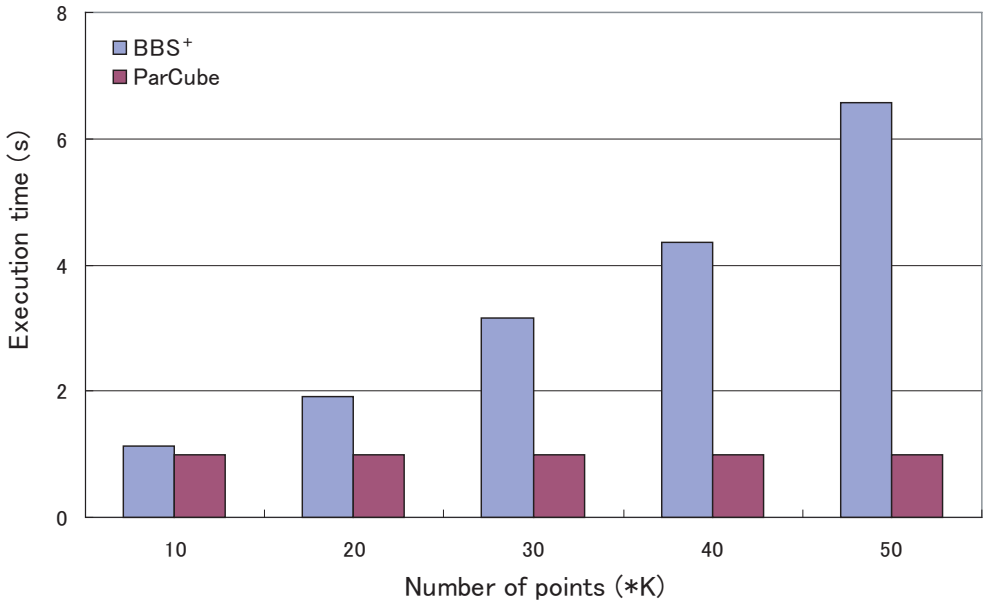



(a) Execution time comparison on Skyline query against number of points (dimensionality=5)

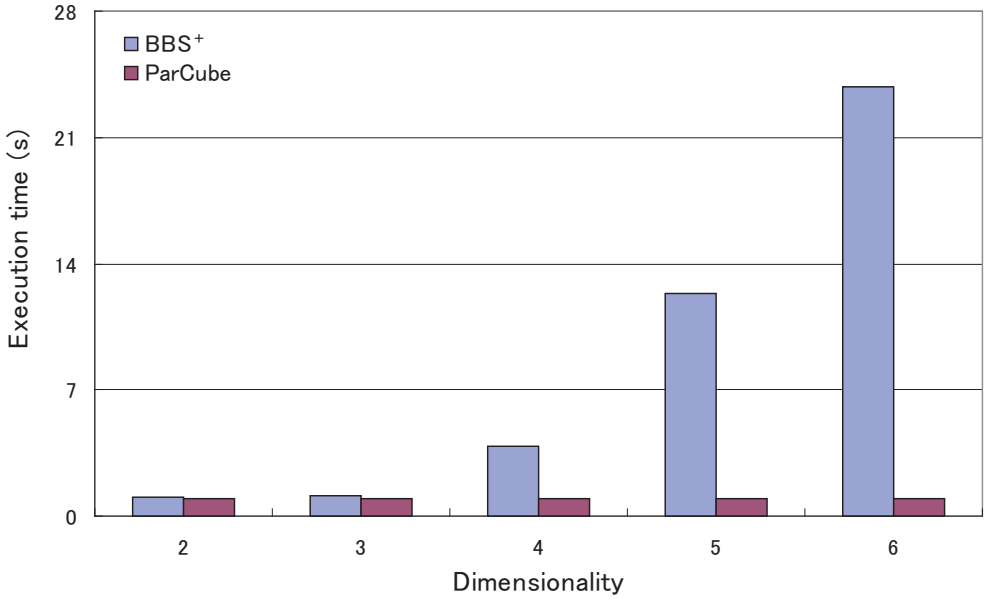



(b) Execution time comparison on Skyline query against dimensionality (point number=50k)

Figure 6.8: Execution time comparison between $SUBSKY$ and $ParCube$ on skyline query




(a) General dominant relationship query against number of points(dimensionality=3)




(b) General dominant relationship query against dimensionality(point number=10K)

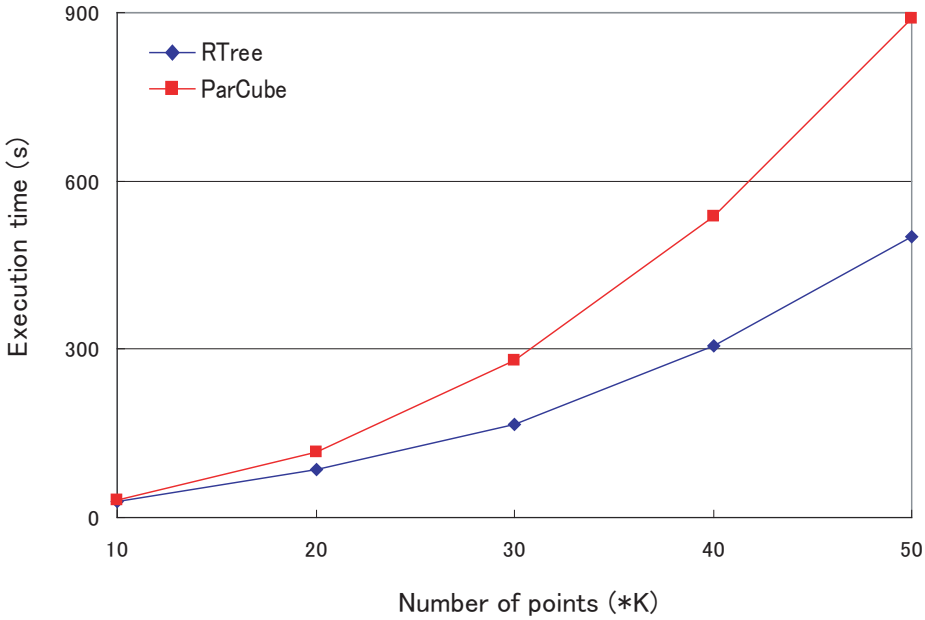
Figure 6.9: Execution time comparison between $BBS^+$ and $ParCube$ on general dominant relationship query

