Exploring Boolean Combinations of Succinct Constraints for Frequent Pattern Mining

by

Zhan Li

A thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada

April 2006

Copyright © 2006 by Zhan Li

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

Exploring Boolean Combinations of Succinct Constraints for Frequent Pattern Mining

BY

Zhan Li

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

OF

MASTER OF SCIENCE

Zhan Li © 2006

Permission has been granted to the Library of the University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilms Inc. to publish an abstract of this thesis/practicum.

This reproduction or copy of this thesis has been made available by authority of the copyright owner solely for the purpose of private study and research, and may only be reproduced and copied as permitted by copyright laws or with express written authorization from the copyright owner.

Thesis advisor

Dr. Carson K. Leung

Author

Zhan Li

Exploring Boolean Combinations of Succinct Constraints for Frequent Pattern Mining

Abstract

In recent years, constraint-based frequent pattern mining has become more important, because constraints allow users to focus their search for frequent patterns and thereby reduce computation. One important type of constraints, called succinct constraints, has nice properties and can be used to optimize the mining process. Existing algorithms have exploited only conjunctions, but not negations or disjunctions, of succinct constraints in the Frequent Pattern tree (FP-tree) framework. However, to support any Boolean combinations of succinct constraints, one needs to be able to effectively handle the negations and disjunctions.

In this thesis, we (a) explore properties of negations and disjunctions of succinct constraints and (b) design and implement an efficient algorithm called FPS* for handling Boolean combinations of succinct constraints in the FP-tree framework. Specifically, we prove that the negations and disjunctions of succinct constraints are also succinct. We extend the existing FPS algorithm to become FPS*, which inherits the functions for handling conjunctions of succinct constraints from FPS and adds new functions for handling negations and disjunctions of succinct constraints. Experimental results show the effectiveness of our FPS* algorithm.

Acknowledgements

First, I would like to acknowledge the inspiration, helpful advice and suggestions I received from my research supervisor, Dr. Carson K. Leung. This thesis would not have been completed without his help and guidance in the whole research process. He is an active researcher in the data mining area with nice personality. It is a great experience to work with him and I benefit in many ways.

In addition, I would like to express my thanks to my thesis examination committee members, Dr. David H. Scuse and Dr. G. Gary Wang, and the chair of my thesis defence, Dr. Peter R. King, who review my thesis and provide helpful comments and suggestions.

I would also like to thank members of the Graduate Studies Committee in Department of Computer Science for their useful comments and suggestions on my thesis proposal.

Special gratitude is reserved for my parents who have encouraged me in so many ways to pursue my life dreams, and my husband, Wei Li, for his understanding and encouragement during this study.

ZHAN LI
B.Eng., Harbin Engineering University, China, 2002

The University of Manitoba
April 2006

This thesis is dedicated to my parents.

Table of Contents

Abstract ii
Acknowledgements iii
Dedication iv
Table of Contents v
List of Tables vii
List of Figures viii

1 Introduction **1**
1.1 Problem Statement 5
1.2 Thesis Organization 6

2 Related Work **8**
2.1 Background: Constraints 8
2.2 Relevant Algorithms 16
2.3 Summary 24

3 Boolean Combinations of Succinct Constraints **25**
3.1 Negations of Atomic Succinct Constraints 26
3.2 Disjunctions of Atomic Succinct Constraints 29
3.3 Boolean Combinations of Succinct Constraints 35
3.4 Summary 38

4 The Design and Implementation of the FPS* Algorithm **40**
4.1 Negations of Atomic Succinct Constraints 41
4.2 Disjunctions of Atomic Succinct Constraints 42
4.3 Summary 48

5 Experimental Results **49**
5.1 Experimental Results for Negation of Atomic Succinct Constraints . . 50
5.1.1 $FPS^*_{Negation}$ for a SAM Constraint 50
5.1.2 $FPS^*_{Negation}$ for a SUC Constraint 54
5.2 Experimental Results for Disjunctions of SAM Constraints 59

5.3 Experimental Results for Disjunctions of SUC Constraints 67
5.4 Experimental Results for Disjunctions of Mixed Constraints 72
5.5 Summary 76

6 Conclusions and Future Work **78**
6.1 Conclusions 78
6.2 Future Work 81

Bibliography **82**

List of Tables

2.1 Characterization of constraints: anti-monotonicity and succinctness . 13
2.2 A transaction database TDB 17
2.3 Auxiliary information for the transaction database TDB 20

5.1 The number of frequent patterns for FP-growth++ and $FPS^*_{Negation}$. 57
5.2 The number of nodes in FP-trees and the number of support counting 57
5.3 The number of frequent patterns for FP-growth++ and FPS^*_{nSAM} . . 61
5.4 The number of frequent patterns for the mushroom dataset 64
5.5 The number of frequent patterns with different number of constraints 66

List of Figures

2.1 The construction of an FP-tree 17
2.2 The $\{e\}$-conditional FP-tree 18
2.3 The FP-tree for the FPS algorithm 21
2.4 The FP-tree for FPS handling conjunctions of SUC constraints 22

4.1 The global FP-tree 44
4.2 The constraint FP-tree for C_1 44
4.3 The constraint FP-tree for C_2 45
4.4 The conditional FP-tree for the $\{a\}$-projected database 45
4.5 The constraint FP-tree for C_3 46
4.6 FP-trees for FPS^{*}_{nMIX} 47

5.1 Results for a SAM constraint with different minsup 51
5.2 Results for a SAM constraint with different selectivity 52
5.3 Scalability for $FPS^{*}_{Negation}$ for a SAM constraint 53
5.4 Results for a SUC constraint with different minsup 54
5.5 Results for a SUC constraint with different selectivity 56
5.6 Scalability for $FPS^{*}_{Negation}$ for a SUC constraint 58
5.7 Results for disjunctions of SAM constraints with different minsup . . 60
5.8 Results for disjunctions of SAM constraints with different selectivity . 62
5.9 Results of FPS^{*}_{nSAM} on the mushroom data with different selectivity 64
5.10 Results for changing the number of SAM constraints 65
5.11 Scalability of FPS^{*}_{nSAM} 67
5.12 Results for disjunctions of SUC constraints with different minsup . . 68
5.13 Results for disjunctions of SUC constraints with different selectivity . 69
5.14 Results for changing the number of SUC constraints 71
5.15 Scalability of FPS^{*}_{nSUC} 71
5.16 Results for disjunctions of mixed constraints with different minsup . . 72
5.17 Results for disjunctions of mixed constraints with different selectivity 74
5.18 Results for changing the number of mixed constraints 75
5.19 Scalability of FPS^{*}_{nMIX} 76

Chapter 1

Introduction

Data mining refers to the search for implicit, previously unknown, and potentially useful information that might be embedded in data. Association rule mining is an important task in data mining with the goal is to find associations among items in the database of transactions. Since its introduction [AIS93], association rule mining has been the subject of numerous studies. A classical example of association rule mining originates from the analysis of market-basket data, where rules like "a customer who buys a set X of items (i.e., a set of merchandise or products) will also buy a set Y of items with $c\%$ probability" are found [HGN00]. Association rule mining is popular because of its direct applicability to many business problems and the inherent understandability of the rules. Moreover, association rules are successfully applicable to a wide range of business problems such as catalogue design, sales campaign analysis, Web log (click stream) analysis, and DNA sequence analysis.

In brief, an *association rule* is an expression $X \Rightarrow Y$, where X and Y are sets of items (or *itemsets*, for short). Given a transaction database TDB (where each

transaction t_i in TDB represents a set of items brought by a customer), the rule $X \Rightarrow Y$ expresses that whenever a transaction t_i contains X then t_i probably also contains Y. The rule confidence is defined as the percentage of transactions containing X and Y among all those transactions containing X [HGN00]. More formally, the probability that both X and Y occur together in the same transaction is called the *support*; the probability that Y occurs among all transactions containing X is called the *confidence*. Then, the association rule mining problem is to find all association rules with support and confidence values greater than or equal to the user-specified minimum support and minimum confidence thresholds. These rules can be found in two steps. Given a collection of transactions, Step 1 finds all subsets of transactions that occur frequently in the database (i.e., finding frequent itemsets, also known as frequent patterns, which satisfy the minimum support threshold). For a transaction with m items, there are $O(2^m)$ subsets. Step 2 uses these frequent patterns to derive interesting association rules. This step is straightforward. Basically, it checks the confidence of all rules $X \Rightarrow Y$ (where X and Y are frequent patterns found in Step 1), and removes those rules with confidence value less than the minimum confidence threshold [HGN00]. For p frequent patterns, there are only $O(p^2)$ possible rules. Therefore, finding frequent patterns dominates the computational complexity of the entire process of association rule mining .

Over the past decade, many algorithms have been proposed to study the problem of frequent pattern mining. These studies can be broadly divided into two generations [NL+98]. In the first generation, most studies focused on performance and efficiency issues. These studies mainly stemmed from two frameworks: the *Apriori framework*

[AS94, AM+96] and the *Frequent Pattern tree (FP-tree) based framework* [HPY00, LP+02, GZ03b, PZ03]. Algorithms in the Apriori framework use a generate-and-test approach. They first generate some candidate patterns, and then test if these patterns are frequent by counting their occurrences in the database. To reduce the number of candidates needed to be counted, the algorithms use the Apriori property [AS94], which states that "all supersets of an infrequent pattern must be infrequent". To further speed up the mining process, the FP-tree based framework was proposed. Algorithms in this framework do not generate candidates. Instead, the algorithms utilize a tree structure, called the FP-tree, to capture the database content, from which frequent patterns are computed. As the FP-tree based algorithms are more efficient than the Apriori-based algorithms, we developed an FP-tree based algorithm in the thesis work.

When compared with studies in the second generation, the studies in the first generation basically considered the data mining exercise in isolation (i.e., no interaction with the user). Consequently, they often generated a very large number of frequent patterns, in which only a small fraction were useful or interesting to the user. Hence, they wasted computation efforts. In contrast, the studies in the second generation realized the importance of a paradigm for *constraint-based mining*, where the user is allowed to express his focus, by means of a rich class of constraints that capture the application semantics. Besides allowing user exploration and control, this paradigm also allows many constraints to be "pushed inside" the mining process so that the search for patterns is confined to only those of interest to the user. Hence, performance can be improved [PHL01].

There are several classes of "semantic" constraints. One of them is called *succinct constraints*. This class of constraints has some nice properties that help optimize mining. These properties include the following. First, if a constraint is succinct, one can directly generate precisely all and only those itemsets satisfying the constraints by using a precise "formula", called a *member generating function (MGF)* [NL+98], which does not require generating and excluding itemsets not satisfying the constraint. For example, itemsets satisfying the succinct constraint "$min(S.Price) > \$10$" (which says that the minimum price in the set S of items is greater than \$10) can be generated solely from items whose individual price is greater than \$10. Second, a majority of constraints are succinct. Third, for constraints that are not succinct, many of them can be induced into weaker constraints that are succinct.

The motivation for this thesis work is as follows. Constraint-based mining allows users to express their focuses, via the use of constraints, so that the search for patterns is confined to only those of interest to users. Regarding the expressiveness of the constraints, users are not confined to express their focuses by using a single (i.e., atomic) constraint. They can freely express their focuses using any Boolean combinations of constraints. So far, there are studies for handling atomic succinct constraints and conjunctions of succinct constraints in both the Apriori-based and the FP-tree based frameworks [NL+98, LLN02, LLN03, Leu04]. For example, an FP-tree based algorithm, called FPS [LLN02], effectively handles succinct constraints and their conjunctions. However, it is not uncommon to have situations where users would like to impose some Boolean combinations of succinct constraints (i.e., involving conjunctions, negations, and disjunctions). For instance, a store manager may want to find

some products that are either expensive or do not have high quantity (e.g., a constraint "$(S.Price \geq \$50) \cup (S.Quantity \leq 1000)$") for promotional purposes. To our knowledge, there does not exist any work on handling negations or disjunctions of succinct constraints. Consequently, the store manager cannot find the products he is interested in for the sales promotion. There are many other similar real-life situations where the user would like to express their interests via some Boolean combinations of constraints. Hence, having a mining system that handles negations and disjunctions of succinct constraints would be useful. Such a system would provide users the ability to freely express their focuses, not only by atomic succinct constraints, but also by any Boolean combinations of succinct constraints.

1.1 Problem Statement

In this thesis, we investigated the following issues: (a) Can we deal with the negation of succinct constraints? (b) Can we deal with the disjunction of succinct constraints? (c) How do we handle negations and disjunctions of succinct constraints efficiently? Specifically, the thesis statement is as follows:

> We explore the properties of negations, disjunctions of succinct constraints. Based on the analysis of properties, we design and implement an efficient algorithm FPS* for handling negations of atomic succinct constraints and disjunctions of subclasses of succinct constraints in the FP-tree framework.

The result of this thesis work is important. Through our exploration, we prove that any Boolean combinations of succinct constraints are succinct. The FPS* algorithm can effectively handle the negations of atomic succinct constraints and disjunctions of subclasses of succinct constraints. The succinct constraints are "pushed inside" the mining process, and thus leading to more efficient mining.

1.2 Thesis Organization

This thesis is organized as follows.

Chapter 2 presents background of constraints and some relevant algorithms. We provide a thorough analysis on succinct constraints and conjunctions of succinct constraints. Among relevant algorithms, we describe how the FPS algorithm efficiently handles succinct constraints and conjunctions of succinct constraints by utilizing the succinctness of succinct constraints and conjunctions of succinct constraints.

After reviewing the previous work in Chapter 2, we start describing our current work (i.e., the contribution of this thesis) in Chapter 3. Specifically, Chapter 3 focuses on the analytical aspect of how to handle negations and disjunctions of succinct constraints. We provide a complete proof of succinctness of negations and disjunctions of succinct constraints.

Chapter 4 focuses on the implementation aspect. We develop the FPS* algorithm that efficiently handles negations of atomic succinct constraints and disjunctions of SUC, SAM and SUC mixed with SAM constraints. All details of FPS* algorithm are provided in this chapter.

Experimental results of the FPS* algorithm are provided in Chapter 5. We test

the algorithm on synthetic databases and real-life databases, and compare results of FPS* with its relevant algorithm to evaluate the efficiency of the FPS* algorithm. We study the experimental results, and find out the most adaptable range of FPS*.

Finally, we give conclusions and future work in Chapter 6.

Chapter 2

Related Work

In this chapter, we provide a thorough review of various classes of constraints. Furthermore, we state the definition and properties of succinct constraints, which are relevant to the rest of this thesis. In addition, we provide a survey of some existing relevant mining algorithms (e.g., FP-tree based algorithms as well as constrained algorithms).

2.1 Background: Constraints

There are several classes of "semantic" constraints. In this section, we provide definitions of various classes of constraints.

Definition 2.1 (Domain and Aggregate Constraints [NL+98]) A constraint can be a domain constraint or an aggregate constraint, as described below.

(1) A **domain constraint** is of one of the following forms:

- $S.A\theta v$, where S is a set variable, A is an attribute, v is a constant from the domain that $S.A$ comes from, and θ is one of the Boolean operators $=, \neq, <, \leq, >, \geq$. It says every element of S stands in relationship θ with the constant value v.
- $v\theta S.A$, where S, A, v are as above, and θ is one of the Boolean operators $\in, \notin$. This simply says the element v belongs (or not) to the set S.
- $V\theta S.A$ or $S.A\theta V$, where S is a set variable, A is an attribute, V is a set of constants ranges over the domain of S, and θ is one of $\subseteq, \not\subseteq, \subset, \not\subset, =, \neq$.

(2) An **aggregate constraint** is of the form $agg(S.A)\theta v$, where agg is one of the aggregate functions min, max, sum, avg, and θ is one of the Boolean operators $=, \neq, <, \leq, >, \geq$. It says the aggregate of the set of numeric values in S stands in relationship θ to v. □

In the examples below, assume we have the transaction database $TDB(TID, Itemset)$ with auxiliary information contained in $itemInfo(Item, Type, Price)$. The constraint "$S.Price$ < \$100" says that all items in the set S are of price less than \$100, and the constraint "\$120 $\in$ $S.Price$" says that there is an item of price \$120 in S. As another example, "$S.Type \supseteq \{snacks, sodas\}$" says that S should include some items whose type is snacks and some items whose type is sodas. These are all domain constraints. The constraint "$max(S.Price)$ > \$100", which says that the maximum price of items in S should be greater than \$100, is an aggregate one. While constraints can be categorized into domain and aggregate constraints based on their forms, they can also be classified into overlapping classes depending on

their properties. These classes include (1) anti-monotone constraints, (2) monotone constraints, (3) convertible constraints, and (4) succinct constraints.

Definition 2.2 (Anti-monotone Constraints [NL+98]) A constraint C_a is **anti-monotone** iff whenever an itemset S violates C_a, so does any superset of S. □

For example, $C_a \equiv S.Price < \$100$ is an anti-monotone constraint, because any superset of S violating C_a (e.g., containing any item with $Price = \$100$) also violates C_a.

Definition 2.3 (Monotone Constraints [PH02]) A constraint C_m is **monotone** iff whenever an itemset S satisfies C_m, so does any superset of S. □

For example, $C_m \equiv max(S.Price) > \$100$ is a monotone constraint, because any superset of S satisfying C_m also satisfies C_m (as adding more items to S does not lower the maximum).

Definition 2.4 (Convertible Constraints [PHL01]) A constraint C is *convertible anti-monotone* provided there is an order R on items such that whenever an itemset S satisfies C, so does any prefix of S (w.r.t. the ordering). A constraint C is *convertible monotone* provided there is an order R on items such that whenever an itemset S violates C, so does any prefix of S. A constraint is **convertible** whenever it is convertible anti-monotone or monotone. □

For example, $C_{conv} \equiv max(S.Price) > \$100$ is a convertible (anti-monotone) constraint. When items in S are arranged in decreasing price order, any prefix of S satisfies C_{conv} if S satisfies C_{conv}.

Definition 2.5 (Succinct Constraints [NL+98]) Succinctness is defined in several steps, as follows: Define SAT_C (Item) to be the set of itemsets that satisfy the constraint C. With respect to the lattice space consisting of all itemsets, SAT_C (Item) represents the pruned space (i.e., the solution space) consisting of those itemsets satisfying C.

(1) An itemset $I \subseteq Item$ is a succinct set if it can be expressed as $\sigma_p(Item)$ for some selection predicates p, where σ is the selection operator (as in relational algebra).

(2) $SP \subseteq 2^{Item}$ is a succinct powerset if there is a fixed number of succinct sets $Item_1, ..., Item_k \subseteq Item$, such that SP can be expressed in terms of the powersets of $Item_1, ..., Item_k$ using union and minus.

(3) A constraint C is **succinct** provided $SAT_C(Item)$ is a succinct powerset. □

Property 2.1 For every succinct constraint C, there is a member generating function MGF_C that can generate precisely all those itemsets satisfying C [LLN03]. □

Hence, a succinct constraint C can simply operate in a generate-only environment (by using MGF_C), rather than in a generate-and-test environment. In other words, one does not need to generate lots of itemsets, test them, and then exclude those violating C. Instead, one can easily enumerate (by using MGF_C) all and only those itemsets that satisfy the succinct constraint C.

