

Mining of Frequent Sets using Pruning, Based on Background Knowledge

Anke Jäger

promotor :
Prof. dr. Bart KUIJPERS

co-promotor :
dr. Vania BOGORNY

Acknowledgements

I want to thank every single person for their contribution to the realization of this thesis.

First, I am thankful to my promotor, prof. dr. Bart Kuijpers, for giving me the chance to write this thesis. I really enjoyed doing research for this thesis and learned a lot from it.

A special thanks goes to my two wonderful advisors, Bart Moelans and Vania Bogorny. They put so much time and effort in the development of my work and kept telling me so many times this was great work. I am really proud I could work together with the two of you!

I also want to thank my parents and friends, for the support and encouragement they have given me during my career as a student and especially during the writing of this thesis. Thank you for always believing in me!!

Abstract

Association rule mining is a technique to find useful patterns and associations in transactional databases. There have been developed a lot of algorithms for this purpose, among which are also APriori and FP-Growth. Though you can find new patterns and associations, the technique of association rule mining usually results in too many rules through which the user has to find those that are interesting to him/her. Among this large amount of association rules, there are also ones that are non-interesting, simply because they are known a priori, like for example *isPregnant* \rightarrow *isFemale*.

Since these rules are not useful, their frequent itemsets also do not need to be generated. This method was already described in [Bog06], where the idea of knowledge constraints was applied to APriori. Because the FP-Growth algorithm is a lot faster than APriori, it seems logical to also apply this method to FP-Growth.

The only drawback for this new algorithm (and also for APriori-KC) was that there were removed too many rules through this elimination of dependences. Therefore, we developed a method to recover the rules that were lost, without too much time going lost.

The main advantage of this new algorithm is that it reduces the number of frequent itemsets significantly and thus also the total number of association rules that is generated.

Samenvatting

Tegenwoordig maakt men zeer veel gebruik van databases. Omdat de hoeveelheid data in die databases almaar blijft groeien en groeien is het voor een mens bijna onmogelijk om op het eerste zicht patronen uit die data te gaan halen. Om deze patronen te verkrijgen, gaat men dan het KDD of *Knowledge Discovery in Databases* toepassen.

In deze thesis hebben we ons vooral toegespitst op een specifiek onderdeel binnen het KDD, namelijk het *datamining proces*. Binnen datamining zijn er nog eens verschillende technieken, alnaargelang het resultaat men wil verkrijgen. Zo hebben we associatieanalyse, clusteren, classificeren, detectie van uitschieters en evolutieanalyse. In deze thesis gaan we ons enkel focussen op de associatieanalyse.

Bij associatieanalyse is het de bedoeling om patronen te vinden door middel van zogenaamde *associatieregels* die tonen dat bepaalde attributwaarden geregeld samen voorkomen in een dataset [AIS93]. Met dit doel voor ogen, zijn er het laatste decennium een heel aantal algoritmes ontwikkeld, zoals bijvoorbeeld APriori [AS94] en FP-Growth [HPYM04].

Vooraleer we beginnen met het zoeken van de associatieregels, dient er eerst op zoek gegaan te worden naar de frequente itemsets. Dit zijn items die frequent samen voorkomen. Het APriori algoritme baseert zich vooral op het feit dat wanneer een bepaalde itemset frequent is, dan ook alle deelverzamelingen van items binnen deze itemset frequent zijn. Dit algoritme zoekt eerst alle frequente 1-itemsets. Daarna zoekt het alle frequente n -itemsets door de frequente $(n - 1)$ -itemsets samen te voegen. Dit totdat er geen nieuwe frequente itemsets meer gevonden kunnen worden.

FP-Growth daarentegen gaat heel anders te werk. Bij dit algoritme gaat men de database scannen, en dan zo een boom opbouwen van de data die zich in de database bevindt. De resulterende boom wordt ook wel een *FP-tree* genoemd. Om de frequente itemsets te vinden gaat men de *conditionele FP-trees* bekijken van ieder item/knoop. Deze conditionele FP-tree vertelt met welke andere items het item voorkomt in de database en hoe vaak die combinatie voorkomt. Zo verkrijgt men dan de frequente itemsets. Dit algoritme staat er trouwens om gekend dat het een stuk sneller is dan APriori, iets wat ons nog van pas kan komen.

Als we de frequente itemsets gevonden hebben met behulp van APriori of FP-Growth, kunnen we hieruit de associatieregels afleiden. Voor elke frequente itemset I gaan we alle niet-lege deelverzamelingen J nemen en dan zo kijken of de regel $J \rightarrow (I - J)$ aan een bepaalde voorwaarde voldoet, namelijk de minimale *confidence*. Dit betekent concreet het percentage van itemsets waarin J voorkomt, die ook $(I - J)$ bevatten.

Associatieanalyse heeft echter één groot probleem: het aantal regels. De gebruiker krijgt zodanig veel associatieregels terug na het uitvoeren van het algoritme, dat het bijna is als zoeken naar een speld in een hooiberg om de associatieregels te zoeken die voor hem/haar interessant zijn. Binnen deze enorme hoeveelheid associatieregels zitten er dan soms ook nog eens tussen die totaal nutteloos zijn omdat ze al op voorhand geweten zijn. Een voorbeeld van zulk een regel is *isZwanger* \rightarrow *isVrouw*. Deze regel geldt altijd vermits een zwanger persoon enkel en alleen maar vrouwelijk kan zijn. Om zo het totale aantal regels te reduceren, gaan we voorkomen dat deze oninteressante regels gegenereerd kunnen worden, door deze *afhankelijkheden* te verwijderen.

Voor dit doel werd er reeds een nieuw algoritme ontwikkeld, namelijk APriori-KC [Bog06]. In plaats van het gewone APriori algoritme uit te laten voeren en daarna alle regels te gaan verwijderen die een afhankelijkheid bevatten, gaat deze methode *voorkomen* dat deze regels gegenereerd worden. Hierbij worden frequente itemsets verwijderd van het moment dat ze gegenereerd worden en een afhankelijkheid blijken te bevatten. En vermits er geen associatieregels gevormd kunnen worden van frequente itemsets die er niet zijn, kunnen er ook geen oninteressante regels voorkomen in het resultaat.

Deze methode van het verwijderen van gekende afhankelijkheden was hier nu enkel toegepast op het APriori algoritme. Maar het is bekend dat FP-Growth qua performantie een stuk beter scoort dan APriori. Daarom gaan we dezelfde methode als bij APriori-KC toepassen om zo dan het FP-Growth-KC algoritme te verkrijgen. De pseudo-code van dit nieuwe algoritme staat in Listing 1, waarbij Φ de verzameling van alle afhankelijkheden voorstelt.

Bij dit algoritme gaan we dan kijken of de boom bestaat uit een enkelvoudig pad. Zo ja, dan kan de boom niet gesnoeid worden, maar dan gaan we gewoon alle mogelijke combinaties van knopen in dat pad af, zoals bij het originele FP-Growth algoritme. Enkel wanneer zulk een combinatie geen afhankelijkheid bevat, wordt hij opgenomen als een frequente itemset.

Wanneer de boom uit meerdere paden bestaat, gaan we anders te werk. Hiervoor zijn de regels 10 tot en met 12 toegevoegd aan het oorspronkelijke algoritme.

We gaan voor iedere $\beta = a_i \cup \alpha$ (waarbij a_i een knoop is in de header van de boom) een conditionele FP-tree opstellen. In het gewone FP-Growth algoritme gaan we daartoe β

Listing 1: FP-Growth-KC($Tree; \alpha$)

```

1 if( $Tree$  bevat 1 enkel pad  $P$ )
2   voor iedere combinatie  $\beta$  van de knopen in  $P$ 
3     if( $\beta$  bevat een afhankelijkheid van  $\Phi$ )
4       genereer een patroon  $\beta \cup \alpha$  met support
5         = minimum support van de knopen in  $\beta$ ;
6 else
7   voor iedere  $a_i$  in de header van  $Tree$ {
8      $\beta = a_i \cup \alpha$  met support =  $a_i$ .support;
9      $Tree_\beta =$  conditionele FP-tree van  $\beta$ ;
10    voor iedere knoop  $b_i$  in  $Tree_\beta$ 
11      if( $b_i \cup \beta$  bevat afhankelijkheid van  $\Phi$ )
12        verwijder  $b_i$  uit  $Tree_\beta$ ;
13      if( $Tree_\beta$  bestaat)
14        FP-Growth-KC( $Tree_\beta, \beta$ );
15    }
16  }
```

zelf verwijderen uit de prefix paden van β en daarnaast ook de knopen die geen minimum support meer halen. Vermits uit die conditionele FP-trees de frequente itemsets – en dus ook later de associatieregels – gevormd worden, is het belangrijk om er in dit stadium reeds voor te zorgen dat er geen frequente itemsets met afhankelijkheden gegenereerd worden. Daarom gaan we de conditionele FP-tree van β – genaamd $Tree_\beta$ – nog wat verder snoeien. We gaan daarvoor voor iedere knoop b_i in $Tree_\beta$ controleren of die in combinatie met β geen afhankelijkheid gaat vormen. Indien blijkt dat $b_i \cup \beta$ daadwerkelijk een afhankelijkheid uit Φ bevat, gaat de bewuste knoop b_i uit de conditionele FP-tree verwijderd worden. Dit resulteert dan uiteindelijk in een conditionele FP-tree die geen enkele oninteressante frequente itemset meer zal voortbrengen.

We zien nu duidelijk dat er geen oninteressante associatieregels gevormd worden, maar er is nog een probleem: door het verwijderen van die afhankelijkheden gaan er ook bepaalde associatieregels die *wel* interessant zouden kunnen zijn, niet gegenereerd worden. Een voorbeeld hiervan is het verwijderen van frequente itemsets die de items *isZwanger* en *isVrouw* bevatten omdat de regel

$$isZwanger \rightarrow isVrouw$$

niet interessant is. Daardoor krijgen we ook geen regels als

$$isZwanger \wedge isBlond \rightarrow isVrouw$$

die ook nutteloos zijn, vermits het aspect van blond zijn hier geen rol speelt. Echter, we kunnen ook geen regels meer krijgen zoals

$$isVrouw \wedge isBlond \rightarrow isZwanger,$$

een regel die wel interessant zou kunnen zijn omdat toevallig een deel van de blonde vrouwen in het geteste gebied zwanger blijkt te zijn.

Deze regels mogen dan wel verloren zijn gegaan, we kunnen ze nog altijd terug halen. We weten al dat de associatieregel $A \rightarrow B$ een confidence heeft van 100% vermits $\{A, B\}$ een afhankelijkheid is. Daarom is het ook niet nodig om regels als $AC \rightarrow BD$ terug te halen, omdat ze toch geen extra waarde toevoegen.

Maar de associatieregel $B \rightarrow A$ is geen afhankelijkheid en heeft dus een confidence die lager ligt dan of juist gelijk is aan 100%. Daarom dienen we dus alle regels terug te halen die B in het antecedent hebben en A in het rechterdeel.

Het terughalen van deze regels mag dan wel iets meer tijd kosten dan het gewone FP-Growth-KC algoritme, toch is het nog altijd een heel stuk sneller dan het FP-Growth algoritme. Het tijdsverschil tussen de gewone FP-Growth-KC en de versie met het terughalen van bepaalde regels speelt hier ook niet zo een grote rol, vermits het in dit geval belangrijker is om *alle* regels die interessant zijn als resultaat terug te krijgen. FP-Growth-KC met het terughalen van regels is equivalent met het verwijderen van oninteressante regels na het uitvoeren van FP-Growth.

Naast het ontwikkelen van dit algoritme, hebben we ook de gratis open source data mining toolkit Weka [WF05] uitgebreid met een aantal extra algoritmes. Zo hebben we onze eigen versie van het APriori algoritme geïmplementeerd en daarnaast ook nog APriori-KC en APriori-prune. Deze laatste gaat na het genereren van de associatieregels de oninteressante regels verwijderen. Deze methode is enkel geïmplementeerd voor het uitvoeren van experimenten.

Naast APriori-gebaseerde algoritmes hebben we Weka ook uitgebreid met een aantal algoritmes die op FP-Growth gebaseerd zijn, meer bepaald diegene die we hierboven vermeld hebben. Hiertoe behoren FP-Growth zelf, FP-Growth-KC, FP-Growth-KC met het terughalen van regels en FP-Growth-prune, waarbij de laatste – net als APriori-prune – enkel geïmplementeerd werd omwille van experimentele doeleinden.

Met deze uitgebreide versie van Weka zijn we dan aan de slag gegaan voor de experimenten, om de efficiëntie van de nieuwe algoritmes te controleren. De grafieken van deze experimenten zijn te vinden in Hoofdstuk 6. Bij deze experimenten zijn onze vermoedens over de performance van FP-Growth-KC bevestigd. Het blijkt stukken sneller te zijn dan FP-Growth en – vooral – dan APriori-KC. Ook is gebleken bij deze experimenten dat het terughalen van associatieregels bij FP-Growth-KC niet zo een groot tijdsverlies met zich meebrengt. Dit zeker niet in vergelijking met het aposteriori verwijderen van oninteressante associatieregels na het uitvoeren van FP-Growth.

Contents

1	Introduction and Motivation	1
1.1	Introduction	1
1.2	Motivation	4
1.3	Outline	4
2	Association Analysis	5
2.1	Frequent Itemset Generation	5
2.1.1	APriori	6
2.1.2	FP-Growth	13
2.2	Rule Generation	21
2.2.1	Generating Frequent Association Rules	22
3	Knowledge Constraints	25
3.1	Well-known dependences	25
3.2	Removing non-interesting rules	26
3.2.1	How?	26
3.2.2	No loss of information?	26
3.3	APriori-KC	27
3.3.1	The Algorithm	27
3.3.2	Example	29
3.3.3	Generalization	31
4	FP-Growth-KC	37
4.1	The Concept	37
4.2	The Algorithm	37
4.3	Example	39
4.4	Rule recovery	42
4.4.1	Problem	42
4.4.2	Solution	43

5	Implementation in Weka	47
5.1	Format of the dependences	49
5.2	Implementation of APriori-based methods	49
5.3	Implementation of FP-Growth-based methods	50
6	Experiments and Evaluation	53
6.1	Experiments with the Mushroom dataset	53
6.1.1	Evaluating APriori-KC and FP-Growth-KC for single dependence elimination	53
6.1.2	Evaluating APriori-KC and FP-Growth-KC for the elimination of 2 dependences	55
6.1.3	Evaluating performance of rule recovery with FP-Growth-KC	57
6.1.4	Evaluating FP-Growth with pruning afterwards and FP-Growth-KC with rule recovery	57
6.2	Experiments with the Geographic dataset	58
6.2.1	Evaluating the frequent set generation of the FP-Growth-KC method	59
6.2.2	Evaluating the rule generation and rule recovery of the FP-Growth-KC method	59
7	Conclusions	62

List of Figures

1.1	Process of knowledge discovery	2
2.1	Generation of candidate itemsets and frequent itemset, with minimum support of 50%	8
2.2	Example of a hash-tree	11
2.3	Hashing a transaction at the root node of a hash-tree	11
2.4	Construction of an FP-tree	17
2.5	An FP-tree representation for the dataset shown in Table 2.4 with a different item ordering scheme	18
2.6	FP-Growth algorithm for finding frequent itemsets ending in D	19
2.7	FP-growth algorithm to find frequent itemsets ending in C	24
3.1	Complete lattice of Table 3.1	30
3.2	Lattices where itemsets are removed	32
3.3	Lattice without itemsets containing A and B , and C and D	33
4.1	Singlepath tree	39
4.2	FP-tree of the transactions in Table 4.1	40
4.3	Conditional FP-tree of <i>pregnant</i>	42
5.1	Weka Explorer GUI	48
5.2	Example of an <code>arff</code> file	48
5.3	APriori	51
5.4	FP-Growth	51
5.5	FP-tree	52
6.1	Generation of the frequent sets	54
6.2	Generation of the association rules	55
6.3	Generation of the frequent sets	56
6.4	Generation of the association rules	56

6.5	Computational time for rule recovery with FP-Growth-KC	57
6.6	Generation of the association rules	58
6.7	Frequent sets for the Geographic dataset	59
6.8	Computational time for generating the frequent sets of the geographic dataset for the APriori-based methods and the FP-Growth-based methods	60
6.9	Association rule generation for the FP-Growth-based methods on the Geo- graphic dataset	60
6.10	Association rules for APriori-KC and FP-Growth-KC with rule recovery . . .	61

List of Tables

2.1	All possible itemsets	7
2.2	Transactional database T	7
2.3	Candidate 2-itemsets	8
2.4	Transactional database T	15
2.5	Frequent items of T , ordered in decreasing support count	16
2.6	List of frequent itemsets	20
2.7	List of found association rules	23
3.1	Transactional database T	29
4.1	Dataset example of a city	40
4.2	Frequent itemsets found by applying the FP-Growth algorithm to the FP-tree in Figure 4.2	41
4.3	Association rules deduced from the frequent itemsets in Table 4.2	41
4.4	Frequent itemsets found by applying the FP-Growth-KC algorithm to the FP-tree in Figure 4.2	43
4.5	Association rules deduced from the frequent itemsets in Table 4.4	43
4.6	Association rules deduced from the frequent itemsets in Table 4.2 after executing FP-Growth-KC	46

Listings

1	FP-Growth-KC(<i>Tree</i> ; α)	v
2.1	Frequent itemset generation of the APriori algorithm	9
2.2	apriori-gen(F_{k-1} : frequent ($k - 1$)-itemsets)	10
2.3	has_infrequent_subset(c : candidate k -itemset; F_{k-1} : frequent ($k-1$)-itemsets) .	10
2.4	FP-Growth(<i>Tree</i> ; α)	18
3.1	APriori-KC algorithm	28
4.1	FP-Growth-KC(<i>Tree</i> ; α)	38

Chapter 1

Introduction and Motivation

1.1 Introduction

Nowadays, databases are used everywhere. Because the amount of data in these databases keeps on growing and growing, it becomes nearly impossible for the human eye, to see patterns in the data at the first glance. This makes the need for automatic analysis solutions very big. Knowledge Discovery in Databases (or KDD) is the part of computer science, that is evolving to find these automatic analysis solutions.

Data mining process The definition of knowledge discovery as it was made in [PSF91] is:

The non-trivial extraction of implicit, unknown, and potentially useful information from data.

Although many people often treat KDD and data mining as synonyms, they are not really the same. In [FPSSU96], they make a clear distinction between them. According to [FPSSU96], knowledge discovery takes the results of data mining (that extracts knowledge from large amounts of data) and tries to transform them carefully into useful and understandable information. So here data mining is just an essential step in the process of knowledge discovery in databases. This KDD process is depicted in Figure 1.1 and exists of the following steps: [HK00]

1. Data cleaning: first noise and inconsistent data are removed, as they are of no use to discover knowledge
2. Data integration: here multiple data sources can be combined. It is also a popular trend to perform data cleaning and data integration as a preprocessing step of knowledge discovery and to store the resulting data in a data warehouse.

