Fast Frequent Pattern Mining

Yabo XU

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
System Engineering & Engineering Management

Supervised by

Prof. Jeffrey Xu, Yu

July 3, 2003

Abstract of thesis entitled:
Fast Frequent Pattern Mining
Submitted by Yabo XU
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in July 3, 2003

Advances in information technology and data collection methods have led to an unprecedented opportunity for data mining to analyze the huge data and extract intelligent and useful information. This thesis focuses on two important problems in data mining applications: frequent pattern mining and sequential pattern mining.

FP-Tree based algorithm have been recognized as the most efficient algorithm for mining frequent patterns. However, their performance suffers the complex data structures and inefficient traversal fashion. In the first part, we propose another simple and compact structure, memory-based prefix-path tree. Upon this structure, a new depth-first frequent pattern discovery algorithm, called PP-Mine, is proposed that outperforms FP-Growth significantly.

In the second part, we move forward to an more difficult problem: sequential pattern mining on biological data. We study the implication of the following biological features on data mining techniques: the bioalphabet is extremely small, biosequences are extremely long, and biological patterns occur in short regions of local similarity and in long regions of global similarity with possible gaps. These features render a different blow up of search space from that in classic transaction sequences, thereby, justify-

ing new ways of pattern growing/pruning and support counting. We present a two-phase algorithm to address these issues: the *segment phase* and the *pattern phase*. The purpose of this two-phase approach is multi-fold: adopt best techniques separately for local similarity and global similarity, exploit local similarity for efficient search of global similarity, and grow patterns rapidly one segment at a time instead of one item at a time. We evaluate this approach on both synthetic and real life data sets.

論文摘要:

信息技術和數據採集技術的發展給數據挖掘帶來了前所未有的機遇去從中分析和提取有用的信息。本文主要研究了兩個數據挖掘應用中的基本問題：Frequent Pattern 挖掘和 Sequential Pattern 挖掘。

基於 FP-Tree 類的算法是當前被認爲最有效的挖掘 Frequent Pattern 的算法。然而，他們的效率總是受限於複雜的數據結構和缺乏效率的遍歷方式。在本文的第一部分中，我們提出了一種簡單而緊湊的數據結構，置於內存的 Prefix-Path 樹。在這種數據結構上，一種新的深度優先的 Pattern 搜索算法，PP-Mine, 被提出，實驗表明，它的效率完全壓過了 FP-Growth.

在本文的第二部分，我們探討了另一個難度更大的問題：對生物序列的 Sequential Pattern 的挖掘。我們研究了生物序列本身的一些特性對數據挖掘技術的一些潛在的作用: 生物序列的組成元素特別的少，但是卻特別的長；在小範圍內會有一些局部相似性，在全局範圍內也會有一些整體的一些相似性。這些特性極大的增加了經典的解決一些交易序列數據挖掘問題的搜索空間，從而，需要找到新的方法來進行 Pattern 的搜索和刪減，以及數據計算。

我們提出了一個分爲兩個階段的算法來解決這個問題：Segment 階段和 Pattern 階段兩階段算法的目的是多方面的：分別采用最適合的技術來處理局部相似性和全局相似性；利用局部的相似性來提高查找整體相似性的效率；一次增長一個 Segment 而不是增長一個單獨的單元。我們分別在人造的數據和真實的生物數據上做了實驗，結果充分證實了我們方法的有效性。

Acknowledgement

I wish to express my deep gratitude to my supervisor Prof. Jeffrey Xu. Yu. I thank him for his continuous encouragement, confidence and support, for creating an free academic atmosphere in these two years. As an advisor, he guided me on the right road to research.

My gratitude and appreciation also goes to Prof. Ke Wang. Part of this work is done in collaboration with him. I thank him for the knowledge and skills they imparted through the collaboration. Working with him is always enjoyable. I also thank him for serving as an external examiner of my thesis.

Last but not least, I am very grateful to my parents for their continuous moral support and encouragement. Their love accompanies me wherever I go.

Contents

Abstract i

Acknowledgement iii

1 Introduction 1
- 1.1 Frequent Pattern Mining 1
- 1.2 Biosequence Pattern Mining 2
- 1.3 Organization of the Thesis 4

2 *PP*-Mine: Fast Mining Frequent Patterns In-Memory 5
- 2.1 Background 5
- 2.2 The Overview 6
- 2.3 *PP*-tree Representations and Its Construction . . 7
- 2.4 *PP*-Mine 8
- 2.5 Discussions 14
- 2.6 Performance Study 15

3 Fast Biosequence Patterns Mining 20
- 3.1 Background 21
  - 3.1.1 Differences in Biosequences 21
  - 3.1.2 Mining Sequential Patterns 22
  - 3.1.3 Mining Long Patterns 23
  - 3.1.4 Related Works in Bioinformatics 23
- 3.2 The Overview 24
  - 3.2.1 The Problem 24

3.2.2 The Overview of Our Approach 25
3.3 The Segment Phase 26
3.3.1 Finding Frequent Segments 26
3.3.2 The Index-based Querying 27
3.3.3 The Compression-based Querying 30
3.4 The Pattern Phase 32
3.4.1 The Pruning Strategies 34
3.4.2 The Querying Strategies 37
3.5 Experiment 40
3.5.1 Synthetic Data Sets 40
3.5.2 Biological Data Sets 46

4 Conclusion **55**

Bibliography **60**

List of Figures

2.1 The memory representation (PP_M-tree) 8
2.2 A PP_M-tree with four items 11
2.3 An Example 13
2.4 PP_M-tree, FP-tree and H-struct for Example 1 where $\tau = 2$ 18
2.5 Scalability 19

3.1 Compressing a sequence 31
3.2 The segment tree in Example 4 33
3.3 The pattern tree in Example 5 36
3.4 C128S32N4D100K, MinLen = 5 42
3.5 C128S32N20D100K, MinLen = 3 43
3.6 C256S64N4D100K, MinLen= 7 44
3.7 C256S64N20D100K, MinLen= 3 45
3.8 C20S8N10000D100K, MinLen = 1 50
3.9 Scalability wrt the database size 51
3.10 Biological data sets 52
3.11 The real life DNA dataset, MinLen = 5 53
3.12 The real life protein dataset, MinLen = 3 54

List of Tables

2.1 The transaction database TDB 7

3.1 The sequence database D 29
3.2 The position lists 29
3.3 Parameters of the data generator 48
3.4 Synthetic data sets 49
3.5 Statistics for C128S32N4D100K, MinLen=5 . . . 49

Chapter 1

Introduction

"The universe if full of magical things patiently waiting for our wits to grow sharper." [1] Now advances in information technology and data collection methods have led to an unprecedented opportunity to analyze the huge data and extract intelligent and useful information.

1.1 Frequent Pattern Mining

Frequent patterns play an essential role in many data mining tasks that try to find interesting patterns from databases, such as association rules, correlations, sequences, episodes, classifiers, cluster and many more of which the mining of association rules is one of the most popular problems. The original motivation for searching association rules came from the need to analyze so called supermarket transaction data, that is, to examine customer behavior in terms of the purchased products. Association rules describe how often items are purchased together. Such rules can be useful for decisions concerning product pricing, store layout and many others.

Since their introduction in 1993 by Argawal et al.[4], the frequent pattern and association rule mining problems have re-

[1] By Eden Phillpotts(1862-1960), English writer, poet, playwright

ceived a great deal of attention. Within the past decade, hundreds of research papers have been published presenting new algorithms or improvements on existing algorithms to solve these mining problems more efficiently.

So our work start from this basic mining problem: Frequent Pattern Mining. FP-Tree based algorithm have been recognized as the most efficient algorithm for mining frequent patterns. However, their performance suffers the complex data structures and inefficient traversal fashion. In the first part of this thesis, we propose a novel and efficient mining problem, called *PP*-Mine, which does not generate any conditional sub-tree, and counting is done as a side-effort of pushing-right operation in an accumulated manner. We will report the comparison with the other popular algorithms in our experimental studies later in this thesis.

1.2 Biosequence Pattern Mining

In the second part, we move forward to another important but more difficult mining problem, the sequential pattern problem. Comparable to frequent patterns, the items occurred in sequential patterns can be repeated and with order, which lead to the larger search space.

One important problem arising from bio-applications is the discovery of sequential patterns that occur in many biosequences (i.e., DNA or protein sequences). Such patterns often correspond to residues conserved during evolution due to an important structural or functional role. The "classic" sequential pattern mining has been studied in market-basket analysis [6, 5, 16, 20, 27, 17], where the task is to find all *frequent* subsequences that have some minimum support by occurring in a required percentage of transaction sequences. Such sequential patterns capture temporal purchasing behaviors of customers.

By treating a biosequence as a transaction sequence, existing algorithms can be applied to biosequences. However, our experiments show that the efficiency demonstrated on classic transaction sequences cannot be retained on biosequences.

In this thesis,we study the implication of the following biological features on data mining techniques: the bioalphabet is extremely small, biosequences are extremely long, and biological patterns occur in short regions of local similarity and in long regions of global similarity with possible gaps. These features render a different blow up of search space from that in classic transaction sequences, thereby, justifying new ways of pattern growing/pruning and support counting.

Below are the main ideas of our approach in addressing the above requirements.

Two-phase pattern growth. We propose a two-phase approach to grow patterns rapidly in length to reduce the frequency of support counting. The first phase finds frequent segments X_i, rather efficiently, above a specified minimum length. The second phase grows patterns $X_1 * \cdots * X_k$ using frequent segments X_i as building blocks. The essence of this two-phase approach is to grow patterns one segment at a time and to exploit information about segments for candidate pruning and support counting in the second phase, as explained below.

Segment-based pruning. Suppose that we know that a pattern $P = X_1 * \cdots * X_{k-1} * X_k$ does not extend into a frequent pattern $P * X$ for some frequent segment X, or does not occur before position i in some sequence s. We can infer this information for any pattern $P' = X_1 * \cdots * X_{k-1} * X'_k$ or $P' = X_1 * \cdots * X_{k-1} * X_k * \cdots * X_{k+i}$, where X_ki s a prefix of X'_k, because each occurrence of P' is an occurrence of P. We exploit these relationships for pattern pruning and pattern matching in a novel search strategy

Query-based counting. Pattern matching against a long

sequence must be significantly faster than scanning the whole sequence. We formalize this problem as querying the smallest end positions of a pattern $X_1 * \cdots * X_k * X_{k+1}$, given such positions of a pattern $X_1 * \cdots * X_k$. This approach benefits from any efficient querying method. We consider one direct access method by indexing local similarity and one sequential scan method by compressing local similarity.

We evaluate this approach on both synthetic and real life data sets.

1.3 Organization of the Thesis

The rest of the paper is organized as follows. Chapter 2 focus on our new algorithm *PP*-Mine. our new approaches on biosequence pattern mining will be presented in Chapter 3. For easy understanding, the background, problem overview, approach details and experimental study sections will be included in each chapter separately. Section 7 gives an brief conclusion.

□ **End of chapter.**

Chapter 2

PP-Mine: Fast Mining Frequent Patterns In-Memory

Summary

In this chapter, we propose a simple and compact structure, memory-based prefix-path tree. Upon this structure, a new depth-first frequent pattern discovery algorithm, called PP-Mine, is proposed that outperforms FP-Growth significantly.

2.1 Background

Recent studies show pattern-growth method is one of the most effective methods for frequent pattern mining [2, 3, 8, 12, 15, 14, 18]. As a divide-and-conquer method, this method partitions (projects) the database into partitions recursively, but does not generate candidate sets. This method also makes use of Apriori property [4]: if any length k pattern is not frequent in the database, its length $(k + 1)$ super-patterns can never be frequent. It *counts* frequent patterns in order to decide whether it can assemble longer patterns. Most of the algorithms use a tree

as the basic data structure to mine frequent patterns, such as the lexicographic tree [2, 3, 8, 12] and the FP-tree [15]. Different strategies were extensively studied such as depth-first [3, 2], breath-first [3, 8], top-down [23] and bottom-up [15].

As one of the most representative pattern-growth algorithm, FP-growth, it start its mining process with the construction of FP-Tree: First scan the database to get the frequent items, then insert the frequent part of every transaction into a prefix-path tree during the second database scan and link all the same items together by a header table. FP-growth explores the FP-Tree by a bottom-up fashion and the conditional FP-Tree will be created when the pattern is extended by any frequent item. The complex node-link across the FP-Tree in a unpredictable manner makes FP-Tree difficult to be materialized on disk and the conditional FP-Tree generation consume some unnecessary memory. Both motivate us to study new faster mining algorithms with simpler data structures. In this chapter, we present our solutions on this problem: a node-link free tree, called *PP*-Tree and *PP*-Mine , a novel mining algorithm which does not generate any conditional trees, and outperforms FP-growth significantly.

2.2 The Overview

Let $I = \{x_1, x_2, \cdots, x_n\}$ be a set of items. An itemset X is a subset of items I, $X \subseteq I$. A transaction $T_X = (tid, X)$ is a pair, where X is an itemset and tid is its unique identifier. A transaction $T_X = (tid, X)$ is said to contain $T_Y = (tid, Y)$ if and only if $Y \subseteq X$. A transaction database TDB is a set of transactions. The number of transactions in TDB that contains X is called the support of X, denoted as $sup(X)$. An itemset X is a frequent pattern, if and only if $sup(X) \geq \tau$, where τ is a threshold called a minimum support. The frequent pattern mining problem is to find the complete set of frequent patterns

in a given transaction database with respect to a given support threshold, τ.