Definition 2.6 (Member Generating Function [NL+98]) There are two types of member generating function:

(1) A succinct powerset $SP \subseteq 2^{Item}$ is said to have a **basic member generating function (basic MGF)** provided there is a function that can enumerate all and only those elements of SP, and that is of the form $MGF \equiv \{X_1 \cup ... \cup X_{k+1} \mid X_i \subseteq \sigma_{p_i}(Item)\ \&\ X_i \neq \emptyset$, for $1 \leq i \leq k, X_{k+1} \subseteq \sigma_{p_{k+1}}(Item)\}$ for some selection predicates $p_1, ..., p_{k+1}$. In this definition, the X_i's (for $1 \leq i \leq k$) that are required to be nonempty are called mandatory item-subsets; whereas X_{k+1} is called the optional item-subset.

(2) A succinct powerset $SP \subseteq 2^{Item}$ is said to have a **general member generating function (general MGF)** provided there is a function that can enumerate all and only those elements of SP, and that is of the form $\bigcup_{j=1}^{N} MGF_j$, for some basic member generating functions $MGF_1, ..., MGF_N$. □

For example, for the constraint "$S.Price < 100$", its MGF is $\{X_1 \mid X_1 \subseteq \sigma_{price<100}(Item) \& X_1 \neq \emptyset\}$. In this case, there is only one mandatory item-subset X_1, but there is no optional item-subset. This corresponds to the pruned space being 2^{Item1}, where $Item1 = \sigma_{price<100}(Item)$. For "$max(S.Price) > 100$", its MGF is $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{price>100}(Item) \& X_1 \neq \emptyset \& X_2 \subseteq \sigma_{price \leq 100}(Item)\}$. There is at least one item with $Price >$ \$100, and it is acceptable to include any additional item with Price = \$100. So, it has the optional item-subset X_2.

Succinct constraints can be of various forms. Table 2.1 shows a characterization of constraints. The second column of the table identifies whether or not the constraints in the first column are anti-monotone, while the third column identifies whether or not the constraints are succinct. From the table, we can easily observe the following:

- A majority of constraints are succinct. This explains why we focus on the

Table 2.1: Characterization of constraints: anti-monotonicity and succinctness [LLN03]

Constraint	Anti-monotone	Succinct
$S.A\ \theta\ cn$, where $\theta \in \{=, \leq, \geq\}$	*yes*	*yes*
$cn \in S.A$	*no*	*yes*
$S.A \supseteq CS$	*no*	*yes*
$S.A \subseteq CS$	*yes*	*yes*
$min(S.A) \leq cn$	*no*	*yes*
$min(S.A) \geq cn$	*yes*	*yes*
$max(S.A) \leq cn$	*yes*	*yes*
$max(S.A) \geq cn$	*no*	*yes*
$sum(S.A) \leq cn$	*yes*	*no*
$sum(S.A) \geq cn$	*no*	*no*
$avg(S.A)\ \theta\ cn$, where $\theta \in \{=, \leq, \geq\}$	*no*	*no*

Note: S is a set variable, A is an attribute of a set S, cn is a constant for A, and CS is a set of constants for A.

succinct constraints in this thesis.

- Some succinct constraints happen to be anti-monotone as well. We denote these succinct anti-monotone constraints as **SAM constraints**. Similarly, we denote those succinct non-anti-monotone constraints as **SUC constraints**. One of the SUC constraints ($S.A \supseteq CS$) is called **superset constraint** because of the special form of its MGF (as explained below).

For SAM constraints, the corresponding MGF is of the form $\{X_1 \mid X_1 \subseteq \sigma_p(Item), X_1 \neq \emptyset\}$, for some selection predicate p. It only contains the mandatory group.

For SUC constraints, the corresponding MGF is of the form $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_p(Item), X_1 \neq \emptyset, X_2 \subseteq \sigma_{\neg p}(Item)\}$, for some selection predicate p. It contains one mandatory group and one optional group.

For superset constraints, the corresponding MGF is of the form $\{X_1 \cup \ldots X_k \cup X_{k+1} \mid X_i \subseteq \sigma_{p_i}(Item) \& X_i \neq \emptyset, \text{for } 1 \leq i \leq k; X_{k+1} \subseteq \sigma_{p_{k+1}}(Item)\}$, where p_{k+1} is of the form $(\neg p_1 \wedge \ldots \wedge \neg p_k)$. It contains k mandatory groups and 1 optional group [LLN03].

Property 2.2 In addition to having Property 2.1, succinct constraints also have the following properties [LLN03]:

(a) A conjunction of succinct constraints CSC is succinct.

(b) For a conjunction of succinct constraints CSC, there is a member generating function MGF_{CSC} that can generate precisely all those itemsets satisfying all the constraints in CSC. □

The complete proof can be found in the paper of Lakshmanan et al. [LLN03]. Here, we only provide a brief explanation. For simplicity, let us consider conjunctions of two succinct constraints. A simpler case is the conjunctions of one SAM constraint and one succinct constraint (e.g., SAM, SUC, or superset constraint). Recall from Definition 2.6 that itemsets generated by the $MGFs$ for SAM constraints contain only mandatory items (which are generated using the mandatory selection predicates), and contain no optional items. Hence, in this case, the MGF contains mandatory items that can be enumerated using the conjunction of the mandatory selection predicates from the both $MGFs$ for the SAM and the SUC or superset constraints, and optional items that can be enumerated using the conjunction of the mandatory selection predicate from the MGF for the SAM constraint and the optional selection predicate from the MGF for the SUC or superset constraint.

For example, let $C_{SAM} \equiv min(Price) \geq 50$ and $C_{SUC} \equiv max(Quantity) \leq 100$. The MGF for $C_{SAM} \wedge C_{SUC}$ is $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{Price \geq 50 \wedge Quantity \leq 100}(Item), X_1 \neq \emptyset, X_2 \subseteq \sigma_{Price \geq 50 \wedge Quantity > 100}(Item)\}$.

Recall that itemsets generated by the $MGFs$ for SUC and superset constraints contain mandatory items (which are generated using the mandatory selection predicates), and may contain optional items (which are generated using the optional selection predicates). For the MGF of two succinct non-SAM constraints (e.g., SUC or superset constraints), the mandatory items can be enumerated using either of the following:

(a) the conjunction of the mandatory selection predicates from both $MGFs$; or

(b) the conjunction of the mandatory selection predicates from MGF_1 and the optional selection predicate from MGF_2, together with the conjunction of the optional selection predicates from MGF_1 and the mandatory selection predicates from MGF_2.

The resulting MGF is a union of the $MGFs$ using part(a) and the $MGFs$ using part(b).

For example, let $C_{SUC_1} \equiv min(Price) \leq 50$, and $C_{SUC_2} \equiv max(Quantity) \geq 100$. The MGF for $C_{SUC_1} \wedge C_{SUC_2}$ is a general MGF consisting of two basic $MGFs$: $MGF_1 \cup MGF_2$, where

$$\begin{aligned} MGF_1 &\equiv \{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{Price \leq 50 \wedge Quantity \geq 100}(Item), X_1 \neq \emptyset, \\ &\quad X_2 \subseteq \sigma_{Price > 50 \vee Quantity < 100}(Item)\} \\ MGF_2 &\equiv \{X_3 \cup X_4 \cup X_5 \mid X_3 \subseteq \sigma_{Price \leq 50 \wedge Quantity < 100}(Item), X_3 \neq \emptyset, \end{aligned}$$

$$X_4 \subseteq \sigma_{Price>50 \wedge Qugntity \geq 100}(Item), X_4 \neq \emptyset,$$

$$X_5 \subseteq \sigma_{Price>50 \wedge Quantity<100}(Item)\}$$

2.2 Relevant Algorithms

There are many algorithms that have stemmed from the FP-growth algorithm [HPY00], which is one of the most famous FP-tree based algorithms for frequent pattern mining. The FP-growth algorithm can be described as follows. It uses an extended prefix-tree structure, called a *Frequent Pattern tree (FP-tree)*, to capture the content of the database. Only frequent items are kept in the tree, and the tree nodes are arranged according to some criteria (e.g., descending frequency order). Frequent patterns are formed by first finding the frequent 1-itemsets (i.e., sets consisting of one item) from the FP-tree, and then recursively growing them. The pattern growth is achieved via the concatenation of each frequent pattern with new ones generated from a conditional FP-tree (i.e., a sub-tree). To elaborate, the FP-growth algorithm starts from a frequent 1-itemset $\{X\}$, examines only its projected database (a "sub-database" consisting of the set of frequent items co-occurring with the frequent 1-itemset $\{X\}$), constructs its conditional FP-tree, and performs mining recursively with such a tree (i.e., to find the frequent supersets of $\{X\}$). See the example below.

Example 2.1 (Construction of the FP-tree) *Consider the transaction database TDB in Table 2.2 with a minimum support threshold of 2.*

We construct an FP-tree as follows. First, a scan of the TDB derives a list of frequent items $\langle(b:4),(a:3),(c:3),(d:3),(e:3)\rangle$ *(where the number after ":"*

Table 2.2: A transaction database TDB

Transactions in TDB	Contents
t_1	a, b, c, d
t_2	b, d, f
t_3	a, b, d, e
t_4	a, b, c, e
t_5	c, e

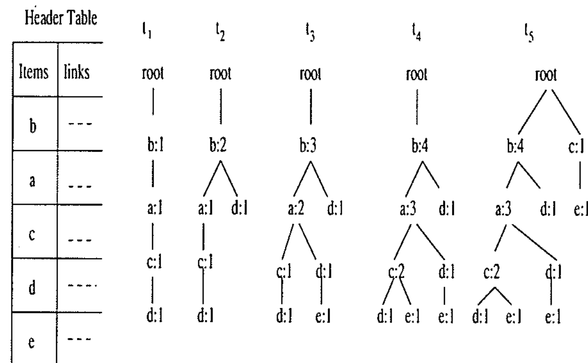

Figure 2.1: The construction of an FP-tree

indicates the support count) in which items are ordered by descending frequency. Note that item f (with a frequency of 1) is infrequent; so, it does not appear in this list of frequent items or the resulting FP-tree. The FP-tree is constructed by scanning the TDB the second time. The scan of the first transaction leads to the construction of the first branch of the tree: $\langle (b:1),(a:1),(c:1),(d:1)\rangle$. *For the second transaction, its (ordered) frequent item list* $\langle b,d\rangle$ *shares a common prefix* $\langle b\rangle$ *with the existing path* $\langle b,a,c,d\rangle$, *so the count of node b is incremented by 1, and a new node (d:1) is created and linked as a child of (b:2). A similar construction scheme applies to the remaining three transactions. After scanning all the transactions, the resulting FP-tree is returned as shown in (the rightmost FP-tree in) Figure 2.1.*

Example 2.2 (Mining Process Based on the FP-growth Algorithm) *Let us examine the mining process based on the (final) FP-tree shown in Figure 2.1.*

For node e, its immediate frequent pattern is $\{e\}$, and it has three paths in the FP-tree: $\langle b, a, c, e \rangle$, $\langle b, a, d, e \rangle$ and $\langle c, e \rangle$. The first path indicates that string "(b, a, c, e)" appears once in the database, or e's prefix path is (bac:1). The second path indicates "(b, a, d, e)" appears once in the database, or e's prefix path is (bad:1). The third path indicates "(c, e)" appears once in the database, or e's prefix path is (c:1). These three prefix paths of e, "$\{$(bac:1), (bad:1), (c:1)$\}$", form e's subpattern-base, which is called the $\{e\}$-projected database (i.e., the projected database under the condition of e's existence). The FP-tree for this projected database (which is called the $\{e\}$-conditional FP-tree) has two paths $\langle b : 2, a : 2, c : 1 \rangle$ and $\langle c : 1 \rangle$. This conditional FP-tree (as shown in Figure 2.2) is then mined recursively, and frequent patterns $\{e, c\}$, $\{e, c, a\}$, $\{e, c, a, b\}$ and $\{e, c, b\}$ are found. Note that item d is infrequent in the $\{e\}$-projected database; so, it does not appear in the $\{e\}$-conditional FP-tree. Similarly, the FP-growth algorithm sets up the $\{d\}$-conditional FP-tree to find frequent itemsets containing d but not e. Then, the algorithm sets up the $\{c\}$- and $\{a\}$-conditional

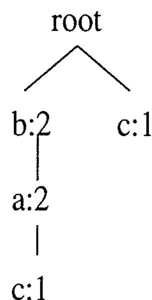

Figure 2.2: The $\{e\}$-conditional FP-tree

FP-trees to find other frequent itemsets in a similar divide-and-conquer manner. For more details, please refer to the work of Han et al. [HPY00].

The above examples showed how the FP-growth algorithm uses FP-trees to efficiently find frequent patterns. FP-trees are suitable for dense datasets where many branches share common prefixes. However, FP-trees have some limitations. For instance, when the dataset is sparse, FP-trees can be big and bushy because of their lack of shared branches. To solve this problem, the PatriciaMine algorithm [PZ03] uses the Patricia trie for efficient handling of sparse datasets. The FP-growth* algorithm [GZ03b] uses an array structure, in addition to FP-trees, to omit the first traversal of the FP-tree and conditional FP-trees to improve the efficiency. The use of the array structure works very well especially when the database is sparse, because the time saved by omitting the first traversal is far greater than the time needed for accumulating counts in the array. However, when the dataset is dense, the traversal of a compact FP-tree is fairly rapid; so, accumulating counts in the array may take more time. As our research more commonly deals with dense datasets, we use FP-trees.

So far, we have discussed only unconstrained tree-based algorithms. In the remaining part of this section, we discuss some relevant constrained algorithms. The FIC algorithm [PHL01] is an algorithm for handling convertible constraints, which cover aggregate succinct constraints as well as some constraints that are neither anti-monotone nor succinct. However, the algorithm does not handle non-aggregate succinct constraints. Moreover, it does not handle aggregate succinct constraints efficiently because it does not explore these constraints. The algorithm requires lots of constraint testing, which is time consuming.

The FPS algorithm [LLN02] improves the FIC algorithm by exploiting succinctness properties. It effectively handles succinct constraints and the conjunction of succinct constraints in the FP-tree framework. *For a SAM constraint, the FPS algorithm only keeps the frequent items that satisfy the SAM constraint (i.e., valid items) in the FP-tree*, and then applies the usual FP-growth algorithm. Since all frequent patterns satisfying a SAM constraint can be formed solely by using the valid items, FPS avoids all unnecessary constraint checking at recursive steps/projected databases. *For a SUC constraint, the FPS algorithm partitions valid items into two groups, namely a "mandatory group" and an "optional group".* Then, all frequent patterns satisfying a SUC constraint must be "extensions" of an item from the mandatory group. For a superset constraint, the FPS algorithm partitions valid items into $k+1$ groups, namely k "mandatory groups" and an "optional group". Then, all frequent patterns satisfying a superset constraint must be "extensions" of a k-itemset that consists of an item from each of the k mandatory groups. Hence, FPS only needs to form projected databases for valid frequent patterns and avoids all unnecessary constraint checking at recursive steps/projected databases.

Example 2.3 (FPS for Handling Atomic SUC Constraint) *Let the transaction database TDB be the same database as in Example 2.1 with the auxiliary information about items as shown in the Table 2.3:*

Table 2.3: Auxiliary information for the transaction database TDB

Item	a	b	c	d	e	f
Price	40	10	25	30	20	35
Quantity	100	75	80	30	35	25

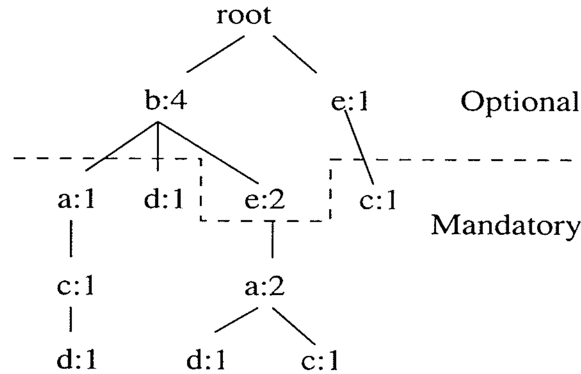

Figure 2.3: The FP-tree for the FPS algorithm

Let the SUC constraint be "$max(S.Price) \geq 25$" and the minimum support threshold be 2. Then, items a, c and d belong to the mandatory group, and b and e belong to the optional group. (Item f is infrequent.) Then, FPS builds an FP-tree in a slightly different way as shown in Figure 2.3, where the dashed line indicates the boundary between mandatory and optional items in the FP-tree. Here, mandatory items appear below optional items. Then, a projected database is formed for each item belonging to mandatory group ($\{d\}$-, $\{c\}$-, and $\{a\}$-projected databases). The usual FP-growth algorithm can be applied recursively to these projected databases to find all frequent patterns satisfying "$max(S.Price) \geq 25$".

The FPS algorithm can also handle conjunctions of succinct constraints. Based on the resulting MGF of the conjunctions of succinct constraints, the items can be divided into mandatory group(s) and optional group. Then, the FPS algorithm controls to construct frequent patterns by extending combinations of items from mandatory group(s).

Example 2.4 (FPS for handling conjunctions of SUC constraints) *Let the transaction database TDB be the same database as in Example 2.1 with the auxiliary information about items as shown in Table 2.3. Let SUC_1 be "$max(S.Price) \geq 25$" (same as the SUC constraint in Example 2.3), SUC_2 be "$min(S.Quantity) \leq 40$", and the minimum support threshold be 2. We already know that for constraint SUC_1, items a, c and d belong to the mandatory group, and b and e belong to the optional group. Then, for constraint SUC_2, items d, e belong to the mandatory group, and a, b and c belong to the optional group. As we learned from the Section 2.1 (for the conjunction of SUC_1 and SUC_2), we can divide items into 3 mandatory groups and 1 optional group. In particular, item d belong to group $Mandatory_1$, items a and c belong to group $Mandatory_2$, item e belong to group $Mandatory_3$, and item b belong to the optional group.*

The FPS algorithm builds a global FP-tree as shown in Figure 2.4. Then, a projected database is formed for each item belonging to group $Mandatory_1$ ($\{d\}$-projected database), and the combination of items belonging to group $Mandatory_2$ and group

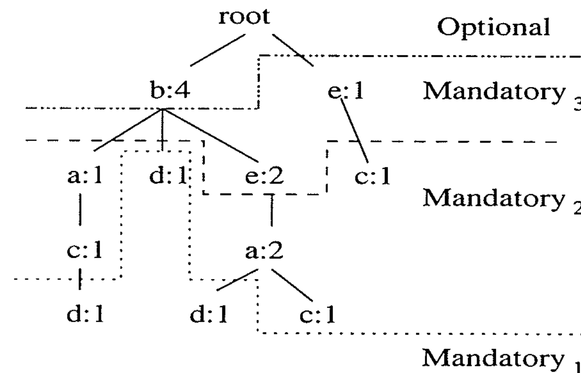

Figure 2.4: The FP-tree for the FPS Algorithm to Handle Conjunctions of SUC Constraints

$Mandatory_3$ ({ce}-, {ae}- projected database). The usual FP-growth algorithm can be applied recursively to these projected databases to find all frequent patterns satisfying conjunctions of the two SUC constraints.

The FPS algorithm is most relevant to our proposed algorithm, and satisfies a performance guarantee called *constraint checking and counting-optimality (ccc-optimality)*.