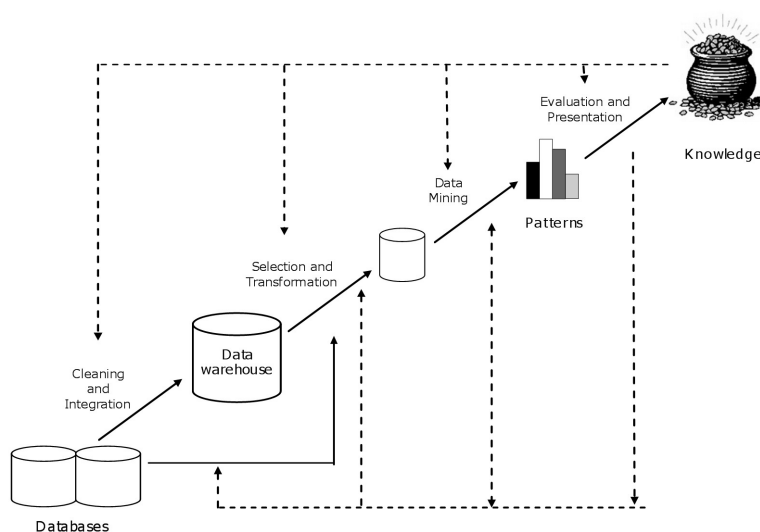


Figure 1.1: Process of knowledge discovery

3. Data selection: we only want to gather knowledge from data that is relevant to the analysis task. Thus we will first select this data, before we start to search for patterns.
4. Data transformation: here data are transformed or consolidated into forms appropriate for mining.
5. Data mining: extract patterns from data.
6. Pattern evaluation: we only want to know the interesting patterns, so these are selected by using interestingness measures.
7. Knowledge presentation: present the mined knowledge to the user. This is done by using visualization and knowledge representation techniques.

In this thesis, we will only go deeper into the data mining step. There are several techniques in data mining, among which are: association analysis, classification and prediction, clustering, outlier detection and evolution analysis.

Association Analysis The first technique, association analysis, tries to discover association rules which show that certain attribute-values occur frequently together in a given set of data [AIS93]. For example, if we have a rule $X \rightarrow Y$, this means that, when we find a certain condition X in a tuple, it is likely that we will also find that a certain condition Y also holds in the same tuple. This association analysis is often used for market basket or transaction data analysis.

Classification Classification is “the process of finding a set of models or functions that describe and distinguish data classes or concepts, for the purpose of being able to use the model to predict the class of objects whose class label is unknown” [HK00]. In other words, we want to distribute certain data over classes – although we don’t know what their class labels are – using a set of training data, from which we already know the class labels. Techniques that are widely used within classification are neural networks, Bayesian learners and ID3.

Though it can be useful to predict the class labels of data objects, we sometimes want to predict the value of missing or unavailable data values. This technique is referred to as prediction.

Clustering Clustering analyses data objects without making use of known class labels. Here the training data – in contrast with classification – does not provide its class labels, simply because they are not known. Clustering can be useful to discover these class labels, by grouping the data objects into certain groups, called clusters, where the data objects in these clusters seem to be very similar to each other. When we look at the similarity of data objects that belong to different clusters, they have to be as less similar as possible. When we have clustered the objects, we can look at a cluster as a class of objects.

Other techniques There also exist some data mining methods, that use outlier detection. These outliers are some data objects that do not comply to the general behaviour of the data. Outlier mining is for example useful for fraud detection, to discover extremely large amounts of money for a given account. Another technique that is used within data mining, is called evolution analysis. This technique tries to find regularities or trends for data objects, whose behaviour changes over time.

Only association analysis will be discussed in the next chapters of this thesis. There are several widely used methods to discover association rules, among which are APriori [AS94] and FP-Growth [HPYM04]. Although these methods are fast, they find a lot of rules, and therefore can still be improved. There still are rules that are found to be non-interesting, because they display implications that are already widely known and are evident. An example of such a rule is *isPregnant* \rightarrow *isFemale*. We can easily remove these rules after they were being generated, but we can also *prevent* them from being generated. The APriori method was extended to APriori-KC [Bog06], where the user gives these well-known dependences to the algorithm as background knowledge. This method was again improved to APriori-KC+ [BMA], where besides the elimination of this well-known dependences, also pairs of predicates with the same feature type are removed (for example, *contains_slum* and *touches_slum*). These methods were mainly developed for use with geographical data, but it is easy to also apply the idea onto other kinds of data.

The FP-Growth algorithm is known to be faster than APriori. Because we also know that the APriori-KC method speeds up the traditional APriori algorithm, it would be logical to apply this same idea of using well-known dependences as background knowledge to improve the FP-Growth algorithm. This will be the main goal of this thesis.

1.2 Motivation

Since there are found too many – non-interesting – rules with the ordinary methods and the FP-Growth algorithm is faster than APriori, we will adapt the APriori-KC method to FP-Growth to find a new algorithm: FP-Growth-KC.

There still seemed to be a little problem with APriori-KC. By removing itemsets that represented well-known dependences, not only the non-interesting association rules were removed, but also some rules that could be interesting. We also took care of this problem, by recovering these removed – interesting – rules.

To show that the newly developed algorithm FP-Growth-KC, together with its extension of rule recovery, still is faster than the APriori algorithms, we also did some experiments.

1.3 Outline

The remaining of this thesis is organized as follows. Chapter 2 shows how association analysis works and how association rules can be generated. It also shows the methods APriori and FP-Growth in more detail.

Chapter 3 presents the idea of well-known dependences and how they can be used to improve existing data mining methods. We will also discuss the APriori-KC method in more detail, and in chapter 4, the new algorithm FP-Growth-KC is presented.

In chapter 5, we will discuss the open source data mining tool Weka, and how we extended this system with the new methods.

Chapter 6 presents the experiments performed with the new algorithm to show that the FP-Growth-KC method indeed is faster than the APriori-KC method. Finally, chapter 7 concludes the thesis and suggests directions of future research.

Chapter 2

Association Analysis

[HK00] The purpose of association rule mining is, given a set of transactions T , find all rules that have a support $\geq \text{minSupport}$ and have a confidence $\geq \text{minConfidence}$ ¹, where support and confidence are defined as follows (with X and Y itemsets):

- $\text{support}(X, Y) = \frac{\text{supportCount}(X \cup Y)}{\# \text{transactions}}$
- $\text{confidence}(X \rightarrow Y) = \frac{\text{supportCount}(X \cup Y)}{\text{supportCount}(X)}$

This is also called the *Association Rule Mining Problem*. This problem can be decomposed into two major subtasks:

1. Frequent Itemset Generation
2. Rule Generation

In the following sections, we will discuss algorithms, for these two subtasks.

2.1 Frequent Itemset Generation

The first thing that we need to take care of is the generation of the frequent itemsets. An itemset can be called frequent, if it has a support $\geq \text{minSupport}$.

Using a brute-force approach to generate the frequent itemsets is absolutely not done. Let's illustrate why: when a dataset, containing k items, is used, there are $2^k - 1$ possible frequent itemsets. Within this brute-force method, this means that every single candidate itemset from these $2^k - 1$ candidate itemsets, needs to be checked for its support count. This is done by comparing each candidate against every transaction. As can be seen, this approach can be very expensive, because it requires $O(NMw)$ comparisons, where N is the number

¹ minSupport and minConfidence are the corresponding support and confidence

of transactions, $M = 2^k - 1$ is the number of candidate itemsets, and w is the maximum transaction width. With the transaction width, we mean the total number of items that is present in a transaction. For example, if a transaction contains the items A , B and C , then its transaction width is 3.

The following section describes 2 methods, APriori and FP-Growth. In both methods, we will assume that the itemsets are ordered lexicographically. This lexicographic ordering can be done with a sorting algorithm, like for example Mergesort, Heapsort ... The two algorithms that are mentioned here both have a complexity of $O(n \log n)$, which is lower than the complexity of generating frequent itemsets. Therefore it is defensible to sort first before the generation of the frequent itemsets. It will become clear later on, why we need to sort the itemsets lexicographic.

2.1.1 APriori

The APriori Principle

The use of support for pruning candidate itemsets is guided by the following principle.

Theorem 2.1 (APriori Principle). *If an itemset is frequent, then all of its subsets must also be frequent.*

Reversely, we can also say that if an itemset is infrequent, then all of its supersets must also be infrequent.

Property 2.1 (Anti-monotonicity property). Let X and Y be two itemsets. A measure f is anti-monotone if

$$X \subseteq Y \Rightarrow f(Y) \leq f(X), \quad (2.1)$$

which means that if X is a subset of Y , then $f(Y)$ must not exceed $f(X)$.

The APriori principle is an example of an anti-monotone property. In this case it means that if an itemset is not frequent because it doesn't reach minimum support, then all of its supersets will not reach minimum support as well.

Let us illustrate this principle with an example. Consider the itemsets in Table 2.1. Suppose now that $\{A, B, D\}$ is a frequent itemset. It is easy to see, that every transaction containing the itemset $\{A, B, D\}$, must also contain all of its subsets: $\{A\}$, $\{B\}$, $\{D\}$, $\{A, B\}$, $\{A, D\}$, $\{B, D\}$. Thus every subset of $\{A, B, D\}$ must also be frequent. Also the opposite counts: when the itemset $\{C, D\}$ is infrequent, then all of its supersets are also infrequent.

The Algorithm

The algorithm was developed by [AS94] and is shown in Algorithms 2.1, 2.2 and 2.3 [TSK05]. First of all, the transaction database is scanned once, to discover all frequent 1-itemsets, which

k	Frequent itemsets
1	$\{A\}, \{B\}, \{C\}, \{D\}, \{E\}$
2	$\{A, B\}, \{A, C\}, \{A, D\}, \{A, E\}, \{B, C\}, \{B, D\},$ $\{B, E\}, \{C, D\}, \{C, E\}, \{D, E\}$
3	$\{A, B, C\}, \{A, B, D\}, \{A, B, E\}, \{A, C, D\},$ $\{A, C, E\}, \{A, D, E\}, \{B, C, D\}, \{B, C, E\},$ $\{B, D, E\}, \{C, D, E\}$
4	$\{A, B, C, D\}, \{A, B, C, E\}, \{A, B, D, E\},$ $\{A, C, D, E\}, \{B, C, D, E\}$
5	$\{A, B, C, D, E\}$

Table 2.1: All possible itemsets

TID	Transaction
T1	A, B, C
T2	A, B, C, D, E
T3	A, C, D
T4	A, B, D, E
T5	A, B, C, D

Table 2.2: Transactional database T

are the items that appear in at least $minSupport$ of the transactions. If we have a transaction database like Table 2.2 and suppose the minimum support is 50% (or a minimum support count of $2.5 \approx 3$), we see that F_1 (see Table 2.1) contains the frequent 1-itemsets $\{A\}, \{B\}, \{C\}$ and $\{D\}$. The itemset $\{E\}$ is not frequent, because it appears in only 2 transactions.

Next, we are going to generate iteratively all new candidate k -itemsets from the frequent $(k - 1)$ -itemsets, that were generated in the previous iteration. This candidate generation is described in Listing 2.2. It takes all frequent $(k - 1)$ -itemsets and compares them to each other, to see if they have $(k - 2)$ items in common. We don't check some random $k - 2$ items, but we compare only the first $k - 2$ items. If all of the $k - 2$ items seem to be the same (and these itemsets are not the same), the itemsets will be joined and thus will form a new k -itemset. Then we are going to check, if this generated k -itemset does not contain an infrequent subset. This is done, because of the APriori principle in Theorem 2.1. If it passes this test, this itemset will be added to C_k , which is the set of candidate k -itemsets. The itemsets are called *candidates*, because we do not know yet, if these itemsets are frequent.

C_2	Support count
{A, B}	4
{A, C}	4
{A, D}	4
{B, C}	3
{B, D}	3
{C, D}	3

Table 2.3: Candidate 2-itemsets

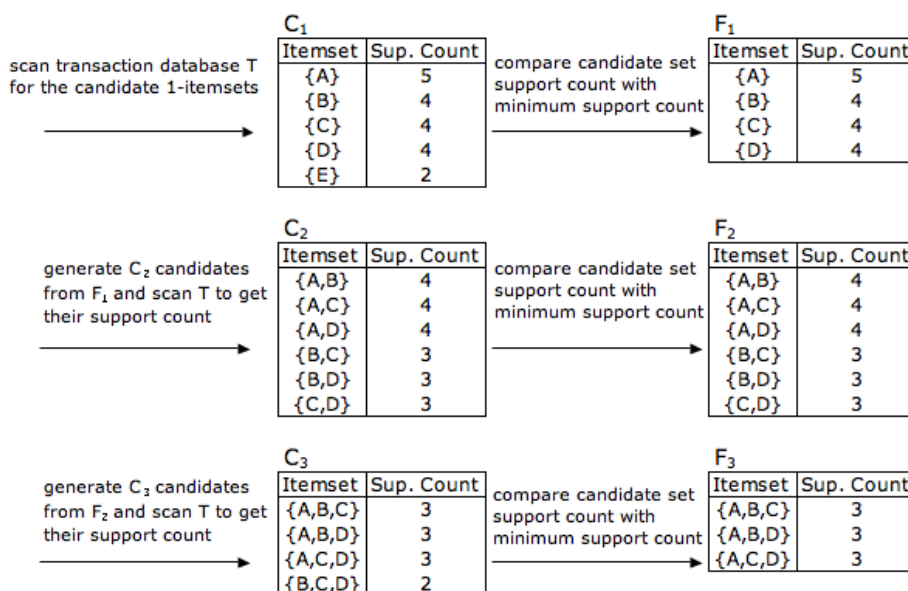


Figure 2.1: Generation of candidate itemsets and frequent itemset, with minimum support of 50%

When we have found the candidate k -itemsets, we are going to check which of these itemsets is frequent. This is done by counting the times that each one of these k -itemsets appears among the transactions in the transaction table. In Table 2.3, you can see the candidate 2-itemsets, that were generated from the frequent 1-itemsets, together with their support count. When the support count for every candidate k -itemset is found, the ones that seem to be frequent, will be added to F_k , the set of frequent k -itemsets. This algorithm will be stopped, if F_k seems to be empty, which means there are no more frequent k -itemsets generated.

You can see the APriori algorithm, applied to the transaction database in Table 2.2, illustrated in Figure 2.1.

Listing 2.1: Frequent itemset generation of the APriori algorithm

```

1   $k = 1$ ;
2  // Find all frequent 1-itemsets
3   $F_k = \{i \mid i \in I \wedge \sigma(\{i\}) \geq N \times \text{minsup}\}$ ;
4  while( $F_k \neq \emptyset$ ) {
5       $k++$ ;
6       $C_k = \text{apriori-gen}(F_{k-1})$ ; // Generate candidate itemsets
7      for each transaction  $t \in T$  {
8           $C_t = \text{subset}(C_k, t)$ ; // Identify all candidates that belong to  $t$ 
9          for each candidate itemset  $c \in C_t$ 
10              $\sigma(c) = \sigma(c) + 1$ ;
11     }
12     // Extract the frequent  $k$ -itemsets
13      $F_k = \{c \mid c \in C_k \wedge \sigma(c) \geq N \times \text{minsup}\}$ ;
14 }
15 Result =  $\bigcup F_k$ 

```

But why do we only need to check the $(k - 2)$ first items during the candidate set generation? In Figure 2.1, the frequent itemsets $\{A, B\}$ and $\{A, C\}$ are merged to form the candidate 3-itemset $\{A, B, C\}$. The algorithm does not need to merge $\{B, C\}$ and $\{C, D\}$, because the first item in both itemsets is different. Indeed, when $\{B, C, D\}$ would be a valid candidate (which is the case in this example), it would be generated by merging the itemsets $\{B, C\}$ and $\{B, D\}$. This example shows that the candidate set generation procedure indeed is complete, and that there are no duplicate k -itemsets generated, because of the lexicographic ordering of the items in the itemsets.

Hash-tree

The way the algorithm is described now, it seems that during the support counting, every itemset of a transaction has to be compared with *every* candidate itemset. But this would slow down the algorithm, as it is possible that there are a lot of transactions and/or a lot of candidate itemsets. To avoid this, the algorithm stores the candidate itemsets of C_k in a *hash-tree*. In a hash-tree, the leaves represent itemsets, while the interior nodes represent a hash-table. Each bucket of such a hash-table at depth d points to another node at depth $d + 1$ (the root of the hash-tree is defined to be at depth 1). An example of a hash-tree is shown in Figure 2.2, where the leaves contain the candidate 2-itemsets.

When we want to add an itemset i to the tree, we start in the root and go from one node

Listing 2.2: apriori-gen(F_{k-1} : frequent $(k-1)$ -itemsets)

```

1 for each itemset  $f_1 \in F_{k-1}$  {
2   for each itemset  $f_2 \in F_{k-1}$  {
3     if ( $f_1[1] == f_2[1] \wedge f_1[2] == f_2[2] \wedge \dots$ 
4        $\wedge f_1[k-2] == f_2[k-2] \wedge f_1[k-1] < f_2[k-1]$ ) {
5        $c = f_1 \bowtie f_2$ ;
6       if (has_infrequent_subset( $c, F_{k-1}$ ))
7         delete  $c$ ; // prune step: remove unfruitful candidate
8       else
9         add  $c$  to  $C_k$ ;
10    }
11  }
12 }
```

Listing 2.3: has_infrequent_subset(c : candidate k -itemset; F_{k-1} : frequent $(k-1)$ -itemsets)

```

1 for each  $(k-1)$ -subset  $s$  of  $c$  do
2   if  $s \notin F_{k-1}$ 
3     return true;
4 return false;
```

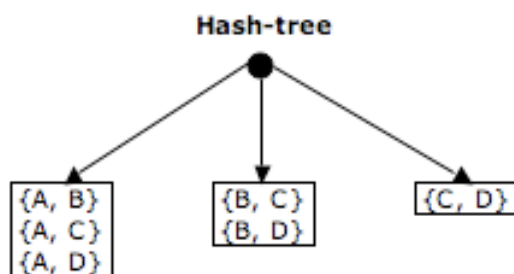


Figure 2.2: Example of a hash-tree

to another until we reach a leaf, where the itemset will be stored. When the itemset i arrives at an interior node at depth d , a hash-function is applied to the d -th element of the itemset. The result of this hash-function decides which branch is taken to go to the next node at depth $d + 1$. If the number of itemsets in a leaf exceeds a predefined threshold, the leaf itself becomes an interior node.