Example 1 *Let the first two columns of Table 3.1 be our running transaction database TDB. Let the minimum support threshold be $\tau = 2$. The frequent items are shown in the third column of Table 3.1.*

Trans ID	Items	Frequent items
100	c,d,e,f,g,i	c,d,e,g
200	a,c,d,e,m	a,c,d,e
300	a,b,d,g,k	a,d,e,g
400	a,c,h	a,c

Table 2.1: The transaction database TDB

2.3 *PP*-tree Representations and Its Construction

The in-memory representation of *PP*-tree, denoted PP_M-tree, is of a tree. Despite the pointers to the children nodes, a node in PP_M-tree consists of item-name, count, and a node-link. The count registers the number of itemsets represented by the portion of the path reaching from the root to this node. The PP_M-tree for Example 1 ($\tau = 2$) are shown in Figure 2.1. Recall, when $\tau = 2$, the frequent items are shown in the third column of Table 3.1.

Given a transactional database TDB and a minimum support (τ_m), an initial PP_M-tree can be constructed as follows. First, we scan the database to find all the frequent items, then, we scan

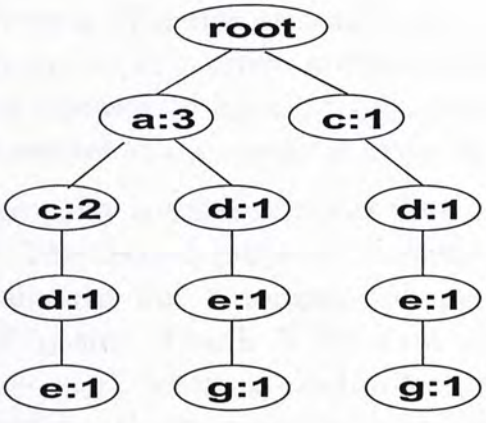


Figure 2.1: The memory representation (PP_M-tree)

the database again to construct PP_M-tree in memory. For each transaction, the infrequent items are removed. The remaining frequent items are sorted in a total order, and are inserted into PP_M-tree. The constructing time for PP_M-tree is slightly less than FP-Tree, because it does not need to build node-links in the tree initially.

2.4 *PP*-Mine

In this section, we propose a novel mining algorithm, called *PP*-Mine, using a PP_M-tree. For simplicity, we use a prefix-path to identify a subtree. Here, the prefix-path is expressed as a dot-notation to concatenate items with a total order. For example, in Figure 2.2, a-prefix identifies the leftmost subtree containing a, and $a.c$-prefix identifies the second subtree rooted at a-prefix. In the following, we use i_j and i_k for a single item prefix-path, and use α, β and γ for a prefix-path in general which are possible empty.

The *PP*-Mine algorithm is based on two properties. The first property states the Apriori property as below.

Property 1 *Given a PP_M-tree of rank N for a set of frequent itemsets $I = (i_1, i_2, \cdots, i_N)$, where a total order ($\preceq$) is defined on I. A pattern represented by $\alpha.i_j.i_k$-prefix can be frequent if the pattern represented by $\alpha.i_j$-prefix is frequent, where $i_j \preceq i_k$.*

The second property specifies subtrees that need to be mined for a pattern. The second property is given on top of two concepts: containment and coverage. We describe them below. Given a PP_M-tree of rank N for a set of frequent itemsets $I = (i_1, i_2, \cdots, i_N)$, where a total order ($\preceq$) is defined on I. We say a prefix-path (representing a subtree), α-prefix, is contained in $i_j.\alpha$-prefix, denoted α-prefix $\subseteq i_j.\alpha$-prefix. In addition, α-prefix $\subseteq \gamma$-prefix, if α-prefix $\subseteq \beta$-prefix and β-prefix $\subseteq \gamma$-prefix. A*coverage* a prefix-path α-prefix is defined as all the β-prefixes that contain α-prefix (including α-prefix itself).

Property 2 *Given a PP_M-tree of rank N for a set of frequent itemsets $I = (i_1, i_2, \cdots, i_N)$, where a total order ($\preceq$) is defined on I. Mining a pattern represented by a prefix-path α-prefix is to mine the coverage of α-prefix.*

For example, Figure 2.2 shows a *PP*-tree with four items {a, b, c, d}. Assume they are in lexicographic order. The coverage of *b.c.d*-prefix includes *b.c.d*-prefix and *a.b.c.d*-prefix. It implies that we only need to check these two subtrees, in order to determine whether the pattern, {b, c, d}, is frequent. Also, the coverage of *c.d*-prefix includes *c.d*-prefix, *b.c.d*-prefix, *a.c.d*-prefix and *a.b.c.d*-prefix. It implies that we only need to check these four subtrees, in order to determine whether the pattern, {c, d}, is frequent.

Based on the above two properties, we derive three main features including two pushing operations and a no-counting strategy below.

- **Push-down**: Processing at a node in a PP_M-tree is to check an itemset represented by the prefix-path from the

root to the node in question. Pushing-down to one of its children is to check the itemset with one more item. Property 1 states the Apriori heuristic. We implement it as a depth-first traversal with building a sub header-table.

- **Push-right**: Mining an itemset requires to identify a minimal coverage in PP_M-tree to mine. Property 2 specifies such a minimal coverage for any prefix-path.

  Pushing-right is a technique that helps to identify the coverage transitively, based on Property 2. In other words, the push-right strategy is to push the child to its corresponding sibling. We implement it as a dynamic link-justification.

  It is the best to illustrate it using an example. In Figure 2.2, after we have mined all the patterns in the leftmost subtree (a-prefix), we push-right $a.b$-prefix to the subtree b-prefix, push-right $a.c$-prefix to the subtree c-prefix, and push-right $a.d$-prefix to the subtree d-prefix. After these push-right operations, the whole coverage of b-prefix: $a.b$-prefix and b-prefix and part of the coverage of c-prefix including $a.c$-prefix and c-prefix are collected together.
  After mining the subtree (b-prefix), $b.c$-prefix is pushed to c, as well as $a.b.c$-prefix transitively. Plus the two subtrees $a.c$-prefix and c-prefix we collected in the former push-right operations, the whole coverage of c-prefix is identified.
  It is worth noting that the subtree $a.c$-prefix does not need to be pushed into the subtree $b.c$-prefix, because the former is to check the itemset $\{a, c, d\}$ excluding $\{b\}$, whereas the latter is to check the item $\{b, c, d\}$ excluding $\{a\}$.

- **No-counting**: Counting is done as a side-effort of pushing-right (dynamic link-justification) in an accumulated manner. For example, after we push-right $a.b$-prefix to the subtree b-prefix, all the prefix-paths and their support counts for b-prefix are collected by dynamic link-justification au-

tomatically. Therefore, all the counting cost is minimized. No extra counting is needed.

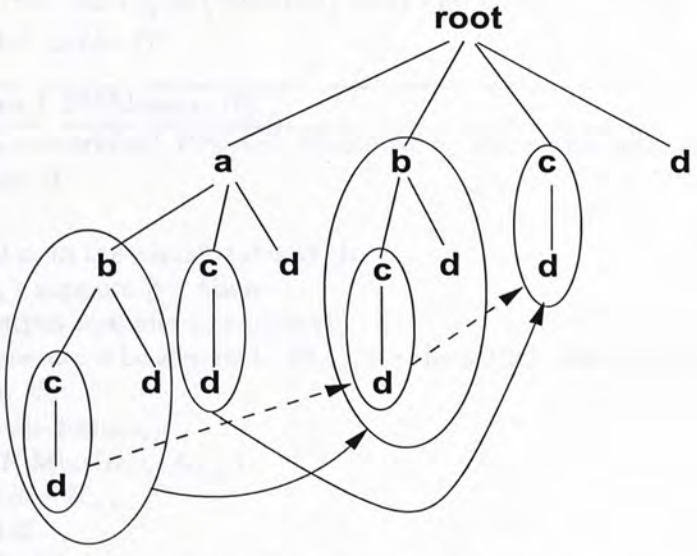


Figure 2.2: A PP_M-tree with four items

The *PP*-Mine algorithm is illustrated in Algorithm 1. The procedure is to check all the items in the header table H passed (line 1-10). In line 2-3, we check if the corresponding count (`num`) for a_i is greater than or equal to the minimum support, τ. Recall that counts are accumulated through pushing-right. If `num` for a_i is greater than or equal to τ, we output the pattern as represented by the path. Then, at line 4, a sub header table is created by removing all the entries before a_i (including a_i). Pushing-down a_i (line 5) is outlined below. Because the coverage of a_i-prefix has already linked through the `link` field in the header-table H (by the previous push-rights), all a_i's j-th children on the link are pushed-down (chained) into the corresponding j-th entry in the sub header table ($H_{\alpha.a_i}$). Line 6 calls *PP*-Mine recursively to check (k+1)-itemset if the length of the path is k. After returning, the sub header table will be deleted. Irrelevant with the minimum support, pushing-right a_i (line 9) is

described below: a) the coverage of a_i's left siblings are pushed-right from a_i to its right siblings, b) all a_i's j-th children on the link are pushed-right (chained) into the corresponding entry in the header table H.

Algorithm 1 *PP*-Mine(α, H)

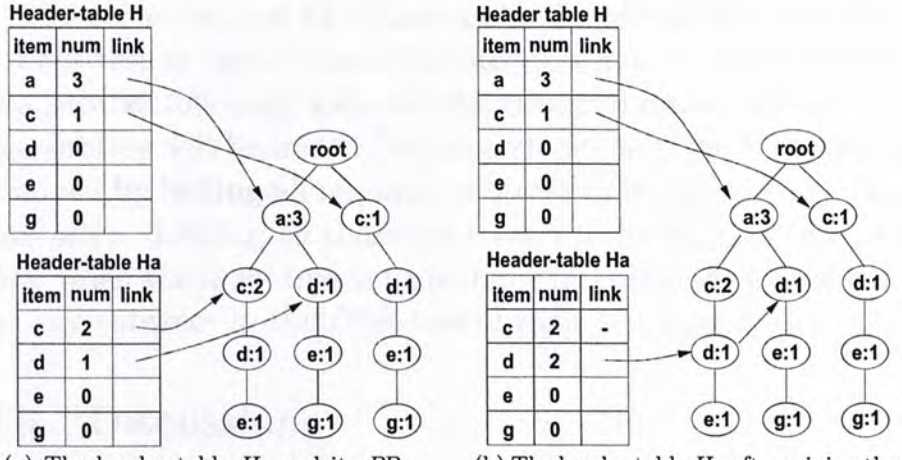
Input: A constructed PP_M-tree identified by the prefix-path, α, and the header table H.

```
1: for all a_i in the header table H do
2:   if a_i's support ≥ τ then
3:     output α.a_i and a_i's support;
4:     generate a header-table, H_{α.a_i}, for the subtree rooted at α.a_i, based
       on H;
5:     push-down(a_i);
6:     PP-Mine(α.a_i, H_{α.a_i});
7:     delete H_{α.a_i};
8:   end if
9:   push-right(a_i);
10: end for
```

Consider the mining process using the constructed PP_M-tree (Figure 2.1(a)). Here, the initial header table H includes all single items in PP_M-tree. Only the children of the root are linked from the header-table, and their counts are copied into the corresponding `num` fields in the header-table. Other `links/nums` in the header-table are initialized as null and zero. (The initial header H is shown in Figure 2.3 (a).)

1. Call *PP*-Mine(root, H). Item a is first to be processed, as the first entry in H. The support of a is 3. It is the exact total support for the item a, because a does not have any left siblings. Next, the subtree a-prefix is to be mined.

   The second header table, H_a, consists of all items in H except for a. Only the children nodes of a are pushed-down into H_a(F igure 2.3 (a)). In H_a, c and d counts are copied



(a) The header table H_a and its PP_M-tree rooted at a-prefix

(b) The header table H_a after mining the PP_M-subtree rooted at $a.c$-prefix

Figure 2.3: An Example

from the node $a.c$ and $a.d$, in the PP_M-tree. Their values are 2 and 1.

2. Call *PP*-Mine(a-prefix, H_a). Item c is picked up as the first entry in H_a. Because c's count (`num`) is 2 (frequent), we output $a.c$. Next, the subtree $a.c$-prefix is to be mined.

   The third header-table is constructed for the subtree of $a.c$-prefix, denoted as H_{ac}, in which d's `num` is 1 and d's link points to the node $a.c.d$. Other fields for e and g are set as zero/null.

3. Call *PP*-Mine($a.c$-prefix, H_{ac}). Item d is picked up. Because d's `num` is 1 (infrequent), return.

4. Backtrack to the subtree a-prefix. Here, the header-table H_a is reset (Figure 2.3 (b)). First, the entry c in H_a becomes null (done). Second, $a.c$'s child, d, is pushed-right into d's entry in the header-table H_a. In other words, the `link` of the entry d in H_a is linked to the node $a.d$ through the node $a.c.d$. The d's count (`num`) in H_a is accumulated to 2, which indicates $\{a, d\}$ occurs 2 times.