Definition 2.7 (ccc-optimality [LN+99]) A computation strategy is **constraint checking and counting-optimal (ccc-optimal)** for a class of constraints provided for every set C of constraints from that class, it satisfies the following conditions:

(1) The computation strategy counts for the frequency of an itemset S iff all subsets of S are frequent and S is valid;

(2) The computation strategy invokes the constraint checking operations on an itemset S only if S is a singleton itemset. □

With FP-trees, the FPS algorithm uses an MGF to generate exactly those itemsets that satisfy the succinct constraints. As a result, the constraint checking operations are done only on items (i.e., 1-itemsets), and frequency counting is done only for valid itemsets having frequent proper subsets. In other words, FPS is ccc-optimal. In this thesis, we also develop a ccc-optimal algorithm.

In addition to the FIC & FPS algorithms, there are some other less relevant constrained mining algorithms. For example, the DualMiner algorithm [BG+03] simultaneously handles both anti-monotone and monotone constraints, but it does not

consider succinct constraints. Kifer et al. [KG+03] proposed the concept of a "witness", which is a single itemset $\{X\}$ on which whether or not the constraint holds can be easily tested. The witness can be applied to mine itemsets with restrictions on their variance that are not covered by succinct constraints. Gade et al. [GWK04] also proposed the notion of block constraints, which cover aggregate succinct constraints but not other types of succinct constraints (e.g., the aggregate succinct constraint "$120 \in S.Price$").

2.3 Summary

In this chapter, we provided an overview of constraints and existing FP-growth related algorithms. Furthermore, we analyzed the properties of succinct constraints and conjunctions of succinct constraints, and learned that conjunctions of succinct constraints are succinct. Among those constrained mining algorithms we reviewed, the FPS algorithm efficiently handles succinct constraints and conjunctions of succinct constraints in the FP-tree based framework by utilizing the succinctness properties of these constraints and their conjunctions. However, there does not exist any work on handling negations or disjunctions of succinct constraints. After reviewing the related work in this chapter, we will describe our contribution of this thesis in the next few chapters (starting from Chapter 3). Specifically, in next two chapters, we will provide (i) our analysis of the properties of negations and disjunctions of succinct constraints and (ii) an algorithm called FPS* to efficiently handle them.

Chapter 3

Analysis of the Properties of Boolean Combinations of Succinct Constraints

In the previous chapter, we reviewed some existing work; in this chapter, we start describing our new work. To elaborate, we learned from Chapter 2 that the properties of conjunctions of succinct constraints have been exploited. In this chapter, we explore the properties of negations and disjunctions of succinct constraints. This exploration is independent of frameworks (the Apriori or the FP-tree framework). The explored properties are useful for handling Boolean combinations of succinct constraints.

3.1 Negations of Atomic Succinct Constraints

Through the exploration of properties of negations of atomic succinct constraints, we have the following lemma.

Lemma 3.1 The negation of atomic succinct constraints satisfies the following conditions:

(a) The negation of an atomic succinct constraint is also succinct.

(b) Let C be an atomic succinct constraint. Then, there exists an $MGF_{\neg C}$ that generates precisely those itemsets that satisfy $\neg C$ (i.e., the negation of C).

Proof: First, let us prove that the negation of an atomic succinct constraint is also succinct.

According to Definition 2.5, if C is an atomic succinct constraint, there exists a powerset $SAT_C(Item)$ for C. For the negation of C, the set of itemsets that satisfy $\neg C$ can be expressed by subtracting from 2^{Item} the set of itemsets that satisfy C, where 2^{Item} is the set of all possible itemsets. So, there exists a succinct powerset $2^{Item} - SAT_C(Item)$ for $\neg C$. Therefore, $\neg C$ is succinct.

The proof of part(b) of Lemma 3.1 is divided into the following three cases, depending on the nature of the constraint (SAM, SUC or superset constraint).

- Case 1 (SAM Constraints). Let C be a SAM constraint. From Table 2.1, it is easy to observe that C is equivalent to one of the forms summarized in the following table.

SAM Constraint C_s	Negation of C_s
S.A θ cn, where $\theta \in \{=, \leq, \geq\}$ S.A $\subseteq$ CS	S.A θ cn, where $\theta \in \{\neq, >, <\}$ S.A $\not\subseteq$ CS
min (S.A) $\geq$ cn max (S.A) $\leq$ cn	min (S.A) $<$ cn max (S.A) $>$ cn

We can observe from the table that some negations of SAM constraints are still SAM constraints, such as "S.A θ cn, where $\theta \in \{\neq, >, <\}$" and "S.A $\not\subseteq$ CS". Their MGFs are in the form of $\{X_1 \mid X_1 \subseteq \sigma_p(Item), X_1 \neq \emptyset\}$, for some selection predicate p.

However, some of the negations of SAM constraints, such as "min (S.A) $\geq$ cn" and "max (S.A) $\leq$ cn", are succinct non-anti-monotone (i.e., SUC constraints). The MGFs for those two are in the form of $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_p(Item), X_1 \neq \emptyset, X_2 \subseteq \sigma_{\neg p}(Item)\}$, for some selection predicate p.

- Case 2 (SUC Constraints) Let C be a SUC constraint. It is easy to observe from Table 2.1 that C is equivalent to one of the following: $cn \in S.A, min(S.A) \leq cn$, or $max(S.A) \geq cn$, as summarized in the table below.

SUC Constraint C_s	Negation of C_s
cn $\in$ S.A min (S.A) $\leq$ cn max (S.A) $\geq$ cn	cn $\notin$ S.A min (S.A) $>$ cn max (S.A) $<$ cn

We can observe from the table that all negations of SUC constraints, such as "cn $\notin$ S.A", "min (S.A) $>$ cn" and "max (S.A) $<$ cn", are succinct anti-monotone (i.e., SAM constraints). The MGFs for these three constraints are in the form of $\{X_1 \mid X_1 \subseteq \sigma_p(Item), X_1 \neq \emptyset\}$, for some selection predicate p.

- Case 3 (Superset Constraints) Let C be a superset constraint. It is easy to observe from Table 2.1 that C is equivalent to $S.A \supseteq CS$, as summarized in the table below.

Superset Constraint C_s	Negation of C_s
S.A $\supseteq$ CS	S.A $\not\supseteq$ CS

For the negations of the superset constraint "S.A $\supseteq$ CS", the form of MGF is much more complicated. It is a general MGF that contains a number of basic MGFs. The number of the basic MGFs is decided by the number of mandatory groups of the MGF for "S.A $\supseteq$ CS". As we know, the MGF for "S.A $\supseteq$ CS" is in the form of $\{X_1 \cup \ldots X_k \cup X_{k+1} \mid X_i \subseteq \sigma_{p_i}(Item) \& X_i \neq \emptyset, \text{for } 1 \leq i \leq k, X_{k+1} \subseteq \sigma_{p_{k+1}}(Item)\}$, where p_{k+1} is of the form $(\neg p_1 \wedge \ldots \wedge \neg p_k)$. It contains k mandatory groups and 1 optional group. To satisfy the negations of this constraint, the items belonging to each mandatory groups cannot be appeared at the same time, which means there are $2^k - 1$ possible types of combination of the succinct sets. Therefore, the MGF for the negation of the constraint "$S.A \supseteq CS$" contains $2^k - 1$ basic MGFs. Among the $2^k - 1$ basic MGFs, there are $2^k - 2$ MGFs in the form of $\{X \cup Y \mid X \subseteq \sigma_{q_j}(Item), \text{for } 1 \leq j \leq 2^k - 2, X \neq \emptyset, Y \subseteq \sigma_{\neg p_1 \wedge \ldots \wedge \neg p_k}(Item)\}$, where q_j is the $2^k - 2$ possible combinations of the succinct sets (such as $\neg p_1 \wedge \neg p_2 \wedge \ldots \wedge p_k$) and there is 1 MGF in the form of $\{X \mid X \subseteq \sigma_{\neg p_1 \wedge \ldots \wedge \neg p_k}(Item)\}$. Therefore, we can also see the negation of superset constraint as disjunctions of $2^k - 2$ SUC and 1 SAM constraints. □

Example 3.1 (Negation of a SAM constraint) *The negation of a SAM constraint may either a SAM or a SUC constraint. For example, the MGF for negation of*

a SAM constraint "S.Price > 100" is in the form of $\{X \mid X \subseteq \sigma_{S.Price \leq 100}(Item), X \neq \emptyset\}$. *The MGF for negation of a SAM constraint "min(S.Price) ≥ 50" is in the form of* $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{S.Price<50}(Item), X_1 \neq \emptyset, X_2 \subseteq \sigma_{S.Price \geq 50}(Item)\}$.

Example 3.2 (Negation of a SUC constraint) *The negation of a SUC constraint is a SAM constraint. For example, the MGF for the negation of a SUC constraint "50 ∈ S.Price" is in the form of* $\{X \mid X \subseteq \sigma_{(S.Price<50) \vee (S.Price>50)}(Item), X \neq \emptyset\}$.

Example 3.3 (Negation of a superset constraint) *The negation of a superset constraint is a succinct constraint with a general MGF. For example, the MGF for the negation of a superset constraint "S.Type ⊇ {snack, soda}" is a general MGF, namely* $\{X_1 \cup Y_1 \mid X_1 \subseteq \sigma_{Type=snack}(Item), X_1 \neq \emptyset, Y_1 \subseteq \sigma_{Type \neq snack \wedge Type \neq soda}(Item)\} \cup \{X_2 \cup Y_2 \mid X_2 \subseteq \sigma_{Type=soda}(Item), X_2 \neq \emptyset, Y_2 \subseteq \sigma_{Type \neq snack \wedge Type \wedge soda}(Item)\} \cup \{X_3 \mid X_3 \subseteq \sigma_{Type \neq snack \wedge Type \neq soda}(Item), X_3 \neq \emptyset\}$. *There are 2 mandatory groups in the MGF for "S.Type ⊇ {snack, soda}", so there are* $2^2 - 1 = 3$ *basic MGFs to form the general MGF for the negation of the constraint.*

So far, we have explored the properties of negations of atomic succinct constraints. We will explore the properties of negations of complex succinct constraints after we explore the properties of disjunctions of succinct constraints.

3.2 Disjunctions of Atomic Succinct Constraints

Through the exploration of properties of disjunctions of succinct constraints, we have the following lemma.

Lemma 3.2 The disjunction of n atomic succinct constraints satisfies the following conditions:

(a) The disjunction of n atomic succinct constraints is also succinct.

(b) Let $C_1, ..., C_n$ be some atomic succinct constraints. Then, there exists an MGF_{DSC} that generates precisely those itemsets that satisfy $C_1 \vee ... \vee C_n$ (i.e., the disjunction of C_i's).

Proof: First, let us prove part(a) of Lemma 3.2.

According to Definition 2.5, there exist n succinct powersets for n succinct constraints. The set of itemsets that satisfy the disjunction of n succinct constraints can be expressed by the union of succinct powersets of the succinct sets that satisfy each succinct constraint. In other words, there exists a succinct powerset $\bigcup_{i=1}^{n} SAT_{C_i}(Item)$ for the disjunction of n succinct constraints. Therefore, the disjunction of n succinct constraints is also succinct.

The proof of part(b) of Lemma 3.2 is divided into the following four cases.

- Case 1 (SAM Constraints). Let us consider the case where $C_1, ..., C_n$ are SAM constraints. The MGF of the disjunction of n SAM constraints is a general MGF that is union of n basic MGFs:

$$\begin{aligned} MGF_1 &= \{X \mid X \subseteq \sigma_{p_1}(Item) \& X \neq \emptyset\} \\ MGF_i &= \{Y \cup Z \mid Y \subseteq \sigma_{p_i \wedge \neg(p_{i-1} \vee ... \vee p_1)}(Item) \& Y \neq \emptyset, \\ &\quad Z \subseteq \sigma_{p_i \wedge (p_{i-1} \vee ... \vee p_1)}(Item)\}, \text{for } 2 \leq i \leq n. \end{aligned}$$

In MGF_1, we get the itemsets satisfying the first SAM constraints. In MGF_i, we only generate itemsets by combining the succinct sets that are newly valid

for the i-th SAM constraint with the succinct sets that have been processed by the previous $MGFs$ (i.e., valid for any of the previous SAM constraints). This result avoids computational redundancy (e.g., n times computation for some itemsets satisfying n SAM constraints). It is a union of n disjointed sets.

- Case 2 (SUC Constraints). For the case of the disjunction of n SUC constraints, the MGF is of the following form:

  $MGF = \{X \cup Y \mid X \subseteq \sigma_{p_1 \vee \ldots \vee p_n}(Item), X \neq \emptyset, Y \subseteq \sigma_{\neg(p_1 \vee \ldots \vee p_n)}(Item)\}$.

  We can observe that the resulting MGF is in a quite simple form, and avoids the computational redundancy. From Chapter 2, we learned that the itemsets satisfying a SUC constraint contains at least one item that belongs to the mandatory group. This means that once an item is valid for a SUC constraint, all its supersets satisfy the SUC constraint. The itemsets satisfying any of the n SUC constraints satisfy the disjunction of the n SUC constraints. Therefore, in the MGF for disjunctions of n succinct constraints, the items satisfying at least one of the n SUC constraints belong to the mandatory group, and the items that do not satisfy any of the n SUC constraints belong to the optional group.

- Case 3 (Superset Constraints). For the case of the disjunction of n superset constraints, the MGF is a union of n MGFs:

  $\bigcup_{i=1}^{n} MGF_i = \{X_1 \cup \ldots \cup X_{k+1} \mid X_j \subseteq \sigma_{p_j}(Item), 1 \leq j \leq k, X_j \neq \emptyset, X_{k+1} \subseteq \sigma_{\neg p_1 \wedge \ldots \wedge \neg p_k}(Item)\}$.

  The MGF_i is the MGF for the i-th superset constraint, and it contains k mandatory groups and one optional group. The value of k varies with each MGF_i.

The resulting MGF is the union of the n MGFs.

- Case 4 (Mixed Succinct Constraints). From the above analysis of the MGF for disjunctions of SUC constraints, we observe that for items belonging to the mandatory group, all their supersets can be enumerated. If we process other succinct constraints (say, SAM or superset constraints) after we process the SUC constraints, then we do not need to consider these mandatory items anymore. Therefore, when we handle the disjunctions of mixed succinct constraints, we first generate the MGF for those SUC constraints. Then, apply the solution for SAM or superset constraints to the remaining items (i.e., items belonging to the optional group for SUC constraints).

  Assume we have m SUC constraints, n SAM constraints and l superset constraints. The result of disjunctions of m SUC, n SAM and l superset constraints is the union of those $1+n+l$ MGFs as follow:

$$\begin{aligned}
MGF_{SUC} &= \{X \cup Y \mid X \subseteq \sigma_{p_1 \vee p_2 \vee \cdots \vee p_m}(Item), X \neq \emptyset, \\
&\quad Y \subseteq \sigma_{\neg(p_1 \vee p_2 \vee \cdots \vee p_m)}(Item)\} \\
MGF_{SAM_1} &= \{X \mid X \subseteq \sigma_{q_1 \wedge \neg(p_1 \vee p_2 \vee \cdots \vee p_m)}(Item), X \neq \emptyset\} \\
MGF_{SAM_i} &= \{X \cup Y \mid X \subseteq \sigma_{q_i \wedge \neg q_{i-1} \wedge \cdots \wedge \neg q_1 \wedge \neg(p_1 \vee p_2 \vee \cdots \vee p_m)}(Item), X \neq \emptyset, \\
&\quad Y \subseteq \sigma_{q_i \wedge q_{i-1} \wedge \cdots \wedge q_1 \wedge \neg(p_1 \vee p_2 \vee \cdots \vee p_m)}(Item)\}, i = 2, 3, ..., n \\
MGF_{superset_t} &= \{X_1 \cup \cdots \cup X_{k+1} \mid X_j \subseteq \sigma_{s_j}(Item), 1 \leq j \leq k, X_j \neq \emptyset, \\
&\quad X_{k+1} \subseteq \sigma_{\neg s_1 \wedge \cdots \wedge \neg s_k}(Item)\}, t = 1, 2, ..., l
\end{aligned}$$

  The resulting MGF is a union of $1+n+l$ MGFs. Here, $p_1, ..., p_m$ in MGF_{SUC} represent selection predicates for m SUC constraints, and $q_1, ..., q_n$ in MGF_{SAM_1}

and MGF_{SAM_i} (where $1 \leq i \leq n$) represent selection predicates for n SAM constraints. $MGF_{superset_t}$ (where $1 \leq t \leq l$) is the MGF for each superset constraint, in which $s_1, ..., s_k$ represent the selection predicates for the tth superset constraint. The value of k represents the number of selection predicates for the tth superset constraint, and varies with each $MGF_{superset_t}$.

In particular, the MGF for disjunctions of SUC mixed with SAM constraints is a union of $1 + n$ disjoint sets, which avoids computational redundancy. □

To gain a deeper understanding of our lemma, let us study the following examples.

Example 3.4 (Disjunctions of two SAM constraints) *Let* $C_{SAM_1} \equiv S.Price < 100$ *and* $C_{SAM_2} \equiv min(S.Price) > 50$. *Suppose for the items contained in transaction* $t_i = \{a, b, c, d, e, f\}$, *(1) items* a, b *have their Price values less than \$50, (2) items* c, d *have their Price values greater than \$50 and less than \$100, and (3) items* e, f *have their Price values greater than \$100. Then,* MGF_{SAM_1} *for* C_{SAM_1} *is* $\{X_1 \mid X_1 \subseteq \{a, b, c, d\} \& X_1 \neq \emptyset\}$. *Similarly,* MGF_{SAM_2} *for* C_{SAM_2} *is* $\{X_2 \mid X_2 \subseteq \{c, d, e, f\} \& X_2 \neq \emptyset\}$. *Taking their union gives a set of patterns satisfying* $C_{SAM_1} \vee C_{SAM_2}$. *However, some patterns, such as* $\{c\}, \{d\}$ *and* $\{c, d\}$, *were generated* twice *(first by* MGF_{SAM_1} *and then by* MGF_{SAM_2}*). To save computation, we use* $MGF_{SAM_1 \vee SAM_2} \equiv \{X_3 \mid X_3 \subseteq \{a, b, c, d\} \& X_3 \neq \emptyset\} \cup \{X_4 \cup X_5 \mid X_4 \subseteq \{e, f\} \& X_4 \neq \emptyset, X_5 \subseteq \{c, d\}\}$, *which generates the same set of patterns as taking the union but each pattern is generated only* once.

Example 3.5 (Disjunctions of two SUC constraints) *Let two SUC constraints* $C_{SUC_1} \equiv max(S.Price) > 100$ *and* $C_{SUC_2} \equiv min(S.Price) < 50$. *The items contained*

in transaction t_i are the same as Example 3.4. Then, MGF_{SUC_1} for C_{SUC_1} is $\{X_1 \cup X_2 \mid X_1 \subseteq \{e, f\} \& X_1 \neq \emptyset, X_2 \subseteq \{a, b, c, d\}\}$ and the MGF_{SUC_2} for C_{SUC_2} is $\{X_3 \cup X_4 \mid X_3 \subseteq \{a, b\} \& X_3 \neq \emptyset, X_4 \subseteq \{c, d, e, f\}\}$. Although taking their union will give a set of patterns satisfying $C_{SUC_1} \vee C_{SUC_2}$, the sets of patterns generating by these two MGFs are overlapping. In other words, some patterns (e.g., $\{a, b, c, d, e, f\}$) were generated more than once. *To save computation, we use $MGF_{SUC_1 \vee SUC_2} \equiv \{X_5 \cup X_6 \mid X_5 \subseteq \{a, b, e, f\} \& X_5 \neq \emptyset, X_6 \subseteq \{c, d\}\}$, which generates the same set of patterns as taking the union but each pattern is generated only* once.