During support counting, the same hash-function is applied to the itemsets contained in each transaction, to partition these itemsets in their appropriate buckets. That way, each itemset in the transaction will only be matched against candidate itemsets that belong to the same bucket.

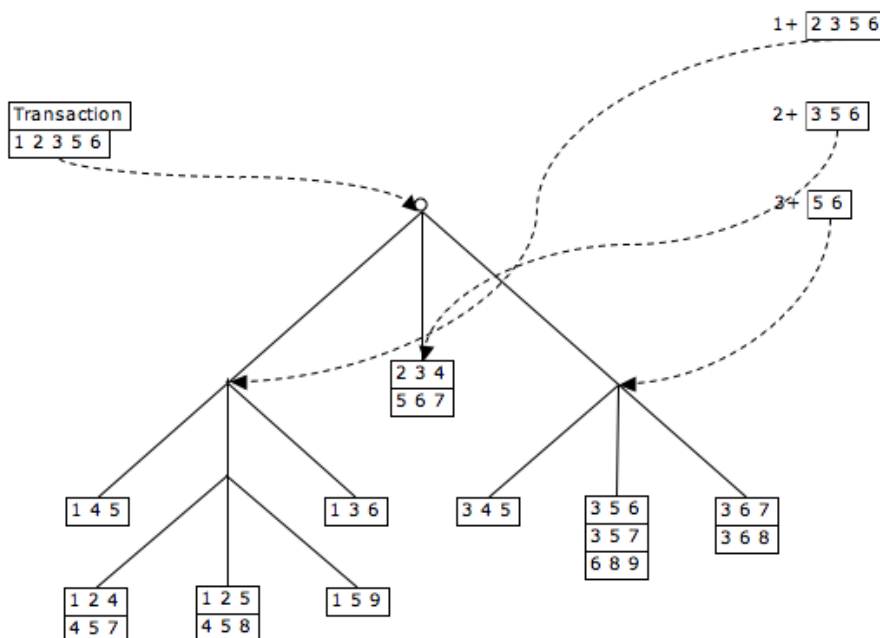


Figure 2.3: Hashing a transaction at the root node of a hash-tree

Figure 2.3 [TSK05] shows an example of a hash-tree structure. Each internal nodes uses the hash-function $h(p) = p \bmod 3$, to decide which branch has to be followed next. When the hash-function returns 1, the leftmost branch has to be picked, if it returns 2, the middle branch and if it returns 0, the rightmost branch. The hash-tree in Figure 2.3 contains 15 candidate 3-itemsets, that are distributed among 9 leaf nodes.

Now consider a transaction $t = \{1, 2, 3, 5, 6\}$. When we want to update the support count of the candidate itemsets, we have to traverse the tree in such a way, that every leaf node containing candidate 3-itemsets belonging to t must be visited at least once. As items are ordered lexicographic, a 3-itemset contained in t must begin with 1, 2 or 3. Therefore, we need to hash the items 1, 2 and 3 separately in the root node. Item 1 is hashed to the left child of the root node, item 2 to the middle child and item 3 to the right child. At the next level of the tree, the transaction is hashed on the second item. This proces continues until the leaf nodes of the hash-tree are reached.

Computational complexity

To analyse the complexity of the APriori-algorithm, we need to take into account three parts [TSK05]:

1. the generation of the frequent 1-itemsets
2. the generation of the candidates
3. the support counting: this is the process that determines the frequency of occurence for each candidate itemset

During the generation of the frequent 1-itemsets, we need to scan the transactional database transaction per transaction, and in meantime update the support count for every item that we encounter. If w_{avg} is the average transaction width and N is the number of transactions, then we need $O(Nw_{avg})$ time to finish this operation.

When we want to generate the candidate k -itemsets from the frequent $(k - 1)$ -itemsets in Algorithm 2.2, we need to check if a pair of frequent $(k - 1)$ -itemsets have $(k - 2)$ items in common. Obviously, this takes $(k - 2)$ equality comparisons. In the worst-case scenario, this algorithm must merge every pair of frequent $(k - 1)$ -itemsets that were found in the previous iteration. When we look at the example that was given in Figure 2.1, we see that to generate the candidate set C_2 , we need to merge every single pair of itemsets in F_1 . So we can conclude that the cost of merging is

$$O\left(\sum_{k=2}^{w_{avg}} (k - 2) |F_{k-1}|^2\right).$$

This formula states that in the worst-case scenario, we need to compare every single F_{k-1} itemset with every other F_{k-1} itemset (which gives us $|F_{k-1}|^2$ possibilities) and that this also needs $(k-2)$ equality comparisons every time. Assuming every transaction has transaction width w_{avg} , it is not possible to generate frequent itemsets that have more than w_{avg} items. Therefore k goes only from 2 to w_{avg} .

During the generation of these candidate k -itemsets, a hash tree is constructed to store these candidates. The maximum depth of this tree is k , the cost of populating this tree with candidates is

$$O\left(\sum_{k=2}^{w_{avg}} k|C_k|\right).$$

During the pruning of the candidates, we need to check that the $k-2$ subsets of every candidate k -itemset are frequent. The cost of looking up a candidate in a hash tree is $O(k)$, so the candidate pruning step requires

$$O\left(\sum_{k=2}^{w_{avg}} k(k-2)|C_k|\right)$$

time.

At last, we need to take in account the cost of counting the support for every candidate itemset. Every transaction that has length $|t|$ produces $\binom{|t|}{k}$ itemsets of size k . This number is also the effective number of hash-tree traversals that needs to be performed for each transaction. Assuming w_{max} is the maximum transaction width and α_k is the cost for updating the support count of a candidate k -itemset in the hash-tree, we can say that the cost of support counting is $O(N \sum_k \binom{w_{max}}{k} \alpha_k)$.

To retrieve the total complexity of the APriori algorithm, we need to take the sum of all of these complexities. This gives us the following complexity:

$$\begin{aligned} O\left(Nw_{avg} + \sum_{k=2}^{w_{avg}} ((k-2)|F_{k-1}|^2 + k|C_k| + k(k-2)|C_k|) + N \sum_k \binom{w_{max}}{k} \alpha_k\right) \\ = O\left(Nw_{avg} + N \sum_k \binom{w_{max}}{k} \alpha_k + \sum_{k=2}^{w_{avg}} ((k-2)|F_{k-1}|^2 + k^2|C_k|)\right) \quad (2.2) \end{aligned}$$

2.1.2 FP-Growth

A second method to generate frequent itemsets, is called FP-Growth and was developed by [HPYM04]. This method is radically different from the APriori-algorithm we have seen before. Whereas the APriori-algorithm generates candidate itemsets, the FP-Growth method puts the dataset in a compact data structure, called an *FP-tree* and extracts frequent itemsets directly from this structure.

Construction of the FP-tree

An FP-tree is a compact representation of the input data, constructed by reading the data transaction per transaction. Each transaction is mapped on a path in the FP-tree and several transactions can have paths that (partially) overlap. An important property of an FP-tree is the node-link property, where nodes with the same label are connected to each other. This makes it easier to traverse the tree.

Property 2.2 (Node-link property). For any frequent item a_i , all the possible patterns containing only frequent items and a_i can be obtained by following a_i 's node-links, starting from a_i 's head in the FP-tree header.

The construction of an FP-tree looks as follows:

1. The data set is scanned once to determine the support count of each item. Items whose support is less than *minSupport* are discarded. Next, the frequent items are sorted in decreasing support count.
2. Create the root of the FP-tree and label it as *null*.
3. For each transaction t , the frequent items are ordered in decreasing support count. This list of ordered items will be added to the tree, as follows:
 - Let f be the first item of the ordered list and T the tree (initially: the root)
 - If T has a child node N such that $f.itemName = N.itemName$, then increment N 's count by 1.
 - If there is no such node, create a new node N that has count 1, let it be linked to T as a child node and its node-link to the nodes that have the same *itemName*.
 - If there still remain items in the ordered list (say L), we need to insert them in the same way into N .

Suppose we have the transactional database T as in Table 2.4, and that *minSupport* = 30%, which is the same as a minimum support count of 3. First, T is scanned to retrieve all the frequent items and these items are sorted in decreasing support count. This gives us the sorted list L in Table 2.5. Initially, the FP-tree contains only the root node, depicted by the *null* symbol. The rest of the FP-tree is constructed in the following way:

1. The first transaction with TID T1 is read. The items of T1 are ordered in decreasing support count, as in L . We only have the root node in the tree, so we will create a node with label B and one with label A and they both get frequency count 1. Having created these nodes, we now must insert them into the tree and thus we create a path $null \rightarrow B \rightarrow A$ to encode the transaction. This is shown in Figure 2.4(a).

TID	Transaction
T1	A, B, E
T2	B, D
T3	A, B, D
T4	B, C
T5	A, D
T6	B, C
T7	A, C
T8	A, B, C, E
T9	A, B, C

Table 2.4: Transactional database T

2. The second transaction, $\{B, D\}$, shares the same prefix item B with the first transaction. As a result, the path of this transaction, which is $\text{null} \rightarrow B \rightarrow D$, overlaps with the path of the previous transaction. Therefore, the frequency count of node B is incremented by 1, and thus becomes 2. The frequency count for the other node, with label D is 1. This FP-tree is shown in Figure 2.4(b).
3. After reading the third transaction, $\{A, B, D\}$, we see that we again have the same situation as after reading the second transaction. The path for this transaction, $\text{null} \rightarrow B \rightarrow A \rightarrow D$, overlaps with the one of the first transaction, $\text{null} \rightarrow B \rightarrow A$. Therefore, the frequency count of both node B and node A are incremented by 1. The node D of this transaction gets a frequency count of 1. We also see that we have an item in common with T2, which is D . But their paths are disjoint, because they do not share a common prefix. To go easily from one node D to another node with label D , we create a node-link between them. After this transaction, we get the FP-tree shown in Figure 2.4(c).
4. This process continues until all of the transactions have been read and mapped onto a path in the FP-tree. The complete FP-tree – that is after reading the transaction with TID T9 – is shown in Figure 2.4(d).

Size of the FP-tree

Typically, the size of an FP-tree is smaller than the size of the uncompressed representation of the data in the transaction table, because usually there are transactions that have one or more items in common. In the best-case scenario, there is only one branch, because all transactions share the same set of items. In the worst-case scenario though, every transaction

Item	Support count
B	7
A	6
C	5
D	3

Table 2.5: Frequent items of T , ordered in decreasing support count

has a different set of items and thus the size of the FP-tree is the same as that of the original data. But, as also the counters for each item and the pointers between nodes need to be saved, the physical storage requirement of the FP-tree in this worst-case scenario is higher.

Also the ordering of the items plays a significant role. Instead of ordering them by decreasing support count, you can also order them by increasing support count. This sometimes results in a more dense FP-tree, but we absolutely cannot conclude that this will always be like this. We can see this in Figure 2.5, where the FP-tree for Table 2.4 was constructed, by ordering the items in increasing support count.

How to generate Frequent Itemsets

The FP-Growth algorithm generates the frequent itemsets by exploring the FP-tree in a bottom-up way. The pseudo-code of the FP-Growth algorithm can be found in Listing 2.4. It is initially called with `FP-Growth(Tree, null)`.

We will explain this algorithm with the example tree shown in 2.4(d). The algorithm will first look for frequent itemsets ending in D , then in C , A and finally in B . Since every transaction is mapped on a path in the FP-tree, we can find the frequent itemsets, ending in D , by examining only the paths that contain node D . These paths can be found easily, since all nodes with label D have a link to the next node with label D . In Figure 2.6(a), we can see the paths that contain node D .

Thus, first the algorithm will search for frequent itemsets, ending in D . This is done by splitting the problem into smaller subproblems. We will first check if the itemset $\{D\}$ itself is frequent. If so, we will consider the subproblem of finding frequent itemsets ending in CD , then in AD and finally in BD . These subproblems will also in turn be divided into smaller subproblems. This is the strategy for finding frequent itemsets ending in D .

1. First we want to retrieve all paths containing node D . These paths are shown in Figure 2.6(a) and they are called *prefix paths*.
2. We need to check if $\{D\}$ itself is a frequent itemset. Therefore, we need to retrieve its support count. This is done by adding the support counts that are associated with node

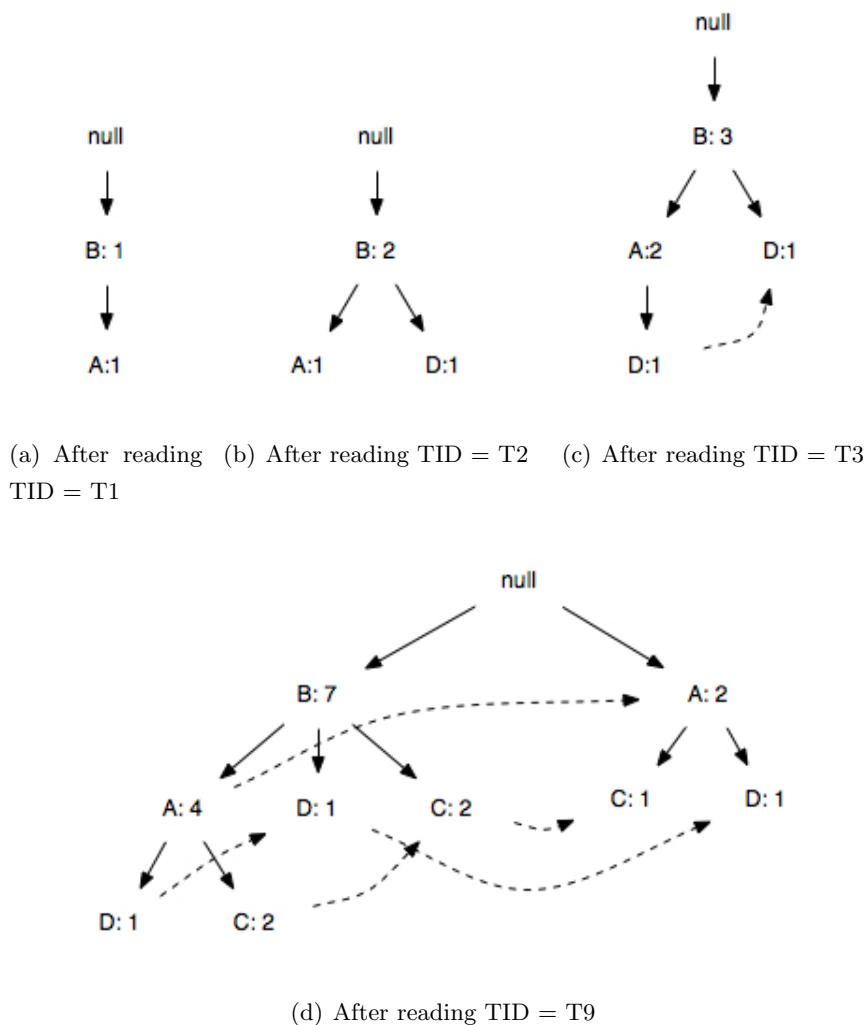


Figure 2.4: Construction of an FP-tree

D . As the support count of $\{D\}$ is 3 and the minimum support count is 3, we can say that this itemset indeed is frequent.

3. Before we are going to divide this problem into the smaller subproblems of finding frequent itemsets in CD , AD and BD , we are going to convert the prefix paths of Figure 2.6(a) into a *conditional FP-tree*. This conversion goes like this:
 - (a) Update the support counts along the prefix paths, as these still contain counts from other transactions.
 - (b) Truncate the prefix paths by removing the nodes with label D . These nodes can be removed, because the support counts have been updated, to only reflect transactions that contain the item D .

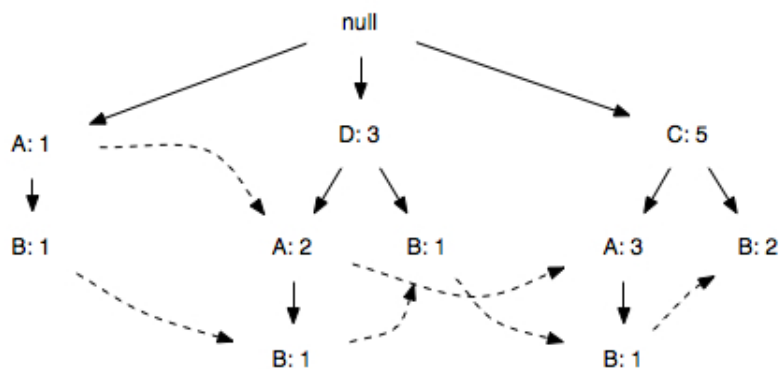
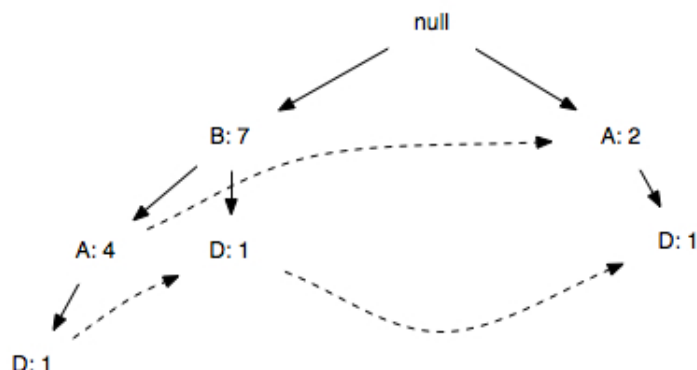
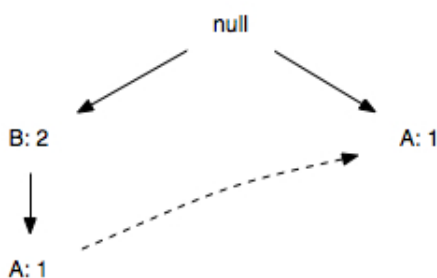


Figure 2.5: An FP-tree representation for the dataset shown in Table 2.4 with a different item ordering scheme

Listing 2.4: FP-Growth($Tree; \alpha$)

```

1 if( $Tree$  contains a single path  $P$ )
2   for each combination  $\beta$  of the nodes in  $P$ 
3     generate pattern  $\beta \cup \alpha$  with support
4       = minimum support of nodes in  $\beta$ ;
5 else
6   for each  $a_i$  in the header of  $Tree$ {
7     generate pattern  $\beta = a_i \cup \alpha$  with support =  $a_i$ .support;
8      $Tree_\beta$  = conditional FP-tree of  $\beta$ ;
9     if( $Tree_\beta$  exists)
10      call FP-Growth( $Tree_\beta, \beta$ );
11   }
12 }
```

(a) Paths containing node D (b) After updating the prefix paths of D **Figure 2.6:** FP-Growth algorithm for finding frequent itemsets ending in D

- (c) It is possible that, after updating the support counts, some of the items may no longer be frequent. We can see this in Figure 2.6(b), where all of the support counts seem to be less than the minimum support count. This means that, for example with node A with a support count of 2, there only are 2 transactions that contain the itemset $\{A, D\}$ and thus is not frequent. Because there are no nodes with a sufficient support count, we can say there are no frequent itemsets ending in D .