The correctness of *PP*-Mine can be showed as follows in brief. A PP_M-tree of rank N has N subtrees. First, we mine patterns in a subtree following a depth-first traversal order. All patterns in a subtree will be mined (vertically). Second, the k-th subtree is mined by linking all required subtrees in its left siblings (horizontally). Linking to those subtrees will be completed at the time when the k-subtree is to be mined. Third, the above holds for any subtrees in the PP_M-tree of rank N (recursively).

2.5 Discussions

In this subsection, we discuss the differences between *PP*-Mine and other similar approaches, FP-growth [15] and H-Mine [18]. We mainly compare the mining phase of the three algorithms, because the cost of constructing the FP-tree/H-struct in memory is almost the same.

- ***PP*-Mine vs FP-growth**: Both FP-growth and *PP*-Mine use the very similar data structure. But, the trees being constructed in memory are different. FP-growth requests that all nodes with the same item-names in FP-tree must be linked in the header table from the beginning. For example, in Figure 2.4(b), all three d-items must be linked from the header table. However, *PP*-Mine does not necessarily require a header table as a part of it (Figure 2.4(a)). The header table is a data structure used during the mining process, and can be easily constructed by only linking the children of the root in PP_M-tree[1] (as shown in Figure 2.3 (a)). With *PP*-Mine, the number of links is minimized. The reduction on the number of links has significant impacts on the performance, because the maintenance cost of those linkings are reduced. It is important to know that

[1]The set of frequent 1-itemset is known.

FP-tree can not be effectively stored on disk because of its complex node-link structures. When mining in memory, FP-growth processes FP-tree in a bottom-up fashion. Conditional FP-trees need to be constructed. Therefore, extra memory space is needed to further mine FP-tree. On the other hand, *PP*-Mine mines PP_M-tree using a depth-first traversal order. Constructing additional conditional FP-trees is replaced by dynamic link adjusting in PP_M-tree.

- ***PP*-Mine vs H-Mine**: While *PP*-Mine uses a tree structure, H-Mine uses a hyper-structure, H-struct, as shown in Figure 2.4(c). The main advantage of H-Mine is the dynamic hyper-link adjusting which was implemented at the expenses of using a hyper-structure. Sharing among itemsets becomes difficult. The hyper-structure has a problem such that the space requirement becomes high for dense dataset. H-Mine needs to be integrated with FP-growth for dense datasets. In fact, two data structures need to be used, namely, FP-tree and H-struct. Detecting whether a dataset is dense at run time is challenging. Relative support is used to detect if a projected dataset is dense. But, the accuracy is arguable. Also, the cost of switching from one structure to another needs to be considered. *PP*-Mine uses PP_M-tree in a novel way, and uses the similar dynamic hyperlink adjusting. In addition, *PP*-Mine uses an accumulation technique, it does not need to count the projected databases. In other words, we only do addition when we adjusting links. We do not need to count the projected database.

2.6 Performance Study

We conducted performance studies to analyze the efficiency of *PP*-Mine in comparison of FP-tree [15] and H-Mine [18]. We

did not compare *PP*-Mine with TreeProjection [3], because, as reported in [15], FP-growth outperforms TreeProjection.

All the three algorithms were implemented using Visual C++ 6.0. The synthetic data sets were generated using the procedure described in [4]. All our experimental studies were conducted on a 900MHz Pentium PC, with 128MB main memory and a 20GB hard disk, running Microsoft Windows/NT.

For a given minimum support τ, we assume that we have to construct PP_M-tree, FP-tree and H-struct in memory from scratch. The constructing time for both H-struct and PP_M-tree is marginally better than FP-tree construction. To give a fair view on this three algorithms, here we only compare the mining-phase of the three algorithms.

We have conducted experimental studies using the same datasets as reported in [15]. We report our results using one of them, T25.I20.D100K with 10K items, as representative. In this dataset, the average transaction size and average maximal potentially frequent itemset size are set to be 25 and 20, respectively, while the number of transactions in the dataset is 100K. There are exponentially numerous frequent itemsets in this dataset, when the minimum support is small. The frequent patterns include long frequent itemsets as well as a large number of short frequent itemsets.

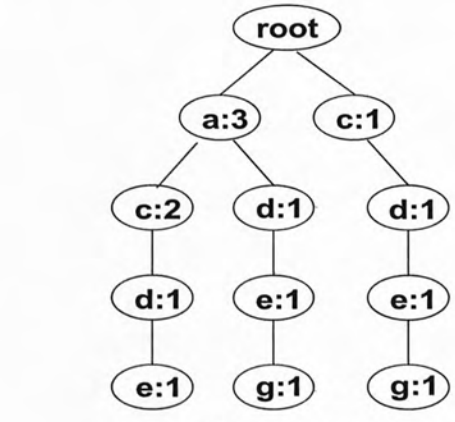
The scalability of the three algorithms, *PP*-Mine, FP-tree and H-Mine, is shown in Figure 2.5 (a). While the support threshold decreases, the number as well as the length of frequent itemsets increases. High overhead incurs for handling projected transactions. FP-growth needs to construct conditional FP-trees using extra memory space repeatedly. H-Mine needs to count every projected transactions. *PP*-Mine does not need to construct conditional trees and uses accumulation technique, which avoids unnecessary counting. From Figure 2.5 (a), we can see *PP*-Mine significantly outperforms FP-growth and H-Mine.

PP-Mine scales much better than both FP-tree and H-Mine.

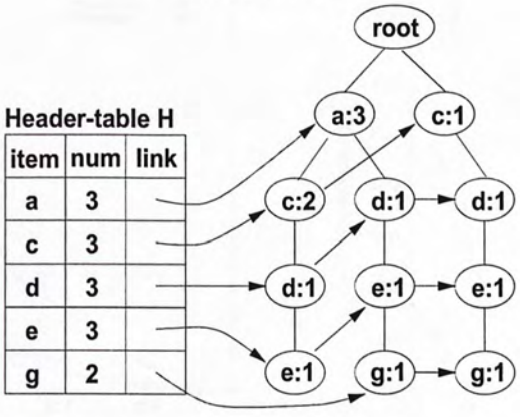
We also compared the mining phase of the three algorithms using a very dense dataset. The dataset was generated with 101 distinct items and 1K transactions. The average transaction size and average maximal potentially frequent itemset size are set to 40 and 10. When the minimum support is 40%, the number of frequent patterns is 65,540. When the minimum support becomes 10%, the number of frequent patterns is up to 3,453,240. As shown in Figure 2.5 (b), *PP*-Mine outperforms both FP-growth and H-Mine significantly. *PP*-Mine has the best scalability while the threshold decreases.

For sparse datasets and small datasets, *PP*-Mine marginally outperforms H-Mine, because both use the similar dynamic link adjusting technique. The effectiveness of *PP*-Mine's accumulation (or non-counting) techniques becomes weaker. Both *PP*-Mine and *H*-Mine outperform FP-growth.

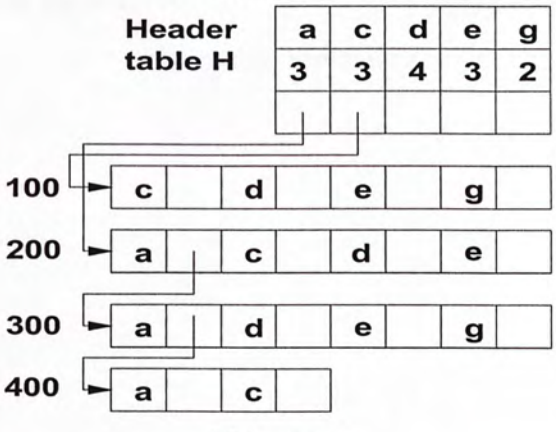
□ **End of chapter.**



(a) PP_M-tree

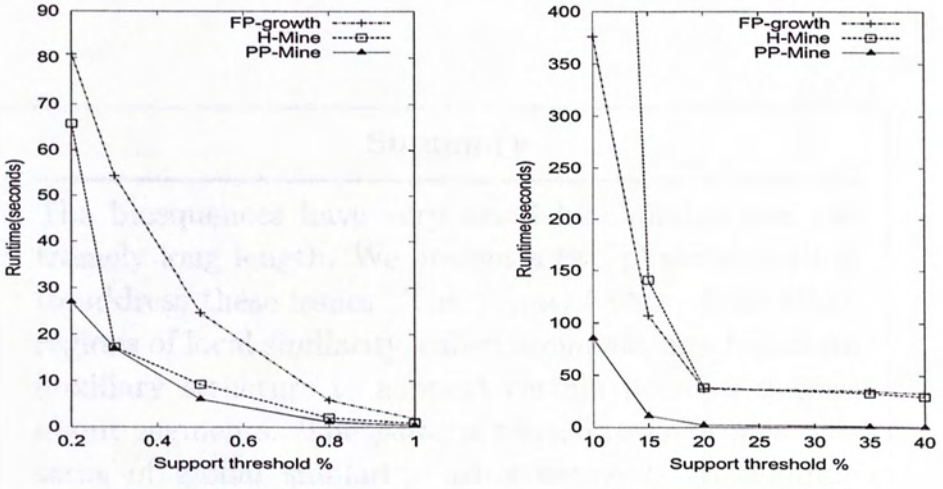


(b) FP-tree



(c) H-struct

Figure 2.4: PP_M-tree, FP-tree and H-struct for Example 1 where $\tau = 2$



(a) small threshold (τ) (b) large threshold (τ)

Figure 2.5: Scalability

Chapter 3

Fast Biosequence Patterns Mining

Summary

The biosquences have very small bioalphabet but extremely long length. We present a two-phase algorithm to address these issues. The *segment phase* finds short regions of local similarity, called *segments*, and builds an auxiliary structure to support certain position queries about segments. The *pattern phase* grows/prunes patterns of global similarity using segments as building blocks by answering position queries. The purpose of this two-phase approach is multi-fold: adopt best techniques separately for local similarity and global similarity, exploit local similarity for efficient search of global similarity, and grow patterns rapidly one segment at a time instead of one item at a time. We evaluate this approach on both synthetic and real life data sets

3.1 Background

3.1.1 Differences in Biosequences

Below, we analyze some differences in biosequences and their implications on data mining techniques.

Biosequences have a very small alphabet. DNA sequences are made up of 4 items (i.e., nucleotides) and protein sequences are made up of 20 items (i.e., amino acids), with repeats allowed. In contrast, customer transaction sequences in market-basket applications are made up of items taken from a collection of 1,000 to 10,000 items (e.g., sales items in a supermarket) [6, 5, 27, 17]. Consequently, while only a tiny fraction of items occurs in each transaction sequence, most (likely all) items occur in each biosequence. This has a drastic implication on all pruning strategies based on *absence of items*. For example, with every item occurring in a biosequence, the hash-partitioning of the hash-tree [5, 20] is not effective because most branches will be searched, and intersecting the idlists/bitmaps of patterns [6, 27] is not effective because idlists/bitmaps are very long for long sequences of a small alphabet.

Biosequences have extremely long length. A DNA or protein sequence is typically a few hundreds in length and can be up to a few thousands (http://www.ncbi.nlm.nih.gov), compared to 10 to 20 for a transaction sequence [6, 27, 5, 17]. Long sequences typically contain long patterns, and the classic "one item at a time" pattern growth [6, 27, 5, 17] means too many database scans and support countings. Also, scanning a long sequence for pattern matching is not a negligible cost, especially because pattern matching is performed frequently. The problem of mining long non-sequential patterns was studied recently [1, 7, 11]. For biosequences, the sequential nature and longer length make the problem significantly harder. See below and Section 2 for more discussions.

Biosequence patterns occur in short regions of local similarity and in long regions of global similarity with gaps. Such patterns have the form of $X_1 * \cdots * X_k$, where each X_i is a segment representing several consecutive items, and * represents a variable length don't care (VLDC). For DNA/protein sequences, each segment X_i represents to a local similarity conserved during evolution due to an important structural or functional role, and VLDCs represent the rest. A minimum requirement, such as minimum segment length, can be specified to remove trivial local similarity. This kind of local similarity has not been exploited in the classic sequential pattern mining [6, 27, 5, 17] for effective pattern pruning and support counting.

In summary, the blow up of classic sequential patterns is due to a large cardinality at each position, whereas the blow up of biosequence patterns is due to a large length. The former can be dealt with by classic *partitioning/indexing/bitmapping* techniques, but the latter requires novel techniques. The *look-ahead* technique [1, 7, 11] assumes that each item occurs *at most once*, as in a non-sequential pattern, and extends a pattern by all remaining items. If the extended pattern is frequent, all (non-maximal) subpatterns can be pruned. However, this technique is not applicable to sequential patterns where there is a lack of the notion of "remaining items" because an item can occur *repeatedly*. On the other hand, bio-applications ultimately require non-maximal patterns. For example, classification rules $X \rightarrow C$, where X denotes a biosequence pattern and C denotes a protein family, are likely non-maximal because they represent generalized characteristics of protein families.

3.1.2 Mining Sequential Patterns

The work on mining sequential patterns was motivated in market-basket analysis [6, 5, 16, 17, 20, 27], where typically sequences

are short and the alphabet is large. Both breadth-first generation [5, 20, 27] and depth-first generation [6, 27, 17] have been studied. The support counting is by scanning sequences against candidate patterns, with hash-partitioning to focus on likely candidates [5, 20], intersecting the idlists/bitmaps of shorter patterns [6, 27], projecting sequences according to scanned prefixes [17]. For long biosequences of a small alphabet, scanning or partitioning sequences is not effective as mentioned in Introduction, and idlists/bitmaps are very large because they code all occurring positions in all sequences. All these methods grow patterns one item at a time and do not exploit local similarity for pattern pruning and support counting.

