Supporting User Interaction for the Exploratory Mining of Constrained Frequent Sets

by

Teresa Mah

B.Sc., University of British Columbia, 1997

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science

in
THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

August 1999

© Teresa Mah, 1999
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date August 26, 1999
Abstract

Data mining is known to some as "knowledge discovery from large databases". It is the technology of taking large quantities of data, and searching through the data looking for previously unknown patterns of information. One can see how this can be useful to entrepreneurs and researchers of all kinds. Retailers can apply data mining to find customer shopping patterns. On a grander scale, meteorologists can use the technology to identify telltale signs of extreme weather conditions, such as tornados or hurricanes. Unfortunately, albeit so useful, data mining has not yet broken out of its shell.

There are two main reasons for this. The first reason is that the mining process is still slow, even with all the research done to optimize the algorithms. The second reason is that there has not been much work done on improving the user interaction aspect of the technology. Most of the systems created so far have resembled a black box. Input is entered in at one end of the black box, and output is received at the other end. There is no concept of human-centred exploration or control of the process, and no mechanism to specify focus in the database.

The work described here provides a glimpse of a new exploratory mining framework that encourages exploration and control. In addition, this new framework is incorporated into the first fully functional prototype capable of constrained frequent set mining. A user of the prototype can specify focus by providing constraints on data to be mined, and can view frequent sets satisfying these constraints before relationships are found. The prototype also allows users to sort or format frequent set
output, and to choose only interesting sets to find relationships on. Furthermore, frequent sets in our system can be mined between sets with different or similar domains, and users can choose other notions of relationship besides confidence. Combining this new exploratory mining paradigm with the faster, more efficient CAP algorithm, we have what we believe is the first in a new generation of fast and human-centred data mining systems.
Acknowledgements

The author would like to thank Dr. Jaiwei Han and Dr. Laks Lakshmanan for their helpful comments on this thesis work and their suggestions for improvement. Dr. George Tsiknis is also acknowledged for dedicating his time and effort in reviewing this writing. Last, but not least, the author would like to thank Dr. Raymond Ng for all his supervision and guidance without which this work could not have been completed.

TERESA MAH

The University of British Columbia
August 1999
Contents

Abstract ii
Acknowledgements iv
Contents v
List of Tables ix
List of Figures x

1 Introduction 1
1.1 Data Mining 1
1.2 Related Work 1
1.3 Significance of Research 3
1.4 Goals 4
1.5 Outline of Thesis 6

2 Related Work 7
2.1 Association Rules 9
  2.1.1 The Apriori Algorithm 10
2.2 The Association Mining Model 12
  2.2.1 The Classical Model 12
  2.2.2 The Exploratory Two-Phase Architecture 13
2.3 Constrained Frequent Set Queries
2.3.1 The Apriori+ Algorithm
2.3.2 The CAP algorithm

3 Algorithm Implementation
3.1 Programming Languages
3.2 Primary Data Structures
3.2.1 Sets
3.2.2 Lists
3.2.3 Transaction Databases
3.3 Generation of Sets
3.3.1 Generating Large One-Item Sets
3.3.2 Generating Candidate and Frequent Sets
3.3.3 Generating Candidate Pairs
3.4 Constraint Management
3.4.1 Reading Constraints
3.4.2 Applying Constraints
3.5 Optimizing for CAP
3.5.1 Antimonotone Constraints
3.5.2 Succinct Antimonotone Constraints
3.5.3 Succinct Constraints
3.5.4 Non-antimonotone Non-succinct Constraints
3.6 Relationship Calculation
3.6.1 Confidence
3.6.2 Correlation
3.6.3 Dependence
3.7 Mining Between Sets of Different Domains
3.8 Running in the Classical Model
3.9 Testing
### 3.9.1 Testing Individual Constraints

### 3.9.2 Testing Multiple Constraints

### 3.10 Calling CAP and Apriori+

### 4 Support for User Interaction

#### 4.1 Using the Two-Phase Architecture

#### 4.2 Constraint Handling

- 4.2.1 Creating Constraints
- 4.2.2 Modifying, Copying, Reordering and Deleting Constraints

#### 4.3 Displaying Candidate Sets

- 4.3.1 The Candidate Set Window
- 4.3.2 Sorting Candidate Sets
- 4.3.3 Viewing Candidate Sets
- 4.3.4 Graphing Sorted Sets

#### 4.4 Relationships

- 4.4.1 Choosing Interesting Sets
- 4.4.2 Selecting Different Metrics
- 4.4.3 Mining between sets of different types

#### 4.5 Selecting Different Algorithm Options

- 4.5.1 The Classical vs. The New Model
- 4.5.2 Apriori+ vs. CAP

#### 4.6 Database Management

- 4.6.1 Item Database Issues
- 4.6.2 Viewing the Transaction Database

#### 4.7 File Operations

#### 4.8 Other Features

- 4.8.1 Display Options
- 4.8.2 Error Checking
- 4.8.3 Constraints Viewer
4.9 Interface Implementation Issues ........................................ 102
  4.9.1 Programming Language ........................................... 102
  4.9.2 Tcl/Tk problems .................................................. 103

5 Conclusion ........................................................................ 105
  5.1 Future Work .................................................................. 107

Bibliography ......................................................................... 109
List of Tables

2.1 Characterization of 1-var Constraints .................................. 21
3.1 Item Database dbfile.sql ..................................................... 35
3.2 Weaker Constraints induced from NONE Constraints ............... 49
3.3 Contingency Table for Association Rules ............................. 53
4.1 Constraint Distance Calculation .......................................... 76
List of Figures

2.1 Apriori Algorithm .............................................. 12
2.2 Two Phase Architecture for Exploratory Association Mining ........ 15
2.3 Syntax of Constraint Constructs .................................. 17
2.4 Apriori+ Algorithm ............................................. 18
2.5 CAP Algorithm .................................................. 24

4.1 CAP Main Window .............................................. 65
4.2 Antecedent Constraints Creation Window ......................... 66
4.3 Edit Popup Menu ............................................... 68
4.4 Modify Constraint Window ..................................... 69
4.5 Copy Constraint Window ....................................... 70
4.6 Candidate Sets Window ........................................ 74
4.7 Sort Candidate Sets Window .................................. 75
4.8 Sets sorted by Frequency ...................................... 77
4.9 Maximal sets sorted by frequency ............................... 78
4.10 Sets viewed by level .......................................... 79
4.11 Chart of sort results for two criteria ......................... 80
4.12 Edit Chart Window ............................................ 81
4.13 Results of the activation/deactivation technique ................ 82
4.14 Candidate Sets Selection Window ............................. 84
4.15 Maximality Selection Window ................................ 85
4.16 Constraint Distance Selection Window .......................... 86
4.17 Specify Relationship Window ........................................ 87
4.18 Related Sets Window .................................................. 88
4.19 Class Constraint Creation Window .............................. 89
4.20 Options Window ....................................................... 90
4.21 View Metadata Window ............................................... 94
4.22 View Transaction File Window ................................. 95
4.23 Saving a File in CAP .................................................. 98
4.24 Font Selection Window ............................................... 99
4.25 Error Window .......................................................... 101
4.26 Constraints Viewer ................................................... 103
Chapter 1

Introduction

1.1 Data Mining

The boom in database technology over the past few decades has resulted in mass quantities of data. Retail outlets keep records of all their sales in transaction logs. Hospitals store patient data in large databases. Governments identify citizens by keeping track of their employment, health and family information. As our society gets even more complex, there will be more and more data to store, resulting in larger and larger databases. This phenomenon is what created interest in one of the newer research areas of databases: data mining. Also known as knowledge discovery from databases (KDD), data mining is the process of extracting useful (and otherwise unknown) patterns and associations from large databases.

1.2 Related Work

There are many different types of patterns which can occur in data. However, as data accumulates, it becomes more and more difficult for us to sift through data in hopes of searching for any of these patterns. This is why we need data mining, because it addresses this problem through automating the search, making the pro-
cess simpler, manageable and more efficient. Data mining is useful in almost all realms of life. Geographers may want to easily locate clusters of similar topography on maps. This type of data mining is called clustering. Visa companies which want to identify fraudulent spending can look to outlier detection algorithms to catch irregular behavior patterns in data. Stock brokers can use sequential data mining programs to identify which purchase sequences of stocks have generated the greatest return, or use classification algorithms to classify stocks which are profitable or not profitable. Medical researchers may want to look through patient records to find links or associations between eating habits and nervous system disorders, and consultants want to look for patterns in customer purchases; for example, when customers buy shoes, they also buy socks and shirts. These last two cases are examples of association rules, which is the focus of this author's research. Association mining is the extraction of related and frequently occurring data in databases. This type of information can be invaluable; with it, doctors can prescribe treatment and suggest appropriate eating habits to prevent various disorders, and consultants who now know which products are frequently purchased together can rearrange items in their departments to maximize sales. A true example of a supermarket rearranging their shelves to place baby products near the alcohol aisle was provoked by having applied association mining to their transaction database, and finding out that "60% of all customers who buy diapers also buy beer".

One of the first association mining algorithms, Apriori, was proposed by IBM researchers Srikant and Agrawal. Since then, many papers have been written focusing on improving the efficiency of Apriori and expanding its applicability. The paper which this research is based on, "Exploratory Mining and Pruning Optimizations of Constrained Association Rules" [17] discusses just that. In association mining, we want to find all pairs of sets within the database which are related. In Apriori, such sets have to satisfy user-supplied thresholds called support and confidence. Support is how frequently a set occurs in the database, while confidence is a measure of
how associated two sets are. Minimum support and confidence can therefore be thought of as constraints on the sets. The CAP algorithm discussed in [17] is the first proposal to allow a user to specify additional SQL-like constraints on the sets. For example, one can specify that they are interested in finding association rules for only half of their items, or that they only want to find out unknown information about their “chip” and “soda” products. The CAP algorithm uses these constraints to achieve faster performance, resulting in speedups up to 80 times greater than that of Apriori.

1.3 Significance of Research

Since Apriori was first proposed, little research has been done regarding the framework of the mining process. In particular, the classical model is lacking in two areas: user exploration and focus. One of the reasons why data mining has not entered the commercial market, despite its wide applicability and usefulness, is because it does not offer support for any user interaction within the process. We supply an association mining program with a database of transactions and a couple of parameters; then we wait for hours before results are finally displayed on the screen. Users are not given the ability to explore their data during the mining process, and there is no means by which focus (into a particular section of the database) can be described. Under the classical model, even if we only want to find information about a small section of the database, we need to run the algorithm on the whole database in order to find the desired rules. This is the reason why data mining is only popular (and even then, only mildly so) in the academic community. One must have considerable knowledge of the database and the mining process before being confident that hours spent waiting at the computer will eventually result in the information that is desired.

Consequently, it is important to reconfigure the classical model to support human-centred exploration, allowing users to explore their data, control and/or
preempt time-consuming processes, and establish focus in the database. For example, instead of mining the whole database, which is what all association mining programs implemented to date have been doing, allowing one to establish focus can concentrate the mining in one small section of the database and decrease execution time. In addition, enabling users to view their data during the mining process will allow them to decide whether or not the rules being generated are interesting or not. With a new model incorporating human-centred exploration and control, we can envision even the common grocery store manager using data mining to improve sales.