Example 3.6 (Disjunctions of two superset constraints) *Let $C_{superset1} \equiv S.Type \supseteq \{soda, milk\}$ and $C_{superset2} \equiv S.Price \supseteq \{50, 100\}$. Suppose for the items contained in transaction $t_i = \{a, b, c, d, e, f\}$, (1) items a & c is soda, and items b & e is milk, (2) items a & b have their price values \$50, and c & d have their price values \$100. Then, $MGF_{superset1 \vee superset2}$ is $\{X_1 \cup X_2 \cup X_3 \mid X_1 \subseteq \{a, c\}, X_1 \neq \emptyset, X_2 \subseteq \{b, e\}, X_2 \neq \emptyset, X_3 \subseteq \{d, f\}\} \cup \{Y_1 \cup Y_2 \cup Y_3 \mid Y_1 \subseteq \{a, b\}, Y_1 \neq \emptyset, Y_2 \subseteq \{c, d\}, Y_2 \neq \emptyset, Y_3 \subseteq \{e, f\}\}$.*

Example 3.7 (Disjunctions of Mixed Succinct Constraints) *Let $C_{SAM1} \equiv S.Price < 80$, $C_{SAM2} \equiv min(S.Price) > 50$, $C_{SUC1} \equiv max(S.Price) > 100$ and $C_{SUC2} \equiv min(S.Price) < 30$. Suppose for the items contained in the transaction $t_i = \{a, b, c, d, e, f, g, h\}$ have the price value as showed in the following table.*

price<30	30<price<50	50<price<80	80<price< 100	100<price
a, b	c	d, e	f	g, h

Then, we can get the MGF for $C_{SAM1} \vee C_{SAM2}$. $MGF_{SAM1 \vee SAM2} \equiv \{X_1 \mid X_1 \subseteq \{a,b,c,d,e\}, X_1 \neq \emptyset\} \cup \{X_2 \cup X_3 \mid X_2 \subseteq \{f,g,h\}, X_2 \neq \emptyset, X_3 \subseteq \{d,e\}\}$. *We can also get the MGF for* $C_{SUC1} \vee C_{SUC2}$. $MGF_{SUC1 \vee SUC2} \equiv \{X_1 \cup X_2 \mid X_1 \subseteq \{a,b,g,h\}, X_1 \neq \emptyset, X_2 \subseteq \{c,d,e,f\}\}$. *Taking their union gives a set of patterns satisfying* $C_{SAM1 \vee SAM2 \vee SUC1 \vee SUC2}$. *However, some patterns, such as* $\{a\}, \{a,b\}$, *and* $\{g\}$, *were generated twice (first by* $MGF_{SAM1 \vee SAM2}$ *and then by* $MGF_{SUC1 \vee SUC2}$*).*

To save computation, we can directly get the MGF for disjunctions of SUC mixed with SAM constraints. $MGF_{MIX} \equiv \{X_1 \cup X_2 \mid X_1 \subseteq \{a,b,g,h\}, X_1 \neq \emptyset, X_2 \subseteq \{c,d,e,f\}\} \cup \{X_3 \mid X_3 \subseteq \{c,d,e\}, X_3 \neq \emptyset\} \cup \{X_4 \cup X_5 \mid X_4 \subseteq \{f\}, X_4 \neq \emptyset, X_5 \subseteq \{d,e\}\}$.

3.3 Negations and Disjunctions of Boolean Combinations of Succinct Constraints

In Section 3.1 and Section 3.2, we explored properties of negations and disjunctions of atomic succinct constraints. There exist negations or disjunctions of Boolean combinations of succinct constraints, such as negations of conjunctions of succinct constraints or disjunctions of negations of succinct constraints. We explore the properties of negations and disjunctions of Boolean combinations of succinct constraints in this section.

Generally, because conjunctions, negations and disjunctions of atomic succinct constraints are succinct, any Boolean combinations of succinct are also succinct. Therefore, there exist MGFs to represent the Boolean combinations of succinct con-

straints.

To handle the negation of succinct constraints, we first convert it into a form that we already have the solution, and then get the result. To elaborate, we analyze the negation of conjunctions of succinct constraints, the negation of disjunctions of succinct constraints and the negation of negation of succinct constraints below.

Let C_i (for $1 \leq i \leq n$) be succinct constraints. First, let us examine the negation of conjunctions of succinct constraints: $\neg(C_1 \wedge C_2 \wedge \ldots \wedge C_n)$. We can easily convert it into the following form: $\neg(C_1 \wedge C_2 \wedge \ldots \wedge C_n) \equiv \neg C_1 \vee \neg C_2 \vee \ldots \vee \neg C_n$. We already have the solution for handling disjunctions of negation of succinct constraints. Therefore, we can handle the negations of conjunctions.

Similar idea can be applied to the negation of disjunctions of succinct constraints: $\neg(C_1 \vee C_2 \vee \ldots \vee C_n)$. We can convert it into the following form: $\neg(C_1 \vee C_2 \vee \ldots \vee C_n) \equiv \neg C_1 \wedge \neg C_2 \wedge \ldots \wedge \neg C_n$. We have already proved that the negation of succinct constraints is succinct. So, the problem becomes to handle the conjunctions of succinct constraints. The paper of Lakshmanan et al. [LLN03] provides the solution.

The negation of negation of succinct constraint is itself: $\neg\neg C_1 = C_1$.

Now, let us explore disjunctions of Boolean combination of succinct constraints. To elaborate, we provide the analysis of disjunctions of conjunctions and disjunctions of negations of succinct constraints below.

Let C_i (for $1 \leq i \leq n$) be succinct constraints. First, let us examine disjunctions of conjunctions of succinct constraints: $(C_1 \wedge C_2) \vee (C_3 \wedge C_4) \ldots \vee (C_{n-1} \wedge C_n)$. According to the paper of Lakshmanan et al.[LLN03], the conjunctions of succinct constraints are succinct, so handling disjunctions of conjunctions of succinct constraints becomes

to handling disjunctions of succinct constraints and we already have the solution.

Similar idea applied to disjunctions of negation of succinct constraints: $\neg C_1 \vee \neg C_2 \vee \ldots \vee \neg C_n$. Because the negation of each succinct constraint is succinct as we have proved, handling disjunctions of negations of succinct constraints becomes to handling disjunctions of succinct constraints, and we already have the solution.

To gain a deeper understanding, let us consider the examples below.

Example 3.8 (Negations of conjunctions of succinct constraints) *Let the constraints and the items in the transaction be the same as Example 3.7. We want to get the MGF for* $\neg(C_{SAM1} \wedge C_{SAM2} \wedge C_{SUC1} \wedge C_{SUC2})$. *We first convert the negation of conjunctions of succinct constraints into the disjunctions of negation of succinct constraints:* $\neg(C_{SAM1} \wedge C_{SAM2} \wedge C_{SUC1} \wedge C_{SUC2}) \equiv \neg C_{SAM1} \vee \neg C_{SAM2} \vee \neg C_{SUC1} \vee \neg C_{SUC2}$. *Then, we get the negation of each succinct constraint.* $\neg C_{SAM1} \equiv S.Price \geq 80$ *that is a SAM constraint,* $\neg C_{SAM2} \equiv min(S.Price) \leq 50$ *that is a SUC constraint,* $\neg C_{SUC1} \equiv max(S.Price) \leq 100$ *that is a SAM constraint and* $\neg C_{SUC2} \equiv min(S.Price) \geq 30$ *that is a SAM constraint. Now, our problem becomes to get the disjunctions of one SUC constraint* ($\neg C_{SAM2}$) *mixed with 3 SAM constraints* ($\neg C_{SAM1}, \neg C_{SUC1}, \neg C_{SUC2}$). *Therefore, we first process the SUC constraint and get* $MGF_{\neg C_{SAM2}} \equiv \{X_1 \cup X_2 \mid X_1 \subseteq \{a, b, c\}, X_1 \neq \emptyset, X_2 \subseteq \{d, e, f, g, h\}\}$. *Then, we process the disjunctions of three SAM constraints. We can observe that* $\neg C_{SAM1}$ *is included to* $\neg C_{SUC2}$. *So we can actually process the disjunctions of* $\neg C_{SUC1}$ *and* $\neg C_{SUC2}$. $MGF_{\neg C_{SAM1} \vee \neg C_{SUC1} \vee \neg C_{SUC2}} \equiv \{X_3 \mid X_3 \subseteq \{d, e, f\}, X_3 \neq \emptyset\} \cup \{X_4 \cup X_5 \mid X_4 \subseteq \{g, h\}, X_4 \neq \emptyset, X_5 \subseteq \{d, e, f\}\}$. *The resulting MGF for* $\neg(C_{SAM1} \wedge C_{SAM2} \wedge C_{SUC1} \wedge C_{SUC2})$ *is the union of* $MGF_{\neg C_{SAM2}}$ *and* $MGF_{\neg C_{SAM1} \vee \neg C_{SUC1} \vee \neg C_{SUC2}}$.

Example 3.9 (Disjunctions of conjunctions of succinct constraints) *Let the constraints and the items in the transaction be the same as Example 3.7. We want to get the MGF for* $(C_{SAM1} \wedge C_{SAM2}) \vee (\neg C_{SUC1} \wedge C_{SUC2})$ *which means* $((S.Price < 80) \wedge (min(S.Price) < 50)) \vee ((max(S.Price) \leq 100) \wedge (min(S.Price) < 30))$. *We will first get the MGFs for two conjunctions and union them.*
$MGF_{(C_{SAM1} \wedge C_{SAM2}) \vee (\neg C_{SUC1} \wedge C_{SUC2})} \equiv \{X \mid X \subseteq \{d,e\}, X \neq \emptyset\} \cup \{Y_1 \cup Y_2 \mid Y_1 \subseteq \{a,b\}, Y_1 \neq \emptyset, Y_2 \subseteq \{c,d,e,f\}\}$.

3.4 Summary

From the paper of Lakshmanan et al. [LLN03], we learned from that the conjunctions of succinct constraints are succinct. In this chapter, we discovered (and proved) that the negations and disjunctions of succinct constraints are also succinct. Because of the succinctness of conjunctions, negations and disjunctions of atomic succinct constraints, Boolean combinations of succinct constraints are also succinct. Therefore, from the theoretical aspect, any succinct constraints connected by conjunctions, negations and disjunctions are succinct. Moreover, we provided specific MGFs for different kind of negations and disjunctions of succinct constraints. The MGFs for negations of atomic succinct constraints and disjunctions of multiple SAM, SUC and SUC mixed with SAM constraints are in either a basic MGF form, or in a form of a union of disjoint sets. Therefore, we can directly generate precisely all and only those frequent patterns satisfying negations of atomic succinct constraints and disjunctions of multiple SAM, SUC or SUC mixed with SAM constraints, avoiding generating and excluding frequent patterns not satisfying the constraints.

Hence, in the next chapter, we will utilize the succinctness of negations and disjunctions of succinct constraints to develop an algorithm called *FPS**, which is an extension of the FPS algorithm and includes the new part to efficiently handle negations and disjunctions of succinct constraints.

Chapter 4

The Design and Implementation of the FPS* Algorithm

The key contribution of Chapter 3 is the MGFs for handling the negations and disjunction of succinct constraints. The MGFs directly generate precisely all and only those itemsets satisfying the negation and disjunction of succinct constraints. The MGFs express "*which*" itemsets are to be generated and returned to the user. In this chapter, we show "*how*" to generate these itemsets. Note that the MGFs proposed in Chapter 3 are *framework independent*; they can be applied in different frameworks, including the Apriori and the FP-tree frameworks. In this chapter, we show how to use the MGFs to generate the valid itemsets in a specific framework, namely the FP-tree framework.

Our algorithm, called *FPS**, is an extension of the existing *FPS algorithm*. *FPS** inherits the functions to handle conjunctions of succinct constraints from FPS, and adds the new functions to handle the negations of succinct constraints and disjunc-

tions of n SAM, n SUC, and SUC mixed with SAM constraints, called $FPS^*_{Negation}$, FPS^*_{nSAM}, FPS^*_{nSUC} and FPS^*_{nMIX} respectively.

4.1 Negations of Atomic Succinct Constraints

The **negation of succinct constraints** can be mapped into a "positive" succinct constraint. Recall from Chapter 3 that (a) the negation of a SAM constraint is a SAM or SUC constraint, (b) the negation of a SUC constraint is a SAM constraint, and (c) the negation of a superset constraint can be expressed as a disjunction of multiple SUC constraints and one SAM constraint. Therefore, we only talk about how to handle negations of a SAM or SUC constraint in this section. Negations of a superset constraint will be covered when we talk about disjunctions of succinct constraints in the next section.

To get the itemsets satisfying the negation of a SAM or SUC constraint, we first convert the negations into a "positive" form, and then apply the FPS algorithm to find the itemsets satisfying the "positive" succinct constraint. We call this $FPS^*_{Negation}$. For example, the negation of the succinct constraint "$min(S.Price) \leq 100$" can be mapped into the succinct constraint "$min(S.Price) > 100$". Then, we can apply the FPS algorithm to find frequent patterns satisfying "$min(S.Price) > 100$". These patterns will be the answer for the negation of "$min(S.Price) \leq 100$".

4.2 Disjunctions of Atomic Succinct Constraints

In the previous chapter, we provided the MGFs for disjunctions of succinct constraints. In particular, the MGFs for disjunctions of n SAM and mixed (SAM mixed with SUC) constraints are unions of disjoint sets, and the MGF for disjunctions of n SUC constraints is a basic MGF. We design the corresponding algorithm for these cases of disjunctions.

First, let us study how we handle the **disjunction of n SAM constraints**. Like many FP-tree based algorithms, the algorithm handles the disjunction of n SAM constraints by using two main operations: (1) the construction of the FP-tree and (2) the recursive growth of valid frequent patterns. Particularly, in this algorithm, besides the global FP-tree and conditional FP-trees, we need to construct constraint FP-trees for each SAM constraint. Basically, a *constraint FP-tree* for C_{SAM_j} is an FP-tree for C_{SAM_j}, in which only all those items that are valid (whether they are processed or unprocessed) for C_{SAM_j} are kept.

We first construct a global FP-tree for all frequent items and use an n-bit vector to represent the status of each item. The n-bit vector is initialized to "0 ... 0". For each frequent item, if it satisfies the ith SAM constraint C_{SAM_i} in the disjunction, the ith bit of the vector corresponding to the item will be set to 1 (where $1 \leq i \leq n$). Then, we can construct the constraint FP-tree for the SAM constraint C_{SAM_1} by scanning the 1st bit of the n-bit vectors to find the valid items for C_{SAM_1}, traversing the global FP-tree and extracting nodes satisfying C_{SAM_1} to form the constraint FP-tree for C_{SAM_1}, and then process the constraint. For the remaining constraints, we process them as follows. We scan the 1st to $(j-1)^{th}$ bits of the n-bit vectors for the valid items

(i.e., items that satisfy C_{SAM_j}). If any one of the 1st to $(j-1)^{th}$ bits in the vector is equal to 1, it means that the corresponding item has been processed by a previous SAM constraint; if not, such an item is valid for C_{SAM_j} as well as unprocessed. After the scan, we build a *constraint FP-tree* for j^{th} constraint. When we construct the constraint FP-tree for these items satisfying C_{SAM_j}, the status of an item ("processed" or "unprocessed") is kept in a flag (instead of an n-bit vector as in the global FP-tree). After the constraint FP-tree is built, the conditional FP-tree will be constructed for unprocessed items by traversing the constraint FP-tree up and down (in the same fashion as in the iCFP algorithm [Leu04]). For the conditional FP-tree, the usual FP-growth algorithm can be applied. We call this algorithm FPS^*_{nSAM}.

Example 4.1 (FPS^*_{nSAM} for handling SAM constraints) *Consider the same transaction database TDB in Table 2.2. Let C_1, C_2 and C_3 be three SAM constraints. Items b, c, d are valid for C_1; items b, a, c are valid for C_2; and, items a, c, d, e are valid for C_3. Let the minimum support threshold be set to 2.*

We first construct the global FP-tree for all frequent items as Figure 4.1. A 3-bit vector for each item is constructed with the FP-tree, which represents the status of each item for each constraint. For example, the vector $\langle 1, 1, 0 \rangle$ for item a indicates that item a satisfies constraints C_1 and C_2 but not C_3.

Then, we scan the 1st bit of each 3-bit vector to find the valid items for C_1 so that we can construct the constraint FP-tree for C_1 (as shown in Figure 4.2). We can see from the figure that a flag is kept and associated with each valid item contained in the constraint FP-tree for C_1. The flag 0 means the item has not been processed. So, the usual FP-growth algorithm can be applied to this constraint FP-tree for C_1

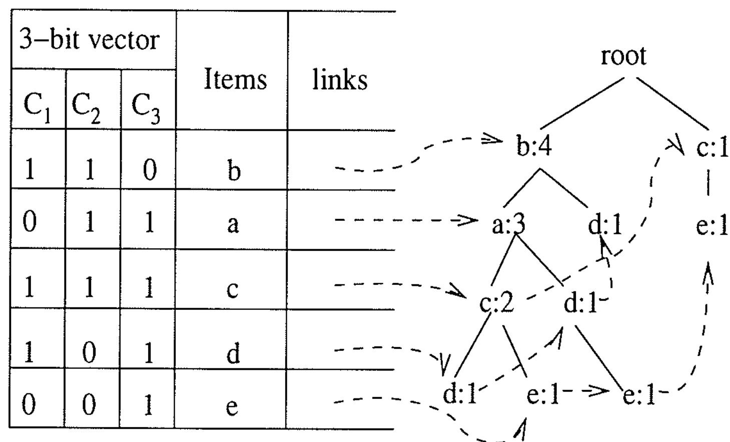

Figure 4.1: The global FP-tree

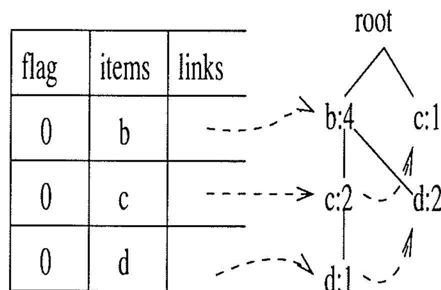

Figure 4.2: The constraint FP-tree for C_1

to find frequent patterns satisfying C_1. In other words, we need to build $\{d\}$-, $\{c\}$-, and $\{b\}$-projected databases and their corresponding conditional FP-trees for finding frequent patterns containing d, c or b. After the mining process for C_1, the constraint FP-tree for C_1 can be discarded. Then, we construct the constraint FP-tree for C_2 (as shown in Figure 4.3).

We can see from the flag in Figure 4.3 that item b and c have been processed. So, we only need to build conditional FP-tree for item a. Following is the conditional FP-

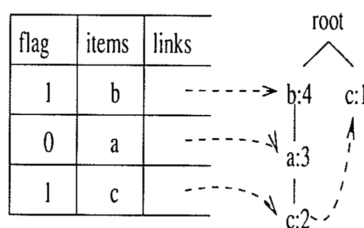

Figure 4.3: The constraint FP-tree for C_2

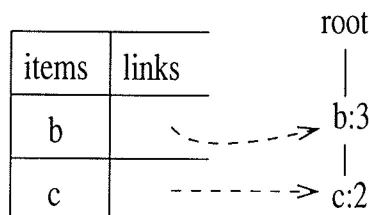

Figure 4.4: The conditional FP-tree for the $\{a\}$-projected database

tree for item a. To construct the conditional FP-tree for the $\{a\}$-projected database, we need to traverse the constraint FP-tree for C_2 up and down [Leu04] to include nodes above and below item a.