3.1.3 Mining Long Patterns

The look-ahead technique extends a pattern by all "remaining items", i.e., items that have not occurred [1, 7, 11]. This technique is not applicable to sequential patterns because an item can occur repeatedly and there is no corresponding notion of remaining items. The *sampling/bordering* techniques [21, 25] find the border between frequent patterns and infrequent ones from a sample of the database, using techniques primarily for non-sequential patterns (such as those in [1, 7, 11]), and then adjust the border on the entire database. These methods still depend on an efficient mining algorithm in that a sample often has a non-trivial size to avoid the bias of sampling. For long sequences of a small alphabet, sampling (like partitioning) is less effective in that it does not reduce sequence length or alphabet size.

3.1.4 Related Works in Bioinformatics

A commonly used biosequence similarity is based on *multiple sequence alignment* (see [24]). This notion is useful when an

entire sequence is similar, but makes no sense for mining short regions of local similarity. Most approaches to local similarity enumerate the solution space [10], thus, are very highly expensive. [22] addresses this problem using heuristics at the expense of missing some patterns. It first ranks patterns using a sample of database and evaluates only highly ranked patterns against the entire database. The pattern growth for the sample is by enumeration, and pattern matching against a sequence is by dynamic programming. Their experiments were conduced only for 150 sequences and two-segment patterns. Our approach finds the complete set of patterns on a large collection of sequences by exploiting novel pattern pruning and support counting methods. Another direction is *approximate pattern matching* [22, 25]. Our work currently considers exact pattern matching.

3.2 The Overview

3.2.1 The Problem

A sequence database D is a collection of sequences $\{s_1, \cdots, s_N\}$. Each sequence is an ordered list of items from a fixed alphabet. The jth item in a sequence occurs at position j. A *segment* refers to one or more items at consecutive positions in a sequence. $|X|$ denotes the number of positions in a segment X. A segment could occur more than once in a sequence, with each occurrence having a start position and end position. A *pattern* has the form $X_1 * \cdots * X_n$ $(n \geq 1)$, where each X_i is a segment and $*$ denotes the variable length "don't care" (VLDC). In matching a pattern $X_1 * \cdots * X_n$ with a sequence s_i, each segment X_j matches itself and each * can substitute for zero or more items. If a match of the pattern is found in s_i, we say that s_i *contains* the pattern, or the pattern *occurs* in s_i.

Definition 1 *The* support *of a pattern is the percentage of the*

sequences in D that contain the pattern. Given a minimum length $MinLen$ and a minimum support $MinSup$, a pattern $X_1 * \cdots * X_n$ is densely frequent *if $|X_i| \geq MinLen$ for $1 \leq i \leq n$ and the support of the pattern is above $MinSup$. The problem of* mining biosequence patterns *is to find all densely frequent patterns.*

A segment X_i captures a short region of local similarity. A pattern $X_1 * \cdots * X_n$ captures a global similarity across a global range separated by gaps $*$. For biosequences where most items occur in most sequences due to the small alphabet size, only local similarities above some minimum length (usually > 1) are non-trivial. The minimum support $MinSup$ conveys a statistical significance requirement on a pattern. In the rest of the paper, a "frequent pattern" means a "densely frequent pattern".

3.2.2 The Overview of Our Approach

We propose a two-phase approach. The *segment phase* finds all frequent segments X_i. The *pattern phase* generates all frequent patterns $X_1 * \cdots * X_k$ using frequent segments X_i as building blocks. The purpose of this two-phase approach is to exploit relationships between frequent segments for the "segment-based pruning" and "query-based counting" discussed in Section 1.2. Two issues are addressed. First, how to search the candidate space $X_1 * \cdots * X_k$ so as to maximize the exploitation of such relationships. We present a novel search strategy, called *2-phased depth-first (2PDF)*, to address this issue. Second, how to tell efficiently if a pattern occurs in a sequence, knowing that it does not occur before some end position? We solve this problem by answering the following position query.

Definition 2 *A* position query *has the form of $Q(X, s, i)$, where X is a frequent segment, s is a sequence id, and i is a position in*

sequence s. The query returns the answer $< s, j >$ if sequence s contains X at a start position greater than i and j is the smallest such start position; or returns nil otherwise.

Precisely, suppose that we know the smallest end position $< s, i >$ of a pattern $P_k = X_1 * \cdots * X_k$ in each sequence s containing the pattern (i.e., P_k does not occur before the position i in s). We find the smallest end positions of a pattern $P_{k+1} = P_k * X_{k+1}$ as follows. For each $< s, i >$ of P_k, if $Q(X_{k+1}, s, i)$ returns nil, s does not contain P_{k+1}; if it returns an answer $< s, j >$, s contains P_{k+1} at the smallest end position $j + |X_{k+1}| - 1$. The support count of P_{k+1} is equal to the number of $< s, j >$ answers. The key to this approach is an efficient method for answering a position query $Q(X, s, i)$. Scanning the whole sequence s is not attractive because most part of a long sequence does not code useful information, nor is materializing all (X, s) pairs because of a high number of frequent segments X and sequences s. We propose an index method and a compression method for answering the position query.

3.3 The Segment Phase

3.3.1 Finding Frequent Segments

We use the *generalized suffix tree (GST)* [22] to find all frequent segments. A GST is an extension of the suffix tree for representing a set of sequences. A suffix starting at position p in a sequence s_i is represented by a leaf containing $s_i : p$. The edges are labeled with items such that the concatenation of the edge labels on the path from the root to the leaf containing $s_i : p$ is the suffix of the sequence s_i that starts at position p. We extract the following information from the GST. (1) The frequent segments of the length $MinLen$, called *base segments* and denoted by B_i, for each base segment B_i, the *position lists* $s_i : p_1, p_2, \cdots$

at which B_i starts, where $p_i < p_{i+1}$. This information is used to build an index. (2) All remaining frequent segments, but no position lists. This information is used to generate patterns.

Theorem 1 *The total length of the positions lists of base segments is no more than the total length of sequences in D.*

Proof. No two base segments occurs at the same position in a sequence (otherwise, they are identical). Thus, the total length of the position lists for the base segments in a single sequence is no more than the length of the sequence.

The time and space needed to construct the GST is $O(|D|)$, where $|D|$ is the total length of the sequences in database D [22]. If the GST of D does not fit in the memory, we can build the GST for one partition of D at a time (that fits in the memory) and scan D once to count the global support for frequent segments extracted from each partition. Below, we present two approaches of exploiting the extracted information for efficient query answering.

3.3.2 The Index-based Querying

This approach provides a direct access to the positions queried by $Q(X, s, i)$ by building a main-memory index. Instead of indexing all frequent segments X, which would be too large, we index only base segments B_i, in the view that every frequent segment can be rewritten into one or more base segments as stated below.

Definition 3 *For two base segments B_1 and B_2 such that the last k items in B_1 are identical to the first k items in B_2, the* k-join *of B_1 and B_2, denoted $B_1 \bowtie_k B_2$, is the segment obtained by overlapping the last k items of B_1 with the first k items of B_2.*

Corollary 1 *A frequent segment X_i can be rewritten into several k-joins of base segments:*

$$B_1 \bowtie_0 B_2 \bowtie_0 \cdots \bowtie_0 B_p \bowtie_k B_{p+1},$$

where $k = 0$ except for the last k-join. $p = \lfloor |X_i|/MinLen \rfloor$ and $k = |X_i| - \lfloor |X_i|/MinLen \rfloor \times MinLen$.

Example 2 *Table 3.1 shows a sequence database containing three sequences, with the alphabet of $\{a, b, c, d\}$. Let $MinSup = 2/3$, and $MinLen = 2$. We have the frequent segments:*

$$ab(2),\ ac(3),\ acd(3),\ acda(2),\ cd(2),\ cda(2),\ da(2),$$

where the integers in the brackets are support counts. The base segments are

$$B_1 = ab,\ B_2 = ac,\ B_3 = cd,\ B_4 = da,$$

and their position lists are given in Table 3.2. $ab * cda$ occurs in s_1 and s_2, so is a frequent pattern. $ab * cda$ can be rewritten as $B_1 * (B_3 \bowtie_1 B_4)$ using only base segments. Similarly, $ab * acda$ is frequent and can be rewritten as $B_1 * (B_2 \bowtie_0 B_4)$.

Definition 4 *The* SP-index (Segment-to-Position index) *has two components, the* root directory *and the* SP-trees. *For each base segment B_i, the root directory has an entry for the root of the SP-tree for B_i. The* SP-tree for B_i *is a B-tree containing an entry $(< s, p >, ptr)$ for each position $s : p$ in the position lists of B_i. $< s, p >$ is the search key of the B-tree. ptr points to the entry $(< s, p + |B_i| >,\ ptr')$ in the SP-tree for some B_j, if it exists, or $ptr = nil$, otherwise. (We have omitted the usual tree pointers in the B-tree.)*

Intuitively, *ptr* in an entry $(< s, p >, ptr)$ links the entries for B_i and B_j (in different SP-trees if $B_i \neq B_j$) that 0-joins, i.e., $B_i \bowtie_0 B_j$, at the start position $s : p$. At any entry $(< s, p >, ptr)$ in the SP-tree for B_1, we can

ID	Sequence
s_1	*abacdab*
s_2	*abcacda*
s_3	*baacdca*

Table 3.1: The sequence database D

Base Segments	Position Lists
ab	$(s_1 : 1, 6), (s_2 : 1)$
ac	$(s_1 : 3), (s_2 : 4), (s_3 : 3)$
cd	$(s_1 : 4), (s_2 : 5), (s_3 : 4)$
da	$(s_1 : 5), (s_2 : 6)$

Table 3.2: The position lists

- *check* if a frequent segment $X = B_1 \bowtie \cdots \bowtie B_q$ occurs at the start position $s : p$ by following the *ptr* pointers (possibly across SP-trees) at most q times starting from the entry $(< s, p >, ptr)$. We refer to this chain of *ptr* pointers as the *join chain* of B_1.

- *move* to the next entry of B_1 in the search key order as provided by the standard B-tree. We refer to this chain of "next entry" as the *occurrence chain* of B_1, the end of which is indicated by a change of sequence id in a search key.

Now we compute the position query using the SP-index as follows.

Compute $Q(X, s, i)$, where $X = B_1 \bowtie \cdots \bowtie B_q$, $q \geq 1$. Search the SP-tree for B_1 by the search key $< s, i >$ for the smallest start position in s greater than i. If the search is not successful, return nil. Assume that the search is successful and ends at an entry $(< s, p >, ptr)$. Check if X occurs at the current start position $s : p$ by following the join chain of B_1. If not, move to the next entry of B_1 in s by following the occurrence chain of B_1, and check again. This "move and check" is repeated until either the end of the occurrence chain of B_1 is reached or the checking is successful. In the former case, return nil. In the latter case, return the key value $< s, j >$ in the last entry of B_1 accessed, which is the smallest start position of X in s greater than i.

The *partial-key technique* for main-memory indexes [9] can be applied to eliminate repeated store of sequence ids s in key values $< s, p >$ in a SP-tree. This not only reduces the index size, but also increases the fanout of the tree structure and reduces the access time.

3.3.3 The Compression-based Querying

Alternatively, we can answer the query $Q(X, s, i)$ by simply scanning the sequence s for the next occurrence of X starting from the position i. For a long sequence, this suffers from scanning a long region not coding useful information. In the compression-based querying, we first compress each input sequence by collapsing consecutive *non-coding regions*, i.e., positions not expandable to either sides into a frequent segment, into a new item ϵ. Intuitively, each ϵ represents a variable length non-coding region between two closest coding regions in a sequence. For large $MinLen$ and $MinSup$ and a large alphabet size, a long sequence tends to contain long non-coding regions and the compressed sequence will shrink in length substantially. The

Input: a sequence s in D and $MinLen$;
Ouput: the compressed sequence of s and length;

```
1: k = 1;
2: for j = 1; j ≤ |s|; j++ do
3:    if s[j, j + MinLen − 1] is a base segment then
4:       fill S[k] with s[j];
5:       k++;
6:       coding ← MinLen − 1;
7:    else if coding > 0 then
8:       fill S[k] with s[j];
9:       k++;
10:      coding − −;
11:   else if k > 1 and S[k − 1] is not ϵ then
12:      fill S[k] with ϵ;
13:      k = k++;
14:   end if;
15: end for;
16: return S and k − 1.
```

Figure 3.1: Compressing a sequence

procedure in Figure 3.1 compresses a given sequence.

Example 3 *For the database in Example 2, the compressed sequences are*

S_1 : *abacdab*
S_2 : *abϵacda (ca is collapsed into ϵ)*
S_3 : *ϵacdϵ (ba and ca are collapsed into ϵ)*

To apply the compression-based approach, we first extract base segments (without position lists this time) and frequent segments from the sequence database D, as described earlier. We then compress each sequence s in D using the algorithm in Figure 3.1. Subsequently in the pattern phase, we shall answer a query $Q(X, S, i)$ by scanning compressed sequences S instead of input sequences s.