1.4 Goals

In [17], such a model has been proposed, encompassing a new exploratory two-phase architecture and the notion of constraints. The bulk of the research described here involves weaving the characteristics of this model into an appropriate design and implementation of the first fully functional prototype capable of constrained frequent set mining. With this prototype, the effectiveness, exploratory abilities and usability of the new framework truly shines.

Our CAP prototype has been the only association mining program to accomplish the following:

- allow for focus by
  - incorporating the optimized CAP algorithm to find constrained frequent sets
  - enabling users to create a variety of constrained frequent set queries, including class, frequency, domain, aggregate and two-variable constraints
  - providing an easy-to-use interface to create, modify, view, reorder, copy and delete constraints
- giving users the ability to select only interesting candidate sets to find relationships on
- providing a selection helper to assist in the selection of interesting candidate sets by frequency, constraint distance, maximality, cardinality, attribute or aggregate values

- allow for human-centred exploration and control of the association mining process by
  - splitting up the process into two phases where the first phase consists of finding the frequent sets and the second phase consists of finding relationships between pairs of sets
  - giving users the ability to reiterate through the two phases if it is found that focus is not established correctly
  - providing a utility to sort candidate pairs based on frequency or “distance” to a constraint, and to view candidates by cardinality or maximality
  - providing a graphical view of sorted candidates and the ability to activate or deactivate candidate representations in graphs based on sort constraint and cardinality
  - providing file operations for users to create, open, save, and print constraint and program output files
  - giving users the option to run under the classical model or the new exploratory model

- allow for different terms of relationships by
  - enabling users to choose from three different relationship metrics: confidence, correlation and dependence
  - giving users the ability to specify relationships between sets which come from different domains
1.5 Outline of Thesis

For the rest of this thesis, we will be discussing the unique features of the CAP prototype in more detail. In Section 2, we give a general survey of the data mining field and provide an overview of the related work which has influenced the research done on this prototype. Section 3 discusses some of the issues concerning the implementation of the CAP algorithm, and the solutions we found to some problems which surfaced. In Section 4, we talk about how we incorporated the new model of association mining into our prototype, and how the system was designed to encourage human-centred exploration, thereby increasing usability and supporting more user interaction in the mining process. Finally, Section 5 contains our conclusions, thoughts about where the prototype is headed, and future work which can be done to further enhance the prototype's features.
Data mining is still a relatively new field, but it already has many applications in the real world. It is for this reason that the author has chosen to contribute to this area, and in particular to the research related to association rules. However, besides association rules, the other fields of data mining are equally interesting and worth discussing.

Many researchers are devoted to creating more efficient outlier detection algorithms, which can find, for example, the players in the NBA who have recently not been playing their regular game, or which hockey players of all time are way above average in terms of injuries. Outlier detection algorithms have always been most efficient with two-dimensional databases. Examples of recent research on outliers includes a paper by Knorr and Ng [14] which proposes new algorithms to efficiently mine for distance-based outliers in large multi-dimensional datasets.

Classification is an extremely popular field in data mining, and is a very common term in the area of business intelligence as well. Classification algorithms are used to classify objects into specific classes. For example, a hospital could use a classification algorithm to classify patients entering the ward as critical or non-critical patients based on their symptoms. With such an algorithm, the system can detect whether a man complaining of chest problems is in danger of undergoing heart failure or has
mild heart burn problems. Classification algorithms usually produce decision trees which in turn generate sets of rules that can be used to classify the objects. Some of the more popular decision tree-based algorithms are CART, CHAID and ID3. The area of statistics has also applied neural networks to solving classification problems [8].

Spatial data mining is also a popular area of research in data mining, since spatial databases are quite large and can contain valuable data such as satellite, medical and geographical information. Usually, manually sifting through the terabytes of information looking for patterns is not desirable, or even imaginable. The field of spatial data mining is devoted to this problem - extracting unknown patterns or features about spatial data, finding regularities within the data and capturing relationships between spatial and non-spatial data. For example, Ng and Han proposed the CLARANS clustering algorithm in [19] and described how it could be applied to databases containing spatial and non-spatial data. Ester, Kriegel, Sander, and Xu later proposed DBSCAN in [11], a density based algorithm which was capable of finding “better” clusters in less time.

Sequential data mining is also an active area of research. One of the first sequential data mining algorithms was created by Srikant and Agrawal in [6] by modifying the Apriori algorithm to handle sequences. With this new version of Apriori, one could find, for example, the sequences of purchases which occur most frequently in transaction databases. Srikant and Agrawal later improved the efficiency and applicability of the algorithm by introducing the GSP algorithm [7], which was capable of handling taxonomies and time constraints between sequences. Mannila, Toivonen, and Verkamo did work in the same field, but concentrated more on finding frequent episodes (sub-sequences) in specified time windows within sequences [15]. They proposed an algorithm to find similar sub-sequences between sequences of data in time-series databases [2]. In this model, the sequences in question can be scaled or translated before similar sub-sequences are found. Sequential data mining can have
many applications, especially in the financial area, where it is usually used to find similar growth patterns in companies, stocks, or product sales.

2.1 Association Rules

In 1993, another area of research in data mining was born - the problem of finding association rules [1]. Association mining is the extraction of related and frequently occurring data in databases. Rules such as "80% of customers who purchase red wine also purchase cheese" are examples of association rules. Apriori, the first algorithm proposed to mine for association rules, was developed in 1994 by Srikant and Agrawal [3]. Since then, many different papers have been written, attempting to improve the algorithm and expanding to the cause. These papers have redefined the term "association rule" to be a general category consisting of a number of distinct varieties. Generalized association rules is one variety. A generalized association rule is a rule which can span is-a hierarchies among items. For example, suppose we have a clothing database with the categories "outerwear" and "shoes", and let "jacket" be a type of outerwear and "hiking boots" be a type of shoe. Then the rule "outerwear \(\Rightarrow\) hiking boots" is an example of a generalized association rule since "outerwear" is at a higher taxonomic level than "hiking boots". Srikant and Agrawal proposed a few algorithms in [4] to handle this, the most basic of which was a simple extension of Apriori which involved itemizing the categories and including the enumerated category IDs in any transaction which contained descendants of the category. Han and Fu modified the single-level Apriori in [13] to handle multiple-level association rules by remapping the transaction database and performing the mining by a progressive deepening of the levels. Four algorithms were proposed in the paper, each one being optimal for different data distributions.

Another type of association rule is a quantitative association rule. It is desirable to find quantitative association rules in databases which contain categorical or quantitative information. For example, the statement "50% of business people be-
tween the ages of 30 and 40 have at least 3 children” is an example of a quantitative association rule. Mining for such information is desired in databases containing, for example, survey-related data or customer information. Srikant and Agrawal suggested in [5] that such rules could be easily found by partitioning and enumerating the domain of each categorical attribute in the database and then applying Apriori, using the partitioned categories as items.

Another popular topic in association mining is developing parallel mining algorithms for finding association rules [12] [21]. Other researchers are concerned with different issues; one recent debate is the appropriateness of using confidence to assess relationship or association. Brin, Motwani and Silverstein in [9] suggested that the dependence ratio or correlation between two sets are more appropriate to calculate relationships than confidence. The algorithm they proposed involved merging the frequent set generation and correlation calculation algorithms into one to increase the pruning power of the correlation algorithm. This topic was later brought up again in [16], which discussed fast parallel searching for correlated association rules. It was this paper that provided the algorithm for chi-squared value calculation used in the CAP prototype.

2.1.1 The Apriori Algorithm

Before delving deeper into the rest of this document, we must provide a detailed explanation of association mining, since the CAP algorithm used in our prototype is an extension of the Apriori algorithm.

The classical association rule algorithm, Apriori, was first proposed in [3]. In the paper, Srikant and Agrawal state that users have to supply two thresholds to run Apriori, a minimum support and a minimum confidence value. These values are then used to generate rules such as “20% of people who purchase a VCR also purchase extended warranty”. The 20% figure in the statement is an example of the confidence value of the rule.
First, let us define the word *itemset*, which will be used quite frequently in the material to follow. An *itemset* is a set containing any number of items. A 3-itemset would therefore be a set containing three items. The minimum support of an itemset \( S \) is the minimum number of transactions (or % of transactions) in the database which contain \( S \). We call any set which satisfies the minimum support threshold *frequent*. For a pair of sets \( S \) and \( T \) to satisfy the minimum confidence threshold, \( S \cup T \) must be frequent in the database, and the probability that \( T \) is in a transaction given that \( S \) is in the transaction, \( P(T|S) \), must be greater than or equal to the minimum confidence threshold.

Given the support and confidence thresholds, Apriori first creates a list of all the items in the database and finds the subset of items from that list that are frequent. Using this frequent 1-itemset list (denoted \( L_1 \)), Apriori then creates 2-itemsets by joining the frequent 1-itemset list with itself. These joined 2-itemsets are called *candidate sets*. The candidate 2-itemsets, \( C_2 \) are then passed through the support test, to find the list of 2-item frequent sets, \( L_2 \). We join the list of 2-item frequent sets with itself to generate the 3-item candidate sets, and filter those 3-item candidate sets which pass the support threshold to make the list of 3-item frequent sets. This process continues until no more candidate sets can be created. The temporary results from this “frequent set generation phase” of Apriori is the list of frequent 1-item, 2-item, 3-item, ... \( k \)-itemsets, or \( L_1 \ldots L_k \).

In the relationship calculation phase, where we find the degree of associativity between two sets, we pair up all frequent sets with each other to create a list of *pairs* of sets. We call the left side of each pair the *antecedent* (\( S \)), and the right side the *consequent* (\( T \)). We can determine whether a set \( S \) is associated with a set \( T \) by first calculating whether \( S \cup T \) is frequent and then calculating the pair's confidence, \( P(T|S) \). If \( S \cup T \) is frequent and the pair's calculated confidence value is greater than the minimum confidence threshold, we say that set \( S \) is *associated* with set \( T \). In particular, we can say that whenever set \( S \) is in a transaction, \( X\% \)
Apriori Algorithm

1. $C_1$ consists of sets of size 1; $k = 1$; $Ans = \emptyset$;
2. while ($C_k$ not empty) {
   (a) conduct db scan to form $L_k$ from $C_k$;
   (b) form $C_{k+1}$ from $L_k$ based on $R_{freq}$; $k++$;
3. for each set $S$ in some $L_k$:
   (a) add $S$ to $Ans$

Figure 2.1: Apriori Algorithm

of the time set $T$ is also in the transaction (where $X$ is the confidence value of the pair). The output of Apriori will be the pairs of sets of items which satisfy both support and confidence thresholds. The algorithm is summarized in Figure 2.1. In the algorithm, $C_k$ is the $k^{th}$ candidate set, $L_k$ is the $k^{th}$ frequent set, $R_{freq}$ is the frequency constraint and $Ans$ is the result. A more detailed account of the Apriori algorithm can be found in Srikant and Agrawal’s original paper [3].