sensitive to the number of points in the datasets, that when the number gets larger, the performance of $ParCube$ construction is much worse than that of $R$-$Tree$ building. However, as illustrated in 6.10 (b), $R$-$Tree$ construction becomes worse as dimensionality grows, which means that $R$-$Tree$ index building is more sensitive to the dimensionality compared with $ParCube$ index building. The reason why the performance of $ParCube$ construction is good, because in high dimensional space, the probability of one point dominates another one, is very low. Hence, the sequential pattern is very few in high dimensional space and the mining process can terminate quickly. In summary, the two index structure have their own advantages and disadvantages, and may fail on building large datasets (i.e., high dimensionality and large number of points). In the future, we should consider on how to efficient index large datasets.

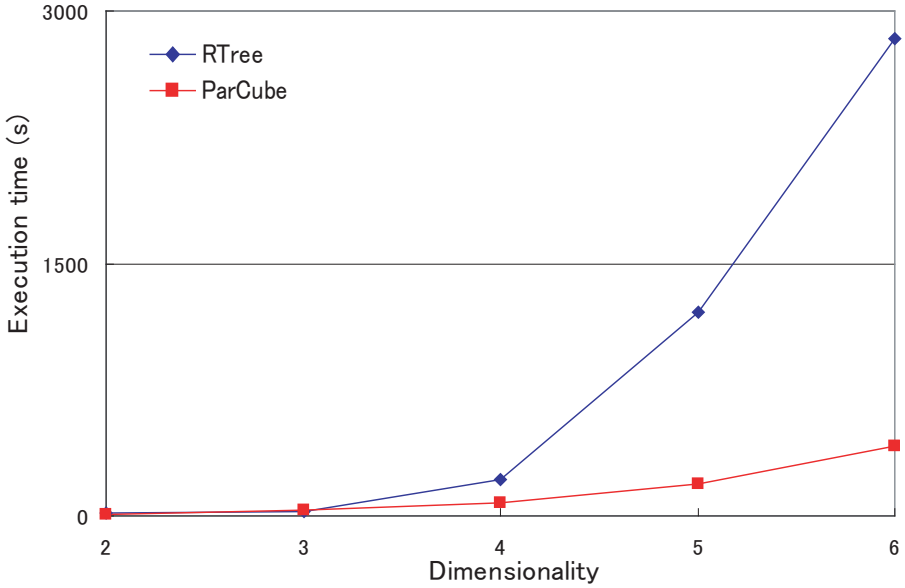
### 6.4.5 Effectiveness of Compression

In this experiment, we explored the compression benefits of $ParCube$ compared with $R$-$Tree$ method.

Figure 6.11 (a) and Figure 6.11 (b) show the compression effect on building the data cube by partial order representation ($ParCube$), compared with $R$-$Tree$. They illustrate that using the compressed data format, DAG, is very efficient on space usage. Similar to query performance, dimensionality has more effect on the compression factor compared with the number of points in the datasets. However, although $ParCube$ largely reduces the space used to store the information, it still consumes a lot. For example, for the dataset (point=10K, dimension=6) whose size is about 409K, $ParCube$ requires 140M to store the partial order information. This will becomes impractical for large datasets. In the future, we would like to reduce more space necessary.

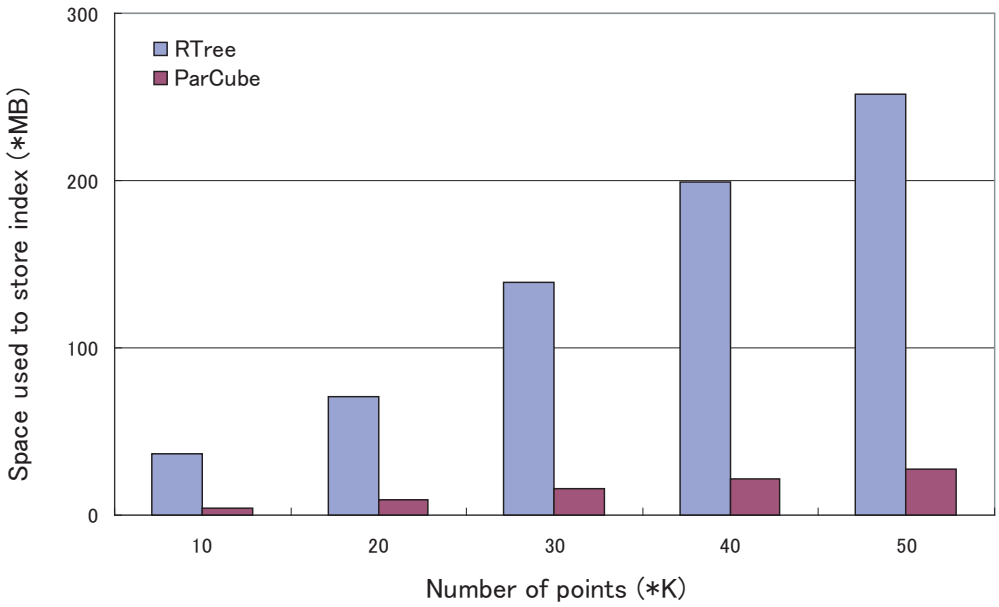



(a) Index build execution time against number of points(dimensionality=3)