Because there are no frequent itemsets ending in D (except the frequent 1-itemset $\{D\}$ itself), we will proceed with the search for frequent itemsets, ending in C . For this, we will start in the same way, as we did to find those ending in D . You can see the prefix paths ending in C in Figure 2.7(a) and the conditional FP-tree of C in Figure 2.7(b). When we add the support counts that are associated with node A , we can find the support count for the itemset $\{A, C\}$. Thus the support count for this itemset is $2 + 1 = 3$ and thus this itemset is

Suffix	Frequent Itemsets
D	{D}
C	{C}, {A,C}, {B,C}
A	{A}, {A,B}
B	{B}

Table 2.6: List of frequent itemsets

frequent. Similarly, we can see that $\{B, C\}$ is also a frequent itemset.

FP-Growth now uses this conditional FP-tree of Figure 2.7(b) to solve the subproblems, which here exist of finding the frequent itemsets ending in AC and then in BC . To find the frequent itemsets ending in AC , we will gather the prefix paths for AC (Figure 2.7(c)) from the conditional FP-tree of C (Figure 2.7(b)). Because the leaf nodes in this conditional FP-tree are all nodes with label A , the prefix paths for AC are the same as the conditional FP-tree for C , but we find only $\{A, C\}$ to be frequent.

Thus the algorithm moves on to the next subproblem: finding the frequent itemsets ending in BC . The algorithm constructs the prefix paths for BC , which is shown in Figure 2.7(d), but concludes that $\{B, C\}$ is the only frequent itemset remaining.

In the same way, we will search for frequent itemsets ending in A and in B . The complete table of frequent itemsets can be found in Table 2.6.

Computational complexity

To build an FP-tree, we need $O(N)$ time where N is the total number of items that were used in the transactions.

At every recursive step in the algorithm, a conditional FP-tree is generated. As all of the subproblems are disjoint, there will never be generated duplicate itemsets. Another advantage is that the support counting of the itemsets happens while generating the common suffix itemsets. When we have an initial FP-tree with k leafs and an average path length w , we can say that the conditional FP-tree is generated $O(kw)$ times. Remark that this is a maximum as it could be possible that – at a certain point – a conditional FP-tree does not contain an itemset with enough support and thus there is no need to generate another conditional FP-tree out of this one.

The run-time performance of the FP-Growth algorithm depends strongly on how compact the dataset is. [HPYM04] made the following conclusions, according to some experiments they did:

- FP-tree achieves good compactness most of the time. Especially in dense datasets, it can compress the database many times.
- When the minimal support count is low, the resulting conditional FP-trees become very bushy (with the worst-case scenario: a full prefix-tree) and the algorithm slows down significantly, as a lot of subproblems have to be generated and the results, returned by each subproblem, have to be merged.

2.2 Rule Generation

Now we have found the frequent itemsets using APriori or FP-Growth, we still need to generate the association rules from these itemsets.

An association rule is an implication $\{X_1, \dots, X_n\} \rightarrow \{Y_1, \dots, Y_m\}$, where $\{X_1, \dots, X_n\} \cap \{Y_1, \dots, Y_m\} = \emptyset$, telling us that, if we encounter the set $\{X_1, \dots, X_n\}$ in a tuple, there is a chance that also the set $\{Y_1, \dots, Y_m\}$ will encounter in this tuple.

From a given k -itemset X , there can be produced $2^k - 2$ association rules, ignoring rules that have empty antecedents ($\emptyset \rightarrow X$) or empty consequents ($X \rightarrow \emptyset$). An association rule can be extracted by partitioning the itemset X into 2 non-empty subsets Y and $X - Y$, so that $Y \rightarrow X - Y$ satisfies the confidence threshold.

Checking the confidence for a particular association rule, does not require an additional read through the transactional database. For example, if we have a frequent itemset $Y = \{x, y, z\}$, we can generate an association rule $\{x\} \rightarrow \{y, z\}$ from it. The confidence of this rule can be calculated by $\sigma(\{x, y, z\})/\sigma(\{x\})$, where $\sigma(X)$ denotes the support count of itemset X .

$$\sigma(X) = |\{t_i | X \subseteq t_i, t_i \in T\}|$$

Because $\{x, y, z\}$ is a frequent itemset, $\{x\}$ must be a frequent itemset too. Since the support counts for both frequent itemsets were already calculated during the frequent itemset generation, we don't need to read the dataset an additional time.

Now suppose we have 2 association rules: $X \rightarrow Y$ and $X_1 \rightarrow Y_1$, where $X_1 \subseteq X$ and $Y_1 \subseteq Y$. When we have the confidence of the first rule, we cannot conclude anything about the confidence of the second rule, as the confidence of this rule can be smaller, equal to or larger than the confidence of the other rule. But if we compare rules that were generated from the *same* frequent itemset Y , we can show that the following theorem holds.

Theorem 2.2. *If a rule $X \rightarrow Y - X$ does not satisfy the confidence threshold, then any other rule $X' \rightarrow Y - X'$, where X' is a subset of X , cannot satisfy the confidence threshold as well.*

Proof. Consider the following two rules:

- $X' \rightarrow Y - X'$
- $X \rightarrow Y - X$

with $X' \subset X$. The confidence of those rules are $\sigma(Y)/\sigma(X')$ and $\sigma(Y)/\sigma(X)$ respectively. X' is a subset of X , so we can say that $\sigma(X') \geq \sigma(X)$. Therefore, the confidence of the first rule must be less than or equal to the confidence of the second rule and thus the theorem is proven. \square

2.2.1 Generating Frequent Association Rules

As we have seen before, the confidence of a rule $A \rightarrow B$ represents the percentage of transactions containing A that also contain B .

$$\text{confidence}(A \rightarrow B) = P(B|A) = \frac{\sigma(A \cup B)}{\sigma(A)} \quad (2.3)$$

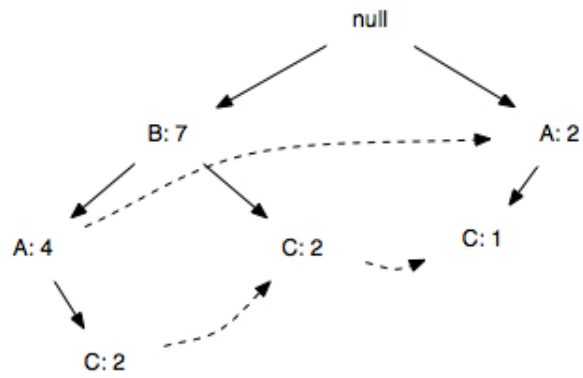
Based on this equation, we can generate association rules in the following way. We will first generate all non-empty subsets of every frequent itemset I . For all of the non-empty subsets J of a frequent itemset I , we will calculate $\frac{\sigma(I)}{\sigma(J)}$, which is the confidence of the rule $J \rightarrow (I - J)$. If the result seems to be larger or equal to *minConfidence*, the association rule is said to be *strong*. This means that both the arguments of *minSupport* and *minConfidence* have been satisfied. The minimum support is satisfied because we have generated these rules from frequent itemsets, so the minimum support still is satisfied.

Now, let us show this with an example. Suppose we use the transactional database T in Table 2.2. One of the frequent itemsets we have found there was the 3-itemset $\{A, B, C\}$. First, we will extract all non-empty subsets of this itemset. The found subsets are: $\{A\}$, $\{B\}$, $\{C\}$, $\{A, B\}$, $\{A, C\}$ and $\{B, C\}$. The resulting association rules are shown in Table 2.7, together with their confidence. When the minimum confidence threshold is, for example, 75%, all rules are strong, except the first one.

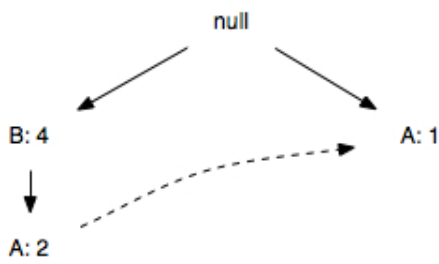
In this chapter, we broadly discussed how to generate frequent itemsets and association rules from these frequent itemsets. We presented the two methods APriori and FP-Growth in detail. The problem with these methods is, that there can be generated too many rules, that are not interesting. Therefore, we will discuss how to overcome this problem in the next chapter.

Rule	Confidence
$A \rightarrow BC$	$3/5 = 60\%$
$B \rightarrow AC$	$3/4 = 75\%$
$C \rightarrow AB$	$3/4 = 75\%$
$AB \rightarrow C$	$3/4 = 75\%$
$AC \rightarrow B$	$3/4 = 75\%$
$BC \rightarrow A$	$3/3 = 100\%$

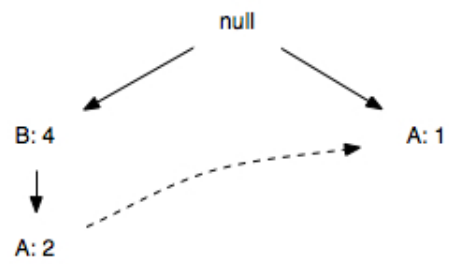
Table 2.7: List of found association rules



(a) Prefix paths for C



(b) Conditional FP-tree for C



(c) Prefix paths for AC



(d) Prefix paths for BC

Figure 2.7: FP-growth algorithm to find frequent itemsets ending in *C*.

Chapter 3

Knowledge Constraints

3.1 Well-known dependences

Within data mining, association rule mining is used to discover patterns in the form of association rules. However, the output of association rule mining algorithms is normally a huge amount of association rules, in which the user has to search those rules that are significant. Among all these rules, many are not interesting, because they express facts that are evident. This is shown in the next example.

Example 3.1. When one would mine data, that describes characteristics of persons, one could possibly find the following rules:

- $isPregnant \rightarrow isFemale$
- $cervicalCancer \rightarrow isFemale$

It is easy to see, that both of these rules are true, but that they are of no use, since they both have a confidence of 100% and are rules that are known apriori.

Because these rules express facts that are well known, we call these non-interesting rules *well-known dependences*. Though we have seen that non-interesting rules have a confidence of 100%, we cannot conclude that the reverse also counts. For example, take the fact that it is dark outside. When we would find a rule like $darkOutside \rightarrow night$ with a confidence of 100% in a dataset of Belgium, this rule would be found non-interesting since in Belgium it is always night when it is dark outside. But when we have a dataset of the Northpole or Southpole, this rule would not always be true, since it can be non-stop dark outside for almost three months. So we cannot call this a well-known dependence since the rule can indeed be interesting sometimes and thus is application dependent.

3.2 Removing non-interesting rules

3.2.1 How?

Well-known dependences are known to be of no use, so they can be removed from the generated set of rules. This would happen during the generation of the rules, i.e. *a posteriori*, when all frequent itemsets were already generated. But instead of letting these rules being generated and just remove them afterwards, it would be more logical to *prevent* these rules from being generated, as is suggested in [Bog06].

How can we do this? We assume the user knows which itemsets can be categorised as well-known dependences. Before the datamining algorithm starts, the user tells the algorithm which combination of items can produce association rules that are non-interesting. These are the *knowledge constraints*, that prevent “useless knowledge” from being generated. During the frequent itemset generation, the algorithm will check that none of these itemsets will be computed, and thus no well-known dependence will be part of the resulting association rules.

3.2.2 No loss of information?

For the removal to be useful, there needs to be proved that no – possibly – interesting information gets lost by applying this method. We could think of 2 possible situations where these interesting rules could get lost.

The first possibility would be when $\{A, B\}$ is treated as a well-known dependence and no frequent itemsets containing both of these items would be generated, such as $\{A, B, C\}$, for example. Now, if the itemset $\{A, B, C\}$ reaches minimum support, then all of its subsets also reached minimum support. We only remove the itemsets that contain both A and B , so the itemsets $\{A, C\}$ and $\{B, C\}$ are still generated, thus also the rules $A \rightarrow C$ and $B \rightarrow C$ are not prevented from being generated. We can conclude here that there gets no information lost in this situation.

Another possible situation is when a pair $\{A, B\}$ is the well-known dependence (because $A \rightarrow B$ is a non-interesting rule) and the removal of all itemsets containing both of these items would prevent rules like $B \rightarrow AC$ from being generated.

Let’s show this with an example. We know that the itemset $\{isPregnant, isFemale\}$ is a well-known dependence because the

$$isPregnant \rightarrow isFemale$$

is non-interesting and thus may not be generated. This means that all of the itemsets containing both of these items, like for example $\{isPregnant, isFemale, isBlond\}$, would not be generated.

Of course, a rule like

$$isPregnant \rightarrow isFemale \wedge isBlond$$

would be non-interesting, because when this rule reaches minimum support, then also

$$isPregnant \rightarrow isBlond$$

reaches minimum support, and thus no information gets lost.

But it is also possible, that a rule like

$$isFemale \rightarrow isPregnant \wedge isBlond$$

could be interesting, because in the researched region a part of the females happen to be pregnant *and* blond.

It thus may happen that possibly interesting rules are not generated. Therefore, we will recover these lost rules out of the association rules we found after the frequent itemset generation. We will discuss this matter later on, in Section 4.4.

3.3 APriori-KC

3.3.1 The Algorithm

One such method to prevent non-interesting rules from being generated, is called APriori-KC (where KC stands for Knowledge Constraints) and was developed by [Bog06]. The pseudo-code of this method can be found in Listing 3.1.

To eliminate the well-known dependences, there is added one more step to the APriori algorithm, that is only performed when k is 2, such that all pairs of elements that were defined to be well-known dependences are removed from C_2 . These dependences are being removed from all of the candidate 2-itemsets, before even the support count of these candidates is performed.

In line 8 of the algorithm, all of the candidate 2-itemsets that contain items with a dependence will be eliminated. The elimination of, for example, $\{isPregnant, isFemale\}$, avoids that later on, there are no association rules generated like $isPregnant \rightarrow isFemale$. Note that association rules like $hair=long \rightarrow isPregnant$ and $hair=long \rightarrow isFemale$ still can be generated, because we do not remove the itemsets $\{isPregnant, hair=long\}$ and $\{isFemale, hair=long\}$.

Because of the removal of these dependences during the frequent itemset generation, APriori-KC is actually faster than the “normal” APriori algorithm itself. Note that this

Listing 3.1: APriori-KC algorithm

```

1  $k = 1$ ;
2 // Find all frequent 1-itemsets
3  $F_k = \{i \mid i \in I \wedge \sigma(\{i\}) \geq N \times \text{minsup}\}$ ;
4 while ( $F_k \neq \emptyset$ ) {
5      $k++$ ;
6      $C_k = \text{apriori-gen}(F_{k-1})$ ; // Generate candidate itemsets
7     if  $k==2$ 
8         delete all pairs with dependences from  $C_2$ ;
9     for each transaction  $t \in T$  {
10          $C_t = \text{subset}(C_k, t)$ ; // Identify all candidates that belong to  $t$ 
11         for each candidate itemset  $c \in C_t$ 
12              $\sigma(c) = \sigma(c) + 1$ ;
13     }
14 // Extract the frequent  $k$ -itemsets
15  $F_k = \{c \mid c \in C_k \wedge \sigma(c) \geq N \times \text{minsup}\}$ ;
16 }
17 Result =  $\bigcup F_k$ 

```

algorithm is not faster in the worst-case scenario, because then there are no dependences that need to be removed and thus APriori-KC acts like the normal APriori. To show that APriori-KC is an improvement of APriori, we are going to discuss the complexity of APriori-KC. With this complexity, we mean the total number of itemsets that will be removed, by removing the well-known dependences.

We first illustrate this with an example. Next, we will deduce a general formula from this example, to calculate the total number of itemsets that will not be generated by removing one or more well-known dependences.

Note that, to show the difference between APriori and APriori-KC, we will treat the fact that some itemsets are not *generated*, as if these itemsets are being *removed*.

3.3.2 Example

In this example, we have a transactional database, like the one in Table 3.1, containing the items A , B , C , D and E .

Suppose the following two things:

1. Every combination that can be made with these items, is a frequent itemset, which means that $minSupport = 0$.
2. $\{A, B\}$ and $\{C, D\}$ are well-known dependences.

The complete lattice of the frequent itemsets can be found in Figure 3.1.

k	Frequent itemsets
1	$\{A\}, \{B\}, \{C\}, \{D\}, \{E\}$
2	$\{A, B\}, \{A, C\}, \{A, D\}, \{A, E\}, \{B, C\}, \{B, D\}, \{B, E\}, \{C, D\}, \{C, E\}, \{D, E\}$
3	$\{A, B, C\}, \{A, B, D\}, \{A, B, E\}, \{A, C, D\}, \{A, C, E\}, \{A, D, E\}, \{B, C, D\}, \{B, C, E\}, \{B, D, E\}, \{C, D, E\}$
4	$\{A, B, C, D\}, \{A, B, C, E\}, \{A, B, D, E\}, \{A, C, D, E\}, \{B, C, D, E\}$
5	$\{A, B, C, D, E\}$

Table 3.1: Transactional database T

Because $\{A, B\}$ and $\{C, D\}$ are well-known dependences, they have to be removed, as well as every other itemset that contains the items A and B or the items C and D . The first itemset that is going to be removed, is $\{A, B\}$. To calculate the total number of frequent

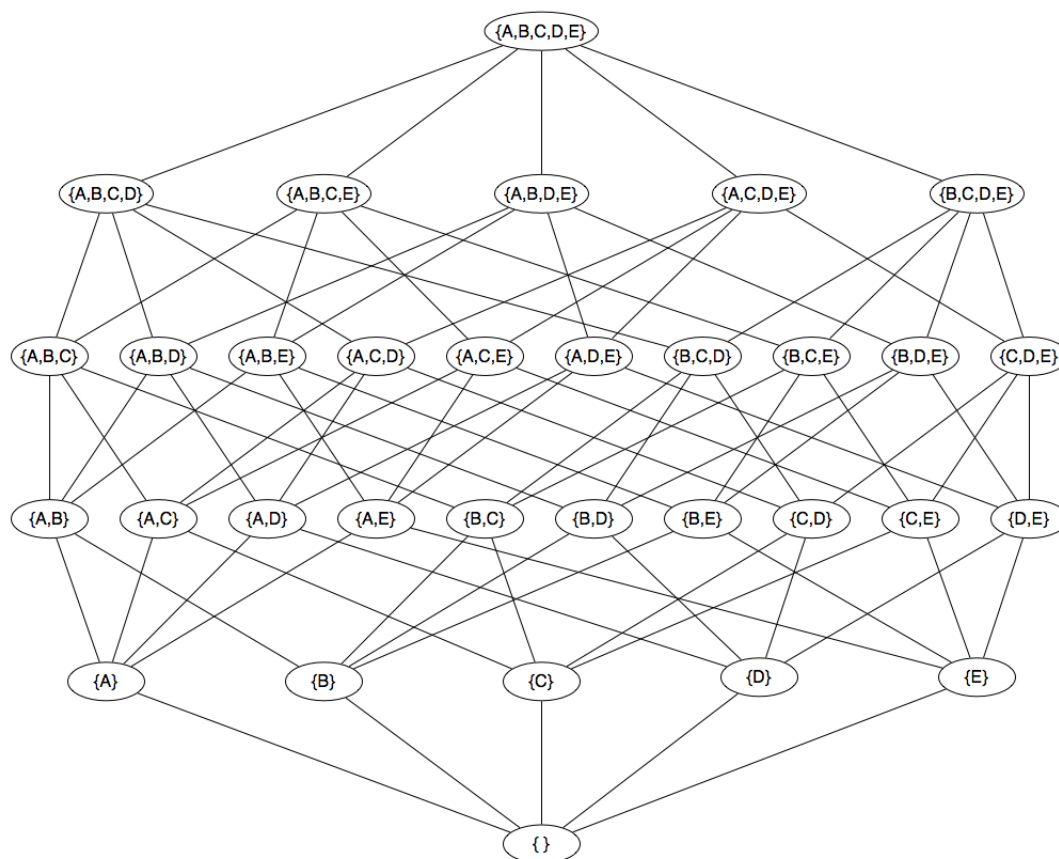


Figure 3.1: Complete lattice of Table 3.1

itemsets that needs to be removed, we need to calculate how many k -itemsets are removed ($2 \leq k \leq 5$).