2.2 The Association Mining Model

2.2.1 The Classical Model

As discussed earlier, there are many problems with the classical association mining model. Users supply the database with a set of transactions, minimum support and confidence values, and run the mining algorithm. Then they sit at their terminals and wait an undetermined amount of time for association rules to appear. This lack of user interaction may not be that bad, for those who know what they are doing and can afford to (and want to) wait at their terminals for only the final output. But for the majority of us, it may be more efficient to provide views of the data generated as the algorithm is being run. This at least gives the user some feedback as to whether the sets being considered and generated are interesting or not.

The problems with the classical framework were first itemized in [17]. They are:
• Lack of user exploration and control

  - The model of association mining resembles a black box. Input is entered in at one end of the black box and received at the other end. The user is not able to explore or preempt the process. What happens if a mistake was made when entering in the data? This lack of user interaction and control can therefore result in hours of wasted time.

• Lack of focus

  - In the classical framework, all frequent sets are found, even ones which the user may not particularly care for. Suppose a user wanted to mine for frequent sets on only snack items. Under the classical framework, the Apriori algorithm would still output all frequent sets, not being able to take the constraint into account. What could have been done in five minutes would now require one hour because of the lack of focus.

• Rigid notion of relationship

  - A rigid notion of relationship is the third problem. The model should allow for other notions of relationship. For example, a user may want to use correlation (as opposed to confidence) to assess the associativity between two sets, or the user may want associations between sets of types to sets of items. The framework should fully allow for the specification of alternative types of relationships.

2.2.2 The Exploratory Two-Phase Architecture

Fixing the weaknesses of the classical model is imperative to advancing association mining to its next level. For years, there have been discussions about allowing the user to guide the mining process, but not many prototypes or programs have incorporated this idea into their design. Of those which have, the problem of the lack of focus has still not been solved. The CAP prototype is the only fully functional
program which has been designed to solve all of the above problems by following the model (proposed in [17]) below:

• open up the black box, allow the user to intervene at specific breakpoints, and provide sufficient feedback
• provide the user with a means to specify the focus through constraints, ensuring that the system only looks for information the user is interested in
• allow the user to specify different notions of relationship and different domains to mine on

The new architecture is one that encourages exploratory association mining by opening up the black box and incorporating the notion of constrained frequent set queries. In this new model, the association mining process is split up into two phases, each phase fully supporting human-centred exploration and user interaction.

In Phase I, the user specifies constraints for the antecedent and consequent. For example, one may want to mine only sets of *types* for the domain of the antecedent, and sets of *items* for the domain of the consequent. After specifying constraints and support thresholds for the antecedent and the consequent, the user runs the algorithm to compute the constrained frequent sets. The list of frequent sets for the antecedent and the consequent will be displayed as the result. At this point in the new two-phase architecture, the user can choose to sort or graph his output, refine his constraints and support thresholds, or proceed to select interesting candidate pairs to find associations. This latter action is called Phase II. In Phase II, users can choose which pairs to find relationships on by manually searching through the output and selecting pairs, or by using automated selection tools which select pairs based on criteria such as support, maximality, cardinality, attribute values, aggregate values, or calculated distance from a constraint. After selecting interesting pairs, users will have a choice of different relationship metrics to test for association. Metrics can include confidence, correlation or dependence ratio. After computing the relationships, the significant related pairs are finally displayed. In this model,
both phases I and II can be reiterated whenever the user needs to refine constraints or change metrics and thresholds. Figure 2.2, taken from [17], illustrates the new framework incorporated into the CAP prototype:

![Two Phase Architecture for Exploratory Association Mining](image)

Figure 2.2: Two Phase Architecture for Exploratory Association Mining

2.3 Constrained Frequent Set Queries

CAQs (originally standing for constrained association queries, but now more properly termed constrained frequent set queries) enable users to express focus in the new two-phase mining model. The user can use CAQs to specify class, domain, frequency and aggregate constraints to apply to the antecedent set, consequent set or both. The use of CAQs here needs a bit of explanation. The syntax for a constraint is: \{(S_1, S_2)|R\} where \(R\) is a set of constraints on set variables \(S_1\) and \(S_2\). Set variables can also have the form \(S.A\) where \(A\) is an attribute. Now, if we assume we have an item database \(ITEM(ITEMID, ITEMNAME, TYPE, PRICE)\), then the set variable \(S.TYPE\) would represent the types of the items in \(S\). Alternatively, we can specify that the domain of \(S\) come directly from the \(TYPE\) attribute using the class constraint \(S \subset TYPE\).

As another example, suppose we want to find all pairs of itemsets where the minimum of the price of \(S_1\) is less than 3 and the count of items in \(S_2\) is greater
than 2. Assuming that the domains of $S_1$ and $S_2$ come from the ITEMID attribute, we would express this in a CAQ as:

$$\{(S_1, S_2) \mid \min(S_1.PRI CE) < 3 \& \text{count}(S_2) > 2\}$$

We can also express mining on different domains with CAQs. The following constraint:

$$\{(S, T) \mid S \subseteq \text{ITEMID} \& S.TY PE = \text{snacks} \& T \subseteq \text{TY PE} \& \text{snacks} \notin T\}$$

finds sets of snack items and non-snack types. This CAQ is an example of how we can use constraints to relax the notion of relationship in the new model. Under the classical model, we would not have been able to specify mining for associations between items and types; the classical model only supports mining between sets with similar domains.

Constrained frequent set queries can represent many different kinds of constraints. Single variable (1-var) constraints specify constraints on only one set variable; two variable (2-var) constraints specify constraints between two different set variables. All these different constrained frequent set queries are fully supported in the CAP prototype. Figure 2.3 shows the different kinds of single and two variable constraints.

### 2.3.1 The Apriori$^+$ Algorithm

The prototype has been implemented to allow users to interactively choose between running Apriori$^+$ or CAP to find candidate pairs. Apriori$^+$ is an extension of Apriori incorporating constraint handling. The simplest way to implement Apriori$^+$ is to test if a set satisfies all constraints after it is found to be frequent; infrequent sets do not need to be tested. In addition, a frequent set which does not satisfy all the constraints is not displayed in the output. Apriori$^+$ is sufficient for all intents and purposes if the goal is to incorporate constraints and generate focus; however, issues
C is a conjunction of constraints on S1, S2 drawn from the following class of constraints.

1. **Single Variable Constraints**: A single variable (1-var) constraint is of one of the following forms.
   
   (a) **Class Constraint**: It is of the form \( S \subseteq A \), where \( S \) is a set variable and \( A \) is an attribute. It says \( S \) is a set of values from the domain of attribute \( A \).

   (b) **Domain Constraint**: It is of one of the following forms.
      
      i. \( S \theta v \), where \( S \) is a set variable, \( v \) is a constant from the domain that \( S \) comes from, and \( \theta \) is one of the boolean operators \( =, \neq, \leq, \geq \). It says every element of \( S \) stands in relationship \( \theta \) with the constant value \( v \).
      
      ii. \( v \theta S \), where \( S, v \) are as above, and \( \theta \) is one of the boolean operators \( \in, \notin \). This simply says that the element \( v \) belongs to (or not) the set \( S \).
      
      iii. \( V \theta S \), or \( S \theta V \), where \( S \) is a set variable, \( V \) is a set of constants from the domain \( S \) ranges over, and \( \theta \) is one of \( \subseteq, \subset, = \) or their corresponding negations.

   (c) **Aggregate Constraint**: It is of the form \( \text{agg}(S) \theta v \), where \( \text{agg} \) is one of the aggregate functions \( \min, \max, \text{sum}, \text{count}, \text{avg} \) and \( \theta \) is one of the boolean operators \( =, \neq, \leq, \geq \). It says the aggregate of the set of numeric values in \( S \) stands in relationship \( \theta \) to \( v \).

2. **Two Variable Constraints**: A two variable constraint (2-var) is of one of the following forms.
   
   (a) \( S_1 \theta S_2 \), where \( S_1 \) is a set variable and \( \theta \) is one of \( \subseteq, \subset \) or their corresponding negations.

   (b) \( (S_1 \circ S_2) \theta V \), where \( S_1, S_2 \) are set variables, \( V \) is a set of constants or \( \emptyset \), \( \circ \) is one of \( \cup, \cap \), and \( \theta \) is one of \( \subseteq, \subset, = \) or their corresponding negations.

   (c) \( \text{agg}_1(S_1) \theta \text{agg}_2(S_2) \), where \( \text{agg}_1, \text{agg}_2 \) are aggregate functions, and \( \theta \) is one of the boolean operators \( =, \neq, \leq, \geq \).

Figure 2.3: Syntax of Constraint Constructs
Apriori\textsuperscript{+} Algorithm

1. $C_1$ consists of sets of size 1; $k = 1$; $Ans = \emptyset$;
2. while ($C_k$ not empty) {
   (a) conduct db scan to form $L_k$ from $C_k$
   (b) form $C_{k+1}$ from $L_k$ based on $R_{freq}$; $k++$
3. for each set $S$ in some $L_k$:
   (a) add $S$ to $Ans$ if $S$ satisfies $R_{other}$

Figure 2.4: Apriori\textsuperscript{+} Algorithm

of efficiency and speed are also important in data mining. The Apriori\textsuperscript{+} algorithm takes even longer to run than Apriori because of the extra overhead incurred when handling constraints. Figure 2.4 summarizes the Apriori\textsuperscript{+} algorithm. In the algorithm, $C_k$ is the $k^{th}$ candidate set, $L_k$ is the $k^{th}$ frequent set, $R_{freq}$ is the frequency constraint, $R_{other}$ are other constraints and $Ans$ is the result. The reader can look to the CAP paper [17] if they want more details on the efficiency of Apriori\textsuperscript{+} and its hybrids.

2.3.2 The CAP algorithm

Optimizations

The prototype runs fast because by default, it executes the optimized CAP algorithm to find constrained frequent sets. We use CAP because it is designed to take advantage of two properties of constraints to achieve greater pruning power during the candidate generation and pruning phases.

The first property of constraints is antimonotonicity. When a set violates an antimonotone constraint, all its supersets also violate the constraint. Frequency is an example of an antimonotone constraint; the reason why we can prune away infrequent candidate sets is because we are guaranteed that all supersets of the infrequent set will also be infrequent due to the antimonotone nature of the constraint.
Similar to the frequency constraint, we can use the antimonotonic nature of other constraints to prune away candidate sets (before counting) which do not satisfy the constraints.