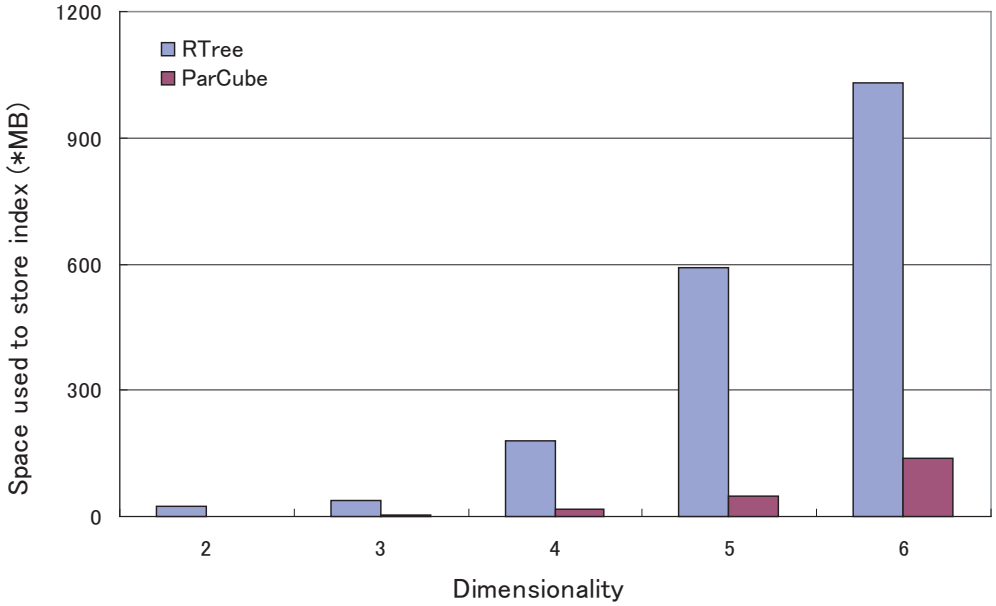



(b) Index build execution time against dimensionality(number of points=10K)

Figure 6.10: Execution time comparison between $R$-$Tree$ building and $ParCube$ construction




(a) ParCube compression effect against point number
(dimensionality=3)




(b) ParCube compression effect against dimensionality
(point number=10K)

Figure 6.11: Compression effect of ParCube against dimensionality and number of points in datasets

# Chapter 7

# Discussion

## 7.1 Extension of Sequential Pattern Mining

Comparing with mining (unordered) frequent (itemset) patterns, mining sequential patterns is one step toward mining more sophisticated frequent patterns in large databases. With the successful development of the sequential pattern mining method, LAPIN, it is interesting to explore how such a method can be extended to handle more sophisticated cases. In this chapter, we will discuss several problems related to the sequential patterns.

### 7.1.1 Constraint-based Mining of Sequential Patterns

For many sequential pattern mining applications, instead of finding all the possible sequential patterns in a database, a user may often like to enforce certain constraints to find desired patterns. The mining process which incorporates user-specified constraints to reduce search space and derive only the user-interested patterns is called constraint-based mining.

Constraint-based mining has been studied extensively in frequent pattern mining, such as [14, 70, 77]. In general, constraints can be characterized based on the notion of monotonicity, anti-monotonicity, succinctness, as well as convertible and inconvertible constraints, respectively, depending on whether a constraint can be transformed into one of these categories if it does not naturally belong to one of them. This has become a classical framework for constraint-based frequent pattern mining.

### 7.1.2 Mining Closed and Maximal Sequential Patterns

A frequent long sequence contains a combinatorial number of frequent subsequences. For a sequential pattern of length 100, there exist $2^{100} - 1$ nonempty subsequences. In such cases, it is prohibitively expensive to mine the complete set of patterns no matter which method is to be applied.

Similar to mining closed and maximal frequent patterns in transaction databases [13, 75], which mines only the longest frequent patterns (in the case of max-pattern mining) or the longest one with the same support (in the case of closed-pattern mining), for sequential pattern mining, it is also desirable to mine only (frequent) maximal or closed sequential patterns, where a sequence $s$ is maximal if there exists no frequent supersequence of $s$, while a sequence $s$ is closed if there exists no supersequence of $s$ with the same support as $s$.

The development of efficient algorithms for mining closed and maximal sequential patterns in large databases is an important research problem. A related study in [105] proposed an efficient closed sequential pattern method, called CloSpan, as a further development of the PrefixSpan mining framework. In a similar way, our algorithms can be extended to efficiently mine closed sequential patterns.

### 7.1.3 Mining Approximate Sequential Patterns

In this study, we have assumed all the sequential patterns to be mined are exact matching patterns. In practice, there are many applications that need approximate matches, such as DNA sequence analysis which allows limited insertions, deletions, and mutations in their sequential patterns. The development of efficient and scalable algorithms for mining approximate sequential patterns is a challenging and practically useful direction to pursue. A related study on mining long sequential patterns in a noisy environment [107] is a good example in this direction.

### 7.1.4 Sequential Patterns Compression

The complete set of frequent patterns is often huge in number, which makes the interpretability of frequent patterns very difficult. The concepts of closed frequent patterns and maximal frequent patterns usually can help in reducing the output size. However, they can only partly alleviate the problem. The size of closed frequent patterns (or

maximal frequent patterns) often remains to be very large and thus it is still difficult for users to examine and understand them.

Recently, several proposals were made to discover $k$ patterns or profiles. This allows users to specify the value of $k$ and thus only discover a small number of patterns or approximation. The concept of top-k patterns is proposed by Han et al. [45]. Although this provides users the option to discover only the $k$ most frequent patterns, this is not a generalization of all frequent patterns satisfying a support threshold. $k$ covering sets was proposed by Afrati et al. [1] to approximate a collection of frequent patterns, i.e. each frequent pattern is covered by at least one of the k sets. The proposal is interesting in generalizing the collection of patterns into $k$ sets. However, the support information is ignored in the approximation and it is unknown how to recover the support of a pattern from the $k$ sets. Support is a very important property of a pattern and plays a key role in distinguishing patterns.