After we construct the FP-tree for the $\{a\}$-projected database and its corresponding conditional FP-tree (shown in Figure 4.4), the usual FP-growth algorithm can be applied.

Finally, we construct the constraint FP-tree for C_3 as shown in Figure 4.5, where (unprocessed) frequent patterns satisfying C_3 (e.g., frequent patterns containing e) can be found.

As a result, we get exactly the frequent patterns that satisfy the disjunction of C_1, C_2, and C_3.

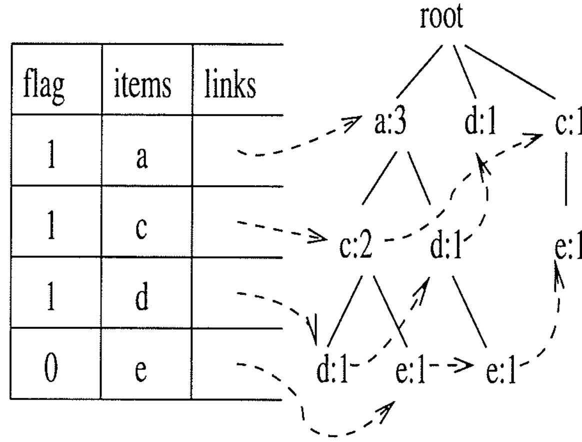

Figure 4.5: The constraint FP-tree for C_3

Then, let us study how we handle the **disjunction of n SUC constraints.** As one can observe from Example 3.5, frequent patterns satisfying the disjunction can be generated by using MGF_{DSUC} of the form $\{X_1 \cup X_2 \mid X_1 \subseteq \{\text{items that satisfy any one of the } n \text{ SUC constraints}\} \& X_1 \neq \emptyset, X_2 \subseteq \{\text{other items}\}\}$. The MGF is in the same form of the MGF for one SUC constraint. Therefore, with a little modification, the FPS algorithm that handles one SUC constraint can be applied to find frequent patterns satisfying the disjunction of n SUC constraints. We call our modified algorithm FPS^*_{nSUC}. With our algorithm, we divide the frequent items into mandatory group and optional group as in the FPS algorithm. However, the key difference between the FPS algorithm (which can handle *conjunction* of SUC constraints) and our algorithm is that items that satisfy **any one of the n SUC constraints** belong to the mandatory group and others belong to the optional group in FPS^*_{nSUC}.

Finally, let us study how we handle the disjunction of m **SUC mixed with**

n **SAM constraints.** The solution we provided is a combination of the solutions to handle m SUC constraints and n SAM constraints. We first process the m SUC constraints by using FPS^*_{nSUC}. And then, we apply the FPS^*_{nSAM} to the items that belong to the optional group for the m SUC constraints and the items that satisfy any one of n SAM constraints. We call this algorithm FPS^*_{nMIX}.

Example 4.2 (FPS^*_{nMIX} **for handling disjunctions of mixed constraints)** *Consider the same transaction database TDB in Table 2.2. Let C_1, C_2 and C_3 be three SAM constraints as Example 4.1. Let items d and e belong to the mandatory group of SUC constraints. Let the minimum support threshold be set to 2.*

We can construct the global FP-tree as shown in Figure 4.6. We only construct the vectors for items that only valid for SAM constraints (not for items that belong to mandatory group of SUC constraints). During the mining process, we construct the FP-tree for $\{e\}$-projected database and $\{d\}$-projected database in the same way as in the FPS^*_{nSUC} algorithm. The $\{e\}$- and $\{d\}$-conditional FP-trees are also shown in

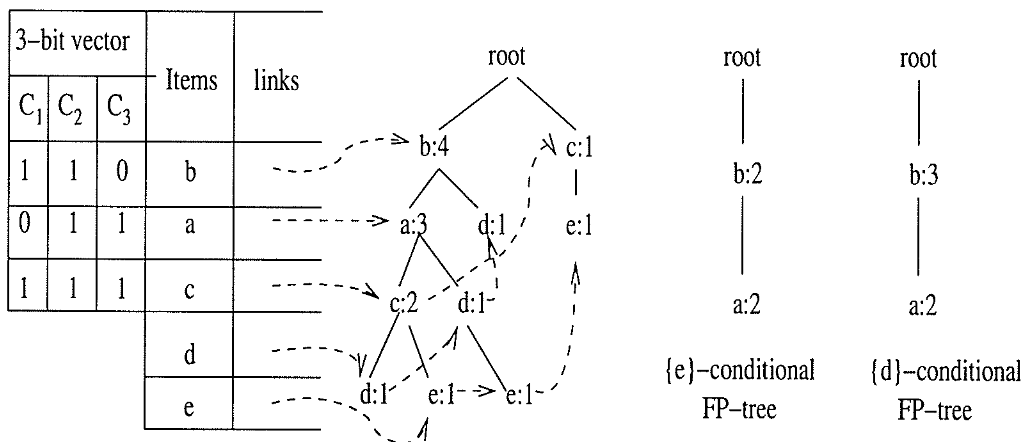

Figure 4.6: FP-trees for FPS^*_{nMIX}

Figure 4.6. After that, we process items b, a and c by using FPS^*_{nSAM} *algorithm (the same solution as in Example 4.1).*

4.3 Summary

In this chapter, we developed the algorithm *FPS**, which is an extension of the existing FPS algorithm. Our FPS* algorithm inherits the functions to handle conjunctions of succinct constraints from FPS, and adds the new functions to handle the negations and disjunctions of succinct constraints. Specifically, the *FPS** algorithm includes $FPS^*_{Negation}$, FPS^*_{nSAM}, FPS^*_{nSUC} and FPS^*_{nMIX} for handling negations of succinct constraint and disjunctions of multiple SAM, SUC, or SUC mixed with SAM constraints. The FPS* algorithm pushes the succinctness inside the mining process so that the search for patterns is confined to only those of interest to the user. This improves efficiency. The experimental evaluation of the FPS* algorithm will be provided in the next chapter.

Chapter 5

Experimental Results

We implemented the FPS* algorithm in C programming language. We used several synthetic databases generated by the program developed at IBM Almaden Research Center [AS94], as well as real-life databases from UC Irvine Machine Learning Repository [BM98] and FIMI Repository [GZ03a], as the testing databases because they are considered to be *benchmark datasets* in the field of data mining. The results produced are consistent. So, unless stated otherwise, all experimental results cited below are based on IBM transaction databases (which consists of 1M records with an average transaction length of 10 items and a domain of 1000 items). All experiments were run in a time-sharing environment in a 1 GHz machine. The reported figures are based on the average of multiple runs. Runtime includes CPU and I/Os that includes the time for both tree construction and frequent pattern mining steps. For lack of existing algorithm handling the negations or disjunctions of succinct constraints, in the experiments, we mainly compared FPS* algorithm with **FP-growth++**, where we first generate all frequent patterns by using the FP-growth algorithm [HPY00] and

then run a post-processing step to check if the frequent patterns satisfy the negations or disjunctions of succinct constraints.

5.1 Experimental Results for Negation of Atomic Succinct Constraints

As we described in Chapter 4, the $FPS^*_{Negation}$ algorithm first converts a negation of succinct constraint into a SAM or SUC constraint, and then performs the corresponding mining process for SAM or SUC constraint.

5.1.1 $FPS^*_{Negation}$ for a SAM Constraint

We set up three experiments to test our $FPS^*_{Negation}$ algorithm for a SAM constraint.

Experiment 5.1 In this experiment, we tested how the *minimum support threshold* values affect the runtime of algorithms. We varied minsup from 0.0025% to 0.025%, and showed results for selectivity equals to 20%, 40%, 60% and 80%. A constraint with $p\%$ selectivity means $p\%$ of items were selected. The *y-axis* of Figure 5.1 shows the runtime, and the *x-axis* shows minsup.

We can observe from Figure 5.1 that when the minsup decreases, the runtimes of both $FPS^*_{Negation}$ and FP-growth++ increase, because more (valid) frequent patterns are generated. In terms of speedup, $FPS^*_{Negation}$ outperforms FP-growth++, especially when selectivity is low, because $FPS^*_{Negation}$ only mines valid items and only generates valid frequent patterns.

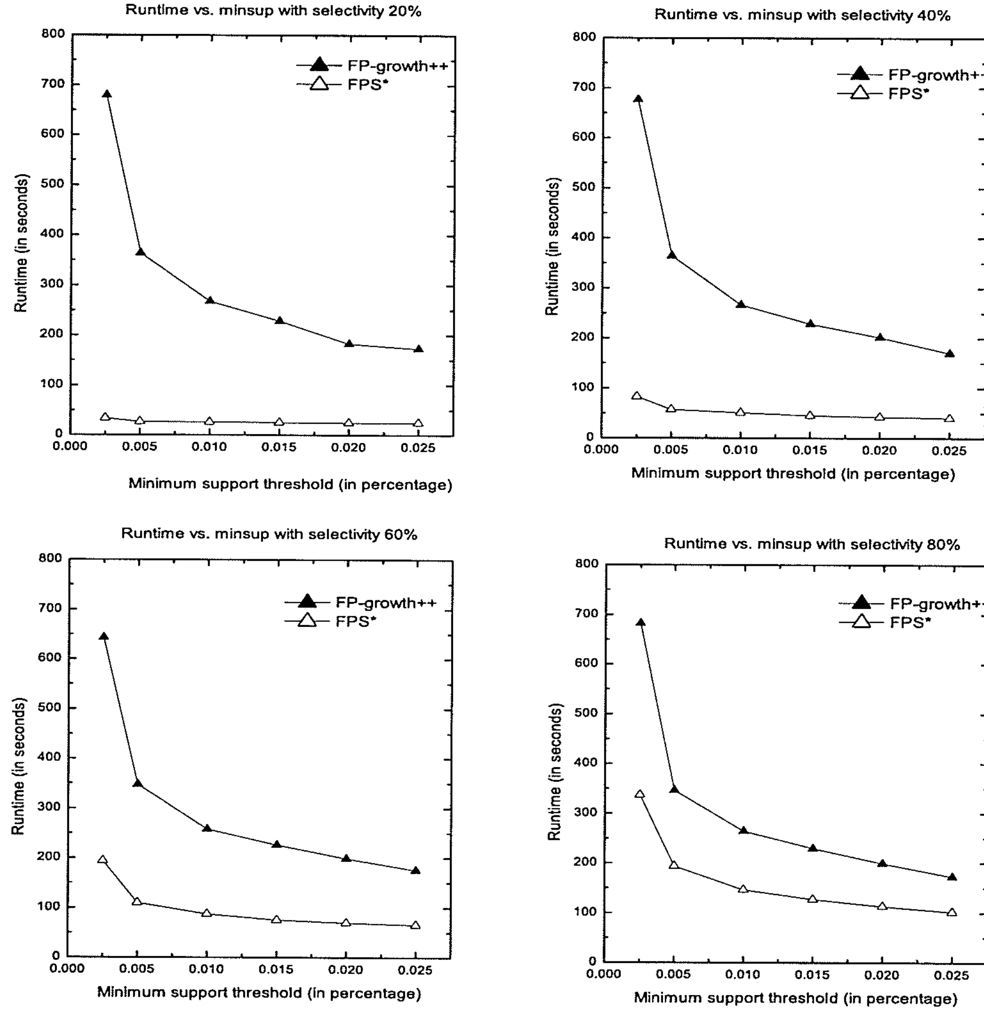

Figure 5.1: Results for a SAM constraint with different minsup (Experiment 5.1)

Experiment 5.2 In this experiment, we tested how the *selectivity* values affect the runtime of the algorithms. We varied selectivity from 20% to 80%, and showed the results for minsup equals to 0.02%, 0.01%, 0.005% and 0.0025%. The *y-axis* of Figure 5.2 shows the runtime, and the *x-axis* shows selectivity. We can observe

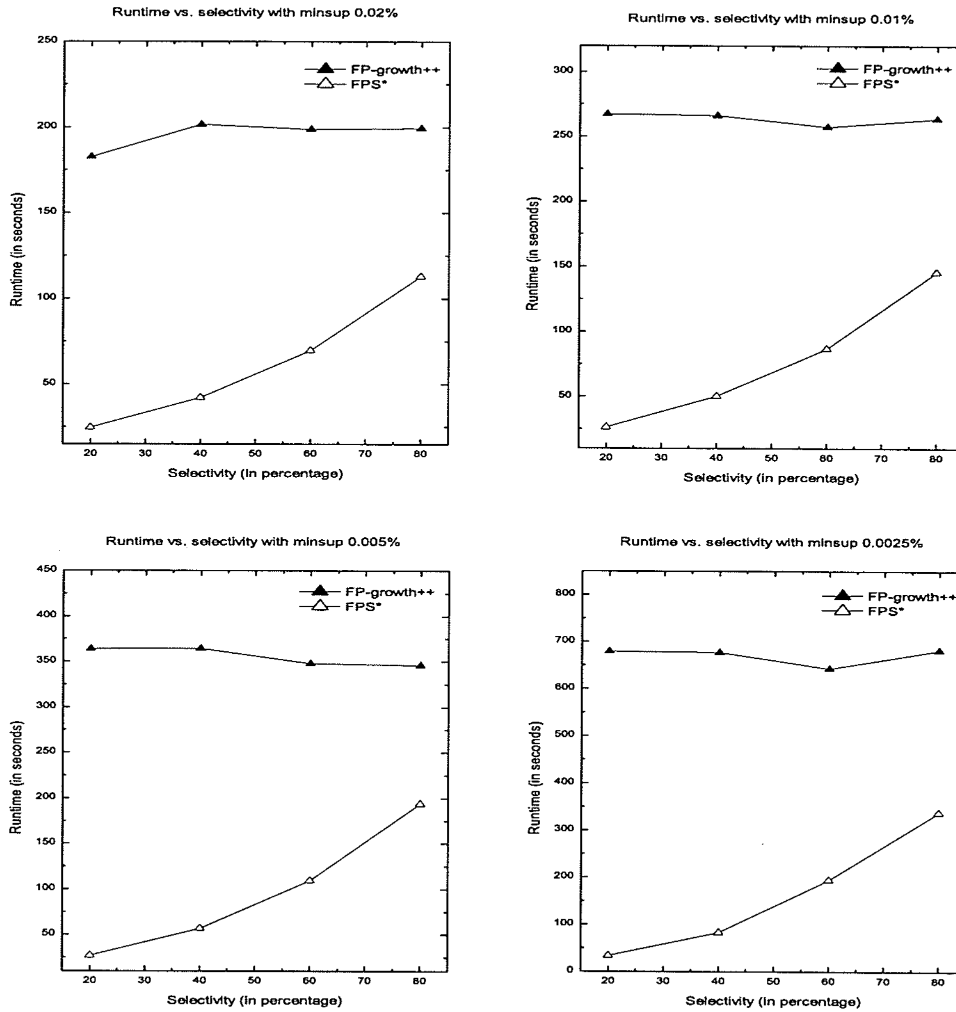

Figure 5.2: Results for a SAM constraint with different selectivity (Experiment 5.2)

from Figure 5.2 that when selectivity increases, the runtime of $FPS^*_{Negation}$ increases while the runtime of FP-growth++ changes slightly. The reason is that $FPS^*_{Negation}$ exploits the succinctness property. When selectivity increases, more valid items are mined, which leads to the generation of more valid frequent patterns. Therefore, the

runtime increases. However, FP-growth++ is not affected so much by selectivity. Recall from the mining process of the FP-growth++ algorithm, we need generate all frequent patterns and then check if the pattern satisfy the SAM constraint. Each frequent pattern needs to be checked regardless of selectivity. So, the runtime of FP-growth++ is almost the same (or changes slightly) with the increase of selectivity.

From the results of these two experiments, we can see the gain in performance of $FPS^*_{Negation}$.

Experiment 5.3 In this experiment, we tested scalability with the number of transactions. We fixed the minsup to 0.025% and selectivity to 20%. The results in Figure 5.3 show that mining with $FPS^*_{Negation}$ for a SAM constraint has linear scalability.

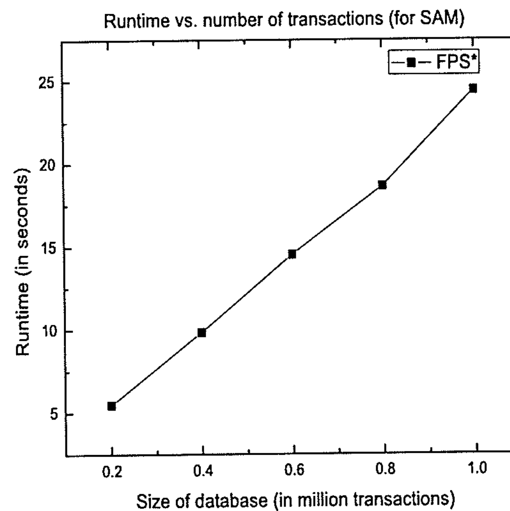

Figure 5.3: Scalability for $FPS^*_{Negation}$ for a SAM constraint (Experiment 5.3)

5.1.2 $FPS^*_{Negation}$ for a SUC Constraint

Similarly, we also set up some experiments to test our $FPS^*_{Negation}$ algorithm for a SUC constraint.

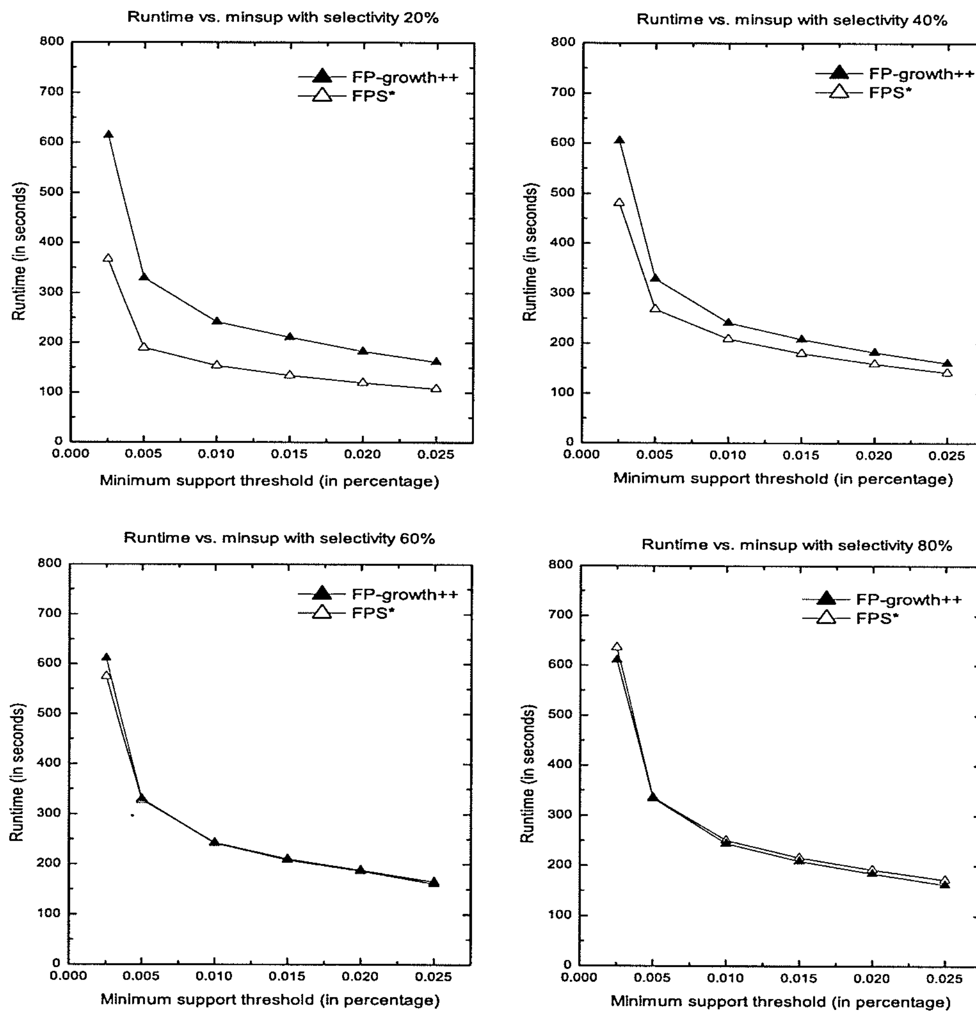

Figure 5.4: Results for a SUC constraint with different minsup (Experiment 5.4)

Experiment 5.4 In this experiment, we tested how the *minsup* values affect the runtime of algorithms. We varied minsup from 0.0025% to 0.025%, and showed results for selectivity equals to 20%, 40%, 60% and 80%. Results are shown in Figure 5.4.