The second property of constraints is succinctness. A constraint is succinct if you can use it to succinctly characterize the set of all itemsets that satisfy its criteria. For example, the constraint \( \min(S.\text{ITEMID}) > 5 \) is a succinct constraint because all candidate sets satisfying it can be succinctly characterized and generated. The 1-item candidate sets satisfying the constraint are all items with ITEMID greater than 5. The 2-item candidate sets are all sets created by joining the one-item candidate sets with each other, and etc. To succinctly generate sets, we need to be able to characterize succinct powersets, which are lattices of candidate sets satisfying succinct constraints. Every succinct constraint has a corresponding member generating function (MGF) which can be used to enumerate all and only elements of the succinct powerset of the constraint. The MGF itself is expressed in terms of selection predicates, which are predicates that occur in the syntax of selection in relational algebra. For example, the predicate \( \sigma_{\text{TYPE} = 'snacks'}(ITEM) \) selects all items from the database whose type is snacks. An MGF itself is expressed in the form:

\[
\{X_1 \cup \ldots \cup X_n \mid X_i \subseteq \sigma_{p_i}(ITEM), 1 \leq i \leq n, \exists k \leq n : X_j \neq \emptyset, 1 \leq j \leq k\}
\]

for some \( n \geq 1 \) and some selection predicates \( p_1, \ldots, p_n \). Let us suppose that we have the constraint \( S.\text{PRICE} > 100 \). This constraint reads, "find all sets whose items have a price greater than 100 dollars." To succinctly generate sets satisfying this constraint, all we have to do is find all items whose price is > 100, since all sets of any size \( k \) must contain only items whose price is greater than 100 dollars. Therefore, our selection predicate is \( \sigma_{\text{PRICE} > 100}(ITEM) \). This predicate yields a set of items which we will call \( \text{Item}_1 \). Then, our MGF for the constraint \( S.\text{PRICE} > 100 \) is:

\[
\{X \mid X \subseteq \text{Item}_1, X \neq \emptyset\}
\]
Now, suppose we have the constraint $S.TYPE \supseteq \{\text{snacks, sodas}\}$. We are looking for all sets which contain at least one snack item and one soda item. To succinctly generate these sets, we need to find all items whose type is snacks, all items whose type is sodas, and all items whose type is neither snacks nor sodas. Let the following be the results from the respective selection predicates:

- Item$_2 = \sigma_{TYPE='snacks'}(ITEM)$
- Item$_3 = \sigma_{TYPE='sodas'}(ITEM)$
- Item$_4 = \sigma_{TYPE \neq \text{sodas} \land TYPE \neq \text{snacks}'}(ITEM)$

Our MGF would then be:

$$\{X_1 \cup X_2 \cup X_3 \mid X_1 \subseteq \text{Item}_2 \land X_1 \neq \emptyset \land X_2 \subseteq \text{Item}_3 \land X_2 \neq \emptyset \land X_3 \subseteq \text{Item}_4\}$$

The CAP prototype takes advantage of MGFs when generating succinct candidate sets. If we have multiple succinct constraints, we can generate the combined powerset satisfying all the constraints by merging their MGFs using the following lemma taken from [17]:

**Lemma 1** Let $C_1, C_2$ be two succinct constraints with respective MGFs : \{${S_i \cup \ldots \cup S_m | S_i \subseteq \sigma_{p_i}(ITEM), 1 \leq i \leq m, \exists m' \leq m : S_i \neq \emptyset, 1 \leq i \leq m'}$, and ${T_i \cup \ldots \cup T_n | T_i \subseteq \sigma_{q_i}(ITEM), 1 \leq i \leq n, \exists n' \leq n : T_i \neq \emptyset, 1 \leq i \leq n'}$. Then : \{${R_{ij} \cup \ldots \cup R_{mn} | R_{ij} \subseteq \sigma_{p_i \land q_i}(ITEM), 1 \leq i \leq m, 1 \leq j \leq n, \land R_{kl} \neq \emptyset, 1 \leq k \leq m', 1 \leq l \leq n'}$ is an MGF for $C_1 \land C_2$.

Each set $R_{ij}$ in the above formula can also be thought of as the intersection of sets $S_i$ and $T_j$. Table 2.1 lists the different 1-variable constraints and whether or not they are antimonotone and/or succinct.

By making use of the anti-monotonicity and succinctness properties of constraints, we can classify constraints into four cases and handle each case separately to maximize the pruning powers of each:

- antimonotone and succinct (SAM) constraints
- succinct non-antimonotone (SUC) constraints
- antimonotone non-succinct (AM) constraints
- non-antimonotone non-succinct (NONE) constraints
1-var Constraint | Anti-Monotone | Succinct
---|---|---
\( S \theta v, \theta \in \{=, \leq, \geq\} \) | yes | yes
\( v \in S \) | no | yes
\( S \supseteq V \) | no | yes
\( S \subseteq V \) | yes | yes
\( S = V \) | partly | yes
\( \min(S) \leq v \) | no | yes
\( \min(S) \geq v \) | yes | yes
\( \min(S) = v \) | partly | yes
\( \max(S) \leq v \) | yes | yes
\( \max(S) \geq v \) | no | yes
\( \max(S) = v \) | partly | yes
\( \text{count}(S) \leq v \) | yes | weakly
\( \text{count}(S) \geq v \) | no | weakly
\( \text{count}(S) = v \) | partly | weakly
\( \text{sum}(S) \leq v \) | yes | no
\( \text{sum}(S) \geq v \) | no | no
\( \text{sum}(S) = v \) | partly | no
\( \text{avg}(S) \theta v, \theta \in \{=, \leq, \geq\} \) | no | no
(frequency constraint) | yes | no

Table 2.1: Characterization of 1-var Constraints

**Antimonotone and succinct constraints**

In CAP, constraints that are both antimonotone and succinct can induce pruning at the very beginning of the mining process. This pruning occurs at level 1; in this case, the list of 1-item candidate sets \( C_1 \) is replaced by \( C^*_1 \) where \( C^*_1 \) is the set of 1-item candidate sets which satisfy all antimonotone and succinct constraints (Strategy I).

**Succinct non-antimonotone constraints**

Succinct non-antimonotone constraints are probably the most difficult to handle in CAP. If we assume we are dealing with only one succinct constraint with an MGF of the form:

\[
\{S_1 \cup \ldots S_m \cup S_n \mid S_1 \subseteq \sigma_{p_1}(Item) \& S_1 \neq \emptyset \& \ldots \& S_m \subseteq \sigma_{p_m}(Item) \& S_m \neq \}
\]
\[ \emptyset \cup S_n \subseteq \sigma_{p_n}(\text{Item}) \]

where frequent sets must contain at least one item from each of \( S_1 \) to \( S_m \), and optionally from \( S_n \), then the following steps (Strategy II) handle succinct constraints:

- Define \( C_1^{S'} = \sigma_{p_1}(\text{Item}) \), \( C_2^{S'} = \sigma_{p_2}(\text{Item}) \) ... \( C_m^{S'} = \sigma_{p_m}(\text{Item}) \), and \( C_n^{S'} = \sigma_{p_n}(\text{Item}) \). For \( k = 1 \ldots n \), generate the corresponding level 1 frequent sets \( L_1^{S'} \), where \( L_1^{S'} \) consists of those sets in \( C_1^{S'} \) that are frequent.

- For \( k = 2 \ldots m \), define \( C_k = L_{k-1} \times L_{k-1} \). When \( k = 2 \) and \( m \) is greater than 1, \( C_2 = \bigcup (L_1^{S'} \times L_1^{S'}) \) where \( a, b = 1, \ldots, m \) and \( a \neq b \).

- For \( k \geq n \), append to sets in \( L_{k-1} \) each item in \( \bigcup (L_j^{S'}) \) for \( j = 1 \ldots n \) to create a set of \( k \)-item sets. A \( k \)-item set \( T \) is in \( C_k \) iff:

\[ \forall T' : T' \subseteq T \text{ and } |T'| = k - 1 \text{ and } T' \cap L_1^{S'} \neq \emptyset \text{ and } L_1^{S'} \neq \emptyset \text{ and } \ldots \text{ and } L_1^{S'} \neq \emptyset \Rightarrow T' \in L_{k-1} \]

- Generate \( L_k \) where \( L_k \) consists of those sets in \( C_k \) that are frequent

### Antimonotone non-succinct constraints

Antimonotone non-succinct constraints are handled in the same way as the frequency constraint. Before counting, test that each candidate set satisfies all antimonotone non-succinct constraints. We can then drop a candidate set from counting if it does not satisfy all antimonotone non-succinct constraints (Strategy III).

### Non-antimonotone non-succinct constraints

Similar to how they are handled in Apriori\(^+\), frequent sets in CAP are tested for satisfaction of non-antimonotone non-succinct constraints after they are generated. The only difference is that in CAP, we take advantage of any weaker constraints which can be induced from the non-antimonotone non-succinct constraints. For example, if we have the non-antimonotone non-succinct constraint \( \text{avg}(S.ITEMID) < v \), we can deduce the weaker succinct constraint \( \text{min}(S.ITEMID) < v \). Depending on whether the weaker induced constraint is antimonotone and/or succinct, strategies
I-III can be used to generate frequent sets with the weaker constraint. After generation of frequent sets from the weaker constraint, the frequent sets still need to be tested for satisfaction of the original non-antimonotone non-succinct constraints (Strategy IV).

Handling Multiple Constraints

The strategies described above were generalized for single 1-variable constraints. It is not always the case that we will only have one constraint of each type; therefore, strategies to handle multiple constraints of each type were given in [17]. To handle multiple constraints, we have to do the following:

**Antimonotone succinct constraints** Apply Strategy I with the combined MGF of all the antimonotone succinct constraints

**Succinct non-antimonotone constraints** Apply Strategy II with the combined MGF of all the succinct non-antimonotone constraints

**Antimonotone non-succinct constraints** Apply Strategy III for all antimonotone non-succinct constraints

**Non-antimonotone non-succinct constraints** Apply Strategy IV to induce weaker constraints for each non-antimonotone non-succinct constraint

Two-Variable Constraints

The optimizations of the CAP algorithm described in [17] can only be applied on 1-variable constraints. Recent work presented at SIGMOD '99 by the authors of [17] discusses how to optimize CAP even further with some of the properties of two-variable constraints. In particular, the term quasi-succinctness has been introduced to describe two-variable constraints that can be reduced to two succinct one-variable constraints to optimize pruning and further increase performance of the CAP algorithm. The optimizations discussed in the SIGMOD '99 paper have
not been incorporated into this work, but more details regarding the matter can be found in [18].

The CAP Algorithm

The CAP algorithm that the prototype uses is now presented in its official form. If we let \( C_{sam} \) be the set of succinct antimonotone constraints, \( C_{suc} \) the set of succinct non-antimonotone constraints, \( C_{am} \) the set of antimonotone constraints and \( C_{none} \) the set of non-antimonotone non-succinct constraints, then we will get the algorithm shown in Figure 2.5.

**CAP Algorithm**

1. if \( C_{sam} \cup C_{suc} \cup C_{none} \) is non-empty, prepare \( C_1 \) as indicated in Strategies I, II and IV; \( k = 1 \);  
2. if \( C_{suc} \) is non-empty \{
   (a) conduct db scan to form \( L_1 \) as indicated in Strategy II;  
   (b) form \( C_2 \) as indicated in Strategy II; \( k = 2 \); \}
3. while \( (C_k \) not empty) \{
   (a) conduct db scan to form \( L_k \) from \( C_k \);  
   (b) form \( C_{k+1} \) from \( L_k \) based on Strategy II if \( C_{suc} \) is non-empty, and Strategy III for constraints in \( C_{am} \); \}
4. if \( C_{none} \) is empty, \( Ans = \bigcup L_k \). Otherwise, for each set in some \( L_k \), add \( S \) to \( Ans \) iff \( S \) satisfies \( C_{none} \)

Figure 2.5: CAP Algorithm
Chapter 3

Algorithm Implementation

One of the main features of the CAP prototype is that it is the first association mining program capable of handling constrained frequent set queries and the first program to use the CAP algorithm, optimized to find constrained frequent sets. This optimization can be shown when running the same set of constraints under CAP, and then under Apriori+, which is also implemented into the prototype. Depending on the constrained frequent set query, the prototype running CAP can reach execution times up to 80 times faster than that of Apriori+

One of the first goals in creating this prototype was implementing the Apriori+ and CAP algorithms. The implementations of the two were similar, except for the differences in how CAP handles constraints based on their antimonotonicity and succinctness. Therefore, Apriori+ was implemented first and the optimizations of CAP were added on afterwards to create the CAP algorithm. The following sections will discuss some of the issues addressed during implementation of the algorithms and how certain facets of the algorithms were coded.
3.1 Programming Languages

The first issue faced was which programming language should be used to implement the algorithms. Since earlier research done on CAP [20] involved the C implementation of Apriori and a simplified CAP program (capable of handling only a handful of constraints), the author used this previous implementation and extended it to include all the functionality of CAP as originally proposed in [17].