Yan et al. [103] proposed an approach to summarizing patterns into $k$ profiles by considering both pattern information and support information; each cluster (profile) is represented with three elements: the master pattern, i.e. the union of the patterns in the cluster, the number of transactions supporting the clusters, the probability of items of the master pattern in the set of transactions supporting the pattern. The supports of frequent patterns can be estimated from the k clusters. It is assumed in [103] that the items in the master pattern are independent in each profile. Cong et al. [30] adopt an alternative probability model to represent a profile composed of a set of frequent patterns. The authors consider the pairwise probabilities that are still easy to compute. From the pairwise probabilities, the authors build simple Bayesian Network to estimate the $n$-dimensional probability distribution, and thus can estimate the supports of the patterns.

All the methods mentioned above deal with itemset patterns. As far as we know, there is no sequential patterns compression technique developed. However, because of the inter-relation between Frequent Itemset Pattern and Frequent Sequential Pattern, we can utilize the strategies introduced above to effectively compress the sequential patterns.

### 7.1.5 Sequential Pattern Mining Over Data Stream

Recently, the data mining community has focused on a new challenging model where data arrives sequentially in the form of continuous rapid streams. It is often referred to as data streams or streaming data. Since data streams are continuous, high-speed and unbounded, it is impossible to mine sequential patterns by using algorithms that require multiple scans.

Mining sequential patterns over data streams is a new research problem in data mining. We can do this job based on the success of itemset mining over stream data.

### 7.1.6 Toward Mining Other Kinds of Structured Patterns

Besides mining sequential patterns, another important task is the mining of frequent substructures in a database composed of structured or semistructured data sets. The substructures may consist of trees, directed-acyclic graphs (i.e., DAGs), or general graphs which may contain cycles. There are a lot of applications related to mining frequent substructures since most human activities and natural processes may contain certain structures, and a huge amount of such data has been collected in large data/information repositories, such as molecule or biochemical structures, Web connection structures, and so on. It is important to develop scalable and flexible methods for mining structured patterns in such databases. There have been some recent work on mining frequent subtrees, such as [110], and frequent subgraphs, such as [57, 104], in structured databases. The strategies can be learned to extend our algorithms to efficiently mine closed subgraph and subtree patterns.

## 7.2 Extension of Skyline Mining

Skyline mining can be seen as the specific case of general dominance relationship analysis. In this section, we will introduce several novel variations of skyline queries.

### 7.2.1 Ranked Skyline Queries

Ranked skyline query was first proposed in [72]. Given a set of points in the $d$-dimensional space $[0, 1]^d$, a ranked (top-$K$) skyline query (i) specifies a parameter $K$, and a preference function $f$ which is monotone on each attribute, (ii) and returns the

$K$ skyline points $p$ that have the minimum score according to the input function. Because the change of the preference function, the dominant relationship between points is also changed. For the special case of dominant relationship, skyline, consider the running example as shown in Figure 6.1, where $K$=1 and the preference function is f(x,y)=2x$^2$+y. The output skyline points should be $c$.

$ParCube$ can easily handle such queries by modifying a little when ordering different points based on the preference function. This modification should be implemented in the process of changing spatial datasets to sequence datasets.

### 7.2.2 Constrained Skyline Queries

Given a set of constraints, a constrained skyline query returns the most interesting points in the data space defined by the constraints [34]. Typically, each constraint is expressed as a range along a dimension and the conjunction of all constraints forms a hyper-rectangle (referred to as the constraint region) in the d-dimensional attribute space. Consider the hotel example as shown in Figure 6.1, where a user is interested only in hotels whose price ($x$-axis) is in the range 2-4. The skyline in this case contains point $a$, as it is the most interesting hotel in the specified range.

$ParCube$ can process such queries with little modification. While extracting skyline or traverse DAG to extracting dominated points, intersecting test is employing to limit the result points that meet the requirement of the constraint.

### 7.2.3 Dynamic Skyline Queries

Assume a database containing points in d-dimensional space with axes $d_1, d_2, \ldots, d_d$. A dynamic (or spatial) skyline query specifies $m\ dimension\ functions\ f_1, f_2, \ldots, f_m$ such that each function $f_i$ (1≤i≤m) takes as parameters the coordinates of the data points along a subset of the $d$ axes [85]. The goal is to return the skyline in the new data space with dimensions defined by $f_1, f_2, \ldots, f_m$. Consider a database that stores the following information for each hotel: (i) its x-, (ii) y- coordinates, and (iii) its price (i.e., the database contains 3 dimensions). Then, a user specifies his/her current location ($u_x, u_y$), and requests the most interesting hotels, where preference must take into consideration the hotels' proximity to the user (in terms of Euclidean distance) and the price. Each point p with coordinates ($p_x, p_y, p_z$) in the original 3D space is transformed to a point p' in the 2D space with coordinates ($f_1(p_x, p_y), f_2(p_z)$).

For this issue, some existing technique can be directly applied with little modification (i.e., BBS [72]). However, our proposed method, $ParCube$ can not easily tackle this problem because we predefine the order between points and during the query process, this order can not be changed. We consider on how to borrow some strategies from mobile data management research field to settle down the issue.

### 7.2.4 Enumerating and K-dominating Queries

Enumerating queries return, for each skyline point $p$, the number of points dominated by $p$. This information may be relevant for some applications as it provides some measure of "goodness" for the skyline points [72]. This issue has been solved by Lin et al. in their paper [63]. However, our proposed method, $ParCube$, is also very well to tackle it. While constructing the partial order, we can easily insert into the node, the number of how many points it dominates. The $K$-dominating query is just a variant of enumerating query, which is the Top-$K$ points which have the most large number of dominating points. While users ask for $K$-dominating points, we just extract and compare the nodes in higher layers until top $K$ nodes are found. The reason is that the points in higher layers in DAG should have more dominating points than those exist in lower layers.