Compute $Q(X, S, i)$, where S is a compressed sequence, X is a frequent segment and i is a position in S. Scan the sequence S starting from the position i and search for the next occurrence of X. Return the position of the next occurrence of X if found, and return nil if not. Note that, in matching X against S, the new item ϵ in S does not match any item in X because it represents a non-coding region.

3.4 The Pattern Phase

This phase generates all frequent patterns $X_1 * \cdots * X_k$ using frequent segments X_i found in the segment phase. The *depth-first generation* is to *extend* the current pattern $X_1 * \cdots * X_k$ by each frequent segment X_{k+1} in the *depth-first manner*, and if $X_1 * \cdots * X_k * X_{k+1}$ is frequent (by support counting), *recursively* extend the pattern $X_1 * \cdots * X_k * X_{k+1}$. However, this simple method is not efficient because the number of candidates X_i at each step can be very large (as X_i can be repeatedly used in a pattern) and the support counting is independent for each pattern. We are interested in exploiting "interactions" between patterns for pruning candidates and sharing support counting. Below, we present a search strategy with this mind.

Definition 5 *The* 2-phased depth-first generation *is the depth-first generation of the pattern tree defined below:*

- *The* segment tree *represents all frequent segments: each node (except the root) is labeled by a base segment B_i, each non-terminal edge is labeled by integer 0, and each terminal edge is labeled by an integer $k \geq 0$, such that a node w represents the frequent segment $B_1 \bowtie_0 \cdots \bowtie_0 B_{p-1} \bowtie_k B_p$, denoted $seg(w)$, where $(root, 0, B_1, 0, \cdots, 0, B_{p-1}, k, B_p)$ is the path from the root to a node w.*

- *The* pattern tree *represents all patterns: every node has the child nodes* $v_1, \cdots, v_n$ *labeled by* $seg(w_1), \cdots, seg(w_n)$ *such that* $w_1, \cdots w_n$ *are the non-root nodes in the depth-first order in the segment tree. A node* v *in the pattern tree represents the pattern* $X_1 * \cdots * X_k$*, denoted* $pat(v)$*, where* $X_1, \cdots, X_k$ *are on the labels on the path from the root to* v*.*

A *snode* refers to a node in the segment tree, and a *pnode* refers to a node in the pattern tree. Similarly, *sroot* and *proot* refer to the root of the segment tree and pattern tree. Note that $pat(proot) = \emptyset$ and $seg(sroot) = \emptyset$.

Example 4 *Figure 3.2 shows the segment tree for Example 2.* w_i *denotes the* i*th node in the depth-first order. The path* $(root, 0, B_2, 1, B_3)$ *or node* w_3 *represents the frequent segment* $acd = B_2 \bowtie_1 B_3$*, the path* $(root, 0, B_2, 0, B_4)$ *or node* w_4 *represents the frequent segment* $acda = B_2 \bowtie_0 B_4$*, and the path* $(root, 0, B_3, 1, B_4)$ *or node* w_6 *represents the frequent segment* $cda = B_3 \bowtie_1 B_4$*.*

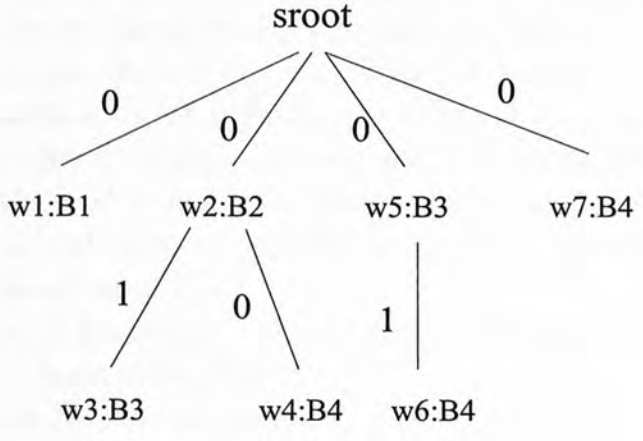


Figure 3.2: The segment tree in Example 4

Consider $P = X_1 * \cdots * X_{k-1} * X_k$, $P' = X_1 * \cdots * X_{k-1} * X'_k$ or $P' = X_1 * \cdots * X_{k-1} * X_k * \cdots * X_{k+i}$, where X_k is a prefix of X'_k. Note that each occurrence of P' is an occurrence of P.

An important property of the 2-phased depth-first generation is that P is generated before P' generation because P is either a left sibling or an ancestor of P'. The following observations, which exploit some information about P for generating P', form the basis of our pruning and querying strategies.

Observation 1. If P does not extend into a frequent pattern $P * X$ for some frequent segment X, neither does P'.

Observation 2. If P does not occur before position i in some sequence s, neither does P'.

We present pruning strategies based on Observation 1 and querying strategies based on Observation 2.

3.4.1 The Pruning Strategies

Consider a pnode v and a snode w. The *pruning signature* of the pattern $pat(v)$ refers to the set of snodes w that failed to extend v, i.e., $pat(v) * seg(w)$ was known not frequent. Let $v.failed$ denote the pruning signature of $pat(v)$. Since $w \in v.failed$ implies that $w' \in v.failed$ for all descendants w' of w, the implementation of $v.failed$ needs to contain only highest possible snodes on a path. We assume that $proot.failed = \emptyset$.

Rationale I. If w failed to extend v (i.e., $w \in v.failed$), for all snodes w' below w, w' will fail to extend v. Therefore, we do not need to extend v by w'.

Pruning I. For every $w \in v.failed$, we can prune the subtree rooted at w from extending v.

Rationale II. For the parent v_p of v, if w failed to extend v_p, w fails to extend v (Observation 1). Therefore, we do not need to extend v by w.

Pruning II. For the parent v_p of v, $v.failed \supseteq v_p.failed$.

Rationale III. For any sibling v_s of v such that the label of v_s is a prefix of the label of v, if w failed to extend v_s, w fails to extend v (Observation 1).

Pruning III. For any sibling v_s of v such that the label of v_s is the parent of the label of v in the segment tree, $v.failed \supseteq v_s.failed$.

From Pruning II, $v_s.failed \supseteq v_p.failed$, thus, Pruning III has priority over Pruning II because of stronger pruning. Pruning II is used only in the case that the parent of the label of v is the sroot, in which case the sibling v_s in Pruning III does not exist. Pruning I and II provide ample opportunities of pruning: whenever either v or w is not a leaf node and w failed to extend v, the whole subtree at w is pruned in the subspace below v in the pattern tree. Pruning III improves on this with even stronger pruning.

Example 5 *Consider the pattern tree in Figure 3.3 generated using the segment tree in Figure 3.2. v_i denotes the ith node in the depth-first generation. Initially, $v_1.failed = \emptyset$. After extending v_1 by w_1, we find that $seg(w_1)*seg(w_1)$ (i.e., $ab*ab$) is not frequent, so $v_1.failed = \{w_1\}$. At node v_3, $v_3.failed = \{w_1\}$ from Pruning II, so we do not extend v_3 by w_1. After extending v_3 by w_2, we find that $seg(w_1)*seg(w_2)*seg(w_2)$ (i.e., $ab*ac*ac$) is not frequent, so $v_3.failed = \{w_1, w_2\}$. From Pruning I, we do not need to extend v_3 by any node in the subtree below w_2 (i.e., w_3 and w_4). At node v_5, since the label of the sibling v_3 (i.e., $seg(w_2)$) is the parent of the label of v_5 (i.e., $seg(w_3)$) in the segment tree, from Pruning III, $v_5.failed = \{w_1, w_2\}$. So, all the pruning at node v_3 applies to node v_5.*

Algorithm. Consider the current path $X_1, \cdots, X_k$ in the 2-phased depth-first generation. Assume that X_i is represented by a path $w_1^i, \cdots, w_n^i$ in the segment tree, which is the current path in the depth-first generation of X_i. Note that $X_i = seg(w_n^i)$. Let

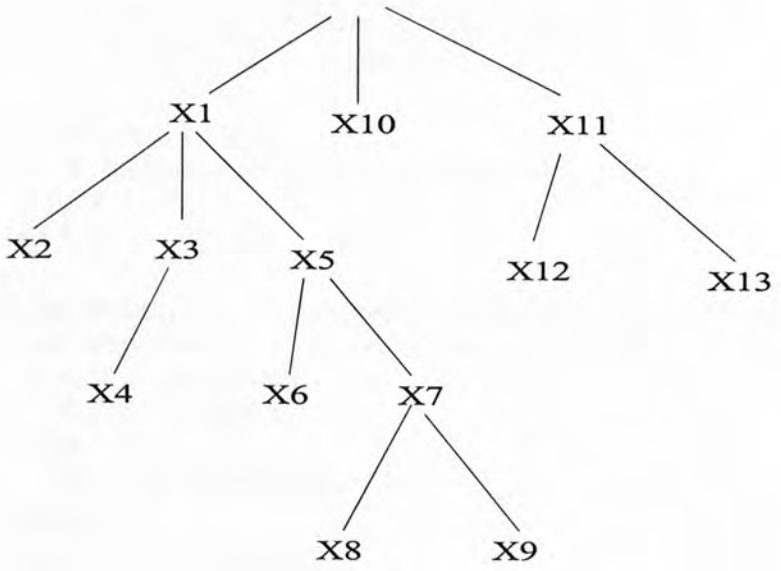


Figure 3.3: The pattern tree in Example 5

$failed_j^i$ denote the pruning signature of $X_1 * \cdots * X_{i-1} * seg(w_j^i)$. For $1 \leq i \leq k$, we push $(w_1^i, failed_1^i), \cdots, (w_n^i, failed_n^i)$ onto the stack π_i in the depth-first generation of X_i. $node(\pi_i)$ and $failed(\pi_i)$ return the top entry on the stack π_i, i.e., w_n^i and $failed_n^i$. Thus, $seg(node(\pi_1)) * \cdots * seg(node(\pi_k))$ represents the current pattern, and $failed(\pi_k)$ gives its pruning signature. $2PDF(k, w)$ in Algorithm 2 finds all frequent patterns that have the prefix $seg(node(\pi_1)) * \cdots * seg(node(\pi_k)) * seg(w')$, where w' is either w or a child node of w. $2PDF(0, sroot)$ finds all frequent patterns.

$2PDF(k, w)$: If w is the sroot, lines 2-4 recursively call $2PDF(k, w')$ for all children w' of w not in $failed(\pi_k)$ (Pruning I). If w is not the sroot, line 6 extends $seg(node(\pi_1)) * \cdots * seg(node(\pi_k))$ by $seg(w)$. If the extension fails, lines 20-21 add w to $failed(\pi_k)$ and remove any descendant of w because we keep only highest possible nodes in $failed(\pi_k)$. Assume that the extension succeeds. Line 13 creates a new pnode for the extension by pushing w and $failed$ onto stack π_{k+1}, where $failed$ is initialized at lines

Algorithm 2 The 2-phased depth-first with pruning strategies

$2PDF(k, w)$: an integer $k \geq 0$, a snode w;

```
1: if w is the sroot then
2:   for all children w' of w, not in failed(π_k) do
3:     2PDF(k, w');
4:   end for;
5: else
6:   if seg(node(π_1)) * ··· * seg(node(π_k)) * seg(w) is frequent then
7:     output seg(node(π_1)) * ··· * seg(node(π_k)) * seg(w);
8:     if π_{k+1} is empty then
9:       failed ← failed(π_k);
10:    else
11:      failed ← failed(π_{k+1});
12:    end if;
13:    push(w, failed, π_{k+1});
14:    2PDF(k + 1, sroot);
15:    pop(π_{k+1});
16:    for all children w' of w, not in failed(π_k) do
17:      2PDF(k, w');
18:    end for;
19:  else
20:    add w to failed(π_k);
21:    remove any descendant of w from failed(π_k);
22:  end if;
23: end if;
```

9 and 11 according to Pruning II and III, respectively, and line 14 recursively calls $2PDF(k+1, sroot)$ for the new pnode. On return, lines 16-18 recursively call $2PDF(k, w')$ for all children w' of w not in $failed(\pi_k)$ (Pruning I). We omit the formal proof of correctness.

3.4.2 The Querying Strategies

In a similar spirit, we can prune the work of support counting based on Observation 2. The *querying signature* of $P_k = X_1 * \cdots * X_k$ refers to the set of $< s, i >$ pairs such that i is the

smallest start position of X_k in any occurrence of P_k in s. This implies that P_k does not occur before the end position $i + |X_k|$ in s. From Observation 2, we have two ways to compute the querying signature (and the support count) of $P_{k+1} = P_k * X_{k+1}$. In the first way, for each $< s, i >$ in the querying signature of P_k, we perform the query $Q(X_{k+1}, s, i + |X_k|)$ to find the smallest start position $< s, j >$ of X_{k+1} in any occurrence of P_{k+1}. The querying signature (resp. the support count) of P_{k+1} is the set (resp. the number) of $< s, j >$ answers to such queries. In the second way, we use the querying signature of $P'_{k+1} = P_k * X'_{k+1}$ instead, where X'_{k+1} is a prefix of X_{k+1}.