3.2 Primary Data Structures

3.2.1 Sets

The first step in any implementation is deciding on the format of the primary data structures. The basic element in association mining, and in both the Apriori and CAP algorithms is the set. For set representation, we chose a static structure over a dynamic one (i.e. linked list) because static structures were found to be more efficient and faster. Previous work done on CAP also experimented with the use of bit vectors to represent sets, but it was concluded that these structures took up too much memory space. Sets in our algorithms are therefore represented by the C structure:

```c
typedef struct {
    short int set[MAXSETSIZE]; /* the set of integers */
    short int count; /* number of items in the set */
} SetList;
```

The `SetList` structure shown above holds only integers. This is because the data structure, created during the earlier implementation of Apriori, never needed to represent string values. However, because the constraint handling in CAP and Apriori\(^+\) requires string comparisons, a set data structure for strings, `StrList`, was created. Like `SetList`, `StrList` can hold up to MAXSETSIZE number of strings.
3.2.2 Lists

Besides set structures, we also needed to create a "list" data structure to hold SetLists used during candidate generation. We called this data structure the LIList structure:

```c
typedef struct {
    int size; /* number of items in each set of the list */
    int max;  /* maximum number of sets allowable */
    int count; /* number of sets in the list */
    int *Support; /* pointer to support values */
    SetList *SL; /* pointer to sets */
} LIList;
```

Whenever a list is created, it is allocated enough memory to hold a maximum number of sets and support values (as dictated by the max variable). At first, LILists were used in the algorithm just to hold candidate and frequent set lists, but as the algorithm was expanded to incorporate constraint handling, LILists were also used to hold temporary results of queries and filters performed on sets.

3.2.3 Transaction Databases

The next step in the implementation was deciding on the format of the transaction database. In order to decrease execution time, we wanted the transaction database reading module to be implemented so that the database is accessed as seldomly as possible. It was therefore decided to format transaction databases as binary files, separated into pages. We paged the transaction database because:

- we do not know how big any one transaction is (it could contain anywhere from 1 ... thousands of items), and
- we can read in multiple pages at a time, thereby minimizing the number of times we have to access the database

The transaction reading package reads 1400 pages from the transaction database at a time. To read transactions, each page is fed through a page parser which parses
The page to extract the individual transactions. A transaction itself is represented by the data structure:

```c
typedef struct {
    int count; /* number of items in the set */
    int *set; /* pointer to items in the set */
} Trans;
```

Basically, during candidate counting, a subset function uses these structures to test if candidate sets of type SetList are contained in transaction sets of type Trans.

### 3.3 Generation of Sets

The general implementation of candidate set generation was quite simple, and involved little more than what was literally described in the Apriori algorithm. Below is a brief discussion of how we generated large one-item sets and k-item candidate and frequent sets.

#### 3.3.1 Generating Large One-Item Sets

Currently, the prototype is hardcoded to read from only one item database, `dbfile.sql`. `dbfile.sql` is a small item database containing the fields `ITEMID`, `TYPE`, `ITEMNAME`, and `PRICE`. The `ITEMID` and `PRICE` fields are integers, while the `TYPE` and `ITEMNAME` fields are strings. The database itself consists of 40 items. Base Apriori subsequently assumes that it is working with item IDs starting with 1 and ending at 40. It creates a SetList to store each item, and scans the transaction database to find the support of the sets. It then compares the support of each itemset to the minimum support and outputs only those 1-itemsets which are greater than or equal to the minimum support value. These 1-itemsets are stored in a current frequent set list (LIList), and are also appended to the list of all frequent sets.
3.3.2 Generating Candidate and Frequent Sets

To generate candidate 2-itemsets, the algorithm joins the frequent 1-itemsets with themselves to create another list, but this time, of 2-itemsets. Apriori then finds the support of sets in this list by reading the transaction database, and testing the new found support of each set against the minimum support value. The new list of frequent 2-itemsets becomes the current frequent set list and gets appended to the list of all frequent sets. To generate candidate k-itemsets, the algorithm joins the (k - 1 item) sets in the current frequent set list with themselves, checks to see that every (k-1) subset of each generated k-item candidate set is frequent, finds the support on the candidate sets, and then compares the actual support of each set against the minimum support value. The new list of frequent itemsets becomes the current frequent set list and is appended to the list of all frequent sets. This of course continues until no more candidate sets can be generated.

3.3.3 Generating Candidate Pairs

Both the CAP and Apriori algorithms can generate frequent sets for just the antecedent, just the consequent or both the antecedent and consequent, depending on the arguments given at the command line. When the user specifies that he wants two-variable answers, CAP/Apriori are called twice - once to find frequent sets for the antecedent, and once to find frequent sets for the consequent. The algorithms then do a cross product of the answer sets to get all the {antecedent, consequent} pairs.

3.4 Constraint Management

Our CAP prototype is the only program which incorporates constraint handling into the mining process. Since no prototype or program has previously implemented this functionality, there was no model we could follow to design the implementation of
constraints. Therefore, the design of the constraint data structures took more time, but produced some general data structures to hold constraints of all types.

3.4.1 Reading Constraints

Since Apriori\textsuperscript{*} and CAP have to filter frequent sets based on constraint satisfaction, we have to be able to read in constraints and store them into our system. Constraints are passed in from the CAP GUI to the algorithm via a file. When the user clicks on the “Run Algorithm...” button in the CAP GUI, the constraints are grouped into their respective types, parsed, and written into a constraint (text) file. The constraint file is then passed as a parameter to the algorithm. We decided to pass constraints from the GUI to the algorithm as a file instead of arguments because there can be an unlimited number of constraints in a query that can be handled by the CAP prototype. An example of a constraint file created by the CAP system is:

```
Field: ITEMID
A Constraints
aggrtag: count TO TYPE is <= 3
S Constraints
domaintag: TO ITEMID is ssuperset 3|4
```

The above file states that there are two constraints on a variable whose domain is the ITEMID attribute. The first constraint, \textit{count}(T0.TYPE) \textit{is} \textless 3, is an antimonotone aggregate constraint (referring to the ‘‘A Constraints’’ line), and the second constraint, \textit{T0.ITEMID} \textit{superset} \text{\{3,4\}}, is a succinct domain constraint.

A constraint reader in the algorithm parses these constraint files and stores the constraints into constraint structures. A constraint structure has the following type definition:

```c
typedef struct constraint {
    char ctype;
    char* agg;
    char* fieldname;
    char* isornot;
    char* criteria;
} /* constraint type: either A for aggregate or D for domain */
/* aggregate */
/* fieldname */
/* either ‘is’ or ‘not’ */
/* ‘>’, ‘subset’, etc */
```

30
float value;  // value in aggregate function */
int stringorvalue;  /* 0 for domain comparison set of
    string type, 1 for integer type */
SetList valueset;  /* domain constraint comparison set
    (integer), i.e. {1, 2} where
    constraint is S0.PRICE is
    subset of {1, 2} */
StrList valueset2;  /* domain constraint comparison set
    (string), i.e. {chips, snacks}
    where constraint is S0.TYPE is
    subset of {chips, snacks} */
struct constraint* next_c;  /* pointer to the next constraint */
}

Each `constraint` data structure holds either an aggregate or domain constraint. To illustrate what fields are used when different types of constraints are stored, we will give a few examples. A constraint structure holding the aggregate constraint \( \text{sum}(S0.PRICE) < 3 \) will have the following fields:

<table>
<thead>
<tr>
<th>FIELD</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctype</td>
<td>A</td>
</tr>
<tr>
<td>agg</td>
<td>sum</td>
</tr>
<tr>
<td>fieldname</td>
<td>PRICE</td>
</tr>
<tr>
<td>isornot</td>
<td>is</td>
</tr>
<tr>
<td>criteria</td>
<td>&lt;</td>
</tr>
<tr>
<td>value</td>
<td>3</td>
</tr>
<tr>
<td>stringorvalue</td>
<td>N/A</td>
</tr>
<tr>
<td>valueset</td>
<td>N/A</td>
</tr>
<tr>
<td>valueset2</td>
<td>N/A</td>
</tr>
<tr>
<td>next_c</td>
<td>null</td>
</tr>
</tbody>
</table>

In the above example, The `valueset` and `valueset2` fields are empty because the comparison in the constraint is with a value and not a set. The domain constraint \( S0.TYPE \text{ not superset } \{\text{chips, snacks}\} \) contains a comparison with the set of strings \{chips, snacks\}; therefore, the `valueset2` field of the structure holding
this domain constraint will contain the StrList \{chips, snacks\}. This is shown in the next table.

<table>
<thead>
<tr>
<th>FIELD</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctype</td>
<td>D</td>
</tr>
<tr>
<td>agg</td>
<td>N/A</td>
</tr>
<tr>
<td>fieldname</td>
<td>TYPE</td>
</tr>
<tr>
<td>isornot</td>
<td>not</td>
</tr>
<tr>
<td>criteria</td>
<td>superset</td>
</tr>
<tr>
<td>value</td>
<td>N/A</td>
</tr>
<tr>
<td>stringorvalue</td>
<td>0</td>
</tr>
<tr>
<td>valueset</td>
<td>N/A</td>
</tr>
<tr>
<td>valueset2</td>
<td>{chips, snacks}</td>
</tr>
<tr>
<td>next_c</td>
<td>null</td>
</tr>
</tbody>
</table>

The `stringorvalue` variable indicates whether or not the items in the comparison set are strings or integers.

Next we had to handle two-variable constraints, which required a different data structure. We wanted to read the two-variable constraints into the system like the one-variable constraints; therefore, we made the two-variable constraint data structure similar to the constraint data structure:

```c
typedef struct tvconstraint {
    char ctype; /* either A for aggregate, R for relational, or S for set */
    char* aggS; /* aggregate of antecedent set */
    char* fieldnameS; /* fieldname of antecedent set */
    char* isornot; /* is or not */
    char* criteria; /* `>', `subset', etc. */
    char* aggT; /* aggregate of consequent set */
    char* fieldnameT; /* fieldname of consequent set */
    float value; /* value to compare against for aggregate constraint */
    char* operators; /* either `union' or `intersect' for relational constraint */
    int stringorvalue; /* 0 for string set, 1 for integer set */
} tvconstraint;
```
SetList valueset; /* integer set used in relational constraint */
StrList valueset2; /* string set used in relational constraint */
struct tvconstraint* next_c; /* pointer to the next two-variable constraint */
} TVCONSTRAINT;

We must note that there will usually be non-applicable fields in the constraint data structures for different types of constraints, and that technically, this is a waste of memory space. However, since there will usually not be an outrageous number of constraints, the unused space is minimal.