From Figure 5.4, we can observe that the runtimes of $FPS^*_{Negation}$ and FP-growth++ increase as the minsup decreases, because more (valid) frequent patterns are generated. $FPS^*_{Negation}$ works well when selectivity is low. However, when selectivity is set to 60% and 80%, the runtime is quite close for both $FPS^*_{Negation}$ and FP-growth++. To analyze those results, we conduct Experiment 5.5.

Experiment 5.5 In this experiment, we tested how the *selectivity* values affect the runtime of algorithms. We varied selectivity from 20% to 80%, and showed results for minsup equals to 0.02%, 0.01%, 0.005% and 0.0025%. From Figure 5.5, we can see that when selectivity is set to 80%, FP-growth++ is even faster than $FPS^*_{Negation}$. The speedup of $FPS^*_{Negation}$ for SUC constraints is not as good as that for the SAM constraint.

As we analyzed in Chapter 3, the MGF for a SUC constraint is as follow: $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_p(Item), X_1 \neq \emptyset, X_2 \subseteq \sigma_{\neg p}(Item)\}$, for some selection predicate p. The valid frequent patterns can be grown by any mandatory items with optional items; in contrast, for SAM constraints, the valid frequent patterns can be grown by only mandatory items. Therefore, when selectivity is high, the percentage of valid frequent patterns is very high for the SUC constraint, but can still be low for the SAM constraint. We designed $FPS^*_{Negation}$ based on the succinctness of SAM and SUC constraints, and it avoids generating invalid frequent patterns. so, it should

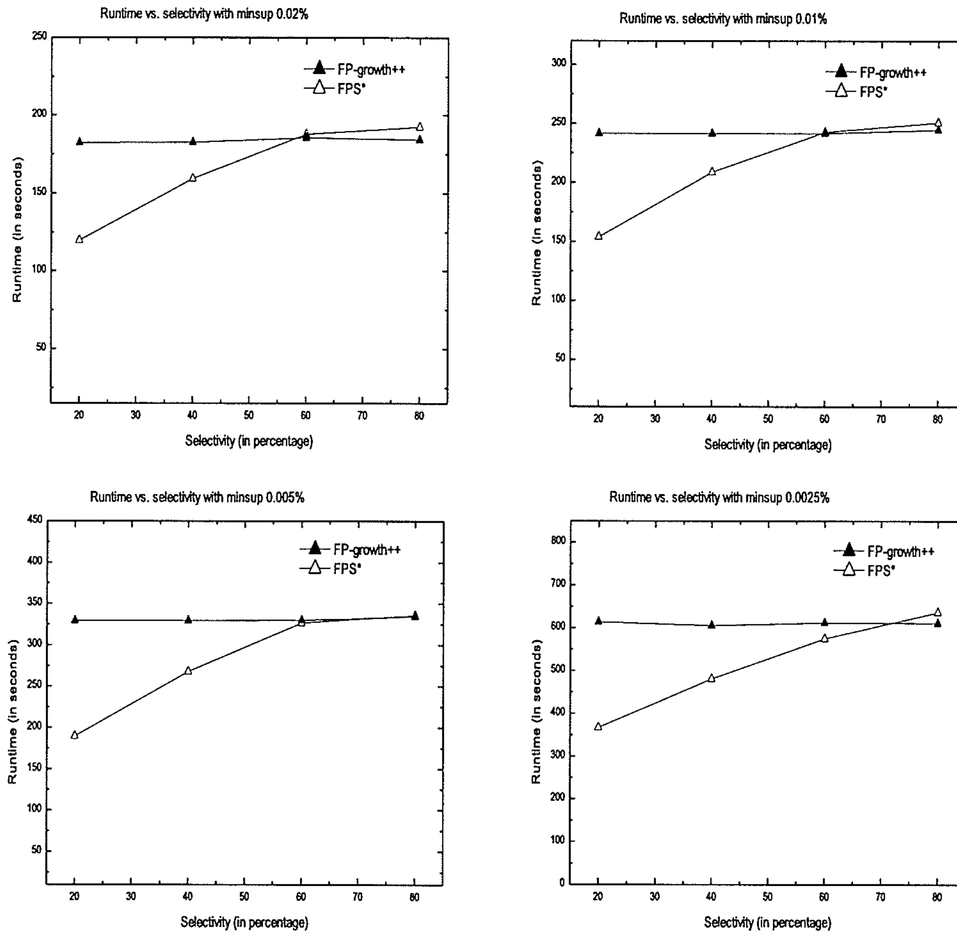

Figure 5.5: Results for a SUC constraint with different selectivity (Experiment 5.5)

outperform FP-growth++. However, in the case where almost all frequent patterns are valid, $FPS^*_{Negation}$ has no advantage.

We can observe from the first two columns of Table 5.1 if the selectivity is set to 80% and minsup is set to 0.0025%, almost all frequent patterns are valid for the SUC constraint.

Table 5.1: The number of frequent patterns for FP-growth++ and $FPS^*_{Negation}$ (Experiment 5.5)

Level of itemsets	FP-growth++	FPS*(SUC)	FPS*(SAM)
L_1	966	776	776
L_2	259408	248795	164837
L_3	754323	747379	367224
L_4	167546	167313	68198
L_5	87717	87697	29419
L_6	37878	37877	10825
L_7	13140	13140	3430
L_8	3517	3517	874
L_9	666	666	156
L_{10}	81	81	18

Recall from the design of $FPS^*_{Negation}$ in Chapter 4, in order to generate only valid frequent patterns, $FPS^*_{Negation}$ needs to sort the items into mandatory group and optional group, and the items within each group are sorted in descending frequency order (i.e., sorted primarily based by group and secondarily by frequency order); in contrast, all items in FP-growth++ are just sorted in descending frequency order. Because of the difference in the ordering schemes, FP-trees (global FP-tree and conditional FP-trees) generated by $FPS^*_{Negation}$ may not be as compact as the FP-trees generated by FP-growth++. A bushy FP-tree leads to longer tree construction time, longer traversal time and longer time for more support counting.

Table 5.2: The number of nodes in FP-trees and the number of support counting (Experiment 5.5)

Algorithms	FPS*	FP-growth++
Number of nodes in the global tree	6876768	6691748
Number of nodes in all trees	40358757	39791147
Number of support count	36217983	35884111

We can observe from Table 5.2 that, when selectivity is set to 80% and minsup is set to 0.0025%, The FP-tree for FPS* contains many more nodes; during the mining process, FPS* did more support counting. Therefore, when the time prolonged by the construction and mining of a bushy FP-tree is almost the same or may be even longer than the time saved by avoiding the generation of invalid frequent patterns, the performance of $FPS^*_{Negation}$ is close to or even worse than FP-growth++.

For SAM constraints, the situations are quite different. First, we can observe from the first and third column of Table 5.1 that the percentage of valid frequent patterns is still low when the selectivity is high. Second, $FPS^*_{Negation}$ does not need any other process (e.g., both $FPS^*_{Negation}$ and FP-growth++ use the same ordering scheme) for generating valid patterns. Therefore, the performance of $FPS^*_{Negation}$ for SAM constraints is better than FP-growth++ even when selectivity is high.

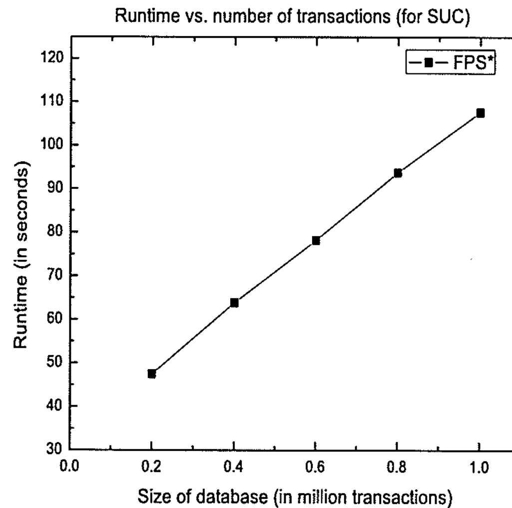

Figure 5.6: Scalability for $FPS^*_{Negation}$ for a SUC constraint (Experiment 5.6)

Experiment 5.6 In this experiment, we tested scalability with the number of transactions. We fixed the minsup to 0.025% and selectivity to 20%. The results in Figure 5.6 show that mining with $FPS^*_{Negation}$ for a SUC constraint has linear scalability.

5.2 Experimental Results for Disjunctions of SAM Constraints

We set up the following experiments to test our FPS^*_{nSAM} algorithm.

Experiment 5.7 In this experiment, we tested how the *minsup* values affect the runtime of algorithms. We set the number of constraints to 4 and varied the minsup for disjunctions of SAM constraints from 0.0025% to 0.025%. We showed the results for selectivity of disjunctions of SAM constraints equals to 20%, 40%, 60% and 80%.

The *y-axis* of Figure 5.7 shows the runtime, and the *x-axis* shows minsup. When minsup decreases, the runtime of FP-growth++ algorithm increases, while FPS^*_{nSAM} changes a little. FPS^*_{nSAM} outperforms FP-growth++, especially when minsup is low.

The mining process of the FP-growth++ algorithm needs to generate all frequent patterns and then to check if the patterns satisfy any of the SAM constraint. When minsup decreases, more frequent patterns are generated. Because FP-growth++ generates all frequent patterns before the post-processing, it is sensitive to minsup. In contrast, FPS^*_{nSAM} is not sensitive to minsup because it exploits succinctness, which

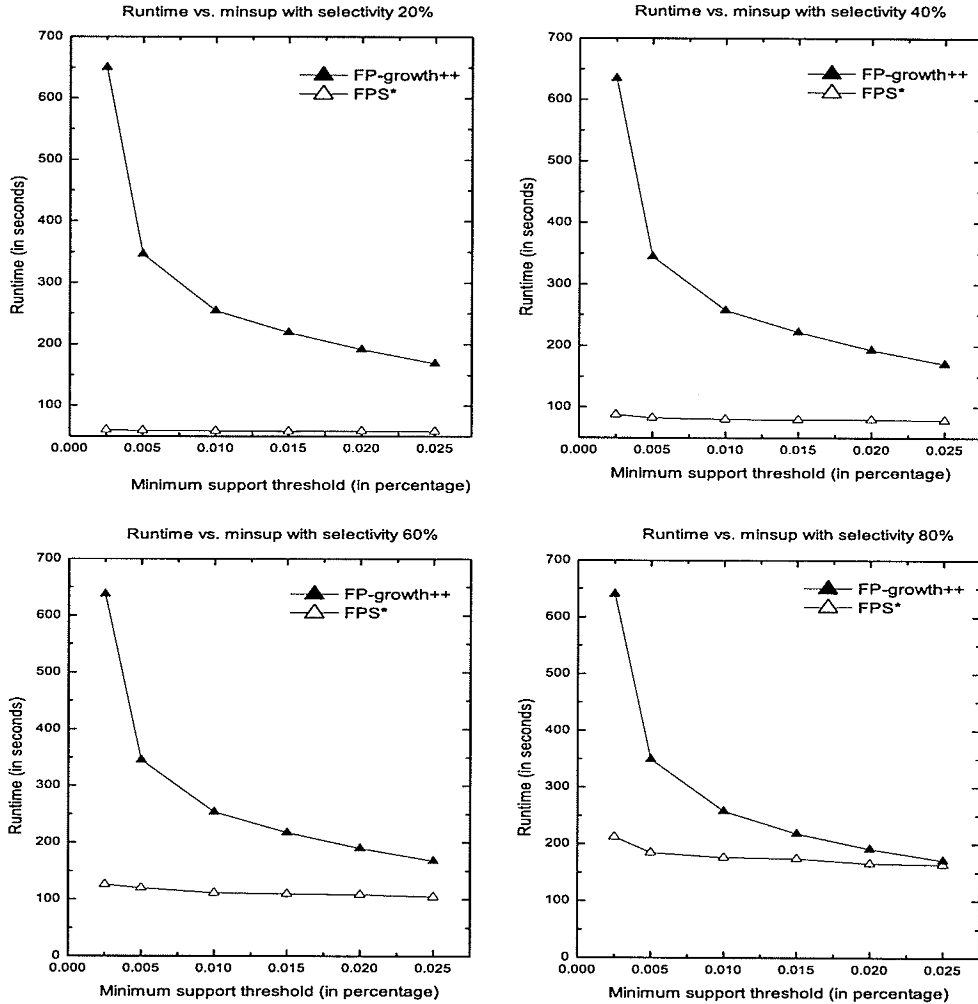

Figure 5.7: Results for disjunctions of SAM constraints with different minsup (Experiment 5.7)

dramatically decreases the number of frequent patterns to be generated (even when the minsup is low).

FPS^*_{nSAM} performs quite well when minsup is low (and even with high) selectivity. If the minsup is low, more patterns (especially patterns with high cardinality) are

Table 5.3: The number of frequent patterns for FP-growth++ and FPS^*_{nSAM} (Experiment 5.7)

Level of itemsets	FP-growth++	FPS^*_{nSAM}
L_1	966	771
L_2	259408	53692
L_3	754323	37975
L_4	167546	1994
L_5	87717	250
L_6	37878	37
L_7	13140	3
L_8	3517	0
L_9	666	0
L_{10}	81	0

frequent. However, this does not necessarily increase the number of patterns satisfying the constraints. We can observe from Table 5.3 that, when selectivity was set to 80% and minsup was set to 0.0025%, a great number of invalid frequent patterns were generated by FP-growth++. By exploiting the succinct property, FPS^*_{nSAM} effectively avoids generating those frequent but not valid patterns.

However, when minsup and selectivity are both high, the gain of performance of FPS^*_{nSAM} is not so much. As we designed FPS^*_{nSAM} in Chapter 4, in order to only generate valid frequent patterns once, FPS^*_{nSAM} needs to construct constraint trees and traverse constraint trees up and down (more time is consuming in this bi-directional traversal than the upward-only traversal in FP-growth++) in some cases. These two processes also take time. If the time for constructing constraint trees and traversing constraint trees up and down is comparable to the time generating invalid frequent patterns and checking, FPS^*_{nSAM} has no obvious advantage over FP-growth++.

Experiment 5.8 In this experiment, we tested how the *selectivity* values affect the runtime of algorithms. We set the number of constraints to 4 and varied the selectivity from 20% to 80%. We showed the results for minsup of disjunctions of SAM constraints equals to 0.02%, 0.01%, 0.005% and 0.0025%.

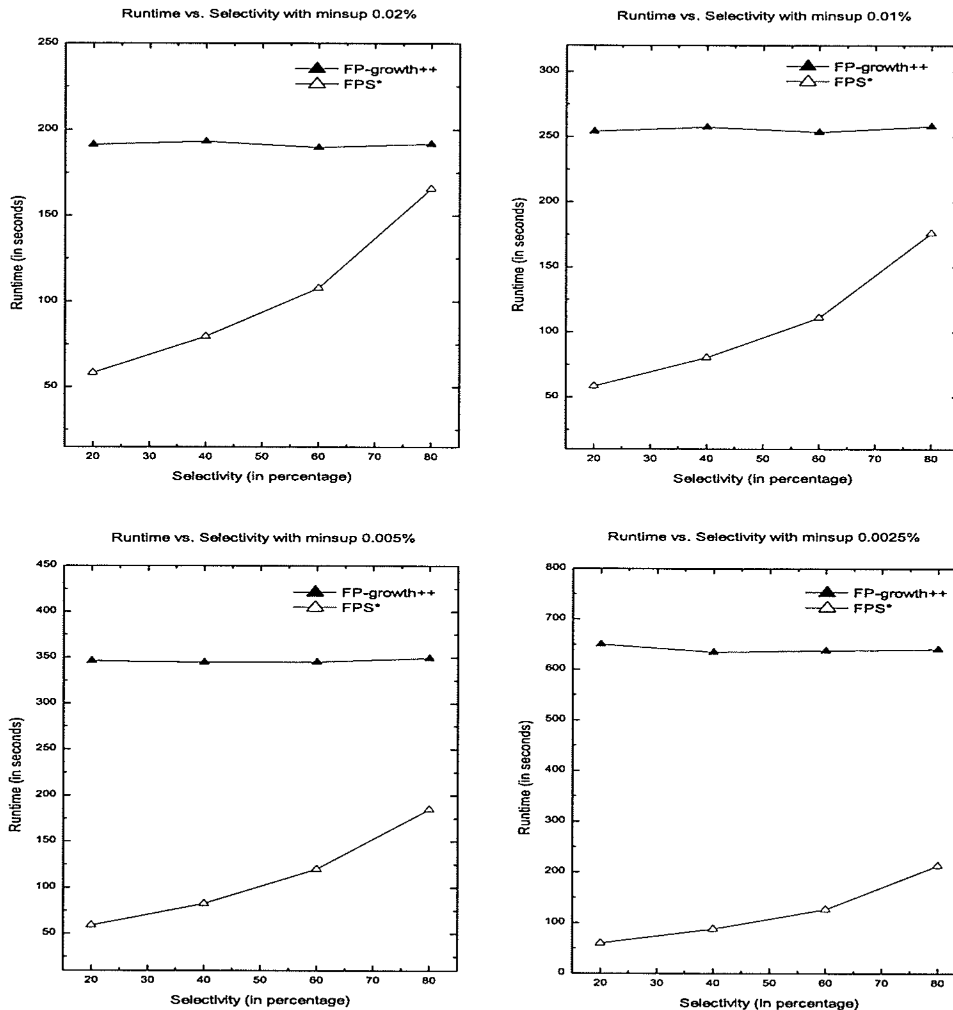

Figure 5.8: Results for disjunctions of SAM constraints with different selectivity (Experiment 5.8)

The *y-axis* of Figure 5.8 shows the runtime, and the *x-axis* shows selectivity. When selectivity increases, the runtime of FPS^*_{nSAM} increases while that of FP-growth++ changes little. In terms of speedup, it is more beneficial to use the FPS^*_{nSAM} algorithm than FP-growth++.