In implementation, we now push $(w_1^i, failed_1^i, ans_1^i)$, $\cdots$, $(w_n^i, failed_n^i, ans_n^i)$ onto the stack π_i, where ans_j^i represents the querying signature of the pattern $X_1 * \cdots * X_{i-1} * seg(w_j^i)$. Let $ans(\pi_i)$ return the top querying signature of π_i, i.e., ans_n^i. The querying strategies for $P_{k+1} = P_k * X_{k+1}$ are as follows.

Querying I. If the parent of X_{k+1} is the sroot, in which case π_{k+1} is empty, compute $Q(X_{k+1}, s, i + |X_k|)$ for each $< s, i >$ in the querying signature of P_k, given by $ans(\pi_k)$.

Querying II. If the parent X'_{k+1} of X_{k+1} is not the sroot, compute $Q(X_{k+1}, s, i)$ for each $< s, i >$ in the querying signature of P'_{k+1}, given by $ans(\pi_{k+1})$.

Algorithm. $2PDF(k, w)$ in Algorithm 3 describes the depth-first generation with both pruning and querying strategies. The main difference from Algorithm 2 is the function $Count(seg(w), k)$ at line 6 for computing the support and querying signature of the candidate $seg(node(\pi_1)) * \cdots * seg(node(\pi_k)) * seg(w)$. If π_{k+1} is empty, $Count(seg(w), k)$ applies Querying I at lines 4-9, if not, Querying II at lines 11-16.

Algorithm 3 The 2-phased depth-first with pruning/querying strategies

$2PDF(k, w)$:

```
1: if w is the sroot then
2:   for all children w' of w, not in failed(π_k) do
3:     2PDF(k, w');
4:   end for;
5: else
6:   (sup, ans) ← Count(seg(w), k);
7:   if sup/N ≥ MinSup then
8:     output seg(node(π_1)) * ··· * seg(node(π_k)) * seg(w);
9:     if π_{k+1} is empty then
10:      failed ← failed(π_k);
11:    else
12:      failed ← failed(π_{k+1});
13:    end if;
14:    push(w, failed, ans, π_{k+1});
15:    2PDF(k + 1, sroot);
16:    pop(π_{k+1});
17:    for all children w' of w, not in failed(π_k) do
18:      2PDF(k, w');
19:    end for;
20:  else
21:    add w to failed(π_k);
22:    remove any descendant of w from failed(π_k);
23:  end if;
24: end if;
```

$Count(X, k)$:

```
1: sup ← 0;
2: ans ← ∅;
3: if π_{k+1} is empty then
4:   for all < s, i >∈ ans(π_k) do
5:     if Q(X, s, i + |seg(node(π_k))|) returns < s, j > then
6:       sup + +;
7:       ans ← ans ∪ {< s, j >};
8:     end if;
9:   end for
10: else
11:   for all < s, i >∈ ans(π_{k+1}) do
12:     if Q(X, s, i) returns < s, j > then
13:       sup + +;
14:       ans ← ans ∪ {< s, j >};
15:     end if;
16:   end for;
17: end if;
18: return (sup, ans);
```

3.5 Experiment

We evaluated the performance of our methods, denoted by 2PDF-Index for index-based querying and 2PDF-Compression for compression-based querying. We compare them with two depth-first sequential pattern mining algorithms, PrefixSpan [17] and SPAM [6], which were shown to outperform earlier algorithms such as [5, 20, 27] [1]. To be consistent with our notion of patterns, we made two modifications to PrefixSpan and SPAM: simplifying each transaction in a customer sequence to a single item, and considering two types of pattern growth, i.e., adding the next item to join the last segment in a pattern, or adding the next item to start a new segment in a pattern. These modifications make PrefixSpan and SPAM more efficient. We do not compare with [22] that does not find all patterns. All experiments were conducted on a PC with 2GHZ CPU and 1GB memory running the Windows 2000 Professional.

3.5.1 Synthetic Data Sets

The first set of experiments was conducted on the synthetic data sets generated as in [5] using the parameters in Table 3.3. We used the data sets named in Table 3.4. The data sets with $N = 4$ (the alphabet size) simulate DNA sequences, the data sets with $N = 20$ simulate protein sequences, and the data set with $N = 10,000$ simulates customer sequences. The average length C of simulated biosequences, i.e., 128 or 256, is much longer than that of simulated customer sequences, i.e., 20. N_I is equal to N because all transactions are singletons. Like in [5], N_S was set to 5000. For a larger C and a smaller N, we use a larger $MinLen$ due to more expected local similarity. For the customer sequence data set, the setting of $MinLen = 1$ yields

[1] For PrefixSpan, we used the *pseudo-projection* technique as suggested in [17], which makes PrefixSpan faster than SPAM.

classic sequential patterns.

Execution time. In Figures 3.4-3.7, the first column plots the execution time in logarithm scale, i.e., $log_{10}T$ where T is the execution time, against $MinSup$. For 2PDFs, this includes the time for both phases. The most significant finding is that both versions of 2PDF are several orders of magnitude faster than PrefixSpan and SPAM on long sequences of a small alphabet. Several factors contributed to this speedup: the reduced frequency of support counting, the pruning of candidates in the pattern phase, and the indexed access or reduced sequence scan in support counting. Table 3.5 shows the number of base segments, frequent segments, and frequent patterns for the data set $C128S32N4D100K$ at $MinLen = 5$. The second finding is that 2PDF-Index is more scalable wrt $MinSup$ than 2PDF-Compression, due to the insensitivity of index access cost to the increase of base segments.

Figure 3.8 shows the execution time for the simulated customer data set and reveals two things. First, the mining task for biosequences is much more difficult than for classic transaction sequences, as indicated by the huge difference in both execution time and minimum support. Second, the 2PDFs, though aimed at long sequences of small alphabets, are also highly competitive for short sequences of large alphabets.

The second column in Figures 3.4-3.8 shows the portion of execution time prior to the depth-first generation, called "Building Time". This refers to the segment phase for 2PDFs and the preparation time for other algorithms. 2PDFs, especially 2PDF-Index, spend more building time than the other algorithms to build the index or compress sequences. However, the reduced overall execution time, as shown above, confirms that this "investment" is worthwhile.

Space consumption. The third column of Figures 3.4-3.8 shows the maximum space required for the current path in