## 7.3 Summary

We have found the inter-connection between sequential pattern mining problem and the dominant relationship analysis. It is built based on the intuitive idea that there is correlation between space (patterns with dominant relationship) and time (sequential patterns). To mine sequential patterns in sequence datasets, is indeed, the same as to find dominant relationship between points in the corresponding spatial datasets. Based on this intrinsic similarity between the two research fields, there are many related issues could be beneficial from each other. As a summary, we could list several common research topics as follows:

- **Top-$K$ strategy**. Due to the rank-aware characteristic of the two problems, finding only the top-$K$ patterns becomes a common topic [45, 63]. Moreover, rank-aware query processing has become a vital need for many other applications, i.e.,

Web search, multimedia and digital libraries similarity search, etc. The answer to a top-k join query is an ordered set of join results according to some provided function (i.e., occurrence frequency) that combines the orders on each attribute (customer or dimension).

- **Compression issue**. Mining the whole result set is always time and space cost, no matter how efficient an algorithm is. Hence, to mine the core patterns (or centers of the clusters) is an important task to simplify the mining process while preserving reasonable semantic meaning. There are some recent papers [76, 105] towards this trend.

- **Approximation method**. Most of the attention has paid to precisely get the results. It is natural to get only approximate results (i.e., with more than 90% precision) to fasten the mining process. This kind of idea is come from Information Retrieval, in which precisely get all the results is impossible and is unnecessary. There are several papers mentioned this issue [53, 55] and even more, a recent paper [82] aimed to get the patterns on uncertain datasets.

- **Constraint based**. General sequential pattern mining and dominance relation query may be not sufficient for users. For example, to mine interesting frequent DNA sequences, minimum gap between items in a pattern should be always satisfied [98]. There is another related paper in dominance relationship analysis field [34]. We believe that, constraint-aware algorithms could be applied directly to real applications. However, note that nearly all the constraint based algorithms are derived from general purpose algorithms and hence, general purpose algorithms are more critical contributing to the performance of the mining process. Constraint based approaches can be seen as real application oriented.

- **Stream data**. Recently, there is emerging an important research issue that stream data mining, whose process is to extract knowledge structures from continuous, rapid data records. The difficulty of data stream mining is that we can read only once or a small number of times using limited computing and storage capabilities. There are several papers related to sequential pattern mining and dominant relationship analysis [26, 62]. This issue should be always a challenging problem and a approximate solution seems reasonable.

# Chapter 8

# Conclusions

In this section, the summary of the thesis and the future research directions will be described.

## 8.1 Summary of the Thesis

In this thesis, we systematically studied the sequential pattern mining problem. we proposed a novel series of algorithms called LAPIN for efficient sequential pattern mining. Our main idea is that the last position of an item $s$ in each customer sequence is very useful and key to judge whether a k-length frequent sequence could grow to a frequent (k+1)-length sequence by appending it with $s$. So LAPIN could reduce searching greatly by only scanning a small portion of the projected database or the ID-List as well as handle long pattern efficiently, which is inherently difficult for most existing algorithms. By thorough experiments and comparison, we demonstrated that LAPINs outperform the existing state-of-the-art algorithms by up to orders of magnitude on dense datasets.

However, we found that the improvement of LAPIN is at the price of much memory consuming when building the list of item's last position because LAPIN uses a bitmap strategy. We further aim to obtain an efficient and balanced pattern mining algorithm with low memory consuming and thus, we proposed an improved algorithm which is based on reuse strategy. The experiments demonstrated that our improved algorithm performs the best in limited resource environments on dense datasets.

SPAM is proved very efficient for long pattern mining and it can outperform Pre-

fixSpan and SPADE by up to an order of magnitude in resource unlimited environments. We proposed a new algorithm named LAPIN_SPAM, which combines the key idea of LAPIN and SPAM. The experiments demonstrated that LAPIN_SPAM significantly outperforms the original SPAM, and is the best under unlimited resource assumption on dense datasets.

We systematically compared and summarized different algorithms on different kinds of datasets. The conclusion is that when the density of $m$ is large enough to overcome the side-effect of LAPIN strategy (i.e., building the item_last_position_table), then our LAPIN algorithm will be faster. For those datasets which have large value of $m$, our algorithms could be orders of magnitude faster than others.

In this thesis, we constructed an effective Web log mining system based on our efficient sequential mining algorithm, LAPIN_WEB, an extension of LAPIN algorithm to extract user access patterns from traversal path in Web logs. Our experimental results and performance studies demonstrate the efficiency of LAPIN_WEB on real Web log datasets. Moreover, we also implemented a visualization tool to help interpret mining results as well as predict users' future requests.

In this thesis, we have found the interrelated connection between sequential pattern mining and the general dominant relationship. Based on this discovery, we proposed efficient algorithms to answer the general dominant relationship queries by using efficient sequential pattern mining algorithms and several other strategies. Extensive experiments illustrate the effectiveness and efficiency of our methods. We believe that the two research fields, sequential pattern mining and dominance relationship analysis, can be beneficial from each other.

## 8.2 Future Research Directions

There are many future work that can be stemmed from this thesis. We list several of them:

- Top-$K$ sequential pattern mining, or Top-$K$ skyline query.
- Sequential pattern result compression, or dominance relationship result compression.

- Approximate mining sequential patterns, or dominance relationship approximate representation on uncertain data.
- Sequential patterns discovering on stream data, or dominance relationship analysis on stream data.
- Constraint based sequential pattern mining, or constraint based dominance relationship analysis.

# Appendix A

# Publication List

## A.1 Journal Papers

Zhenglu Yang, Yitong Wang, and Masaru Kitsuregawa. Effective Algorithms for Sequential Pattern Mining. *DBSJ Letters*, Vol. 5, No. 1, pp. 53-56, Jun. 2006.