- # 2-itemsets: only the itemset $\{A, B\}$ is removed
 \Rightarrow **1 itemset**
- # 3-itemsets: take all of the 1-itemsets, that can be formed with the remaining items. These are: $\{C\}$, $\{D\}$ and $\{E\}$. Combine the itemset $\{A, B\}$, with each of these itemsets, to form a 3-itemset. Remove the resulting three 3-itemsets.
 \Rightarrow **3 itemsets**
- # 4-itemsets: take all of the 2-itemsets, that can be formed with the remaining items. These are: $\{C, D\}$, $\{C, E\}$, $\{D, E\}$. Combine the itemset that you want to remove $\{A, B\}$, with each of these itemsets. This also results in 3 more itemsets, that have to be removed.
 \Rightarrow **3 itemsets**

- # 5-itemsets: take all of the 3-itemsets, that can be formed with the remaining items. Here, this is only one: $\{C, D, E\}$. Combine the itemset $\{A, B\}$, with this itemset. This results in one 5-itemset, that has to be removed.

\Rightarrow **1 itemset**

- **TOTAL: 1 + 3 + 3 + 1 = 8 itemsets**

Next, all itemsets that contain the items C and D are removed. We will act as if the itemset $\{A, B\}$ is not yet removed. Then the calculation of the number of itemsets that is removed is analog to the calculation for the itemsets eliminated by removing $\{A, B\}$ and thus also 8 itemsets are being removed.

We can see the lattices with $\{A, B\}$ removed and $\{C, D\}$ removed in Figure 3.2(a) and 3.2(b). Note that the total number of itemsets, that is removed, is not $(8 + 8) = 16$, because we have counted the itemsets $\{A, B, C, D\}$ and $\{A, B, C, D, E\}$ twice. So the total number of itemsets, that is removed, is actually 14. The lattice without the itemsets containing A and B , and C and D , is shown in Figure 3.3.

This leads to the following formula:

$$\begin{aligned}
 & (\# \text{ removed by } \{A, B\}) + (\# \text{ removed by } \{C, D\}) \\
 & \quad - (\#\{A, B\} \text{ and } \{C, D\} \text{ in common}) \\
 & \qquad \qquad \qquad = 8 + 8 - 2 = 14 \quad (3.1)
 \end{aligned}$$

3.3.3 Generalization

We still suppose that every possible combination of items, is a frequent-itemset.

The total number of frequent itemsets is then represented by I , where

$$I = \sum_{k=1}^n \binom{n}{k} = \sum_{k=1}^n \frac{n!}{k!(n-k)!} \quad (3.2)$$

where n is the number of items. To reduce this formula, the *binomial theorem* is used, where

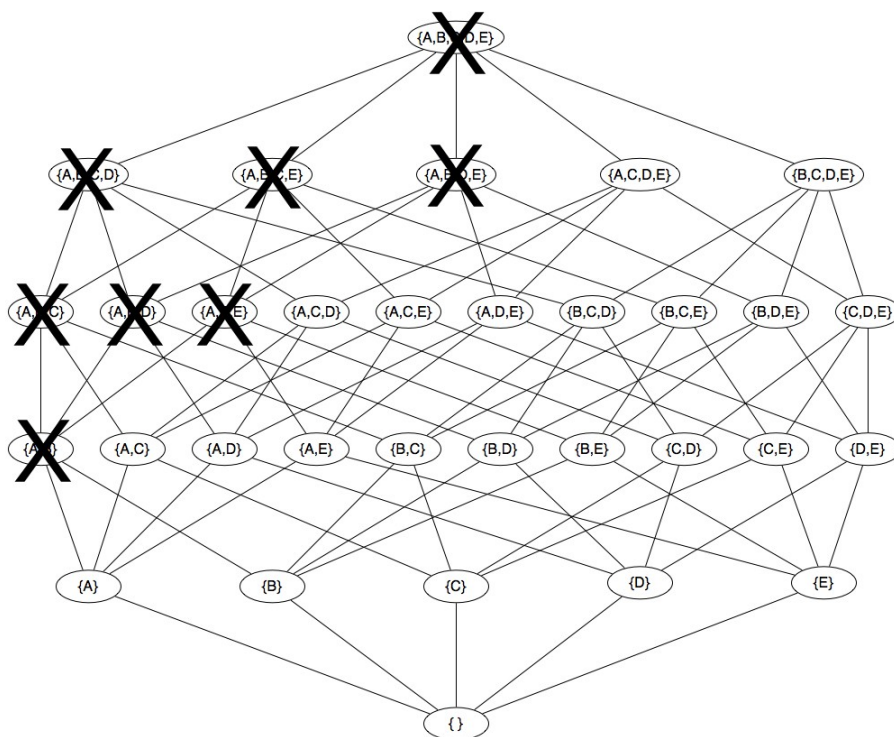
$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k} \text{ for } n \geq 0 \quad (3.3)$$

So actually Formula 3.2 is the same as:

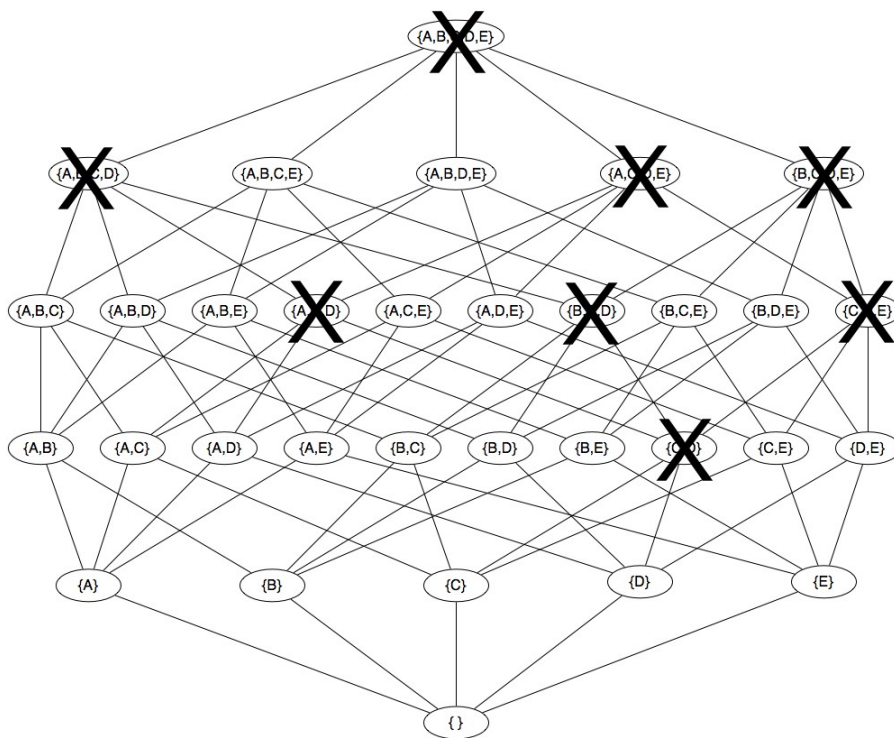
$$I = \sum_{k=0}^n \binom{n}{k} - \binom{n}{0} = \sum_{k=0}^n \binom{n}{k} 1^n 1^{n-k} - 1 = 2^n - 1 \quad (3.4)$$

The number of itemsets that remain, after the removal of an itemset, is represented by R .

Note that in this section, we will only discuss the ideal situation, where *all* of the itemsets



(a) Lattice where $\{A, B\}$ is removed



(b) Lattice where $\{C, D\}$ is removed

Figure 3.2: Lattices where itemsets are removed

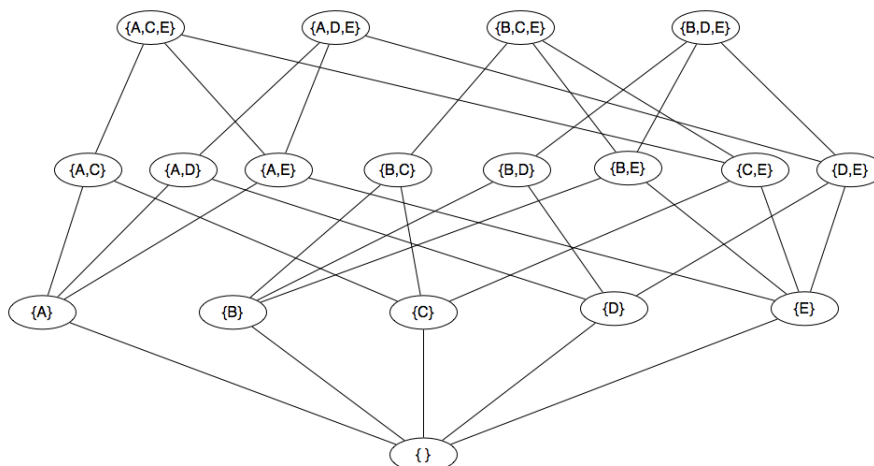


Figure 3.3: Lattice without itemsets containing A and B , and C and D

containing a well-known dependence will be removed. This ideal situation can be regarded as the upper limit for all other situations, since in this situation the maximal number of frequent itemsets is removed.

Removing 1 well-known dependence I_1

Suppose, that I_1 is a 2-itemset and there are n items.

The total number of itemsets that is removed in this ideal situation, then is:

$$\underbrace{\binom{n-2}{0}}_{\alpha_1} + \underbrace{\binom{n-2}{1}}_{\alpha_2} + \underbrace{\binom{n-2}{2}}_{\alpha_3} + \dots + \underbrace{\binom{n-2}{n-2}}_{\alpha_4} \quad (3.5)$$

Where:

1. α_1 : the number of 2-itemsets that can be made with I_1 and the remaining $n - 2$ items. I_1 already is a 2-itemset, so this is 1.
2. α_2 : the number of 3-itemsets that can be made with I_1 and the remaining $n - 2$ items.
3. α_3 : the number of 4-itemsets that can be made with I_1 and the remaining $n - 2$ items.
4. α_4 : the number of n -itemsets that can be made with I_1 and the remaining $n - 2$ items. This is also 1.

Formula 3.5 can be reduced to:

$$\sum_{k=0}^{n-2} \binom{n-2}{k} = 2^{n-2} \quad (3.6)$$

When I_1 would be a 3-itemset, we would get the following formula for the total number of itemsets that is removed:

$$\begin{aligned}
& \underbrace{\binom{n-3}{0}}_{\alpha_1} + \underbrace{\binom{n-3}{1}}_{\alpha_2} + \underbrace{\binom{n-3}{2}}_{\alpha_3} + \dots + \underbrace{\binom{n-3}{n-3}}_{\alpha_4} \\
&= \sum_{k=0}^{n-3} \binom{n-3}{k} \\
&= 2^{n-3}
\end{aligned} \tag{3.7}$$

We can now conclude the following:

Theorem 3.1. *When removing 1 j -itemset ($2 \leq j \leq n$) and the number of items is n , the number of itemsets that remain is:*

$$R = I - 2^{n-j} = 2^n - 2^{n-j} - 1 \tag{3.8}$$

Removing 2 well-known dependences I_1 and I_2 with $I_1 \cap I_2 = \emptyset$

Suppose that:

- I_1 is an i -itemset ($2 \leq i \leq n$)
- I_2 is a j -itemset ($2 \leq j \leq n$)
- n is the number of different items

The total number of itemsets that is removed, is then calculated by:

$$\underbrace{2^{n-i}}_{\alpha_1} + \underbrace{2^{n-j}}_{\alpha_2} - \beta \tag{3.9}$$

Where:

1. α_1 : the number of itemsets that is removed, by removing I_1
2. α_2 : the number of itemsets that is removed, by removing I_2
3. β : the number of itemsets that are removed by both I_1 and I_2 . This is what we have seen in Section 3.3.2 by removing both $\{A, B\}$ and $\{C, D\}$.

To calculate β , we need to search the number of all itemsets that contain the $(i+j)$ -itemset, that is formed by $I_1 \cup I_2$. If $i+j > n$, then there will be no itemsets that are in common, and so $\beta = 0$.

$$\beta = \begin{cases} 0 & \text{if } n - i - j < 0 \\ \sum_{k=0}^{n-i-j} \binom{n-i-j}{k} = 2^{n-i-j} & \text{if } n - i - j \geq 0 \end{cases}$$

Theorem 3.2. *When removing 2 itemsets I_1 and I_2 , that do not contain the same items, the number of remaining itemsets is:*

$$R = 2^n - 2^{n-i} - 2^{n-j} + 2^{n-i-j} - 1 \quad (3.10)$$

Removing 2 well-known dependences I_1 and I_2 with $I_1 \cap I_2 \neq \emptyset$

Suppose that:

- I_1 is a 2-itemset: $\{x, y\}$
- I_2 is a 2-itemset: $\{y, z\}$

Here we have to distinguish two possibilities:

1. There is no transitivity between the rules they generate
2. There is transitivity between the rules they generate

Definition 3.1. There is transitivity between 2 associations, when the first one is of the type $x \rightarrow y$ and the association rule of the other one is of the type $y \rightarrow z$, where the confidence of **both** rules is 100%. This transitivity then means that also $x \rightarrow z$ is an association rule with a confidence of 100%.

First, suppose we can make use of this transitivity rule. This means that the association rule generated by I_1 is $x \rightarrow y$ and the association rule generated by I_2 is $y \rightarrow z$. Because they both are well-known dependences, their association rules also both have a confidence of 100%. This means that every time there appears an x , there also is a y , and every time there is a y , there also is a z . So whenever there appears an x , there also appears a z .

From this, we can conclude that also $x \rightarrow z$ is an association rule with a confidence of 100%, and thus the itemset $\{x, z\}$ is another well-known dependence, which also has to be removed.

We now have 3 itemsets that have to be removed: $\{x, y\}$, $\{y, z\}$ and $\{x, z\}$. So the formula appears to be:

$$R = I - 2^{n-2} - 2^{n-2} - 2^{n-2} \quad (3.11)$$

But by removing each of these itemsets, the itemsets that contain x , y and z are removed too, which results in counting them 3 times instead of 1 time.

Theorem 3.3. *When removing two 2-itemsets I_1 and I_2 that overlap and where there is transitivity, the number of remaining itemsets is:*

$$\begin{aligned} R &= 2^n - 1 - 3 \cdot 2^{n-2} + 2 \cdot 2^{n-3} \\ &= 2^n - 1 - 3 \cdot 2^{n-2} + 2^{n-2} \\ &= 2^n - 2 \cdot 2^{n-2} - 1 \\ &= 2^n - 2^{n-1} - 1 \end{aligned} \quad (3.12)$$

When there is no transitivity between the rules of I_1 and I_2 , so their association rules are $x \rightarrow y$ and $z \rightarrow y$ or $y \rightarrow x$ and $y \rightarrow z$, we cannot remove the itemset $\{x, z\}$. This results in the following theorem:

Theorem 3.4. *When removing two 2-itemsets I_1 and I_2 that overlap and where there is no transitivity, the number of remaining itemsets is:*

$$\begin{aligned}
 R &= 2^n - 1 - 2 \cdot 2^{n-2} + 2^{n-3} \\
 &= 2^n - 2^{n-1} + 2^{n-3} - 1 \\
 &= 2^{n-1} + 2^{n-3} - 1
 \end{aligned}
 \tag{3.13}$$

We discussed the APriori-KC algorithm, and see that this algorithm is really faster than the original algorithm of APriori, because some frequent itemsets were not generated due to the fact of containing a well-known dependence and thus would produce rules that were useless. This speed-up would only take place, if we did not encounter the worst-case scenario, where there would be no well-known dependences given by the user, and thus the algorithm would act just like the original APriori.

Since FP-Growth is faster than APriori, we will apply this method of dependence elimination to FP-Growth, as we will see in the next chapter.

Chapter 4

FP-Growth-KC

4.1 The Concept

The APriori algorithm – apart from generating both redundant and well-known patterns – is not as fast as other algorithms like FP-Growth. As we have seen in Section 2.1.2, the FP-Growth algorithm needs less time, to accomplish the same thing as the APriori algorithm. The authors of the algorithm claim that this is because the FP-Growth algorithm does not generate candidate itemsets. According to [Goe02], this cannot be said, as in fact FP-Growth generates a lot more candidate itemsets than the APriori algorithm. This is because FP-Growth uses the same candidate generation technique as in APriori, but it just leaves out the pruning step.

Since the well known FP-Growth algorithm seems to be faster than APriori, we will apply the same technique of using knowledge constraints to this algorithm.

4.2 The Algorithm

When we discussed the APriori-KC algorithm in Section 3.3, we saw that itemsets that represented a well-known dependence, were removed in the pruning step of the algorithm. But when we want to apply the same method of dependence removal to FP-Growth, it is impossible to do this in the same way as there is no pruning step in the FP-Growth algorithm. Thus we need to check the presence of well-known dependence elsewhere in the algorithm.