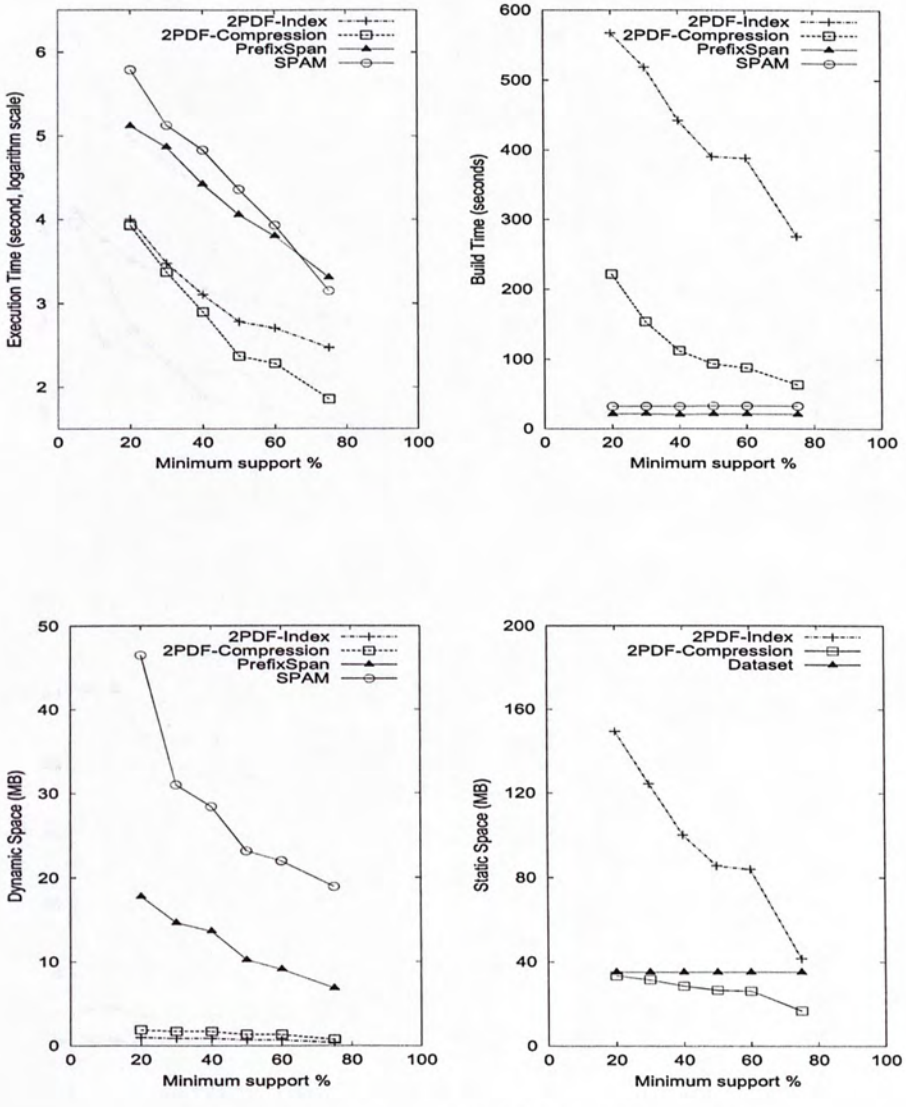


Figure 3.4: C128S32N4D100K, MinLen = 5

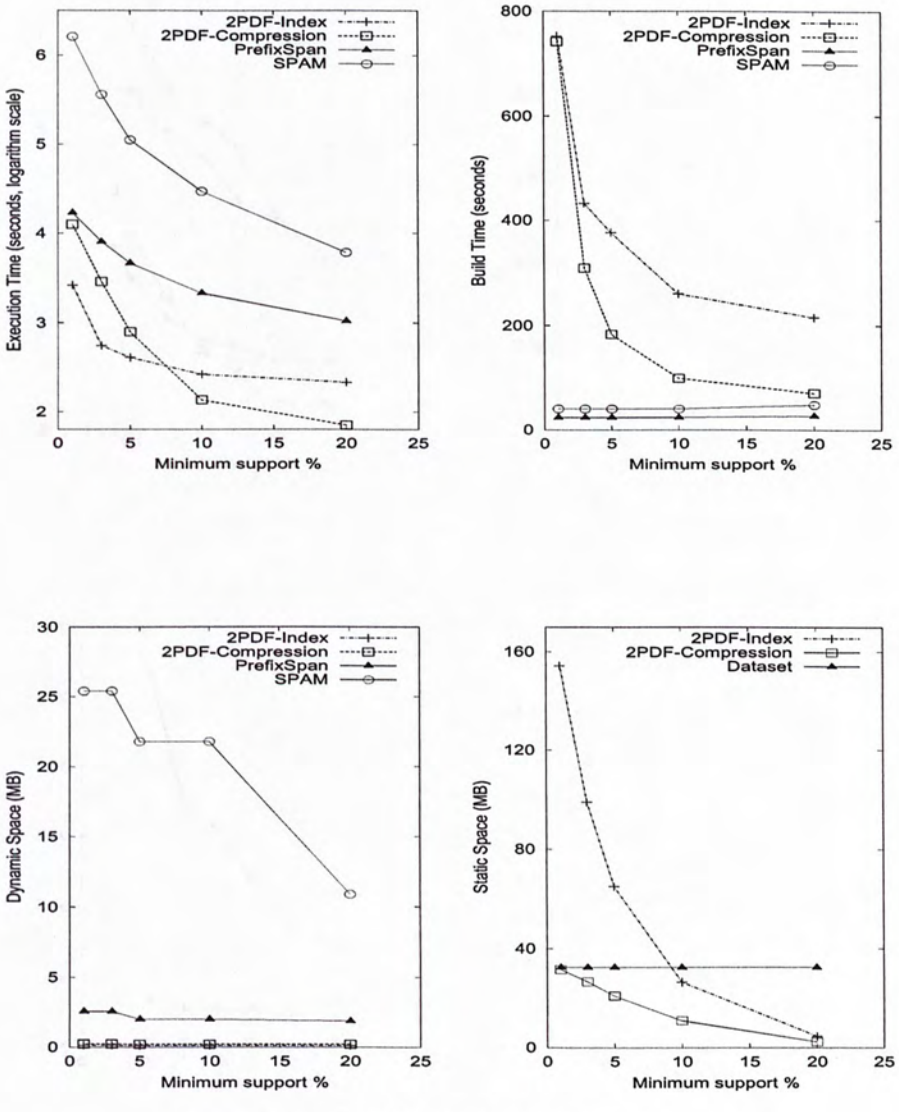


Figure 3.5: C128S32N20D100K, MinLen = 3

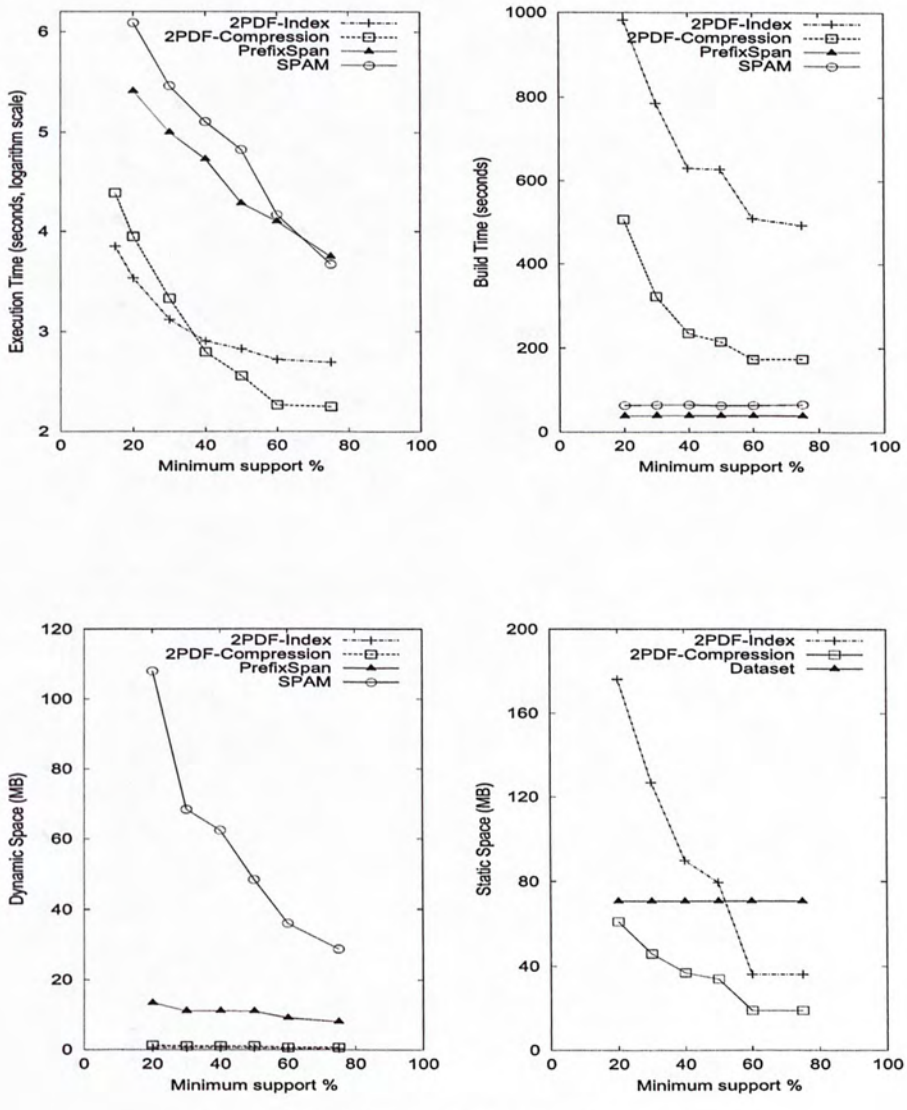


Figure 3.6: C256S64N4D100K, MinLen= 7

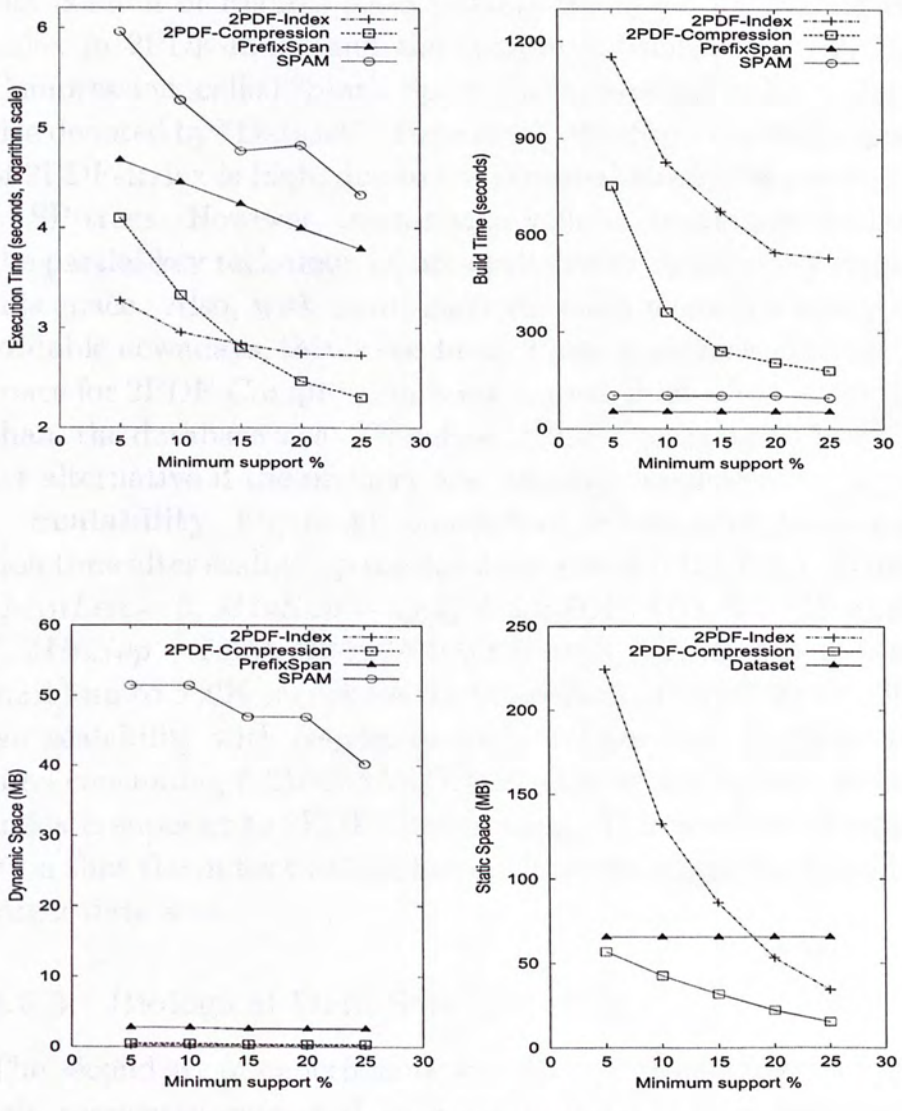


Figure 3.7: C256S64N20D100K, MinLen= 3

depth-first generation, called "Dynamic Space". Both 2PDFs have little dynamic space consumption because only one position is kept for each containing sequence of a pattern. The last column of Figures 3.4-3.8 shows the space for storing the index in 2PDF-Index and the compressed database in 2PDF-Compression, called "Static Space", compared to the input database size denoted by "Dataset". For a small $MinSup$, the static space of 2PDF-Index is high, due to the repeated store of sequence ids in SP-trees. However, main-memory index techniques such as the partial-key technique [9] are available to significantly reduce this space. Also, with multi-gigabyte main memories easily affordable nowadays, this is less likely a major concern. The static space for 2PDF-Compression is always less than, often much less than, the database size. Therefore, 2PDF-Compression is a better alternative if the memory size becomes a concern.

Scalability. Figure 3.9 shows, from left to right, the execution time after scaling up the database size of $C128S32N4D100K$ ($MinLen = 5$, $MinSup = 30\%$), $C256S64N4D100K$ ($MinLen = 7$, $MinSup = 25\%$), $C20S8N10000D100K$ ($MinLen = 1$, $MinSup = 0.2\%$) up to 500K sequences. Both versions of 2PDF show a linear scalability with respect to the database size. On the most time-consuming $C256S64N4D100K$ (the center figure), 2PDF-Index is superior to 2PDF-Compression. This confirms the intuition that the index method has a better scalability for handling larger data sets.

3.5.2 Biological Data Sets

The second set of experiments was conducted on DNA and protein sequences extracted from the website of National Center for Biotechnology Information (http://www.ncbi.nlm.nih.gov). The DNA data set was extracted by specifying the conjunction of the search category "Nucleotide", the range [200:300]

for sequence length, and the submission date from 2002/12 to 2003/02. The protein data set was extracted by specifying the conjunction of the search category "Protein", the range [150:250] for sequence length, and submission date from 2002/12 to 2003/02. The description of these data sets as of the extraction time is given in Table 3.10.

Figures 3.11-3.12 show the execution time, building time, dynamic space and static space. The comparison of 2PDFs with PrefixSpan and SPAM is similar to that for synthetic data sets in Section 6.1. This experiment confirms the superiority of the proposed methods on real life biosequence data.

□ **End of chapter.**

Symbol	Meaning in [5]	Adopted to biosequences
D	Number of customers	Number of sequences
C	Average number of transactions per customer	Average length of sequences
T	Average number of items per transaction	1
S	Average length of maximal potentially frequent sequences	no change
I	Average size of itemsets in maximal potentially frequent sequences	1
N_S	Number of maximal potentially frequent sequences	no change
N_I	Number of maximal potentially frequent itemsets	equal to N
N	Number of items	4 or 20

Table 3.3: Parameters of the data generator

Simulated category	Name	C	S	N	D	Size (MB)	MinLen
DNA	$C256S64N4D100K$	256	64	4	100K	70.6	7
sequences	$C128S32N4D100K$	128	32	4	100K	35.1	5
Protien	$C256S64N20D100K$	256	64	20	100K	65.3	3
sequences	$C128S32N20D100K$	128	32	20	100K	32.5	3
Customer sequences	$C20S8N10000D100K$	20	8	10,000	100K	4.7	1

Table 3.4: Synthetic data sets

MinSup	# base segment	# segment	# pattern
5%	223	1288	557722
10%	142	602	471015
15%	101	313	18502
20%	91	240	6131