## A.2 International Conference Papers

Zhenglu Yang, Yitong Wang, and Masaru Kitsuregawa. An Effective System for Mining Web Log. In *Proceedings of 8th Asia-Pacific Web Conference (APWeb'06)*, pp. 40-52, Harbin, China, Jan. 2006.

Zhenglu Yang, Yitong Wang, and Masaru Kitsuregawa. PAID: Mining Sequential Patterns by Passed Item Deduction in Large Databases. In *Proceedings of 10th International Database Engineering & Applications Symposium (IDEAS'06)*, pp. 113-120, Delhi, India, Dec. 2006.

Zhenglu Yang, Botao Wang, and Masaru Kitsuregawa. General Dominant Relationship Analysis based on Partial Order Models. In *Proceedings of 22nd ACM Symposium on Applied Computing (SAC'07)*, pp. 470-474, Soul, Korea, Mar. 2007.

Zhenglu Yang, Yitong Wang, and Masaru Kitsuregawa. LAPIN: Effective Sequential Pattern Mining Algorithms by Last Position Induction for Dense Databases. In *Proceedings of 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, pp. 1020-1023, Bankok, Thailand, Apr. 2007.

Zhenglu Yang, Lin Li, Botao Wang, and Masaru Kitsuregawa. Towards Efficient Dominant Relationship Exploration of the Product Items on the Web. In *Proceedings of the 16th international conference on World Wide Web (WWW'07)*, pp. 1205-1206, Banff, Alberta, Canada, May 2007.

Zhenglu Yang, Lin Li, Botao Wang, and Masaru Kitsuregawa. Towards Efficient Dominant Relationship Exploration of the Product Items on the Web. In *Proceedings of 22nd National Conference on Artificial Intelligence (AAAI'07)*, pp. 1483-1488, Vancouver, Canada, Jul. 2007.

Lin Li, Zhenglu Yang, Botao Wang, and Masaru Kitsuregawa. Dynamic Adaptation Strategies for Long-Term and Short-Term User Profile to Personalize Search. In *Proceedings of joint conference of the 9th Asia-Pacific Web Conference and the 8th International Conference on Web-Age Information Management (APWeb/WAIM'07)*, pp. 228-240, Huangshan, China, Jun. 2007.

Lin Li, Zhenglu Yang, and Masaru Kitsuregawa. Aggregating User-Centered Rankings to Improve Web Search. In *Proceedings of 22nd National Conference on Artificial Intelligence (AAAI'07)*, pp. 1884-1885, Vancouver, Canada, Jul. 2007.

Lin Li, Zhenglu Yang, Kulwadee Somboonviwat, Masaru Kitsuregawa. User-assisted similarity estimation for searching related web pages. In *Proceedings of the 18th ACM Conference on Hypertext and Hypermedia (HT'07)*, pp. 11-20, Manchester, UK, Sep. 2007.

## A.3 Workshop Papers

Zhenglu Yang, Yitong Wang, and Masaru Kitsuregawa. Effective Sequential Pattern Mining Algorithms for Dense Database. In *Proceedings of National Data Engineering WorkShop (DEWS'06)*, Japan, Mar. 2006.

Zhenglu Yang and Masaru Kitsuregawa. LAPIN-SPAM: An Improved Algorithm for Mining Sequential Pattern. In *Proceedings of International Special Workshop on Databases For Next Generation Researchers (SWOD'05)*, in conjunction with $ICDE$, pp. 8-11, Japan, Apr. 2005.

Zhenglu Yang and Masaru Kitsuregawa. Effective Mining Sequential Pattern by Last Position Induction. In *Proceedings of National Data Engineering WorkShop (DEWS'05)*, Japan, Mar. 2005.

# References


[1] F. Afrati, A. Gionis, and H. Mannila. Approximating a collection of frequent sets. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 12–19, 2004. 120

[2] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of International Conference on Very Large Data Bases*, pages 506–521, 1996. 100

[3] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of International Conference on Very Large Data Bases*, pages 487–499, 1994. 1, 4, 10, 45, 84

[5] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of International Conference on Data Engineering*, pages 3–14, 1995. 1, 2, 4, 8, 9, 22, 51, 84, 99, 104, 106

[6] R. Agrawal and E. Wimmers. A framework for expressing and combining preferences. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 297–306, 2000. 101

[7] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using a bitmap representation. In *Proceedings of ACM SIGKDD International Confer-*

*ence on Knowledge Discovery and Data Mining*, pages 429–435, 2002. 2, 4, 6, 8, 12, 17, 22, 23, 29, 33, 39, 40, 51, 59, 61, 69, 84, 86, 99

[8] T. L. Bailey and C. Elkan. Fitting a mixture model by expectation maximization to discover motifs in biopolymers. In *Proceedings of International Conference on Intelligent Systems for Molecular Biology*, pages 28–36, 1994. 1

[9] J. L. Balcazar and G. Casas-Garriga. On horn axiomatizations for sequential data. In *Proceedings of International Conference on Database Theory*, pages 215–229, 2005. 100

[10] W.-T. Balke, U. Guentzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *Proceedings of International Conference on Extending Database Technology*, pages 256–273, 2004. 95, 98

[11] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *Proceedings of International Conference on Very Large Data Bases*, pages 564–575, 2004. 101

[12] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *SIGKDD Explorations*, 2(2), 2000. 99

[13] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 85–93, 1998. 1, 29, 119

[14] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining on large, dense data sets. In *Proceedings of International Conference on Data Engineering*, pages 188–197, 1999. 118

[15] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler. Genbank. *Nucleic Acids Research*, 30(1):17–20, 2002. 51

[16] C. Bettini, X. S. Wang, and S. Jajodia. Mining temporal relationships with multiple granularities in time sequences. *Data Engineering Bulletin*, 21(1):32–38, 1998. 99

[17] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, 1999. 100

[18] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proceedings of International Conference on Data Engineering*, pages 431–440, 2002. 101