As FP-Growth also uses a candidate generation technique (with the building of conditional FP-trees), it would be logical to insert a test for well-known dependences during this step. This means that the algorithm would look like Listing 4.1. In this algorithm, Φ is the set of well-known dependences, which are the knowledge constraints.

Listing 4.1: FP-Growth-KC($Tree; \alpha$)

```

1 if( $Tree$  contains a single path  $P$ )
2   for each combination  $\beta$  of the nodes in  $P$ 
3     if( $\beta$  contains no dependence of  $\Phi$ )
4       generate pattern  $\beta \cup \alpha$  with support
5         = minimum support of nodes in  $\beta$ ;
6 else
7   for each  $a_i$  in the header of  $Tree$ {
8      $\beta = a_i \cup \alpha$  with support =  $a_i$ .support;
9      $Tree_\beta =$  conditional FP-tree of  $\beta$ ;
10    for each node  $b_i$  in  $Tree_\beta$ 
11      if( $b_i \cup \beta$  contains dependence of  $\Phi$ )
12        remove  $b_i$  from  $Tree_\beta$ ;
13    if( $Tree_\beta$  exists)
14      call FP-Growth-KC( $Tree_\beta, \beta$ );
15  }
16 }
```

As you can see, this algorithm does not differ very hard from the original FP-Growth algorithm. Line 3 and Line 10 to 12 in Listings 4.1 are the only lines that were added, to make sure that no well-known dependences would appear amongst the found frequent itemsets.

When the tree is not a single path, we need to build a conditional FP-tree for every $\beta = a_i \cup \alpha$, where a_i is a node in the header of the tree. In the original FP-Growth algorithm, we will prune β and the nodes that do not reach minimum support, during the building of this conditional FP-tree for β . As the frequent itemsets will be deduced from these conditional FP-trees, we will prune this tree further to make sure the well-known dependences are removed. We will therefore check for every node b_i in the conditional FP-tree of β if $b_i \cup \beta$ produces any of the well-known dependences of Φ . Since these nodes are then useless, because they only lead to frequent itemsets with well-known dependences and thus also to non-interesting association rules, we can remove them from the tree. This results in a conditional FP-tree, that will only give us the interesting frequent itemsets.

When we have a tree, that contains only one single path, it is not possible to prune well-known dependences from this tree, as it still has to be possible to combine the different items of the dependence with the other items. Thus when we would prune this tree, this would

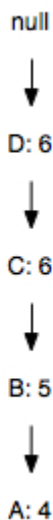


Figure 4.1: Singlepath tree

result in a tree where too many nodes were removed and thus not all of the – still interesting – frequent itemsets could be found.

This can be easily understood with the following example. Assume we have a tree like the tree in Figure 4.1 and that $\{A,B\}$ is the dependence and that all items in this tree reach minimum support. When we would remove A and B from the tree, the only frequent itemsets that would be found are $\{C\}$, $\{D\}$ and $\{C,D\}$. But these itemsets are not the only ones that are frequent: also $\{A\}$, $\{B\}$, $\{A,C\}$, $\{B,C\}$, $\{A,D\}$, $\{B,D\}$, $\{A,C,D\}$, $\{B,C,D\}$ are frequent. Therefore we cannot prune a singlepath tree. When we combine the items (as in the original FP-Growth algorithm), we must then check if this combination does not contain a dependence of Φ . Only then it is found to be a frequent itemset.

4.3 Example

To illustrate how the new algorithm is working, we will show the use of it on an example.

For this, we use the transactional database shown in Table 4.1. This represents tests that were performed on the small amount of 10 persons in a city, according to their characteristics: the gender, haircolour and whether they are pregnant or not. To represent these characteristics, we use the attributes *Female*, *Blond* and *Pregnant*. The value ‘yes’ for a particular characteristic, means that the person in question has this feature. Contrary, the value ‘-’ means that the person in question does not have this characteristic.

PersonID	Female	Blond	Pregnant
1	-	yes	-
2	yes	-	-
3	yes	-	yes
4	yes	yes	yes
5	-	-	-
6	-	yes	-
7	yes	yes	yes
8	yes	-	yes
9	-	yes	-
10	yes	-	-

Table 4.1: Dataset example of a city

First we will construct the FP-tree of this dataset, according to the algorithm shown in Section 2.1.2. Assuming a minimum support count of 1, we will get the FP-tree that is shown in Figure 4.2.

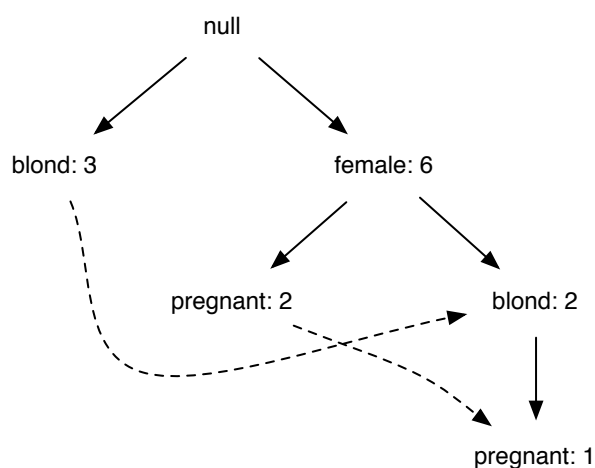


Figure 4.2: FP-tree of the transactions in Table 4.1

When we use this FP-tree as input for the original FP-Growth algorithm, we find the frequent itemsets listed in Table 4.2.

When deducing the association rules from these frequent itemsets – supposing a minimum confidence of 10% – we get 12 rules in total – listed in Table 4.3 – amongst which are the rules $pregnant \rightarrow female$ and $blond \wedge pregnant \rightarrow female$. As is very logical, both of these

k	Frequent itemsets
1	{female}, {blond}, {pregnant}
2	{female, pregnant}, {blond, pregnant}, {female, blond}
3	{female, blond, pregnant}

Table 4.2: Frequent itemsets found by applying the FP-Growth algorithm to the FP-tree in Figure 4.2

rules will have a confidence of 100% since every pregnant person is female and it does not matter whether this pregnant person's hair colour is blond or not.

k	Association Rules	Confidence
2	$pregnant \rightarrow female$	100%
	$blond \rightarrow female$	40%
	$female \rightarrow blond$	33%
	$pregnant \rightarrow blond$	33%
	$blond \rightarrow pregnant$	20%
	$female \rightarrow pregnant$	50%
3	$blond \wedge pregnant \rightarrow female$	100%
	$female \wedge pregnant \rightarrow blond$	33%
	$blond \wedge female \rightarrow pregnant$	50%
	$pregnant \rightarrow blond \wedge female$	33%
	$female \rightarrow blond \wedge pregnant$	17%
	$blond \rightarrow female \wedge pregnant$	20%

Table 4.3: Association rules deduced from the frequent itemsets in Table 4.2

Since it is very evident that every pregnant person is female, this is a well-known dependence and it can be removed to avoid non-interesting rules.

We will start over again with the same FP-tree, but this time we will use the FP-Growth-KC algorithm instead. Like in the original FP-Growth algorithm, we will first look for frequent itemsets ending in *pregnant*, than those ending in *blond* and finally the ones that end in *female*.

First taking the prefix paths of *pregnant*, we can see that {*pregnant*} itself has a support of 3 and thus is frequent itself. Converting the prefix paths into the conditional FP-tree of *pregnant*, we get the tree in Figure 4.3.

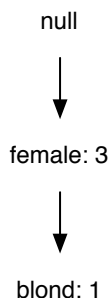


Figure 4.3: Conditional FP-tree of *pregnant*

We already know that the appearance of the node with label “*female: 3*”, will cause the normal FP-Growth algorithm to produce the frequent itemsets $\{female, pregnant\}$ and $\{female, blond, pregnant\}$, which at their turn will lead to non-interesting rules. Therefore the node with label “*female: 3*” will be removed from the tree. As the items *blond* and *pregnant* do not form a well-known dependence together, the node with label “*blond: 1*” will not be removed. The algorithm now goes on with the pruned conditional FP-tree of *pregnant*. Since this is only one single path, the only extra frequent itemset that is found, is the itemset $\{blond, pregnant\}$ with a support count of 1.

We will now look for the frequent itemsets ending in *blond*. As there are no dependences which involve being blond, the FP-Growth-KC algorithm will continue like the original FP-Growth algorithm.

At last, we will continue to search the frequent itemsets ending in *female*. As this item has a support count of 3, we can count the itemset $\{female\}$ to the frequent itemsets. Since the conditional FP-tree of this item does not exist (only the root with label *null*, no further items), we can say there are no further frequent itemsets to be found and the FP-Growth-KC algorithm has finished. The frequent itemsets are listed in Table 4.4. When we deduce the association rules from these frequent itemsets (again with a minimum confidence of 10%), we get those in Table 4.5.

4.4 Rule recovery

4.4.1 Problem

In the example in the previous section, we found that there are no more non-interesting rules generated, due to applying the FP-Growth-KC algorithm. But when we compare the rules

k	Frequent itemsets
1	{female}, {blond}, {pregnant}
2	{blond, pregnant}, {female, blond}

Table 4.4: Frequent itemsets found by applying the FP-Growth-KC algorithm to the FP-tree in Figure 4.2

Association Rules	Confidence
$blond \rightarrow female$	40%
$female \rightarrow blond$	33%
$pregnant \rightarrow blond$	33%
$blond \rightarrow pregnant$	20%

Table 4.5: Association rules deduced from the frequent itemsets in Table 4.4

generated after FP-Growth (Table 4.3) and those generated after FP-Growth-KC (Table 4.5), we see that not only the non-interesting rules are gone: also some interesting rules are not generated anymore, because of the removal of the well-known dependence.

The cause of this can be found in the fact that by removing a well-known dependence $\{A, B\}$, because $A \rightarrow B$ with 100% confidence is logical, every single itemset that contains both of these items is removed. Though, it is still possible that the rule $B \rightarrow A$ has less than 100% confidence and thus cannot be called well-known. We can see this in the example where the rule $pregnant \rightarrow female$ is very obvious and has a confidence of 100%. The rule $female \rightarrow pregnant$ only has a confidence of 50%, since it would be rather a coincidence if all females in a city would be pregnant. But the latter is not generated, because all frequent itemsets that contain both the items $pregnant$ and $female$ are removed. Other interesting rules that are not generated after applying FP-Growth-KC are:

- $blond \wedge female \rightarrow pregnant$
- $female \rightarrow blond \wedge pregnant$

4.4.2 Solution

To overcome this problem, we will try to recover the interesting rules that were lost.

We already know that the rule $A \rightarrow B$ has a confidence of 100%, because $\{A, B\}$ is a well-known dependence. Therefore recovering rules like $AC \rightarrow B$ or $A \rightarrow BC$ is unnecessary, as these still express obvious facts.

Theorem 4.1. *When a rule $A \rightarrow B$ has a confidence of 100%, the rule $AC \rightarrow B$ also has a confidence of 100%.*

Proof. The association rule $A \rightarrow B$ has a confidence of 100%, which means that the support of the itemset $\{A, B\}$ is the same as the support of $\{A\}$. When we want to calculate the confidence of the rule $AC \rightarrow B$, we get:

$$\text{conf}(AC \rightarrow B) = \frac{\text{supp}(\{A, B, C\})}{\text{supp}(\{A, C\})} = \frac{\text{supp}(\{A, C\})}{\text{supp}(\{A, C\})} = 1$$

Thus the confidence of the rule $AC \rightarrow B$ is 100% and the theorem is proven. \square

Theorem 4.2. *When a rule $A \rightarrow B$ has a confidence of 100%, then the rule $A \rightarrow BC$ has a confidence that is equal to the confidence of the rule $A \rightarrow C$.*

Proof. The association rule $A \rightarrow B$ has a confidence of 100%, which means that the support of the itemset $\{A, B\}$ is the same as the support of $\{A\}$. We will denote the confidence of the rule $A \rightarrow C$ with c , with $0\% \leq c \leq 100\%$.

The confidence of the rule $A \rightarrow BC$ then is calculated as follows:

$$\text{conf}(A \rightarrow BC) = \frac{\text{supp}(\{A, B, C\})}{\text{supp}(\{A\})} = \frac{\text{supp}(\{A, C\})}{\text{supp}(\{A\})} = c$$

\square

So we can conclude that recovering rules that have A in the antecedent and B in the consequent will not give us new information and thus is not necessary.

But since the rule $B \rightarrow A$ is not a well-known dependence and thus has a confidence less than or equal to 100%, it is necessary to recover those rules that have B in the antecedent and A in the consequent. This rule recovery takes place during the normal generation of the association rules. Why this is not afterwards will be explained later on.

First, before the rule generation, we will recover the rules that are the reverse of the well-known dependences. Therefore, we will scan the well-known dependences and generate the rule $B \rightarrow A$ (which is the reverse of $A \rightarrow B$ with a confidence of 100%). Then we will check if this rule reaches minimum confidence and if so, we will keep it. Otherwise, it will be removed.

The next step is the generation of the association rules from the found frequent itemsets. Since only rules with B in the antecedent and A in the consequent could possibly be interesting, we will try to recover those rules. During the generation of the rules, we will check for every rule (no matter what its confidence is) if it has B in the antecedent or A in the consequent. If it has B in the antecedent, we will add A to the consequent; if it has A in

the consequent, B will be added to the antecedent. When we have done so, the confidence of the new rule $BC \rightarrow AD$ needs to be calculated. Hereby we need to distinguish two different situations:

1. The rule was recovered from the rule $C \rightarrow AD$:

When we calculate the confidence of the rule $C \rightarrow AD$, we get that

$$\text{conf}(C \rightarrow AD) = \frac{\text{supp}(\{A, C, D\})}{\text{supp}(\{C\})}.$$

The confidence of the new rule can then be calculated as follows:

$$\text{conf}(BC \rightarrow AD) = \frac{\text{supp}(\{A, B, C, D\})}{\text{supp}(\{C\})} = \frac{\text{supp}(\{A, C, D\})}{\text{supp}(\{B, C\})}.$$

To get the support of the itemset $\{B, C\}$, we need to scan the frequent itemsets to find this itemset. But for the support for the itemset $\text{supp}(\{A, C, D\})$, we do not need to do this since it equals $\text{conf}(C \rightarrow AD) \times \text{supp}(\{C\})$, which were both calculated already.

2. The rule was recovered from the rule $BC \rightarrow D$:

$$\begin{aligned} \text{conf}(BC \rightarrow D) &= \frac{\text{supp}(\{B, C, D\})}{\text{supp}(\{B, C\})} \\ \text{conf}(BC \rightarrow AD) &= \frac{\text{supp}(\{A, B, C, D\})}{\text{supp}(\{B, C\})} = \frac{\text{supp}(\{A, C, D\})}{\text{supp}(\{B, C\})} \end{aligned}$$

Since we already have the support of the itemset $\{B, C\}$, we only need to scan the frequent itemsets once to find the support of $\{A, C, D\}$.

This rule recovery takes places *during* the process of the rule generation and not with the rules that were generated after the rule generation finished. This is done because otherwise it is not possible to recover *all* of the rules. Suppose we have the dataset of Table 4.1 and we use a minimum confidence of 30%. If we executed FP-Growth-KC and first generated the rules (without the recovery), we would get the rules summarized in Table 4.6(a). When we would then try to recover all of the rules that have *female* in the antecedent and *pregnant* in the consequent, we would get the association rules shown in Table 4.6(b). But as we have seen before in Table 4.3 also the rule

$$\text{blond} \wedge \text{female} \rightarrow \text{pregnant}$$

can be called an interesting rule *and* it has a confidence of 50% but it is not recovered. Normally this rule can only be recovered from the association rule

$$\text{blond} \rightarrow \text{pregnant}$$

but since this rule only has a confidence of 20% – and thus is not generated – the association rule

$$blond \wedge female \rightarrow pregnant$$

cannot be recovered. Therefore the rule recovery should take place during the rule generation.

Table 4.6: Association rules deduced from the frequent itemsets in Table 4.2 after executing FP-Growth-KC

(a) Before rule recovery

Association Rules	Confidence
$blond \rightarrow female$	40%
$female \rightarrow blond$	33%
$pregnant \rightarrow blond$	33%

(b) After rule recovery

Association Rules	Confidence
$blond \rightarrow female$	40%
$female \rightarrow blond$	33%
$pregnant \rightarrow blond$	33%
$female \rightarrow pregnant$	50%

The new algorithm FP-Growth-KC with rule recovery can be called equivalent to FP-Growth where the rules are pruned afterwards. We will not give a formal proof of this, but explain why this is so. To make it a little easier to follow, we will call FP-Growth where the rules are pruned afterwards *Method 1* and FP-Growth-KC with rule recovery *Method 2*.

When $A \rightarrow B$ is the uninteresting rule that may not be generated, then Method 1 will prune all of the rules that have A in the antecedent and B in the consequent. The rules that have B in the antecedent and A in the consequent will not be touched.

Method 2 does not generate any rule that has A in the antecedent and B in the consequent. The rules with B in the antecedent and A in the consequent are in the normal FP-Growth-KC method (without rule recovery) not generated no such frequent itemset exists where A and B appear together. Therefore we will test during the rule generation for rules that have B in the antecedent or A in the consequent, which makes it possible to recover all of the rules with B in the antecedent and A in the consequent that would otherwise be lost.

Chapter 5

Implementation in Weka

In this chapter, we present Weka-KC, which extends the data mining toolkit Weka 3.5.5 [WF05] with the option to use knowledge constraints while mining the data. Weka is a free and open source data mining toolkit that allows the user to perform the whole KDD process by providing friendly and clear graphical user interfaces. The application implements several widely used data mining algorithms for classification, clustering and association rule mining. The main objective of Weka-KC is to provide a tool for using knowledge constraints in order to reduce time and effort while mining the data.

The prototype that is presented in this chapter provides changes to the standard APriori algorithm, whereas it will be possible to use knowledge constraints. Also the FP-Growth algorithm was implemented, as well as FP-Growth-KC and the method to recover the rules that was explained in Section 4.4. For testing purposes, also the APriori and FP-Growth algorithm where the pruning happens afterwards, were implemented in Weka-KC.

Weka is fully implemented in Java and has a preprocessing module named `weka.Explorer` to read the data that have to be mined. The GUI of this module is shown in Figure 5.1. At this interface, the user can choose the data to be mined, by opening an `arff` file (the standard input file format for Weka), a website or a database. An `arff` file consists of two sections (in the order that they are mentioned):

1. Header section: this section contains the name of the relation and the names of the attributes, together with their types.
2. Data section: this section contains the data instances

An example of an `arff` file is shown in Figure 5.2.