Our FPS^*_{nSAM} algorithm is sensitive to selectivity. A change of selectivity means the number of items mined gets changed, which then changes the number of valid frequent patterns. As we known, FPS^*_{nSAM} only mines valid items and only generates valid frequent patterns. Therefore, the performance of FPS^*_{nSAM} is sensitive to selectivity. FP-growth++ is not affected so much by selectivity, because each frequent pattern needs to be checked no matter which selectivity is chosen. By exploiting succinctness, FPS^*_{nSAM} outperforms FP-growth++.

In addition to testing on the IBM synthetic data, we also tested our FPS^*_{nSAM} on the mushroom dataset (a real-life dataset) in Experiment 5.9. The mushroom dataset consists of 8124 records with the transaction length of 23 items and a domain of 119 items.

Experiment 5.9 In this experiment, we tested how the *selectivity* values affect the runtime of algorithms by using the mushroom dataset. We set the minsup to 1%, which is usually used for the mushroom dataset, and varied the selectivity from 20% to 100%.

We can observe that the results tested on the mushroom dataset are consistent with our experimental results on IBM datasets, and the speedup is better. When selectivity is 20%, the speedup of FPS^*_{nSAM} is more than 60. When selectivity is 100%,

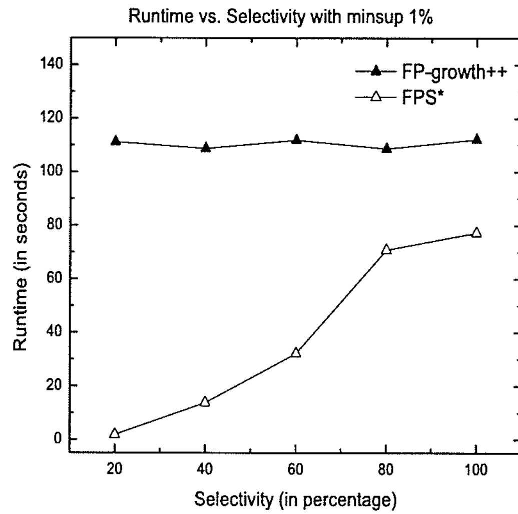

Figure 5.9: Results of FPS^*_{nSAM} on the mushroom dataset with different selectivity (Experiment 5.9)

Table 5.4: The number of frequent patterns for FP-growth++, and FPS^*_{nSAM} tested on the mushroom dataset (Experiment 5.9)

Level of itemsets	FP-growth++	FPS^*_{nSAM}
L_1	96	96
L_2	2435	1757
L_3	28597	17889
L_4	194233	112721
L_5	863271	474392
L_6	2712221	1419177
L_7	6334603	3161183
L_8	11364164	5425907
L_9	15979209	7344625
L_{10}	54188568	23813175

FPS^*_{nSAM} still outperforms FP-growth++. Therefore, our FPS^*_{nSAM} is particularly adapted to real-life (dense) datasets.

Table 5.4 showed the number of frequent patterns tested on the mushroom dataset, when selectivity was set to 100% and minsup was set to 1%. We can observe that FPS^*_{nSAM} avoids generating a great number of invalid frequent patterns. Therefore, the advantage of FPS^*_{nSAM} is obvious for handling dense datasets.

Experiment 5.10 In this experiment, we tested how the *number of constraints* affects the runtime of algorithms. We set the minsup to 0.005%, the selectivity of disjunctions of SAM constraints to 40%, the selectivity for each SAM constraint to 30% (where frequent patterns satisfying these SAM constraints may overlap—the percentage of overlap varied).

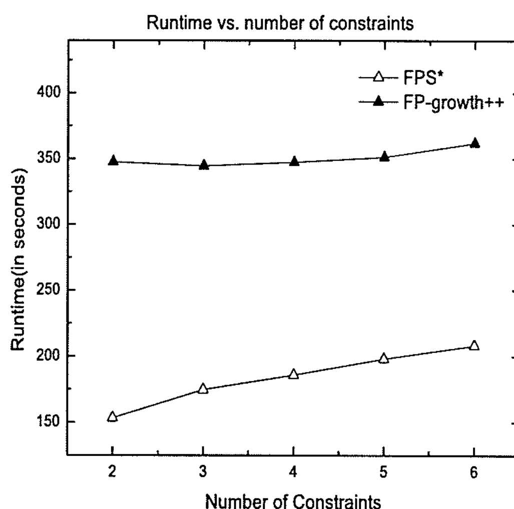

Figure 5.10: Results for changing the number of SAM constraints (Experiment 5.10)

The *y-axis* of Figure 5.10 shows the runtime, and the *x-axis* shows the number of SAM constraints. We can observe that the runtime of FPS^*_{nSAM} increases a little

bit with the increase of the number of constraints. In terms of speedup, it is more beneficial to use the FPS^*_{nSAM} algorithm than FP-growth++.

One would expect that the performance of FPS^*_{nSAM} is only affected by the number of frequent patterns, but not affected by the number of constraints. However, in order to avoid generating frequent patterns that satisfy more than one SAM constraint multiple times, we need to construct a constraint FP-tree for each SAM constraint. When the number of constraints increases, more constraint FP-trees are constructed. Fortunately, the runtime only increases slightly. Thus, FPS^*_{nSAM} still outperforms FP-growth++.

Moreover, the runtime increases also because more frequent patterns are generated. When we fix both the selectivity of disjunctions of SAM constraints and the selectivity of each SAM constraint (and vary the number of patterns that are overlapped), more frequent patterns are valid.

Table 5.5: The number of frequent patterns with different number of constraints (Experiment 5.10)

Level of frequent patterns	Number of constraints				
	2	3	4	5	6
L_1	384	384	384	384	384
L_2	25748	26646	26861	27063	27095
L_3	5658	6111	6292	6478	6487
L_4	724	845	869	897	901
L_5	148	174	180	190	190
L_6	28	28	29	31	31
L_7	3	3	3	3	3

Table 5.5 shows the number of valid frequent patterns with same selectivity of disjunctions of SAM constraints (40%), same selectivity of each SAM constraint (30%), and same minsup (0.005%). We can observe that, when the number of constraints

increases, the number of frequent patterns increases. It is also a reason why runtime increases as the the number of constraints increases.

Experiment 5.11 In this experiment, we tested scalability with the number of transactions. We fixed the minsup to 0.025%, selectivity to 20% and the number of SAM constraints to 4. The results in Figure 5.11 shows that mining with FPS^*_{nSAM} has linear scalability.

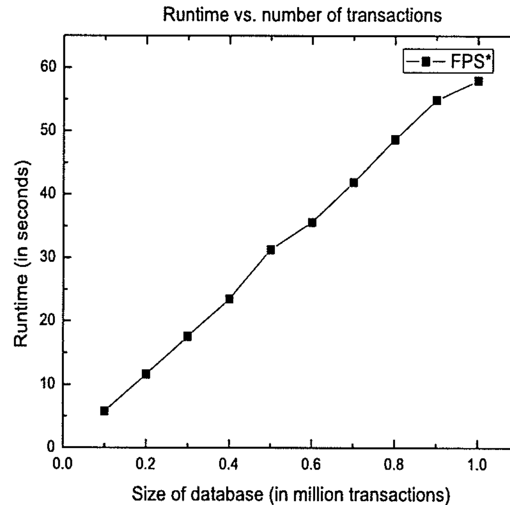

Figure 5.11: Scalability of FPS^*_{nSAM} (Experiment 5.11)

5.3 Experimental Results for Disjunctions of SUC Constraints

Similar to what we did for disjunctions of n SAM constraints, we set up experiments to test FPS^*_{nSUC}.

Experiment 5.12 In this experiment, we tested how the *minsup* values affect the runtime of algorithms. We set the number of constraints to 4, and varied the minsup for disjunctions of SUC constraints from 0.0025% to 0.025%. We showed the results for selectivity of disjunctions of SUC constraints equals to 20%, 40%, 60% and 80%.

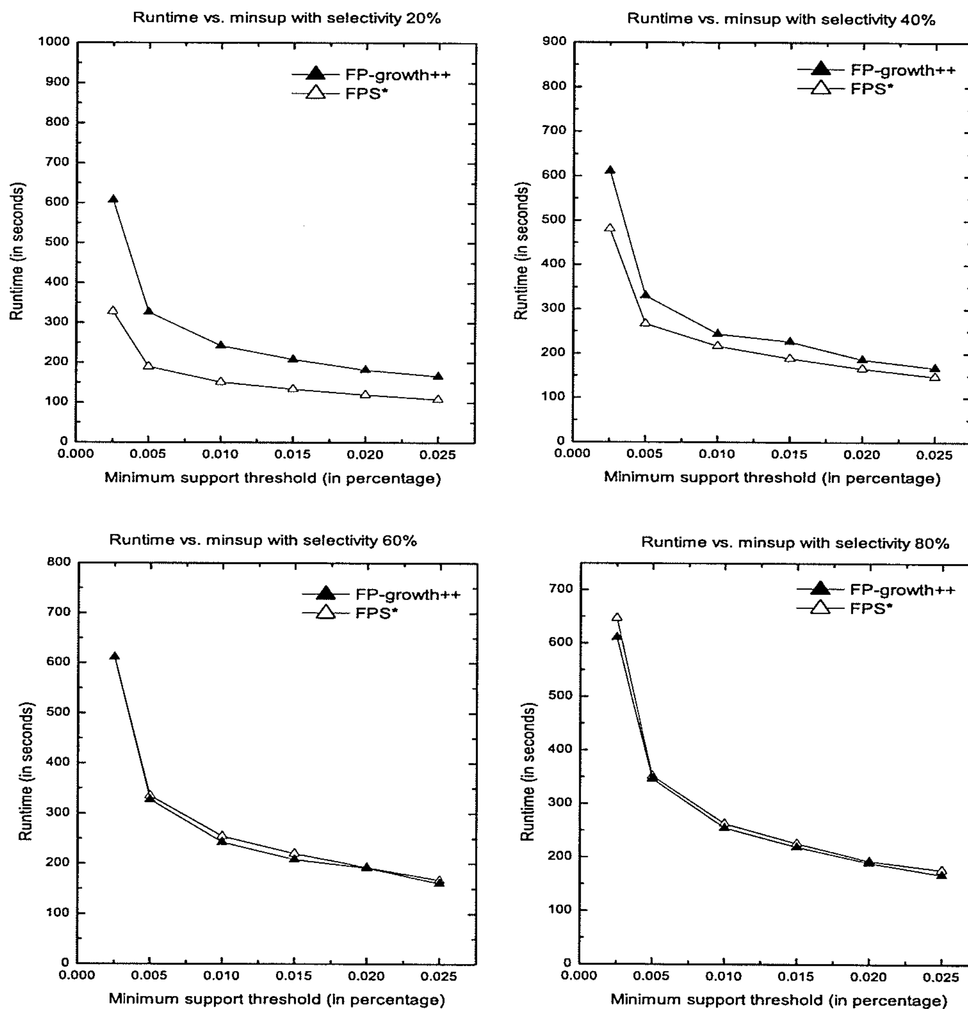

Figure 5.12: Results for disjunctions of SUC constraints with different minsup (Experiment 5.12)

Experiment 5.13 In this experiment, we tested how the *selectivity* values affect the runtime of algorithms. We set the number of constraints to 4 and the selectivity varied from 20% to 80%. We showed the results for minsup of disjunctions of SUC constraints equals to 0.02%, 0.01%, 0.005% and 0.0025%.

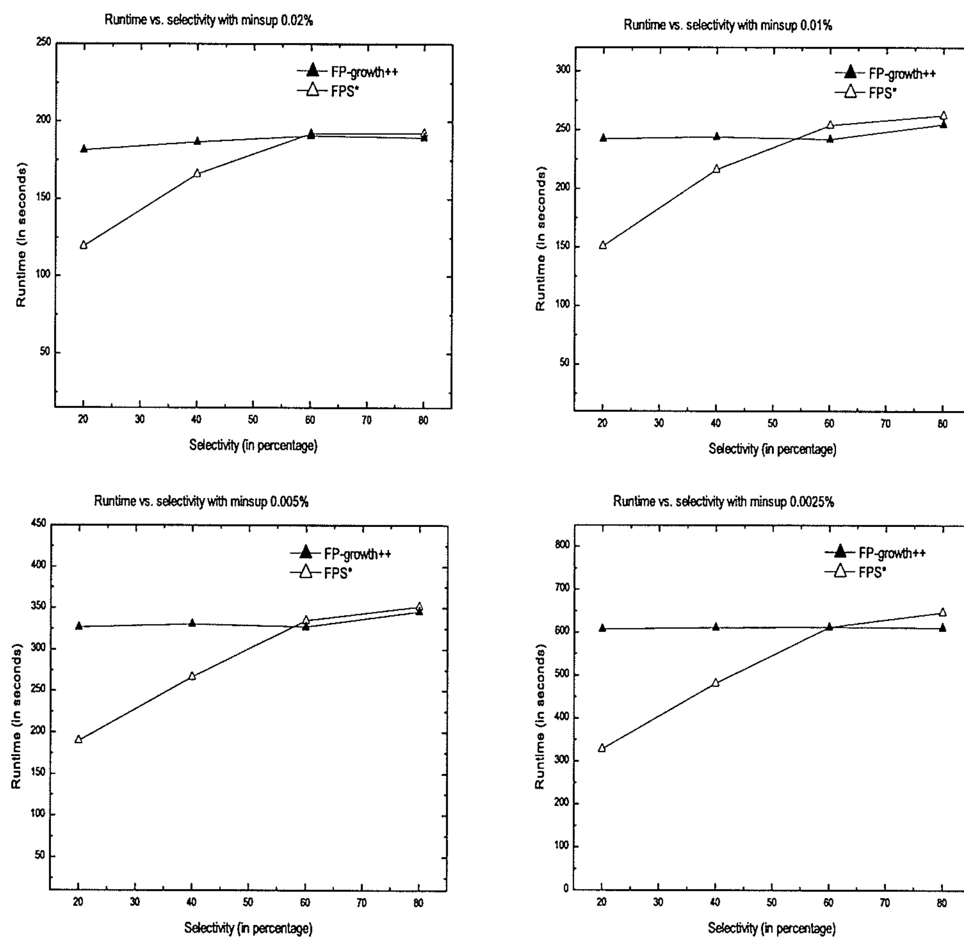

Figure 5.13: Results for disjunctions of SUC constraints with different selectivity (Experiment 5.13)

From Figure 5.12 and Figure 5.13, we observe that the performance of FPS^*_{nSUC} has the same trend as $FPS^*_{Negation}$. By analyzing those two algorithms, we find that the basic process steps of those two algorithms are similar. They both sort the items into a mandatory group and an optional group, and only generate valid frequent patterns based on the items from mandatory group. The only difference is the mandatory group for FPS^*_{nSUC} contains items that satisfy the disjunction of n SUC constraints while $FPS^*_{Negation}$ contains items that satisfy one SUC constraint. Therefore, the performance of FPS^*_{nSUC} shows the same trend as $FPS^*_{Negation}$. The analysis of the results of FPS^*_{nSUC} is also the same as the analysis of $FPS^*_{Negation}$ for a single SUC constraint.

Experiment 5.14 In this experiment, we tested how the *number of constraints* affects the runtime of algorithms. We set the minsup to 0.005%, the selectivity of disjunctions of SUC constraints to 40%, the selectivity for each SUC to 30% (the overlap of those SUC constraints varied).

The *y-axis* of Figure 5.14 shows the runtime, and the *x-axis* shows number of SUC constraints. We can see that both algorithms are not affected by the number of SUC constraints. In terms of speedup, it is more beneficial to use FPS^*_{nSUC} algorithm than FP-growth++.

By analyzing those two algorithms, we find that the change in the number of SUC constraints only affects the time for checking constraints. The checking time is only a small percentage of the runtime. Therefore, the number of constraints does not affect the runtime for FPS^*_{nSUC} and FP-growth++ too much.

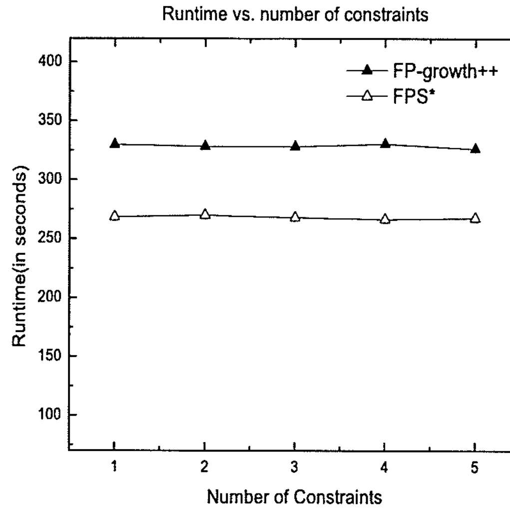

Figure 5.14: Results for changing the number of SUC constraints (Experiment 5.14)

Experiment 5.15 In this experiment, we tested scalability with the number of transactions. We fixed the minsup to 0.025%, selectivity to 20% and the number of SUC constraints to 4. The results in Figure 5.15 shows that mining with FPS^*_{nSUC} has linear scalability.

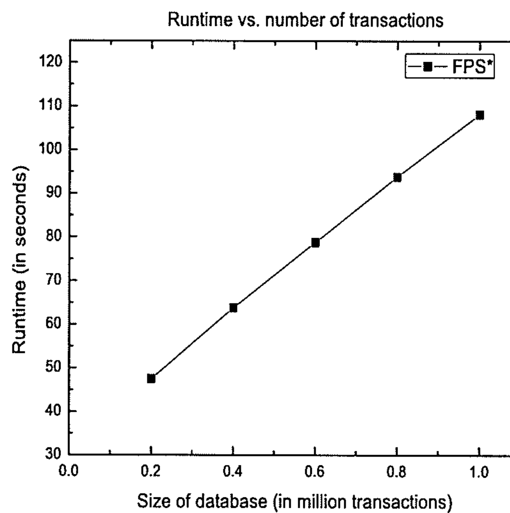

Figure 5.15: Scalability of FPS^*_{nSUC} (Experiment 5.15)

5.4 Experimental Results for Disjunctions of SUC Mixed with SAM Constraints

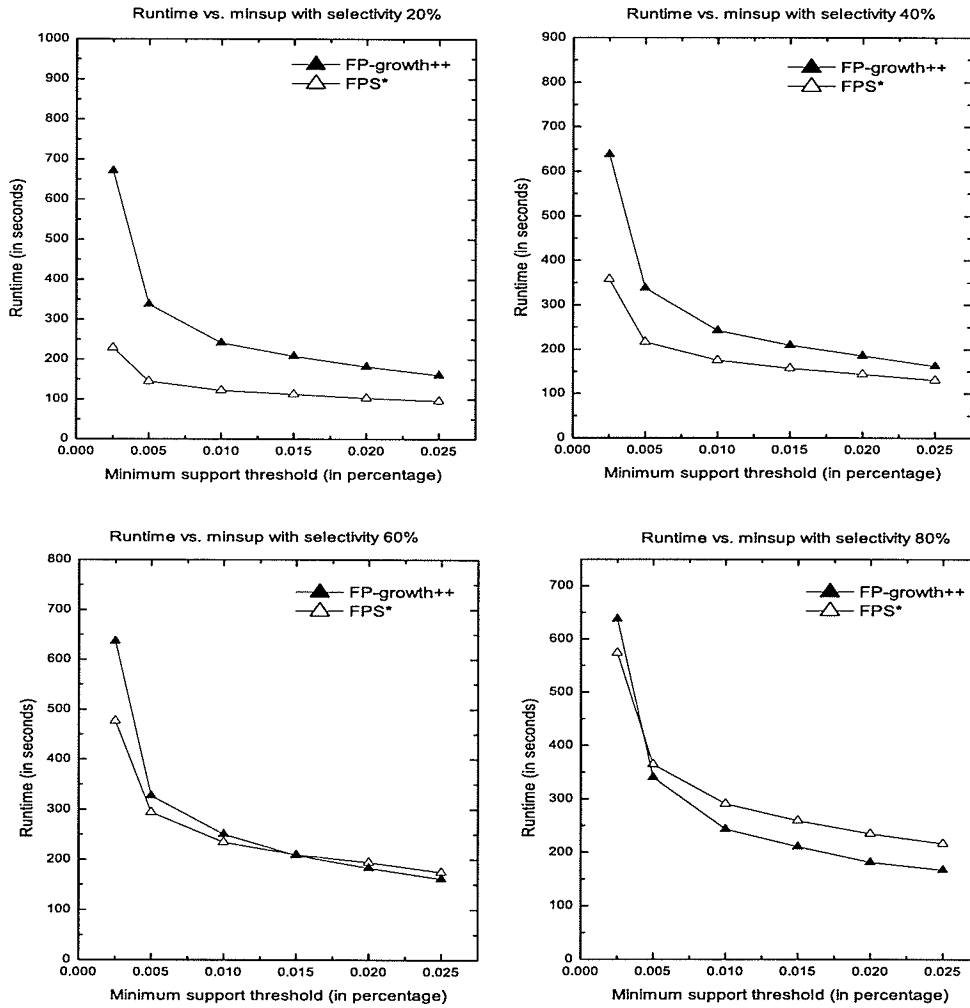

Figure 5.16: Results for disjunctions of mixed constraints with different minsup (Experiment 5.16)

FPS^{*}_{nMIX} is a combination of FPS^{*}_{nSUC} and FPS^{*}_{nSAM}. If the selectivity for

SAM constraints is set to 0, FPS^*_{nMIX} becomes FPS^*_{nSUC}. Similarly, if the selectivity for SUC constraints is set to 0, FPS^*_{nMIX} becomes FPS^*_{nSAM}. We set up four experiments to test FPS^*_{nMIX}.