[19] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of International Conference on Data Engineering*, pages 421–430, 2001. 3, 95, 96, 97, 98, 102, 111

[20] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: generalizing association rules to correlations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 265–276, 1997. 1

[21] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: a maximal frequent itemset algorithm for transactional databases. In *Proceedings of International Conference on Data Engineering*, pages 443–462, 2001. 99

[22] G. Casas-Garriga. Towards a formal framework for mining general patterns from ordered data. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining Workshop on MRDM*, pages 14–26, 2003. 100, 109

[23] G. Casas-Garriga. Summarizing sequential data with closed partial orders. In *Proceedings of SIAM International Conference on Data Mining*, pages 380–391, 2005. 100, 101, 104, 106, 107

[24] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006. 95, 98

[25] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. Tung, and Z. Zhang. On high dimensional skylines. In *Proceedings of International Conference on Extending Database Technology*, pages 478–495, 2006. 95, 98

[26] G. Chen, X. Wu, and X. Zhu. Sequential pattern mining in multiple streams. In *Proceedings of IEEE International Conference on Data Mining*, pages 585–588, 2005. 124

[27] J. Chomicki. Querying with intrinsic preferences. In *Proceedings of International Conference on Extending Database Technology*, pages 34–51, 2002. 101

[28] J. Chomicki. Preference formulas in relational queries. *ACM Transactions on Database Systems*, 24(4):1–39, 2003. 101

[29] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proceedings of International Conference on Data Engineering*, pages 717–719, 2003. 95, 97, 98

[30] G. Cong, B. Cui, Y. Li, and Z. Zhang. Summarizing frequent patterns using profiles. In *Proceedings of International Conference on Database Systems for Advanced Applications*, pages 171–186, 2006. 120

[31] R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining world wide web browsing patterns. *Knowledge and Information Systems*, 1(1):5–32, 1999. 82, 83

[32] D. Cristofor, L. Cristofor, and D. A. Simovici. Galois connection and data mining. *Jounal of Universal Computer Science*, 6(1):60–73, 2000. 99

[33] B. A. Davey and H. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1990. 100

[34] E. Dellis, A. Vlachou, I. Vladimirskiy, B. Seeger, and Y. Theodoridis. Constrained subspace skyline computation. In *Proceedings of ACM Conference on Information and Knowledge Management*, pages 415–424, 2006. 122, 124

[35] G. Dong and J. Li. Efficient mining of emerging patterns: discovering trends and differences. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 43–52, 1999. 1

[36] E. Eskin and P. Pevzner. Finding composite regulatory patterns in dna sequences. In *Proceedings of International Conference on Intelligent Systems for Molecular Biology*, pages 354–363, 2002. 1

[37] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996. 1

[38] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999. 99

[39] M. N. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In *Proceedings of International Conference on Very Large Data Bases*, pages 223–234, 1999. 2, 99

[40] Google. http://www.google.com. 2007. 82

[41] K. Gouda, M. Hassaan, and M. J. Zaki. Prism: A prime-encoding approach for frequent sequence mining. In *Proceedings of IEEE International Conference on Data Mining*, 2007. 22

[42] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of International Conference on Data Engineering*, pages 152–159, 1996. 100

[43] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–27, 2003. 101

[44] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Proceedings of International Conference on Data Engineering*, pages 106–115, 1999. 1, 2, 99

[45] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proceedings of IEEE International Conference on Data Mining*, pages 211–218, 2002. 120, 123

[46] J. Ho, L. Lukov, and S. Chawla. Sequential pattern mining with constraints on large protein databases. In *Proceedings of International Conference on Management of Data (COMAD)*, 2005. 51

[47] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multiparametric ranked queries. In *Proceedings of the*

*ACM SIGMOD International Conference on Management of Data*, pages 259–270, 2001. 3, 95, 101

[48] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *Proceedings of International Conference on Data Engineering*, page 66, 2006. 95, 98

[49] M. Kamber, J. Han, and J. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 207–210, 1997. 1

[50] W. Kie$\beta$ling. Foundations of preferences in database systems. In *Proceedings of International Conference on Very Large Data Bases*, pages 311–322, 2002. 3, 95, 100, 101

[51] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proceedings of ACM Conference on Information and Knowledge Management*, pages 401–407, 1994. 1

[52] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. Proceedings of acm sigkdd international conference on knowledge discovery and data mining-cup 2000 organizer's report: Peeling the onion. *SIGKDD Explorations*, 2:86–98, 2000. 48, 68

[53] V. Koltun and C. H. Papadimitriou. Approximately dominating representatives. In *Proceedings of International Conference on Database Theory*, pages 204–214, 2005. 95, 98, 124

[54] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of International Conference on Very Large Data Bases*, pages 275–286, 2002. 95, 97, 98

[55] H. C. Kum, J. Pei, W. Wang, and D. Duncan. Approxmap: Approximate mining of consensus sequential patterns. In *Proceedings of SIAM International Conference on Data Mining*, pages 311–315, 2003. 2, 124

[56] H. Kung, F. Luccio, and F. Preparata. On finding the maxima of a set of vectors. *Journal of ACM*, 22(4), 1975. 97

[57] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of IEEE International Conference on Data Mining*, pages 313–320, 2001. 121

[58] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978. 100

[59] C. Lee, C. Grasso, and M. F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002. 100

[60] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *Proceedings of International Conference on Data Engineering*, pages 220–231, 1997. 1

[61] C. Li, B. C. Ooi, A. K. Tung, and S. Wang. Dada: A data cube for dominant relationship analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006. 3, 95, 98, 99, 100, 103, 104

[62] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *Proceedings of International Conference on Data Engineering*, pages 502–513, 2005. 95, 98, 124

[63] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *Proceedings of International Conference on Data Engineering*, pages 86–95, 2007. 123

[64] C. Luo and S. Chung. Efficient mining of maximal sequential patterns using multiple samples. In *Proceedings of SIAM International Conference on Data Mining*, pages 64–72, 2005. 2