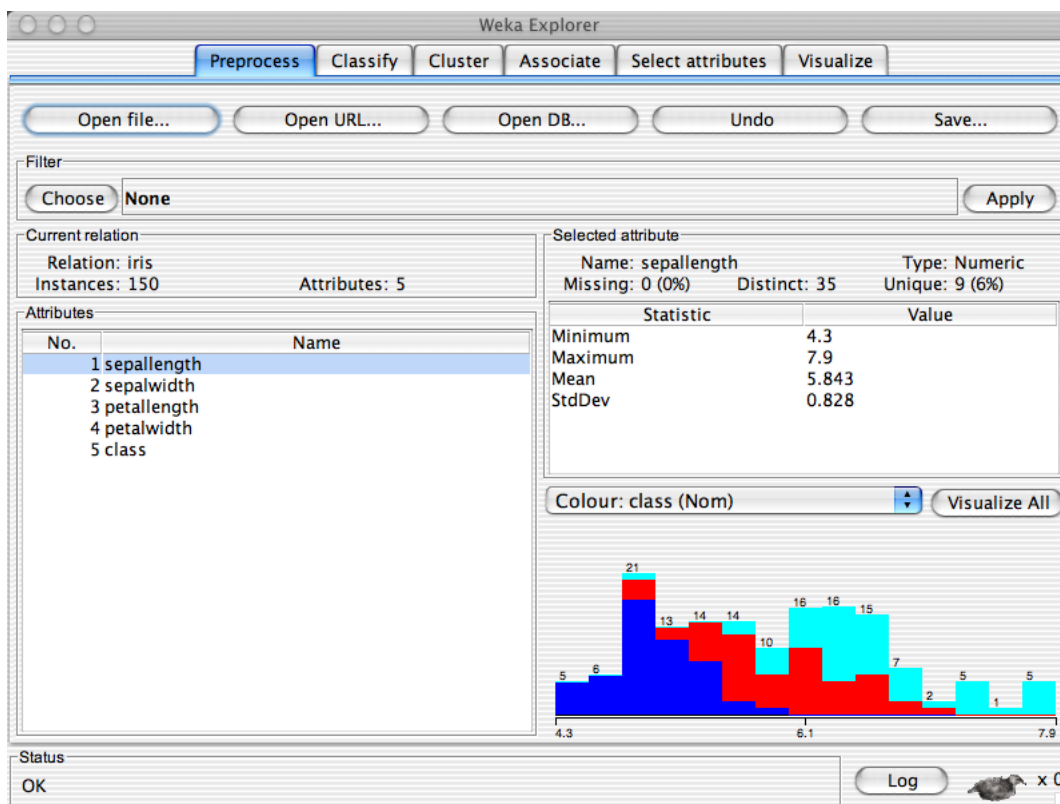


Figure 5.1: Weka Explorer GUI

```
@relation weather.symbolic

@attribute outlook sunny, overcast, rainy
@attribute temperature hot, mild, cool
@attribute humidity high, normal
@attribute windy TRUE, FALSE
@attribute play yes, no

@data
sunny,hot,high,FALSE,no
sunny,hot,high,TRUE,no
overcast,hot,high,FALSE,yes
rainy,mild,high,FALSE,yes
rainy,cool,normal,FALSE,yes
rainy,cool,normal,TRUE,no
```

Figure 5.2: Example of an arff file

5.1 Format of the dependences

As said before, we only work with dependences that contain 2 items, in other words with itemsets of the form $\{A, B\}$.

For the algorithms to use these dependences, we put them in a text file in the following format:

$$\{attribute1 = attrValue1, attribute2 = attrValue2\}$$

where *attribute1* is the item in the antecedent of the dependence and *attrValue1* its value and *attribute2* the item in the consequent of the dependence with value *attrValue2*.

Note that the order of the items between the bracelets really *does* make a difference since $attribute1 = attrValue1 \rightarrow attribute2 = attrValue2$ is the rule that has a confidence of 100% and not $attribute2 = attrValue2 \rightarrow attribute1 = attrValue1$!

5.2 Implementation of APriori-based methods

Because there was already an implementation of APriori present in the current version of Weka, we had to give our implementation of APriori another name: *APrioriModified*. This version has several options that are shown in Figure 5.3:

algoType The type of APriori-based algorithm you want to run:

- APriori: the original APriori algorithm
- APriori-KC: the APriori algorithm with the elimination of dependences a priori, during the frequent itemset generation
- APriori-prune: the APriori algorithm where all of the rules are generated and those that contain a dependence are removed afterwards

buildRules If the association rules have to be generated or not

dependences The file that contains the dependences, formatted like in Section 5.1

minConfidence The minimum confidence

minSupport The minimum support

5.3 Implementation of FP-Growth-based methods

Beside the APriori-based algorithm, we also implemented FP-Growth and its derivatives, that are summarized in the description of the options. The options of the FP-Growth algorithm in Weka-KC can be found in Figure 5.4:

algoType The type of FP-Growth-based algorithm you want to run:

- FP-Growth: the original FP-Growth algorithm
- FP-Growth-KC: the FP-Growth algorithm with the elimination of dependences, during the frequent itemset generation
- FP-Growth-KC with recovery: the same algorithm as FP-Growth-KC, except that the lost association rules are recovered
- FP-Growth-prune: the FP-Growth algorithm where all of the rules are generated and those that contain a dependence are removed after the rule generation

buildRules If the association rules have to be generated or not

dependences The file that contains the dependences, formatted like in Section 5.1

minConfidence The minimum confidence

minSupport The minimum support

showFPTree If the FP-tree needs to be printed with the results. When you chose to print the FP-tree, you can find a tree like in Figure 5.5.

With this extended version of Weka, we can now do experiments with our newly developed algorithms. The results of these experiments are summarized in the next chapter.

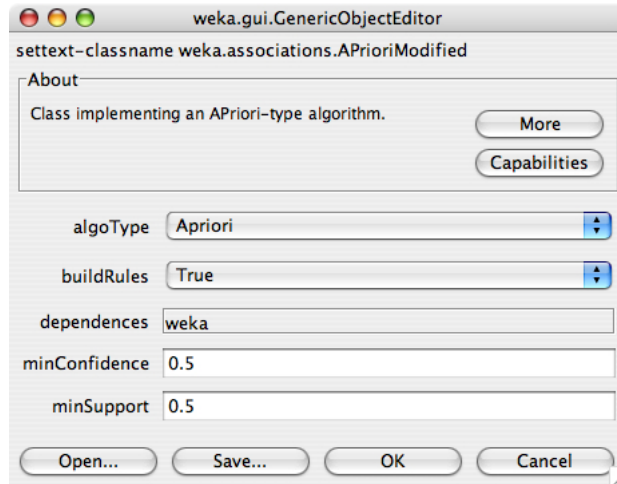


Figure 5.3: APriori

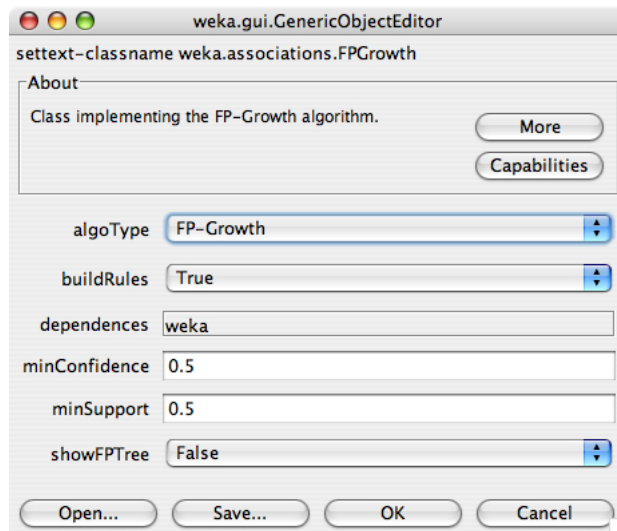


Figure 5.4: FP-Growth

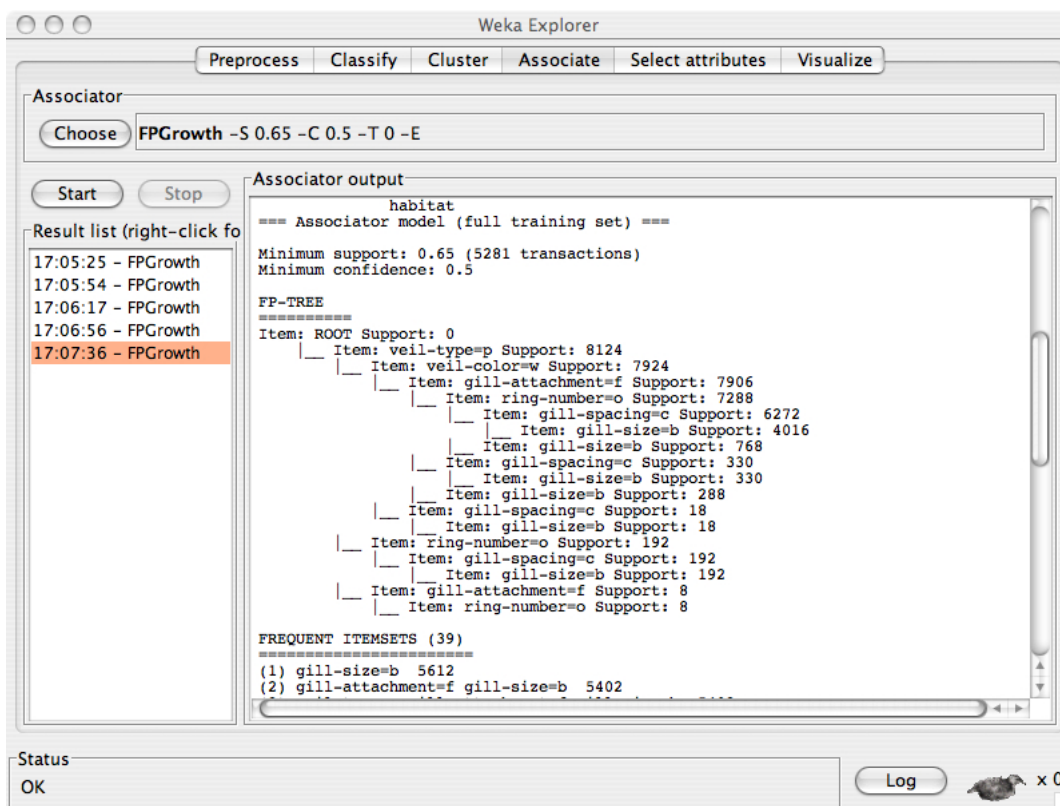


Figure 5.5: FP-tree

Chapter 6

Experiments and Evaluation

In this chapter we present many experiments that were done with Weka-KC, described in Chapter 5, in order to evaluate and validate the newly developed data mining methods. For this tests, we used two different datasets:

1. Mushroom dataset¹: this dataset contains 22 different attributes and 8124 instances. Every tuple describes the characteristics of one single mushroom. The dependences were chosen randomly. We ran the original FP-Growth method on this dataset and took 2-frequent rules that had a confidence of 100%.
2. Geographic dataset: this dataset contains 15 attributes and 514 instances. This dataset was also used in 3.3.

In every experiment, we use a number of different minimum supports, as well as a number of different well-known dependences.

6.1 Experiments with the Mushroom dataset

In this section we evaluate APriori-KC and FP-Growth-KC for the Mushroom dataset. We used a minimum confidence of 50% in each experiment. The experiments that are described in this section, were all performed with a PowerPC G4, 1.42 GHz, with 512 MB of RAM memory and Mac OSX 10.4 as operating system.

6.1.1 Evaluating APriori-KC and FP-Growth-KC for single dependence elimination

The first experiment that we performed was to run both algorithms on the Mushroom dataset where only one pair of dependences had to be removed.

¹<http://www.cs.umb.edu/~rickb/files/UCI/mushroom.arff>

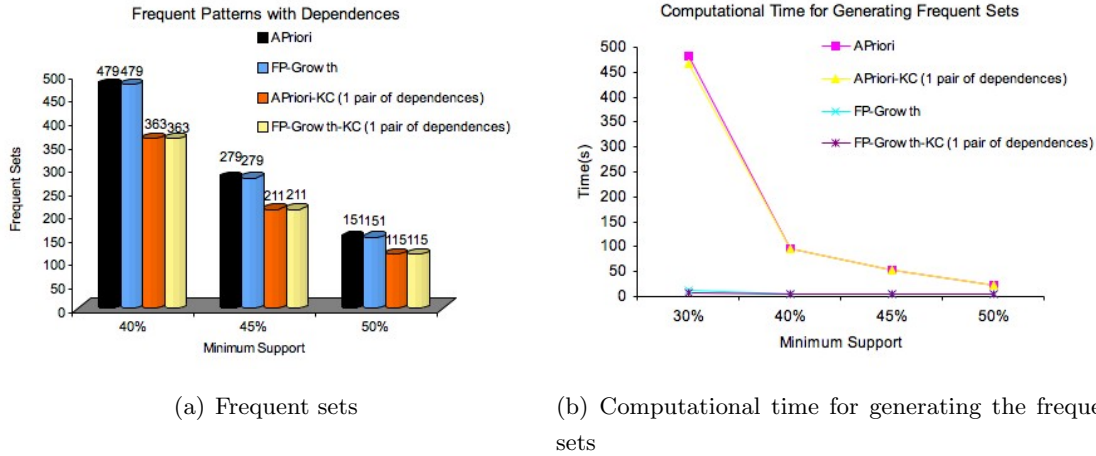


Figure 6.1: Generation of the frequent sets

Figures 6.1 and 6.2 show the result of this experiment. In Figure 6.1(a), you can see the number of frequent itemsets. It is clear that APriori and FP-Growth generate the same frequent itemsets. Here we can also see that – by the elimination of one dependence – APriori-KC and FP-Growth-KC both remove the same frequent sets and thus we see that they both generate the same frequent itemsets. The graphic also shows that the number of original frequent itemsets is reduced in almost 20%, only by removing one single dependence.

Figure 6.1(b) shows the time that is necessary to generate the frequent itemsets for the different methods. The times for both APriori and APriori-KC and for FP-Growth and FP-Growth-KC almost overlap here. This is because the higher the support becomes, the lower the number of itemsets that has to be removed and thus the closer their computational times will lay to each other.

In Figure 6.2, we see the results for generating the association rules. Here we compared 4 different methods: the original APriori method, APriori-KC with removing 1 dependence, the original FP-Growth method and FP-Growth-KC with rule recovery for removing 1 dependence. We did this to show that the FP-Growth-KC algorithm, with or without rule recovery, indeed is faster than APriori and APriori-KC. In Figure 6.2(a) we give an overview of the number of association rules that are generated. We did not put the number of association rules generated by the FP-Growth method in the graph, as these association rules are exactly the same as those generated by APriori.

What we can see in this graph is that almost 34% of the association rules that are generated by APriori (with a minimum confidence of 50%), are not generated by APriori-KC when removing one dependence and this for all different values of minimum support. But when we

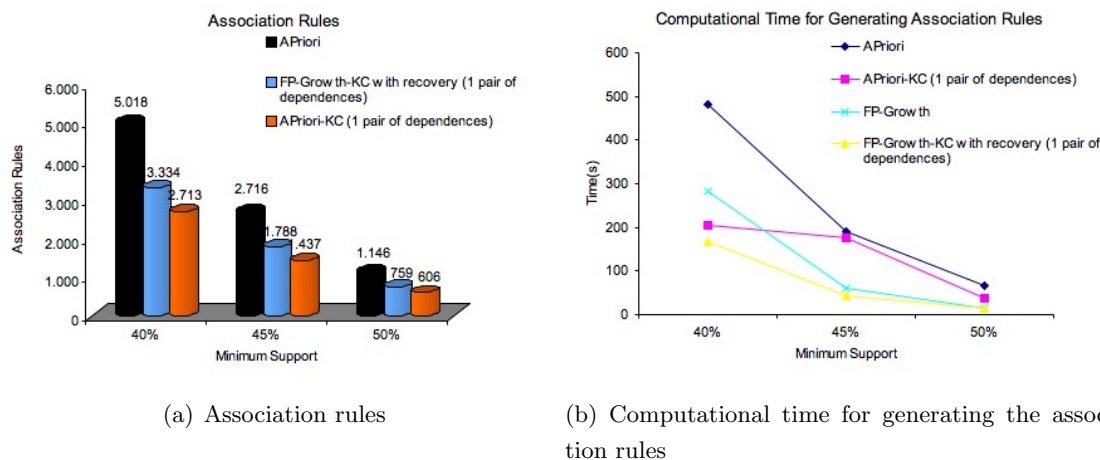


Figure 6.2: Generation of the association rules

look at the FP-Growth-KC method with the recovery, we can see that a lot of association rules were lost with just applying the APriori-KC method, as a lot of rules are recovered in the FP-Growth-KC method. When we look at the computational time for generating these association rules in Figure 6.2(b), it becomes clear that the APriori-KC method is faster than the original APriori, something we already knew. But another thing that becomes visible when looking at the graph is that not only FP-Growth-KC is faster than APriori-KC, but also the FP-Growth-KC method with the rule recovery.

6.1.2 Evaluating APriori-KC and FP-Growth-KC for the elimination of 2 dependences

The second experiment that was performed, was comparing the methods as we removed 2 pairs of dependences.

In Figure 6.3(a) we can see the number of frequent itemsets that were generated by the different methods. As you can see, APriori-KC and FP-Growth-KC still remove the same frequent itemsets, so they will give the same result for the generation of the frequent itemsets. The frequent itemsets are reduced by approximately 36% for every different value of *minSupp*.