A linked list of constraints of each type is created during the constraint reading phase. In the algorithm, a list called “Aconstraints” holds all antimonotone (only) constraints, and a list called “TVconstraints” holds all two-variable constraints. In retrospect, this is not really necessary for the Apriori+ algorithm, since it handles all types of constraints similarly. However, since CAP and Apriori+ use the same constraint reader, the GUI and the algorithm does not distinguish between CAP and Apriori+ when writing constraints into constraint data structures.

3.4.2 Applying Constraints

Querying the Item Database

When CAP or Apriori+ are executed, the first thing they do is read in the item database, dbfile.sql; each tuple (item) in the database is then stored in a dbrecord structure:

typedef struct dbrecord {
    short int itemid;       /* ITEMID field */
    char itemname[CBUFSIZE]; /* ITEMNAME field */
    char itemtype[CBUFSIZE]; /* TYPE field */
    short int itemprice;    /* PRICE field */
} DBRECORD;

The dbrecords for all items are kept in an array representing the database. In general, we should not be storing the whole database in memory, since real-life
databases have on average hundreds of thousands of records. However, since the research is a prototype of what can be done with association mining, we decided to keep everything simple, but fast, which is why we wanted to read the database from memory instead of storage.

Constraint verification on sets is implemented in a query/filter package. The query functions query the database array to retrieve all the necessary information. For the reader's reference, the item database dbfile.sql is printed out in Table 3.1.

The query module contains many different types of query functions. Examples include functions which query information such as the TYPE, PRICE, and ITEMNAME of an item, the ITEMIDs of all items whose TYPE is "snacks", or all ITEMIDs of items whose price is > $3.00. In the prototype, queries are implemented by performing linear searches through the database array.

Filtering Sets

To find if a set satisfies a given constraint, we have to filter the set by testing the constraint on it. The filter package contains functions which check if sets satisfy constraints. Base functions such as min, max, count, avg, sum, subset, superset, and = were first created. These functions were then used in other more general procedures which took in arguments such as sets, and names of aggregates or set comparators to apply to the sets. Switch statements directed control to the appropriate base functions. A wrapper function, OKConstraint, was created, reading in a constraint list and a set as arguments and returning TRUE or FALSE, depending on whether the set satisfied all constraints in the constraint list. In Apriori⁺, after generation of each group of k-item frequent sets, the OKConstraint function is used to check that each k-itemset satisfies the constraints. Only itemsets which satisfy all constraints will be displayed. It is necessary to note that in Apriori⁺, all k-itemsets are used in the generation of the (k+1)-item candidate sets, regardless
<table>
<thead>
<tr>
<th>ITEMID</th>
<th>ITEMNAME</th>
<th>TYPE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Doritos</td>
<td>snacks</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Cheetos</td>
<td>snacks</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Tostitos</td>
<td>snacks</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Bugles</td>
<td>snacks</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Coke</td>
<td>soda</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Sprite</td>
<td>soda</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Jolt</td>
<td>soda</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Swiss</td>
<td>cheese</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>Cheddar</td>
<td>cheese</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>Mozzarella</td>
<td>cheese</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>Parkay</td>
<td>butter</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>Imperial</td>
<td>butter</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>Foremost</td>
<td>butter</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>Tropicana</td>
<td>juice</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>Welch's</td>
<td>juice</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>Sunkist</td>
<td>juice</td>
<td>6</td>
</tr>
<tr>
<td>17</td>
<td>Whisky</td>
<td>alcohol</td>
<td>10</td>
</tr>
<tr>
<td>18</td>
<td>Scotch</td>
<td>alcohol</td>
<td>12</td>
</tr>
<tr>
<td>19</td>
<td>Gin</td>
<td>alcohol</td>
<td>11</td>
</tr>
<tr>
<td>20</td>
<td>Tequila</td>
<td>alcohol</td>
<td>10</td>
</tr>
<tr>
<td>21</td>
<td>Oreo</td>
<td>cookies</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>Bretons</td>
<td>cookies</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>Ritz</td>
<td>cookies</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>Arrowroot</td>
<td>cookies</td>
<td>3</td>
</tr>
<tr>
<td>25</td>
<td>Timeout</td>
<td>chocolate</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>Aero</td>
<td>chocolate</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>Choclairs</td>
<td>chocolate</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>Kitkat</td>
<td>chocolate</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>Shreddies</td>
<td>cereal</td>
<td>4</td>
</tr>
<tr>
<td>30</td>
<td>Cheerios</td>
<td>cereal</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>Muslix</td>
<td>cereal</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>Honeycomb</td>
<td>cereal</td>
<td>3</td>
</tr>
<tr>
<td>33</td>
<td>Carrot</td>
<td>vegetable</td>
<td>3</td>
</tr>
<tr>
<td>34</td>
<td>Radish</td>
<td>vegetable</td>
<td>2</td>
</tr>
<tr>
<td>35</td>
<td>Broccoli</td>
<td>vegetable</td>
<td>3</td>
</tr>
<tr>
<td>36</td>
<td>Tomato</td>
<td>vegetable</td>
<td>2</td>
</tr>
<tr>
<td>37</td>
<td>Onion</td>
<td>vegetable</td>
<td>2</td>
</tr>
<tr>
<td>38</td>
<td>Celery</td>
<td>vegetable</td>
<td>3</td>
</tr>
<tr>
<td>39</td>
<td>Cauliflower</td>
<td>vegetable</td>
<td>3</td>
</tr>
<tr>
<td>40</td>
<td>Lettuce</td>
<td>vegetable</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.1: Item Database dbfile.sql
of whether or not the individual sets satisfy the constraints.

Query, filter and aggregate functions also exist for two-variable constraints. For example, a function `applytwoagg` has been implemented which tests if an aggregate applied on one itemset is less than or greater than an aggregate applied on another itemset. Another function, `applytworeset`, tests if one itemset is a subset, strict subset, superset or strict superset of another itemset. In addition, two-variable constraint checking is implemented similar to one-variable constraint checking. The algorithms read two-variable constraints one by one from the TVconstraint list, and call the appropriate two-variable filter or query function on the pair of itemsets being checked. Pairs which satisfy all two-variable constraints are outputted by the algorithms.

One of the reasons why we hardcoded the attributes of `dbfile.sql` into the algorithm was to facilitate the implementation of the query functions. Had we made the database reader and query modules more general to accept different types of databases with different attributes, we would have had to concern ourselves more with database implementation issues rather than focusing on creating working versions of Apriori* and CAP to complete our prototype. It was subsequently decided to hardcode the metadata of the item database in the algorithms, but leave the records within the item database changeable.

### 3.5 Optimizing for CAP

Even though Apriori* was sufficient for our needs to express focus, we wanted the prototype to run as fast as possible, since speed is a quality that modern data mining programs running Apriori and its hybrids have still not yet achieved. Data mining will never reach its peak unless the process becomes almost instantaneous, and this is the reason why the CAP algorithm is implemented into the prototype. We want our prototype to incorporate the fastest algorithm possible to find the constrained frequent sets.
Consequently, after Apriori\textsuperscript{+} was completely implemented, we added on the optimization strategies described in [17] to create the CAP algorithm. In general, it was thought that this only involved changing how the base algorithm handled the different types of constraints. The task was this simple for most constraints, but proved to be quite a chore when it came time to implement the handling of succinct constraints. Nevertheless, the final product was worth the trouble when we saw that the prototype could run a handful of constraints on a 10,000 transaction database in under a couple of seconds.

3.5.1 Antimonotone Constraints

The first and easiest step taken during the implementation of CAP was incorporating the selectivity of antimonotone constraints. According to Strategy III on page 22, a set that does not satisfy an antimonotone constraint can be immediately pruned from the list of candidate sets and does not need to be counted.

The functions to do the checking were already implemented for Apriori\textsuperscript{+}, so this extension was trivial. Strategy III was implemented as follows: after all candidate sets were generated, we tested the sets by checking that each set satisfied all antimonotone constraints. If a set satisfied all antimonotone constraints, it got counted; otherwise, it was pruned from the candidate list, and was not used to generate future candidate sets.

3.5.2 Succinct Antimonotone Constraints

The second type of constraint to account for was constraints which were both antimonotone and succinct. According to Strategy I on page 21, we can replace the level 1 list of candidate sets with only those level 1 sets which satisfy all succinct antimonotone constraints. To incorporate this into the CAP algorithm, functions which generate lists of sets satisfying given criteria had to first be created. For example, the function \texttt{QueryIDOfType} was created to take one string argument \texttt{querystring},
find all items whose type is the same as *querystring*, and return them in a list. Then when, for example, the constraint \( S0.TYPE \subseteq \{\text{chips, snacks}\} \) is encountered, the \texttt{QueryIDOfType} function is executed, finding all items whose TYPE is "chips" or "snacks" and concatenating the items into the new level 1 candidate set list. To handle multiple constraints which are antimonotone and succinct, the algorithm intersects the list of sets which satisfy the individual constraints, thereby getting the list of 1-item sets which satisfy all antimonotone succinct constraints. This list of 1-item sets then replaced the original Level 1 candidate sets.

### 3.5.3 Succinct Constraints

The third type of constraint dealt with, and the one the author had the most difficulty with was the succinct constraints. The problems with the succinct constraints will be discussed in the later sections. For now, let us describe how we implemented the succinct constraints by first itemizing the four main types of succinct constraints in Table 2.1:

1. \( v \in S \)
2. \( S \supseteq V \)
3. \( \min(S) \leq v \)
4. \( \max(S) \geq v \)

The CAP algorithm has to know the MGFs of all the succinct constraints, since it was previously discussed in Section 2.3.2 that all succinct constraints have a moment generating function (MGF) which is capable of characterizing or generating the itemsets which satisfy the constraint.

So, first the MGFs of all the succinct (only) constraints a user can specify in the prototype had to be found. These are given below for the reader's reference; sets holding the results of queries are denoted by \( I_t \):

1. \( v \in S \)
• find all items where \( v \) is the value of the \( S \) attribute \((I_1)\)
• find all items where \( v \) is not the value of the \( S \) attribute \((I_2)\)
• MGF: \( \{X_1 \cup X_2 \mid X_1 \subseteq I_1 & X_1 \neq \emptyset & X_2 \subseteq I_2\} \)

2. \( S \supseteq V \)

• for each \( \text{value}_i \) in the set \( V \) (where \( V \) has \( n \) items)
  - find all items where \( \text{value}_i \) is the value on the attribute of \( S \) \((I_i)\)
  - find all items which do not have any of the values in \( V \) as the value on the attribute of \( S \) \((I_n)\)
  - MGF: \( \{X_1 \cup \ldots \cup X_n \mid X_1 \subseteq I_1 & X_1 \neq \emptyset & \ldots & X_n \subseteq I_n\} \)

3. \( \min(S) \leq v \)

• find all items whose \( S \) attribute is \( \leq v \) \((I_1)\)
• find all items whose \( S \) attribute is \( > v \) \((I_2)\)
• MGF: \( \{X_1 \cup X_2 \mid X_1 \subseteq I_1 & X_1 \neq \emptyset & X_2 \subseteq I_2\} \)

4. \( \max(S) \geq v \)

• find all items whose \( S \) attribute is \( \geq v \) \((I_1)\)
• find all items whose \( S \) attribute is \( < v \) \((I_2)\)
• MGF: \( \{X_1 \cup X_2 \mid X_1 \subseteq I_1 & X_1 \neq \emptyset & X_2 \subseteq I_2\} \)

From now on, we will use the notation \( X_i \) to refer to all the required \( (\neq \emptyset) \) sets (for \( 1 \leq i \leq n - 1 \)) of an MGF. Non-required sets will be denoted by \( X_n \).