Experiment 5.16 In this experiment, we tested how the *minsup* values affect the runtime of algorithms. We set the number of constraints to 6 (3 SUC constraints mixed with 3 SAM constraints), and varied the minsup for disjunctions of SUC mixed with SAM constraints from 0.0025% to 0.025%. We showed the results for selectivity of disjunctions of SUC mixed with SAM constraints equals to 20%, 40%, 60% and 80%.

Experiment 5.17 In this experiment, we tested how the *selectivity* values affect the runtime of algorithms. We set the number of constraints to 6, and varied the selectivity from 20% to 80%. We showed the results for minsup of disjunctions of SUC mixed with SAM constraints equals to 0.02%, 0.01%, 0.005% and 0.0025%.

From Figures 5.16 and 5.17, we can observe that the performance of FPS^*_{nMIX} is a compromise of FPS^*_{nSUC} and FPS^*_{nSAM}. We can see that when minsup is low, FPS^*_{nMIX} outperforms FP-growth++ (even with a high selectivity). However, in the case that selectivity and minsup are both high, the performance of FPS^*_{nMIX} is close or worse than FP-growth++. The reason is the high selectivity of SUC and SAM constraints leads to a large number of valid frequent patterns, and the extra processing time of FPS^*_{nMIX} (e.g., constraint tree construction time for SAM constraints) is longer than the time saved by avoiding the generation of invalid frequent patterns.

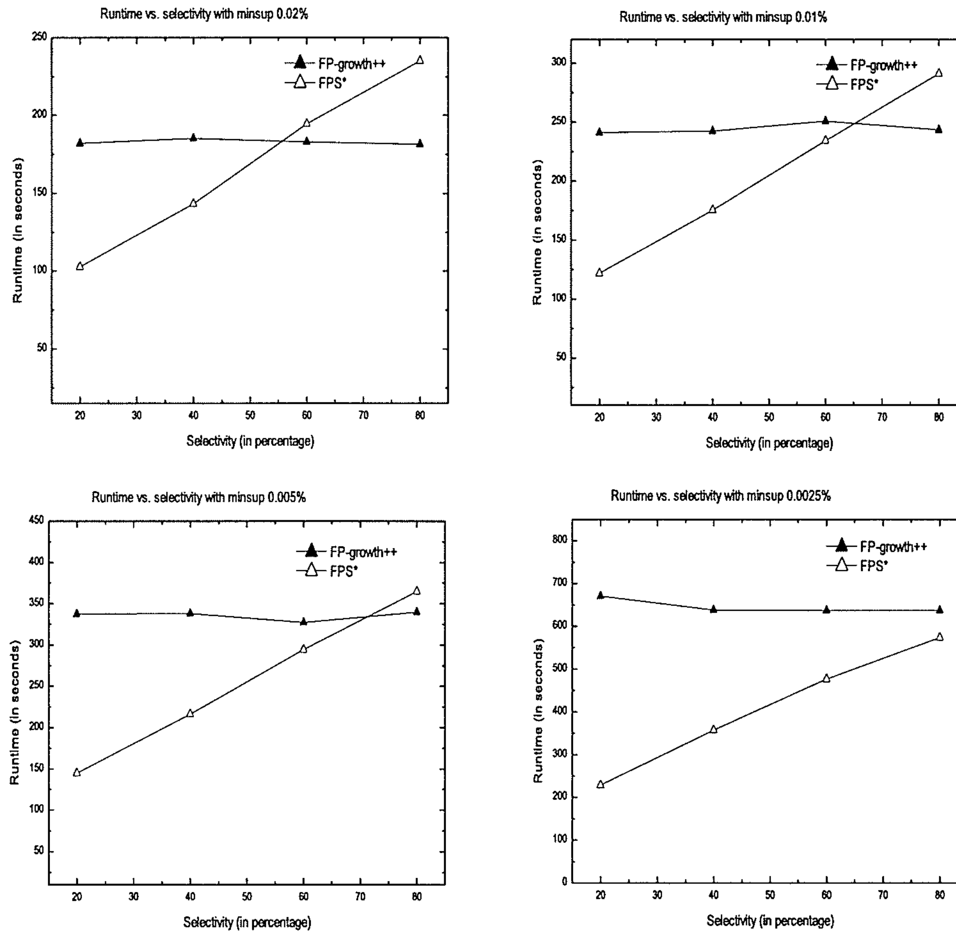

Figure 5.17: Results for disjunctions of mixed constraints with different selectivity (Experiment 5.17)

Experiment 5.18 In this experiment, we tested how the *number of constraints* affects the runtime of algorithms. We set the minsup to 0.005%, the selectivity of disjunctions of SUC mixed with SAM constraint to 40%, and the selectivity of each SUC and SAM constraints to 20% (the overlap of those SUC and SAM constraints varied).

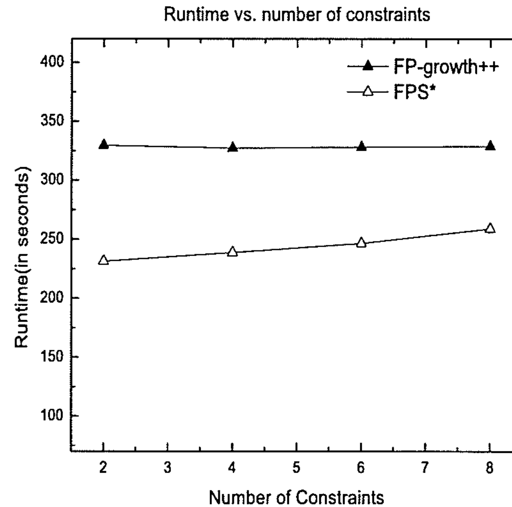

Figure 5.18: Results for changing the number of mixed constraints (Experiment 5.18)

The *y-axis* of Figure 5.18 shows the runtime, and the *x-axis* shows number of constraints. We can observe that the performance of FPS^*_{nMIX} is a compromise of FPS^*_{nSUC} and FPS^*_{nSAM}. The runtime increases slightly with the increase of the number of constraints.

Experiment 5.19 In this experiment, we tested scalability with the number of transactions. We fixed the minsup to 0.025%, selectivity to 20% and the number of constraints to 6. The results in Figure 5.19 shows that mining with FPS^*_{nMIX} has linear scalability.

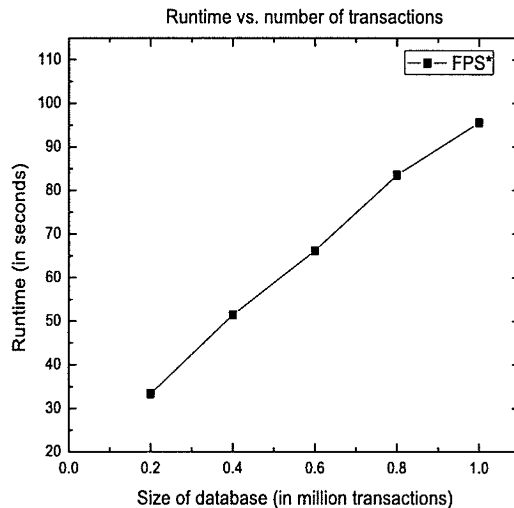

Figure 5.19: Scalability of FPS^*_{nMIX} (Experiment 5.19)

5.5 Summary

In this chapter, we showed the experimental results for our FPS* algorithm. The experimental results verified the performance improvements brought by the exploitation of succinctness in FPS*.

We tested how *minsup* and *selectivity* affect the performance of FPS*. We also tested how the *number of constraints* affects the performance of FPS^*_{nSAM}, FPS^*_{nSUC} and FPS^*_{nMix}, which are components of FPS*.

We find that FPS* for handing SAM constraints ($FPS^*_{Negation}$ for handling SAM constraints and FPS^*_{nSAM}) has obvious improvement: the use of succinctness avoids generating a great number of invalid frequent patterns. Even when selectivity is high, FPS* outperforms FP-growth++. The gap increases as the minsup becomes smaller.

Similarly, FPS* for handling SUC constraints ($FPS^*_{Negation}$ for handling SUC constraints and FPS^*_{nSUC}) and disjunctions of SUC mixed with SAM constraints (FPS^*_{nMIX}) performs well when minsup is low (0.0025%) or selectivity is low (lower than or equal to 60%). We also find that the runtime of FPS^*_{nSAM} and FPS^*_{nMIX} increases slightly with the increase of the number of constraints, while the runtime of FPS^*_{nSUC} is not affected.

In addition, we tested the scalability of FPS*, and find that FPS* has linear scalability.

In summary, FPS* effectively utilizes the succinctness of negation and disjunction of succinct constraints and is particularly adapted to low selectivity conditions.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Frequent pattern mining is an important task in data mining. In recent years, the importance of constraint-based frequent pattern mining has been highlighted. There are several classes of "semantic" constraints. One of them is called succinct constraints. Among all classes of constraints, a majority of them are succinct. For constraints that are not succinct, many of them can be induced into weaker succinct constraints. Moreover, succinct constraints have some nice properties that can be used to optimize the mining process. For example, frequent patterns satisfying succinct constraints can be efficiently generated by the member generating function (MGF). The MGFs are framework independent; they can be applied in different frameworks, including the Apriori and Frequent Pattern tree (FP-tree) frameworks. The properties of succinct constraints and conjunctions of succinct constraints have been exploited by Lakshmanan et al. [LLN03]. We learnt that atomic succinct constraints can be

divided into SAM (succinct and anti-monotone) constraints, SUC (succinct non-anti-monotone) constraints and superset constraint (one particular SUC constraint with a special form of MGF) according to their different forms of MGFs. We also learnt that conjunctions of atomic succinct constraints are succinct.

In this thesis research, we explored the properties of negations and disjunctions of succinct constraints to complete the exploration of properties for any Boolean combinations of succinct constraints. We proved that negations and disjunctions of atomic succinct constraints are succinct. Because of the succinctness of conjunctions, negations and disjunctions of atomic succinct constraints, Boolean combinations of succinct constraints are also succinct. Therefore, from the theoretical aspect, any succinct constraints connected by conjunctions, negations and disjunctions are succinct. Moreover, we provided specific MGFs for different kind of negations and disjunctions of succinct constraints. The MGFs for negation of atomic succinct constraints and disjunctions of multiple SAM, SUC and SUC mixed with SAM constraints are in either a basic MGF form, or in a form of a union of disjoint sets. Therefore, we can directly generate precisely all and only those frequent patterns satisfying negations of atomic succinct constraints and disjunctions of n SAM, SUC or SUC mixed with SAM constraints, avoiding generating and excluding frequent patterns not satisfying the constraints.

Based on the MGFs we got for negations of atomic succinct constraints and disjunctions of multiple SAM, SUC and SUC mixed with SAM constraints, we designed and implemented an FPS* algorithm, which pushes the succinctness inside the mining process in the FP-tree based framework. Our FPS* algorithm, which is an extension

of the existing FPS algorithm, inherits the functions to handle conjunctions of succinct constraints from FPS, and includes the new functions to handle the negations of succinct constraints and disjunctions of multiple SAM, SUC, or SUC mixed with SAM constraints, called "$FPS^*_{Negation}$", "FPS^*_{nSAM}", "FPS^*_{nSUC}" and "FPS^*_{nMIX}" separately.

The experimental results verified the performance improvements by exploiting succinctness in FPS*. We tested how *minsup*, and *selectivity* affect the performance of FPS*. For FPS^*_{nSAM}, FPS^*_{nSUC} and FPS^*_{nMix}, we also tested how the *number of constraints* affects the performance. We found that FPS* for handing SAM constraints (negation of a SAM constraint and disjunctions of SAM constraints) has obvious improvement, for the succinctness avoids generating a great number of invalid frequent patterns. Even when selectivity is high, FPS* outperforms FP-growth++. The gap increases with the minsup becoming smaller. FPS* for handling SUC constraints (negation of a SUC constraint and disjunctions of SUC constraints) and disjunctions of SUC mixed with SAM constraints performs well when minsup is low (0.0025%) or selectivity is low (lower than or equal to 60%). We also found that the runtime of FPS^*_{nSAM} and FPS^*_{nMix} increases slightly with the increase of the number of constraints, while the runtime of FPS^*_{nSUC} is not affected. In addition, we tested the scalability of FPS*, and found that FPS* has linear scalability. In summary, FPS* effectively utilizes the succinctness of negation and disjunction of succinct constraints and is particularly adapted to low selectivity conditions.

6.2 Future Work

In ongoing work, we will try to optimize the MGFs for disjunctions of superset constraints and any Boolean combinations containing superset constraints (e.g., superset constraints mixed with SAM or SUC constraints). Once the optimized MGFs are found, the extension of FPS* will be developed correspondingly.

We also want to explore how parallel computing can help with the FPS* algorithm. We observed that constraint-tree construction and mining time occupies a high percent of the whole running time for the FPS^*_{nSAM} algorithm. Constraint tree construction and mining for each SAM constraint is an independent process from each other. Therefore, parallel computing may apply to the FPS* algorithm and improve the efficiency of the algorithm.

In addition, we would like to explore the possibility to handle dynamic changes of Boolean combinations of succinct constraints. Dynamic changes of atomic succinct constraints for frequent pattern mining have been explored by Lakshmanan et al. [LLN03] and Leung [Leu04]. However, when users express their focuses by using Boolean combinations of succinct constraints, they may also want to change the constraints for having an influence on subsequent computation. Therefore, we are interested in investigating handling dynamic changes of Boolean combinations of succinct constraints.

Bibliography

[AIS93] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases", In *Proc. SIGMOD 1993*, pp. 207-216.

[AM+96] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo, "Fast Discovery of Association Rules", In *Advances in Knowledge Discovery and Data Mining*, AAA/MIT Press, 1996, ch. 12.

[AS94] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", In *Proc. VLDB 1994*, pp. 487-499.

[BG+03] C. Bucila, J.E. Gehrke, D. Kifer, and W. White, "DualMiner: A Dual-Pruning Algorithm for Itemsets with Constraints", *Data Mining and Knowledge Discovery*, Vol. 7, Issue 4, July 2003, pp. 241-272.

[BM98] C.L. Blake and C.J. Merz, UCI Repository of Machine Learning Databases. Department of Information and Computer Science, University of California, Irvine, CA, USA, 1998. `www.ics.uci.edu/~mlearn/MLRepository.html`

[GWK04] K. Gade, J. Wang, and G. Karypis, "Efficient Closed Pattern Mining in the Presence of Tough Block Constraints", In *Proc. SIGKDD 2004*, pp. 138-147.

[GZ03a] B. Goethals and M.J. Zaki (eds), *Proc. FIMI Workshop 2003.*

[GZ03b] G. Grahne and J. Zhu, "Efficiently Using Prefix-trees in Mining Frequent Itemsets", In *Proc. FIMI Workshop 2003.*

[HGN00] J. Hipp, U. Guntzer, and G. Nakhaeizadeh, "Algorithms for Association Rule Mining - A General Survey and Comparison", *ACM SIGKDD Explorations*, Volume 2, Issue 1, June 2000, pp. 58-64.

[HPY00] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation", In *Proc. SIGMOD 2000*, pp. 1-12.

[KG+03] D. Kifer, J.E. Gehrke, C. Bucila, and W. White, "How to Quickly Find a Witness", In *Proc. PODS 2003*, pp. 272-283.

[Leu04] C.K.-S. Leung, "Interactive Constrained Frequent-Pattern Mining System", In *Proc. IDEAS 2004*, pp. 49-58.

[LLN02] C.K.-S. Leung, L.V.S. Lakshmanan, and R.T. Ng, "Exploiting Succinct Constraints using FP-trees", *SIGKDD Explorations*, Volume 4, Issue 1, June 2002, pp. 40-49.

[LLN03] L.V.S. Lakshmanan, C.K.-S. Leung, and R.T. Ng, "Efficient Dynamic Mining of Constrained Frequent Sets", *ACM TODS*, Volume 28, Issue 4, Dec. 2003, pp. 337-389.

[LN+99] L.V.S. Lakshmanan, R.T. Ng, J. Han, and A. Pang, "Optimization of Constrained Frequent Set Queries with 2-variable Constraints", In *Proc. SIGMOD 1999*, pp. 157-168.

[LP+02] J. Liu, Y. Pan, K. Wang, and J. Han, "Mining Frequent Item Sets by Opportunistic Projection", In *Proc. SIGKDD 2002*, pp. 229-238.

[NL+98] R.T. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang, "Exploratory Mining and Pruning Optimizations of Constrained Association Rules", In *Proc. SIGMOD 1998*, pp.13-24.

[PH02] J. Pei and J. Han, "Constrained Frequent Pattern Mining: A Pattern-Growth View", *SIGKDD Explorations*, Volume 4, Issue 1, June 2002, pp. 31 - 39.

[PHL01] J. Pei, J. Han, and L.V.S. Lakshmanan, "Mining Frequent Itemsets with Convertible Constraints", In *Proc. ICDE 2001*, pp. 433-442.

[PZ03] A. Pietracaprina and D. Zandolin, "Mining Frequent Itemsets using Patricia Tries", In *Proc. FIMI Workshop 2003.*

[WH+04] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi, "Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining", In *Proc. PAKDD 2004*, pp. 441-451.