[65] H. Mannila and C. Meek. Global partial orders from sequential data. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 161–168, 2000. 100, 104

[66] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 210–215, 1995. 2, 99

[67] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997. 1

[68] F. Masseglia, F. Cathala, and P. Poncelet. The psp approach for mining sequential patterns. In *Proceedings of European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 176–184, 1998. 99

[69] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006. 95, 98

[70] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 13–24, 1998. 118

[71] B. Ozden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Proceedings of International Conference on Data Engineering*, pages 412–421, 1998. 99

[72] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 467–478, 2003. 95, 97, 98, 111, 121, 123

[73] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 175–186, 1995. 12

[74] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *Proceedings of ACM Conference on Information and Knowledge Management*, pages 251–258, 1999. 2

[75] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of International Conference on Database Theory*, pages 398–416, 1999. 99, 119

[76] J. Pei, A. W.-C. Fu, X. Lin, and H. Wang. Computing compressed multidimensional skyline cubes efficiently. In *Proceedings of International Conference on Data Engineering*, pages 96–105, 2007. 124

[77] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proceedings of International Conference on Data Engineering*, pages 433–442, 2001. 118

[78] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Int'l Workshop on Data Mining and Knowledge Discovery*, pages 11–20, 2000. 99

[79] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1424–1440, November 2004. 2, 4, 8, 19, 20, 22, 23, 32, 35, 40, 51, 57, 59, 84, 85, 99, 106, 107

[80] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu. Mining access pattern efficiently from web logs. In *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 396–407, 2000. 84

[81] J. Pei, J. Han, and W. Wang. Mining sequential patterns with constraints in large databases. In *Proceedings of ACM Conference on Information and Knowledge Management*, pages 18–25, 2002. 99

[82] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *Proceedings of International Conference on Very Large Data Bases*, pages 15–26, 2007. 124

[83] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985. 97

[84] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. In *Technical report RJ10026, IBM*, 1996. 100

[85] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *Proceedings of International Conference on Very Large Data Bases*, pages 751–762, 2006. 122

[86] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. *Data Mining and Knowledge Discovery*, 4(2-3):163–192, 2000. 1

[87] S. W. Smith and J. Tygar. Security and privacy for partial order time. In *ISCA International Conference on Parallel and Distributed Computing Systems*, pages 70–79, 1994. 100

[88] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of International Conference on Extending Database Technology*, pages 3–17, 1996. 2, 4, 8, 11, 12, 16, 22, 84, 99

[89] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM transactions on Computer Systems*, 3(3):204–226, 1985. 100

[90] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proceedings of International Conference on Very Large Data Bases*, pages 301–310, 2001. 98

[91] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):377–391, 2006. 95, 98

[92] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *Proceedings of International Conference on Data Engineering*, page 65, 2006. 95, 97, 98, 111, 112

[93] W. G. Teng, M. Chen, and P. Yu. A regression-based temporal pattern mining scheme for data streams. In *Proceedings of International Conference on Very Large Data Bases*, pages 93–104, 2003. 2

[94] S. Tono, H. Kitakami, K. Tamura, Y. Mori, and S. Kuroki. Efficiently mining sequence patterns with variable-length wildcard regions using an extended modified prefixspan methods. In *Proceedings of International Conference on Intelligent Systems for Molecular Biology*, 2005. 51

[95] P. Tzvetkov, X. Yan, and J. Han. Tsp: Mining top-k closed sequential patterns. In *Proceedings of IEEE International Conference on Data Mining*, pages 347–358, 2003. 2

[96] J. Wang and J. Han. Bide: efficient mining of frequent closed sequences. In *Proceedings of International Conference on Data Engineering*, pages 79–90, 2004. 99

[97] J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 236–245, 2003. 99

[98] K. Wang, Y. Xu, and J. X. Yu. Scalable sequential pattern mining for biological sequences. In *Proceedings of ACM Conference on Information and Knowledge Management*, pages 178–187, 2004. 124

[99] K. Wu, P. Yu., and A. Ballman. Speedtracer: A web usage mining and analysis tool. *IBM Systems Journal*, 37(1):89–105, 1998. 84

[100] P. Wu, C. Zhang, Y. Feng, B. Y.Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *Proceedings of International Conference on Extending Database Technology*, pages 112–130, 2006. 95, 98

[101] T. Xia and D. Zhang. Refreshing the sky: The compressed skycube with efficient support for frequent updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006. 95, 98, 100

[102] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proceedings of International Conference on Very Large Data Bases*, pages 476–487, 2003. 100

[103] X. Yan, H. Cheng, J. Han, and D. Xin. Summarizing itemset patterns: A profile-based approach. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 314–323, 2005. 120

[104] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of IEEE International Conference on Data Mining*, pages 721–724, 2001. 121

[105] X. Yan, J. Han, and R. Afshar. Clospan: mining closed sequential patterns in large datasets. In *Proceedings of SIAM International Conference on Data Mining*, pages 166–177, 2003. 2, 86, 99, 106, 109, 119, 124

[106] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 344–353, 2004. 107

[107] J. Yang, P. S. Yu, W. Wang, and J. Han. Mining long sequential patterns in a noisy environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 406–417, 2002. 99, 119

[108] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proceedings of International Conference on Very Large Data Bases*, pages 241–252, 2005. 95, 98, 100

[109] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning Journal*, 42:31–60, 2001. 2, 4, 8, 14, 15, 22, 33, 35, 40, 51, 62, 84, 85, 99

[110] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2002. 121

[111] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proceedings of SIAM International Conference on Data Mining*, pages 398–416, 2002. 99

[112] M. J. Zaki and M. Ogihara. Theoretical foundations of association rule. In *ACM Proceedings of the ACM SIGMOD International Conference on Management of Data Workshop on Research Issues in Data Mining and Knowledge Discovery*, 1998. 99

[113] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 159–170, 1997. 100