With the MGFs of all the succinct constraints, we were now able to implement the succinct constraint handler. If constraints (1), (2) (only in situations where \( |V| = 1 \)), (3), or (4) exist, the CAP algorithm takes all items from the required sets of the respective MGF as the level 1 candidate sets. However, in the case that succinct antimonotone constraints have also been created, then the succinct 1-item candidate set list would be intersected with the succinct antimonotone 1-
item candidate set list to get a list of all 1-itemsets satisfying both SAM and SUC constraints.

If a succinct superset constraint exists where \(|V| > 1\), then the CAP algorithm proceeds as detailed in [17]. In Level 1, we first find the support of each item in the sets \(X_1 \ldots X_n\). The frequent required sets are stored in \(L^X_i\) for \(1 \leq i \leq n - 1\), while the frequent non-required set is stored in \(L^X_n\). All the 1-itemsets in \(L^X_i\) for \(1 \leq i \leq n - 1\) are then joined to produce the candidate 2-item set list. Support is found on this candidate set list and the frequent sets are then computed. After the computation of the \((n - 1)^{th}\) level of frequent sets, where \(n - 1\) is the count of the minimum number of required sets in the MGF, the next and all future iterations of candidate sets are created by joining \(L_{n-1}\) with \(L^X_1 \cup L^X_2 \cup \ldots \cup L^X_n\). We finally find the frequent sets to create \(L_n\). In addition, starting with \(C_{n+1}\), all sets created have to pass an additional pre-candidate test before counting to ensure that it is a "valid" candidate set. The pre-candidate test is shown in the following lemma.

**Lemma 2** A set \(T\) is in \(C_k\) iff: \(\forall T': T' \subset T\) and \(|T'| = k - 1\) and \(T' \cap L^{S_1}_{1} \neq \emptyset \) and \(L^{S_2}_{1} \neq \emptyset\) and \(\ldots\) and \(L^{S_m}_{1} \neq \emptyset\) \(\Rightarrow T' \in L_{k-1}\)

**Multiple Succinct Constraints**

The above discussion only described how the prototype handles single succinct constraints. Next, we had to account for cases where a query contained multiple succinct constraints. The CAP paper proposed that the MGFs of multiple succinct constraints could be merged into one MGF using Lemma 1. Candidate sets satisfying all succinct constraints could then be generated using this one MGF. The actual implementation of this strategy in C is described in the paragraphs below.

For the MGF of each succinct constraint, a list \(mgf\_list\) containing the required set(s) \((X_1 \ldots X_{n-1})\) and the non-required set \((X_n)\) is stored. For example, suppose we are dealing with the following succinct constraint:

\[(1) \ S0.PRICE \subseteq \{1, 2, 3, 4\}\]
Further, suppose the required sets of the MGF of this constraint are:

1a. \{1, 5\} (items with a price of 1)
1b. \{2, 7\} (items with a price of 2)
1c. \{3, 8\} (items with a price of 3)
1d. \{4, 6\} (items with a price of 4)

and the non-required set is:

1e. \{9, 10\} (items whose price is not 1, 2, 3, or 4)

Then, in the implementation, the sets (1a), (1b), (1c), (1d), and (1e) will be stored in the given order in the \textit{mgflist} for the constraint. Now, let our second constraint be:

\begin{equation}
\text{(2) } S0.TYPE \supset \{\text{chips, snacks}\}
\end{equation}

The \textit{mgflist} corresponding to this constraint will hold required sets:

2a. \{1, 2, 3, 4\} (chip items)
2b. \{5, 6, 7, 8\} (snack items)

and the non-required set:

2c. \{9, 10\} (neither chips nor snack items)

According to Lemma 1, in order to generate the candidate 1-item sets satisfying these two constraints, we have to intersect each constraint’s \textit{mgflist} with the other constraint’s \textit{mgflist}. This results in a total of 15 sets created by the intersections.

<table>
<thead>
<tr>
<th>Constraint 1/Constraint 2</th>
<th>1a</th>
<th>1b</th>
<th>1c</th>
<th>1d</th>
<th>1e</th>
</tr>
</thead>
<tbody>
<tr>
<td>2a</td>
<td>Set 1</td>
<td>Set 2</td>
<td>Set 3</td>
<td>Set 4</td>
<td>Set 9</td>
</tr>
<tr>
<td>2b</td>
<td>Set 5</td>
<td>Set 6</td>
<td>Set 7</td>
<td>Set 8</td>
<td>Set 10</td>
</tr>
<tr>
<td>2c</td>
<td>Set 11</td>
<td>Set 12</td>
<td>Set 13</td>
<td>Set 14</td>
<td>Set 15</td>
</tr>
</tbody>
</table>

Sets 1-8 are the “required sets” of the new MGF; that is, they cannot be \(\emptyset\). Frequent sets satisfying the two constraints must contain at least one item from each of the sets 1-8. Items from sets 9-15, on the other hand, are optional. In the implementation, sets 1-15 are stored in a new \textit{mgflist} in their enumerated order. Each set in the
mgflist can then be accessed through pointers. Note that the non-required sets are always stored at the end of the mgflist.

The items in each set of the mgflist are subsequently tested for support. If an item is found to be frequent, it is left in its set; otherwise, the item is taken out from its set. The resulting mgflist points to the frequent-item sets $L_1^{X_1} \ldots L_1^{X_n}$ described in Strategy II on page 22. This mgflist is used throughout the algorithm to succinctly generate candidates.

**Succinct Constraints - Problem 1**

Interestingly, testing the multiple succinct constraint handler we implemented indicated that there was a flaw in the MGF merging formula. According to [17], multiple MGFs can be combined into one MGF using lemma 1. However, we noticed that this formula was not generalized enough for all cases. Consider the case where we have the constraints:

1. $S0.TYPE \supseteq \{juice, chocolate\}$
2. $S0.ITEMNAME \supseteq \{Tropicana, Aero\}$

*Tropicana* is a type of juice and *Aero* is a type of chocolate. Now, suppose we have the following required lists for each constraint’s MGF:

1a. juice items: $\{1, 3, 5, 7, 9\}$
1b. chocolate items: $\{2, 4, 6, 8, 10\}$
2a. Tropicana: $\{1\}$
2b. Aero: $\{2\}$

Then according to the multiple MGF combination formula, we get the table below when intersecting the required lists of the two MGFs:

<table>
<thead>
<tr>
<th>Constraint 1/Constraint 2</th>
<th>1a</th>
<th>1b</th>
</tr>
</thead>
<tbody>
<tr>
<td>2a</td>
<td>${1}$</td>
<td>${}$</td>
</tr>
<tr>
<td>2b</td>
<td>${}$</td>
<td>${2}$</td>
</tr>
</tbody>
</table>

42
When joining MGFs, intersections of required sets must not equal the empty set. If any of the results is the empty set, then there are no sets satisfying the constraint(s). In the example given above, empty sets were produced; consequently, there should not be any frequent sets. However, surprisingly, we did in fact find sets satisfying the constraints such as \{Tropicana, Aero\} and \{Tropicana, Aero, Bretons\}.

This problem was thought over carefully. One solution suggested was to take only the sets in the diagonals of the table produced above as the required sets (i.e. (1a) intersect (2a) and (1b) intersect (2b)). However, since items in a list can come in any random order, it is difficult to choose one particular diagonal to work from. This problem of course, becomes even more difficult to solve when extended to more than two constraints. It was finally decided to leave the matter unresolved, and instead modify our algorithm to handle only multiple succinct constraints whose MGFs, when joined, produce only one required set - thereby eliminating all diagonal problems.

**Succinct Constraints - Problem 2**

Another problem arised when implementing the superset constraint. With the superset constraint, we have to make sure that when generating the candidate sets \(C_1 \ldots C_{n-1}\) (where \(n - 1\) is the number of required sets in the constraint’s MGF), each item in each set of \(C_k\) has to come from a different required set \(X_i\). This invariant is held true in the implemented program by storing, for each candidate and frequent set, a set \textit{indexset} containing indices to the different required lists. More specifically, for each item in each set of the \textit{mgfist}, the index of the required set \(X_i\) that the item comes from is stored for the item. When joining \(L_1\) with \(L_1\), the algorithm checks to make sure the index from one item of the join and the index from the other item of the join is not the same.

For example, suppose we were dealing with two-item sets, and we wanted to join \(\{1, 2\}\) with \(\{1, 4\}\). Now, further suppose that the succinct constraint was
So, \( TYPE \supset \{\text{chips, snacks, beer}\} \), and we have the following required lists:

1. chip items: \{1, 3, 5, 7\}
2. snack items: \{2, 4, 6, 8\}
3. beer items: \{9, 10, 11, 12\}

Then, joining \{1, 2\} and \{1, 4\} should give us \{1, 2, 4\}; however, this set does not satisfy the given superset constraint. The types of \{1, 2, 4\} are a subset of \{chips, snacks, beer\} as opposed to a superset.

As said earlier, the author solved the problem by storing temporary sets for each itemset which track which required sets items in the itemset come from. The itemset \{1, 2\} would have a corresponding indexset of \{1, 2\} (item 1 comes from list 1 and item 2 comes from list 2) and the itemset \{1, 4\} would have a corresponding indexset of \{1, 2\} as well. When joining the itemsets \{1, 2\} and \{1, 4\} into a 3-item set, we check that their indexsets, when joined, produce a 3-item set with unique items. This is not the case in the example; therefore, the algorithm does not produce the candidate set \{1, 2, 4\}.

**Succinct Constraints - Problem 3**

The last problem which came up while implementing succinct constraints was multiple generation of the same itemsets during candidate creation. When a succinct constraint exists, the first \( n - 1 \) iterations of CAP (where \( n - 1 \) is the number of required sets in the MGF function) are handled the same way as they are in Apriori+. Generate the 1-item frequent sets, join to create the 2-item candidate sets, join the 2-item frequent sets ... create the \((n - 1)\)th-item candidate sets. At the \((n - 1)\)th level however, we do not join the \((n - 1)\)-item frequent sets together to create the \(n\)-item candidate sets. We create the \(n\) item candidate sets by individually adding frequent items from \(L_1^{X_1} \ldots L_1^{X_n}\) to each \(n\)-item set. The generated \(n\)-item sets then have to pass the pre-candidate test given in Lemma 2 before counting.
With this proposed framework, we run into the problem that a candidate set can be generated more than once. To illustrate this, suppose we have the following 3-itemsets:

- \{1 2 3\}
- \{1 2 4\}
- \{1 3 4\}
- \{2 3 4\}

And our list of frequent items is \{1\}, \{2\}, \{3\} and \{4\}. Then, individually adding each frequent item to each frequent 3-itemset, we get:

- \{1 2 3 4\}
- \{1 2 3 4\}
- \{1 2 3 4\}
- \{1 2 3 4\}
- \{1 2 3 4\}

In the above example, it can be seen that we end up generating multiple copies of the same candidate set. The original fix to this problem was to check, for each candidate being generated, whether or not the same candidate had already been previously generated and added to the current candidate list. Unfortunately, this solution, in addition to all the pre-candidate testing we have to perform, compensates performance and efficiency.