Table 3.5: Statistics for C128S32N4D100K, MinLen=5

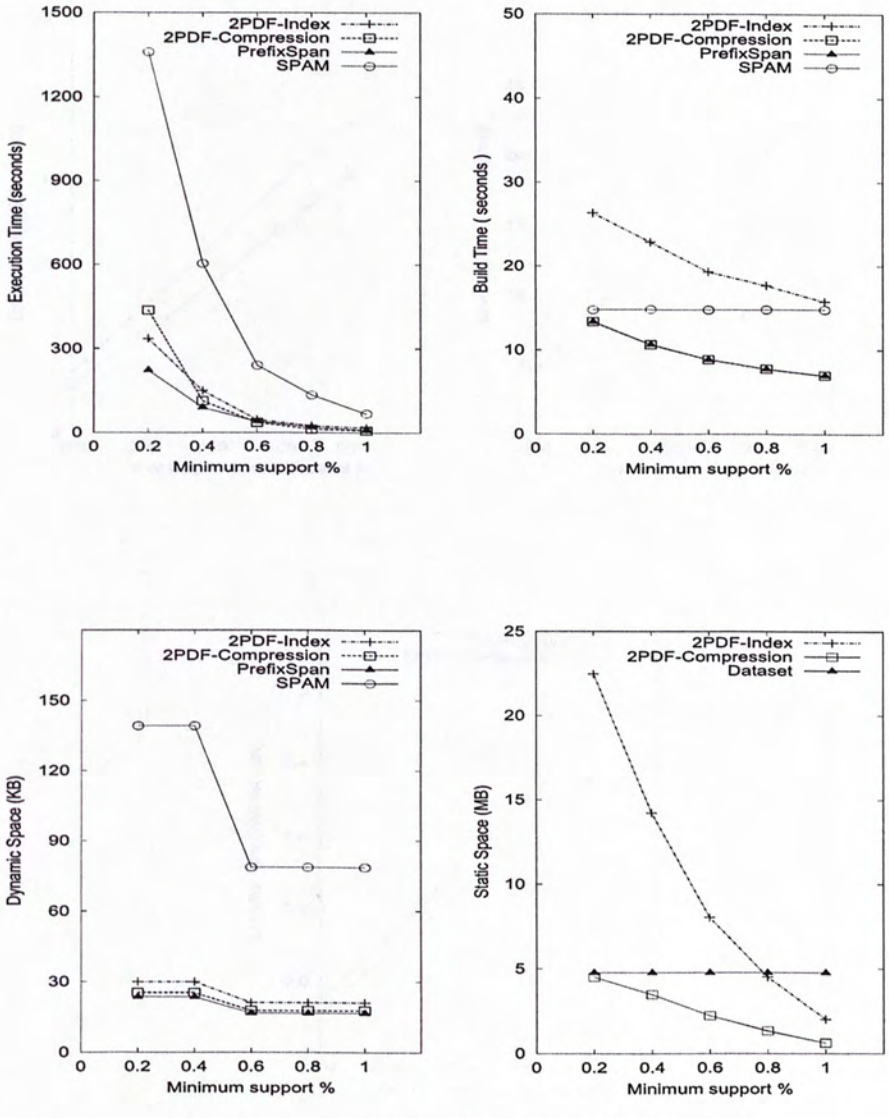


Figure 3.8: C20S8N10000D100K, MinLen = 1

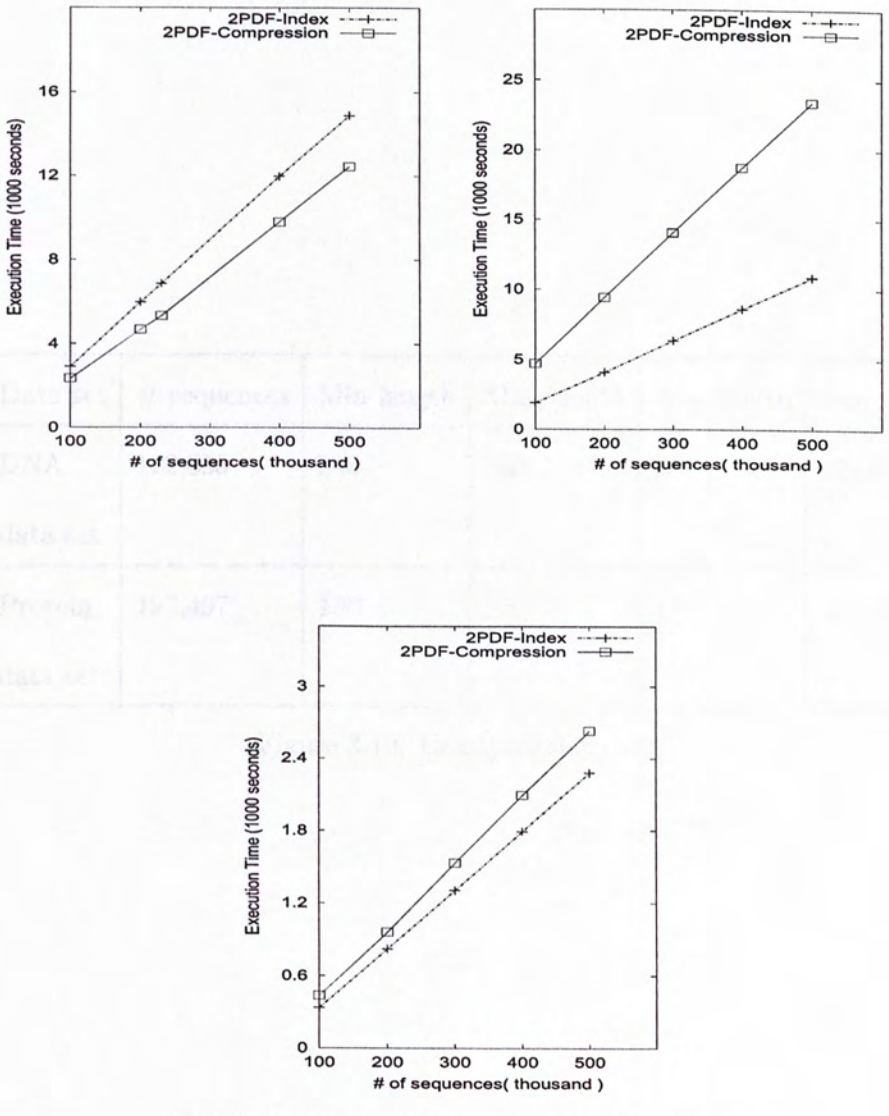


Figure 3.9: Scalability wrt the database size

Data set	# sequences	Min length	Max length	Avg length	Size (MB)
DNA data set	122,855	200	300	254	124.9
Protein data set	192,497	150	250	198	152.9

Figure 3.10: Biological data sets

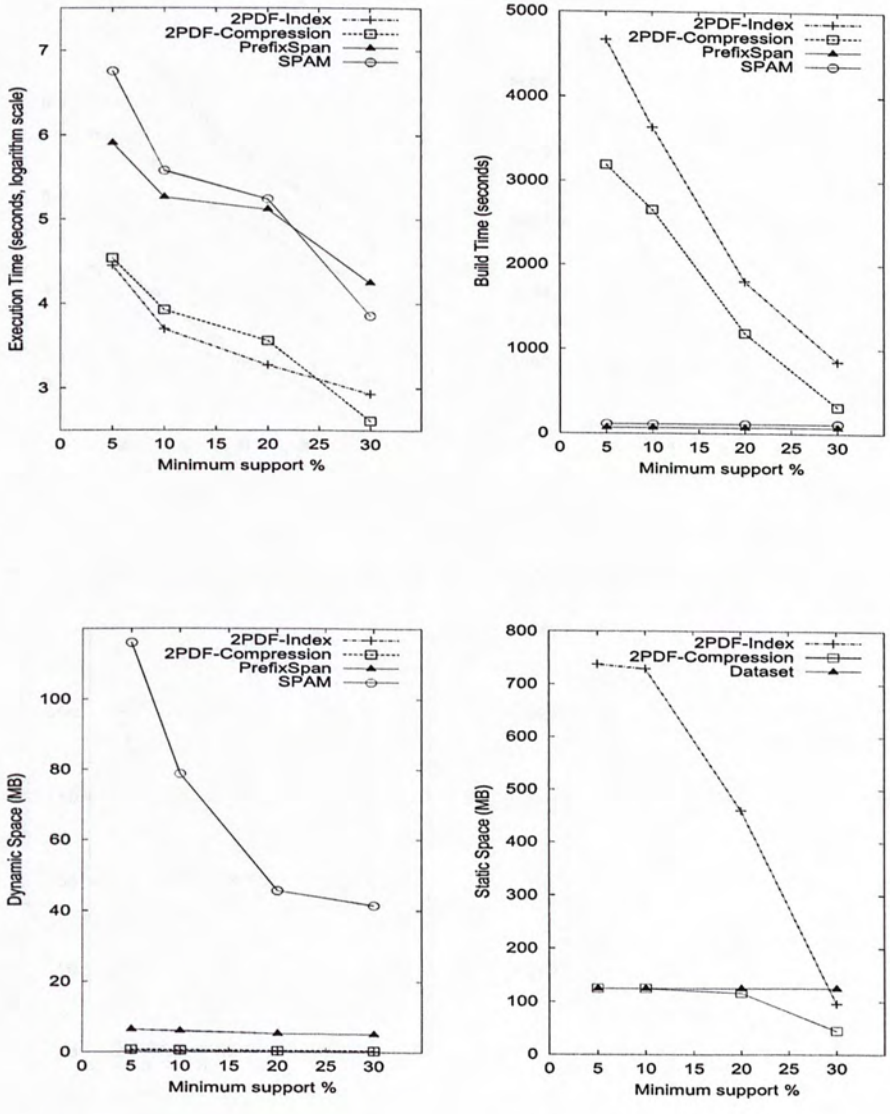


Figure 3.11: The real life DNA dataset, MinLen = 5

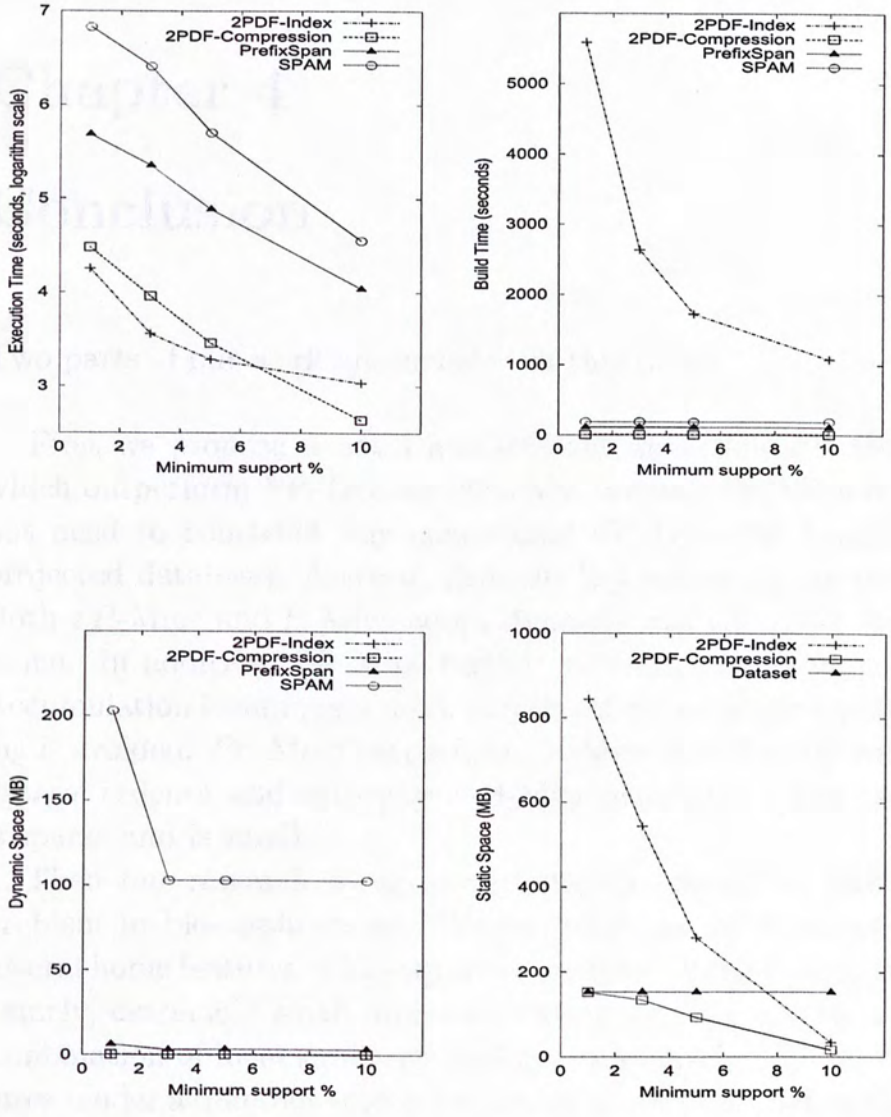


Figure 3.12: The real life protein dataset, MinLen = 3

Chapter 4

Conclusion

Two parts of our work are included in this thesis.

First we propose a novel and efficient algorithm *PP*-Mine, which outperform FP-Tree significantly, because *PP*-Mine does not need to construct any conditional FP-Trees for handling projected databases. Instead, dynamic link adjusting are used. Both *PP*-Mine and H-Mine adopt dynamic link adjusting technique. In addition, *PP*-Mine further minimizes counting cost. Accumulation technique is used, and therefore, unnecessary counting is avoided. *PP*-Mine outperforms H-Mine significantly when dataset is dense, and outperforms H-Mine marginally when dataset is sparse and is small.

Then our research focus move onto the sequential mining problem in bio-applications. We analyzed the implications of several home features of biosequences on data mining techniques, namely, extremely small alphabet, extremely long length, and combination of local similarity and global similarity. These features render a different blow up of search space from that in classic transaction sequences, and traditional indexing/partitioning/ bitmapping techniques are not effective for mining such sequences. To address this issue, we proposed to extract and exploit local similarity before searching for global similarity. This is done by indexing or compressing local similarity in the first phase. In

the second phase, we search for global similarities using local similarities as building blocks, exploiting relationships between them for pruning candidates and sharing support counting, and benefiting from indexed or compressed local similarities. The experiments on both synthetic and real life data sets demonstrated significant speed up over classic methods.

□ **End of chapter.**

Bibliography

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *SIGKDD*, 2000.

[2] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Proc. 6th ACM SIGKDD Int. Conf. on Knowledge discovery and data mining*, pages 108–118. ACM Press, 2001.

[3] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61:350–371, 2001.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[5] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 11th Int. Conf. Data Engineering, ICDE*, pages 3–14. IEEE Press, 6-10 1995.

[6] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using a bitmap representation. In *ACM SIGKDD*, pages 215–224, 2002.

[7] R. J. Bayardo. Efficiently mining long patterns from databases. In *ACM SIGMOD*, pages 85–93, 1998.

[8] R. J. Bayardo. Efficiently mining long patterns from databases. In *1998 ACM SIGMOD Intl. Conference on Management of Data*, pages 85–93. ACM Press, 05 1998.

[9] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *SIGMOD*, 2001.

[10] A. Brazma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to the automatic discovery of patterns in biosequences. In *Technical Report, Department of Informatics, University of Bergen, Norway*, 1995.

[11] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, 2001.

[12] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *2001 Intl. Conference on Data Engineering,ICDE*, pages 443–452, 04 2001.

[13] B. Dunkel and N. Soparkar. Data organization and access for efficient data mining. In *Proc. of 15th IEEE Intl. Conf. on Data Engineering*, pages 522–529, 03 1999.

[14] J. Han and J. Pei. Mining frequent patterns by pattern-growth: Methodology and implications. In *ACM SIGKDD Explorations*. ACM Press, 12 2001.

[15] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, 05 2000.

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

[17] J. Pei, J. Han, B. Asl, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE*, 2001.

[18] J. Pei, J. Han, H. Lu, S. Nishio, and D. Y. S hiwei Tang. H-mine:hyper-structure mining of frequent patterns in large databases. In *2001 IEEE Conference on Data Mining*. IEEE, 11 2001.

[19] P. Shenoy, J. R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 22–33. ACM Press, 05 2000.

[20] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. 5th Int. Conf. Extending Database Technology, EDBT*, volume 1057, pages 3–17. Springer-Verlag, 25-29 1996.

[21] H. Toivonen. Sampling large databases for association rules. In *VLDB*, pages 134–145, 1996.

[22] J. Wang, G.W.chirn, T. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: some preliminary results. In *SIGMOD*, 1994.

[23] K. Wang, L. Tang, J. Han, and J. Liu. Top down fp-growth for association rule mining. In *Proc. of 6th Pacific-Asia conference on Knowledge Discovery and Data Mining*, 2002.

[24] M. Waterman. *Mathematical methods for DNA sequence analysis*. CRC Press, Boca Raton, FL, 1989.

[25] J. Yang, W. Wang, P. Yu, and J. Han. Mining long sequential patterns in a noisy environment. In *ACM SIGMOD*, pages 406–417. ACM, 2002.

[26] M. J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000.

[27] M. J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning Journal, Special Issue on Unsupervised Learning*, 42(1/2):31–60, 2001.