The computational time for generating these frequent itemsets is shown in Figure 6.3(b). Here we can see that there is a bigger difference in time between APriori and APriori-KC than in Figure 6.1(b), but that this is not true for FP-Growth and FP-Growth-KC. The difference between APriori and APriori-KC is so big, because none of the frequent itemsets that contains a dependence is generated in the latter method. However, with FP-Growth-KC still all of the frequent itemsets are collected by building the FP-tree in the beginning. The

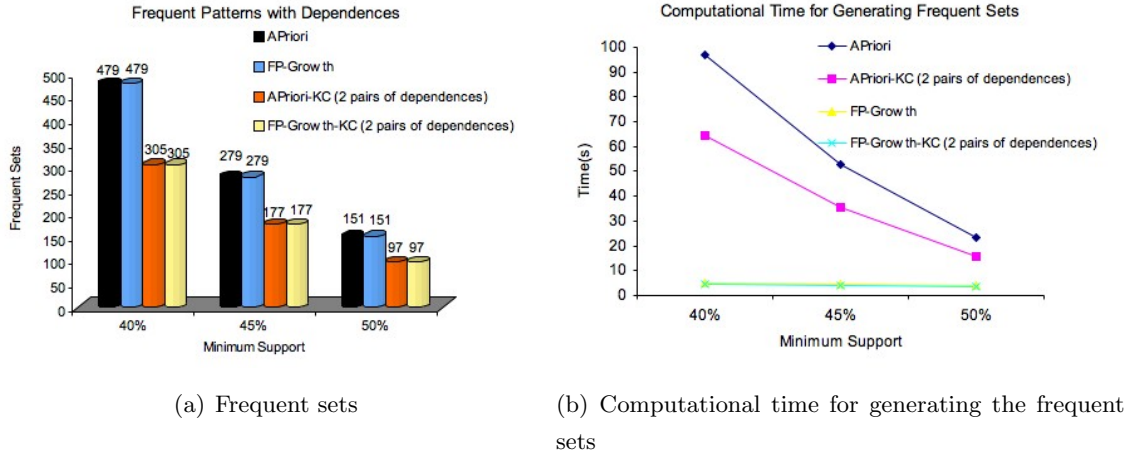


Figure 6.3: Generation of the frequent sets

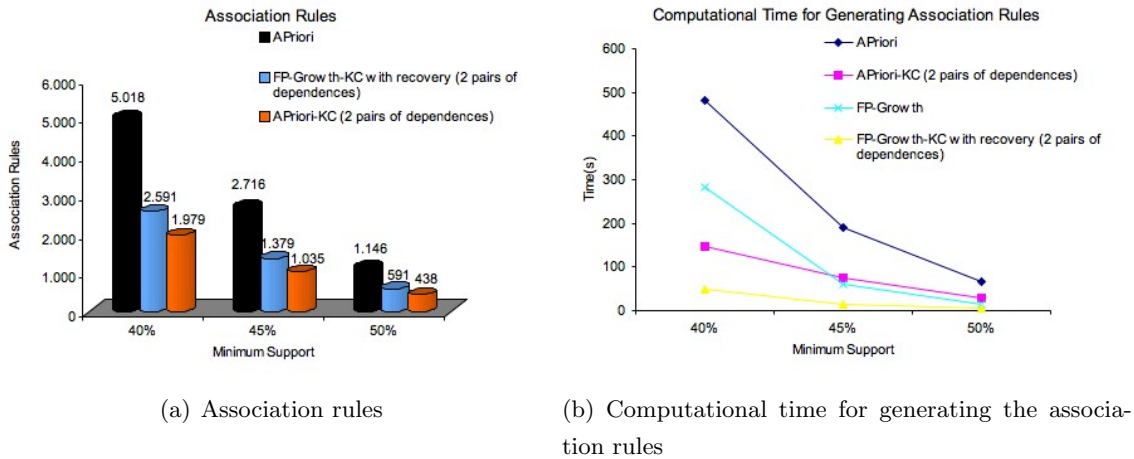


Figure 6.4: Generation of the association rules

pruning only starts when we make the different conditional FP-trees. But the beginning of FP-Growth and FP-Growth-KC is the same, as they both generate the same FP-tree, so there is not much difference in time between the two methods.

Figure 6.4 shows us the results for the generation of the association rules. Because more dependences are removed, the number of association rules will also be lower than the ones in Figure 6.2(a). We can also see here that – as with removing a single dependence – a lot of rules are lost with APriori-KC. Those rules are then recovered by the FP-Growth-KC method. Also the computational time is the best for the FP-Growth-KC method that puts back the rules, just like in Figure 6.2(b).

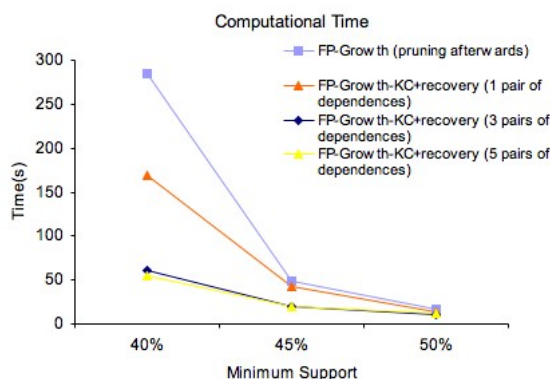


Figure 6.5: Computational time for rule recovery with FP-Growth-KC

6.1.3 Evaluating performance of rule recovery with FP-Growth-KC

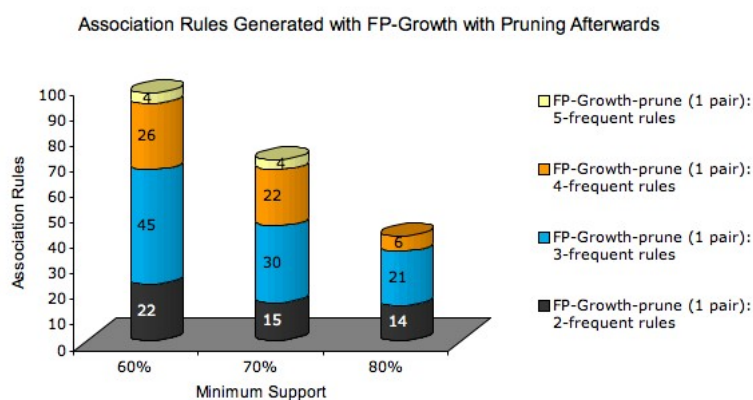
The following experiment was to evaluate the rule recovery with FP-Growth-KC. For this, we ran the original FP-Growth method and pruned the rules a posteriori. This means that we first generated *all* of the rules and after that just removed those that contained a dependence. Beside that, we also ran the FP-Growth-KC method for the elimination of 1 pair of dependences, 3 pairs and 5 pairs.

The result of this experiment is shown in Figure 6.5. The FP-Growth method is shown only once here, because the time for removing dependences stays the same whether there are 3 or 5. This can be explained by the fact that we always have to scan the whole set of association and just remove those that contain a dependence. In this graph we can see that the rule recovery may take some more time than just executing the normal FP-Growth-KC method, but it is still faster than the standard method of pruning the rules a posteriori.

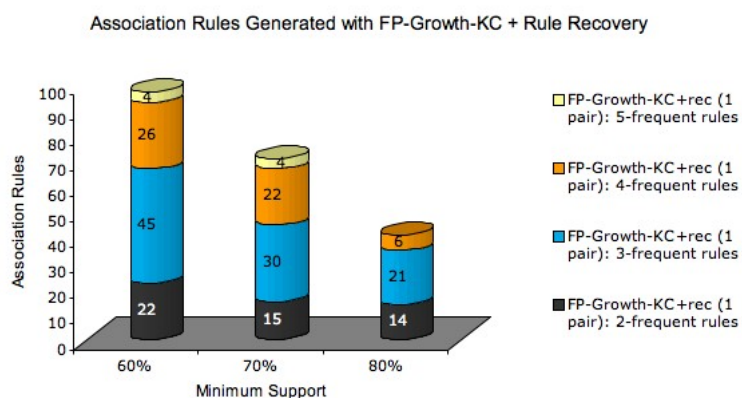
6.1.4 Evaluating FP-Growth with pruning afterwards and FP-Growth-KC with rule recovery

The last experiment was to compare the results (the association rules that were generated) from the FP-Growth algorithm that prunes the rules a posteriori to the results of the FP-Growth-KC algorithm with rule recovery. In contrast to the other experiments, we took a minimum confidence of 90% instead of 50%.

The results of this experiment can be observed in the graphs in Figure 6.6. The graph in Figure 6.6(a) illustrates the number of association rules that were generated by pruning the rules afterwards. As can be seen, the graph in Figure 6.6(b) shows exactly the same number of association rules, which means that they both produce the same association rules. The



(a) FP-Growth with pruning afterwards



(b) FP-Growth-KC with rule recovery

Figure 6.6: Generation of the association rules

only difference is that the rule recovery is much faster than pruning the rules afterwards, as we have already shown in Figure 6.5.

6.2 Experiments with the Geographic dataset

In this section we compare APriori-KC and FP-Growth-KC for the Geographic dataset. We used a minimum confidence of 90% in each experiment. The experiments that are described in this section, were all performed with a Genuine Intel CPU T2400, 1.83 GHz, with 1 GB of RAM memory and Windows XP as operating system, because the Geographic dataset was available on this machine.

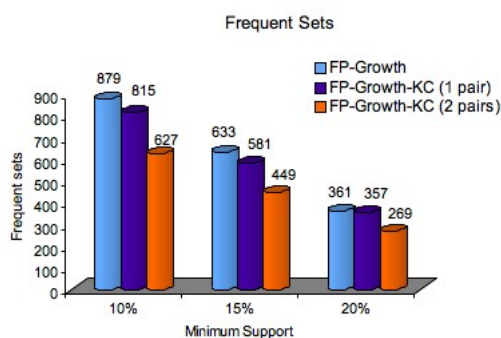


Figure 6.7: Frequent sets for the Geographic dataset

6.2.1 Evaluating the frequent set generation of the FP-Growth-KC method

In the first experiment with the Geographic dataset, we will check the efficiency of the frequent set generation for the FP-Growth-KC algorithm. Herefore, we executed the original FP-Growth algorithm and the FP-Growth-KC algorithm. The latter was executed for eliminating a single pair of dependences and for 2 pairs of dependences. As a minimum support, we took 10%, 15% and 20% instead of the 40%, 45% and 50% with the Mushroom dataset.

The total number of frequent itemsets that was generated by the algorithm can be found in the graph in Figure 6.7. This shows us that the removal of one dependence does not reduce the total number of frequent itemsets that much, in comparison with the original FP-Growth algorithm. On the other hand, removing 2 pairs of dependences makes a big difference in this case, because it reduces the total number of frequent itemsets by almost 20% for every different value of minimum support.

When we look at the time that was needed to accomplish the task of generating frequent itemsets in Figure 6.8(a), we see that the more dependences that are removed, the less time is needed to find the frequent itemsets. When we would try to do the same experiment but with the APriori-based methods, we would get the performance graph in Figure 6.8(b). The APriori-based methods to remove dependences thus need much more time to find the frequent itemsets.

6.2.2 Evaluating the rule generation and rule recovery of the FP-Growth-KC method

The next experiment is to compare the generation of association rules for the original FP-Growth method and the FP-Growth-KC method with rule recovery, one time with a single dependence elimination, another time with 2 pairs.

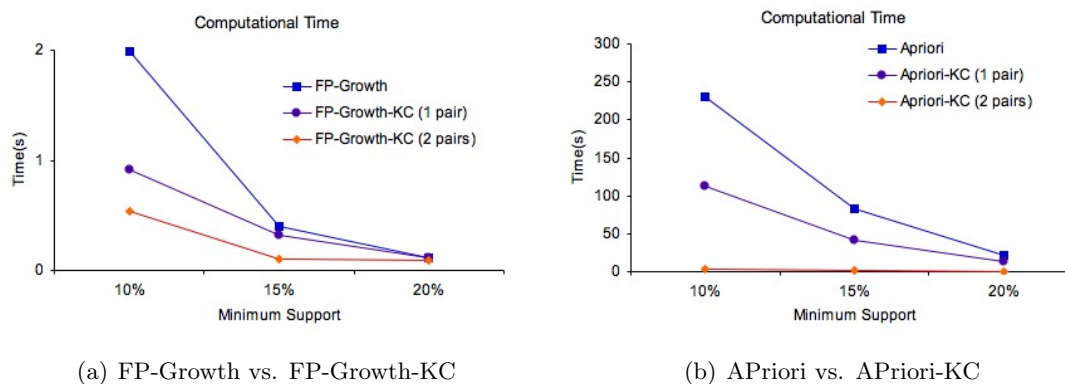


Figure 6.8: Computational time for generating the frequent sets of the geographic dataset for the APriori-based methods and the FP-Growth-based methods

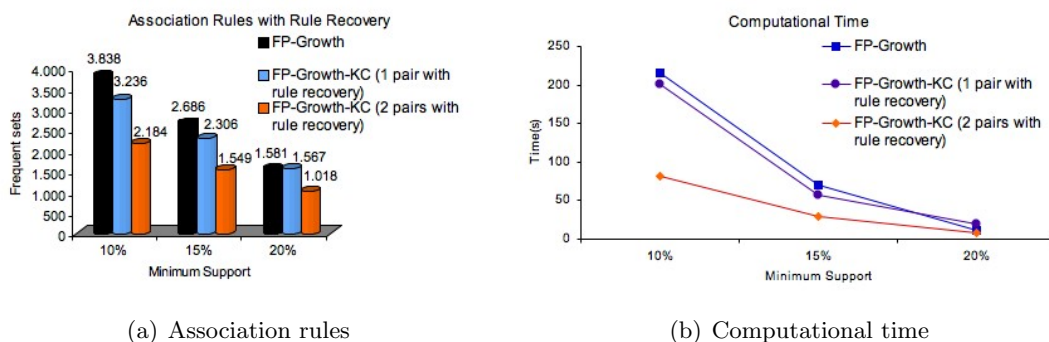


Figure 6.9: Association rule generation for the FP-Growth-based methods on the Geographic dataset

In the graph of Figure 6.9(a) we can see the number of association rules that were generated after the frequent itemsets were found. Figure 6.9(b) shows us the time that the algorithms needed to generate the association rules. The FP-Growth-KC with rule recovery comes out best most of the times, though when the support becomes higher, recovering from a single dependence elimination becomes slower than the FP-Growth algorithm.

Comparing the rule generation of APriori-KC and FP-Growth-KC with rule recovery for the elimination of 2 pairs of dependences, gives us the results in Figure 6.10. The elimination of two dependences shows that APriori-KC is more efficient than FP-Growth-KC (see Figure 6.10(b)). However, we may not forget that APriori-KC does not recover the association rules. As we can observe in Figure 6.10(a), FP-Growth-KC recovers about 20% of the rules that were not generated by APriori-KC, for any value of minimum support.

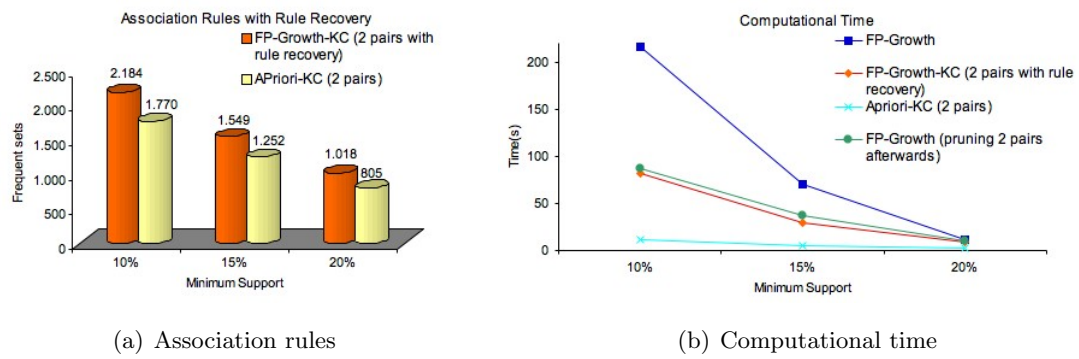


Figure 6.10: Association rules for APriori-KC and FP-Growth-KC with rule recovery

Chapter 7

Conclusions

In this thesis we mainly considered the problem of mining only interesting rules, because there are rules that are known a priori to be uninteresting and that thus will only hinder the data mining process.

We saw that there already existed a method, named APriori-KC [Bog06], to overcome this problem. It removed the itemsets that contained a dependence, what made that none of the frequent itemsets contained a dependence and thus no non-interesting association rules were generated. Since the FP-Growth algorithm was known to have a better performance than APriori, we applied the same idea of knowledge constraints to this algorithm.

Nevertheless, the new algorithm FP-Growth-KC, just like APriori-KC, still contained a problem. Though they avoided that non-interesting rules were generated, also some interesting rules were not generated through the elimination of the dependences. For this problem, we showed that it was possible to recover all of these rules.

This newly developed method has some general advantages: less frequent itemsets and association rules are generated than with the original algorithm, but still more association rules than with the APriori-KC algorithm because of the recovery of the lost rules, which is an improvement. Another advantage is that the removal of the dependences decreases the computational time that is needed to generate the frequent itemsets and the association rules.

Experiments with the implementations in Weka-KC showed that the FP-Growth-KC algorithm was indeed faster than APriori-KC. Only with the rule recovery, it could happen that FP-Growth-KC was a little slower than APriori-KC. But getting *all* of the rules instead of only a part is still more important than the time aspect here. Through experiments, we also found that running FP-Growth and pruning the rules afterwards and running FP-Growth-KC with rule recovery both gave the same results and thus can be called equivalent. Only the method with rule recovery is much faster than pruning the rules a posteriori.

Future works

This thesis only considered the elimination of dependences that contained only 2 items. For there can also exist dependences that exist out of more than 2 items, like for example the rule

$$age > 60 \wedge isProfessor \rightarrow isSeniorProfessor,$$

it would be useful to do some research on removing such dependences.

The rule recovery was only developed for the FP-Growth-KC algorithm, since we thought this would be the faster than APriori-KC. But we were very surprised to see that FP-Growth-KC with rule recovery for 2 pairs of dependences was slower than APriori-KC. Therefore we will also try to apply this rule recovery to the APriori-KC algorithm.

The new algorithm that was developed, is only based on the FP-Growth algorithm. But APriori and FP-Growth are not the only algorithms and there exist some algorithms that are even faster than the FP-Growth algorithm, like for example the CLOSET algorithm [PHM00]. Therefore we should also research if it is possible to apply the same method of the knowledge constraints to these algorithms.

Bibliography

- [AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *SIGMOD Conference*, pages 207–216. ACM Press, 1993.
- [AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994.
- [BMA] Vania Bogorny, Bart Moelans, and Luis Otavio Alvares. Filtering frequent geographic patterns with qualitative spatial reasoning. In *Proc. of the IEEE ICDE International Workshop on Spatio-Temporal Data Mining (STDM'07)*. Istanbul (2007).
- [Bog06] Vania Bogorny. *Enhancing Spatial Association Rule Mining in Geographic Databases*. PhD thesis, Universidade Federal Do Rio Grande Do Sul, 2006.
- [FPSSU96] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.
- [Goe02] B. Goethals. *Efficient Frequent Pattern Mining*. PhD thesis, transnationale Universiteit Limburg, 2002.
- [HK00] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.
- [HPYM04] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, 8(1):53–87, 2004.

- [PHM00] Jian Pei, Jiawei Han, and Runying Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.
- [PSF91] Gregory Piatetsky-Shapiro and William J. Frawley, editors. *Knowledge Discovery in Databases*. AAAI/MIT Press, 1991.
- [TSK05] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.
- [WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen, de gevraagde informatie in te vullen (en de overeenkomst te ondertekenen en af te geven).

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:

Mining of Frequent Sets using Pruning, Based on Background Knowledge

Richting: **Master in de informatica**

Jaar: **2007**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Ik ga akkoord,

Anke Jäger

Datum: **21.08.2007**