Another solution was developed, described as follows. In the simplest case, for any \(k\)-item candidate set created, there should be \(k \times (k - 1)\)-item sets which can be used to generate the candidate. Therefore, there will be \(k\) instances of the same candidate set being generated. The reader can verify this in the example above. For the succinct case, there may not necessarily be \(k\) instances of repeated generation, since not all permutations will be valid if they do not satisfy the succinct constraints.

To show what we mean, let us apply this to the candidate set as follows.
\{1, 2, 3, 4, 5\}

Suppose that item 1 is an item from required set \(X_1\) and item 2 is an item from required set \(X_2\). Since items 1 and 2 have to be in each set, the only possible permutations which can now be used to generate the set \{1, 2, 3, 4, 5\} are:

- \{1, 2, 3, 4\}
- \{1, 2, 4, 5\}
- \{1, 2, 3, 5\}

In this case, the number of generations of the same candidate set is only three, and not five. Given this, we can therefore preliminarily hypothesize that the number of repeated generations is directly related to the number of "required" items in a set.

Using the same set \{1, 2, 3, 4, 5\}, let us now say that the items \{1, 3\} belong to the required set \(X_1\), and the items \{2, 4\} belong to the required set \(X_2\). Then, the permutations that can be used to generate \{1, 2, 3, 4, 5\} are:

- \{1, 2, 3, 4\}
- \{1, 2, 4, 5\}
- \{2, 3, 4, 5\}
- \{1, 3, 4, 5\}
- \{1, 2, 3, 5\}

The number of repetitions is now five again. Using these two examples, a simple formula was created to calculate the number of times a candidate set \(A\) of size \(k\) should be repeatedly generated by a succinct constraint. If we let:

\[
m_1 = \text{number of items in } A \text{ from required set } X_1
\]
\[
m_2 = \text{number of items in } A \text{ from required set } X_2
\]
\[
\vdots
\]
\[
m_{(n-1)} = \text{number of items in } A \text{ from required set } X_{n-1}
\]
we can then set the variables $p_{m_i}$ for $1 \leq i \leq (n - 1)$ where:

$$p_{m_i} = \begin{cases} 
0 & \text{if } |m_i| > 1 \text{ or } |m_i| < 1 \\
1 & \text{if } |m_i| = 1 
\end{cases}$$

The final formula is:

$$\text{Number of generations of } A = \left( \frac{(k - p_{m_1} - p_{m_2} - \cdots - p_{m_{n-1}})}{2} \right)$$

With the above equation, we can modify the algorithm to skip the pre-candidate testing phase. By comparing whether the value of Formula 3.5.3 on a candidate set is equal to the actual number of times the set is repeatedly generated, we have essentially performed pre-candidate testing, and can therefore remove the test completely from the process. For $k \geq n - 1$, the algorithm generates candidate sets by appending items in $L_{X_i}$ to $L_{X_n}$. As each set is created, it is checked to see if it has already been previously added to the candidate list. If the set has not yet been generated, it is added to the candidate list, and a counter for the set is created and incremented. If the set has been previously generated, then the counter will already exist and is incremented. The set itself will not be added again to the candidate list.

When checking if a candidate set is valid, we apply Formula 3.5.3 to see if the calculated value corresponds to the count of how often the set has been created (as stored by the set's counter). By doing this, we can automatically deduce if a set is a valid candidate without having to check all its subsets.

### 3.5.4 Non-antimonotone Non-succinct Constraints

The last type of constraint to handle was the non-antimonotone non-succinct constraints. In general, satisfaction of non-antimonotone non-succinct constraints is checked after counting is performed on a candidate set. Those sets that do not satisfy all of the non-antimonotone non-succinct constraints are not displayed in
the output. All itemsets are, however, used in the processing of the next level of candidates. This is identical to how constraints were handled in Apriori+.

The CAP paper suggests a modified way to handle non-antimonotone non-succinct constraints. When a non-antimonotone non-succinct constraint is passed to the CAP algorithm, we can add a weaker constraint similar to the non-antimonotone non-succinct constraint and process itemsets with the weaker constraint. The weaker constraint should of course be an antimonotone, succinct, or antimonotone succinct constraint to maximize the pruning capabilities of the CAP framework. For example, the constraint \( \text{avg}(S0.\text{ITEMID}) < 5 \) can induce a weaker constraint, \( \text{min}(S0.\text{ITEMID}) < 5 \). This weaker succinct constraint can then be used to prune all sets which do not satisfy its criteria. In turn, any sets which do not satisfy the \( \text{min} \) constraint of course do not satisfy the \( \text{avg} \) constraint. After counting is completed, all remaining sets are still checked for satisfaction of the original non-antimonotone non-succinct constraint, since sets which, i.e. satisfy the weaker \( \text{min} \) constraint may still not satisfy the original \( \text{avg} \) constraint.

Implementation-wise, as non-antimonotone non-succinct constraints are read into constraint structures, we check to see if weaker constraints can be induced from them. If a weaker constraint can be induced, it is stored into a constraint structure and the structure is then added to the corresponding constraint list (Antimonotone, Succinct, or Antimonotone Succinct).

Table 3.2 lists some of the weaker constraints that are implied by non-antimonotone non-succinct constraints.

### 3.6 Relationship Calculation

After optimizing the prototype with CAP, the next item on the list was to implement the relationship calculation algorithm. The prototype shows that the new model can handle more flexible notions of relationship, and does this by providing three different types of relationships acknowledgable by the relationship calculation
algorithm, versus only one in the classical model. In fact, users can calculate relationships based on confidence, correlation or dependence and can provide relevant thresholds for each.

The relationship calculation algorithm itself takes in six arguments. The command line to call the relationship calculation algorithm is:

```
capapr metric minSup minThresh pairedItemsetFile Strans Ttrans
```

The first argument is the metric type, which is either "confidence", "correlation" or "dependence". The second argument is a minimum support value used only for confidence calculation. The third argument is the minimum threshold value, which is the value a set must have before it is deemed "significant". The fourth argument is the name of a paired itemset file containing the pairs of itemsets we want to find relationships on, and the last two arguments are the file names of the transaction databases that we will be extracting itemset support from.

The paired itemset file is a file created by the GUI containing {antecedent, consequent} pairs the user is interested in. Lines in the paired itemset file have the format:

```
supportS | S | supportT | T
```

where `supportS` is the support of the antecedent set S in transactions, and `supportT` is the support of the consequent set T. The lines below are taken from a paired itemset file created by the CAP GUI:

```
49
```
The actual itemsets and supports represented by these lines are:

- Antecedent \{1, 2, 3, 4\} with support of 2501 and consequent \{2, 3, 4, 5\} with support of 3502
- Antecedent \{1, 2, 3, 4\} with support of 2501 and consequent \{2, 3, 4, 6\} with support of 1231
- Antecedent \{1, 3, 4, 5\} with support of 1210 and consequent \{1, 3, 5, 6\} with support of 2312
- Antecedent \{1, 3, 4, 5\} with support of 1210 and consequent \{2, 4, 5, 6\} with support of 2121

To create the paired itemset file, the system collects all pairs the user selects, and extracts their supports using the frequent set output from Phase I (the CAP algorithm) that it has stored internally. This is done in the GUI to decrease execution time of the relationship calculation algorithm. Not all metric types will require supplying the supports for both \(S\) and \(T\) in the paired itemset file. The confidence metric requires the support for the antecedent, but not the consequent. The dependence metric requires support values for both the antecedent and consequent, and the correlation metric does not require support values for either one. Reasons for requiring or not requiring one support value or the other are related to the relationship calculation formula of the chosen metric. For example, confidence does not require the consequent support value to calculate confidence values, whereas dependence does.

In the relationship calculation algorithm, lines from the paired itemset file are first read in and parsed with a parsing function separating the sets and the support
values. Each pair of sets and support values are subsequently stored in an association
rule record:

```c
typedef struct {
    int support;
    char STstatus;  /* support of S union T */
    SetList S;     /* 'y' if we know from input 
                    the support of S union T, 
                    'n' otherwise */
    int supportS;  /* support of the antecedent 
                    set */
    char Sstatus;  /* 'y' if we know from input the 
                    support of S, 'n' 
                    otherwise */
    SetList T;     /* the consequent set */
    int supportT;  /* support of the consequent 
                    set */
    char Tstatus;  /* 'y' if we know from input 
                    the support of T, 'n' 
                    otherwise */
} rule;
```

After a rule structure is created for a pair of itemsets, the structure is then stored
in the list of all rules to be "operated" on:

```c
typedef struct {
    int count;
    rule L[MAXRULELIST];
} rulelist;
```

The rulelist is implemented as an array with a maximum size of MAXRULELIST. For
now, MAXRULELIST is set to 300 due to memory restrictions on some of the machines
CAP was tested on.

After the paired itemset file is parsed and the rulelist is created, the rulelist
is passed to a function findRuleSupport, which finds support for the antecedent
or consequent by counting from the transaction database(s). For each transaction,
each \{antecedent, consequent\} pair is tested to see if the antecedent, consequent, or
antecedent \cup\ consequent is a subset of the current transaction The choice of what
sets to find support for is dependent on the metric chosen. The support count is incremented in the appropriate fields (supportS, supportT and/or support) if the corresponding set is a subset of the transaction.

After supports are found, the rule structure is passed to the corresponding function (i.e. findConfidence, findCorrelation, findDependence) which uses the support values to calculate a relationship’s significance value in terms of the metric chosen. The function then compares the calculated significance to the minimum threshold value supplied by the user. If the calculated significance is greater than or equal to the threshold, the itemset pair is printed out along with its significance value; otherwise, the pair is not displayed.

3.6.1 Confidence

In the prototype, confidence is calculated from the conditional probability formula \(P(T|S)/P(S)\). In simpler terms, the formula calculates the probability of the consequent and antecedent sets appearing in a transaction divided by the probability of the antecedent occurring in a transaction. During transaction file scanning, the findRuleSupport function serves to find the support of \(S \cup T\). The support for \(S\) does not need to be counted because it can be extracted from the paired itemset file created by the GUI.

Confidence is only significant if \(S \cup T\) is frequent; therefore, the support of \(S \cup T\) must be checked to be frequent before confidence can be calculated on an itemset pair. A minimum support value for \(S \cup T\) is passed to the system when confidence is selected as the relationship metric. If the confidence calculated is greater than or equal to the confidence threshold and the support of the union of the antecedent and consequent is greater than the minimum support value, the itemset pair is displayed along with its confidence value in the output.
### 3.6.2 Correlation

Some researchers believe that correlation is a better measure of association than confidence [9]. Given this, the prototype allows the user to choose correlation as a relationship metric versus confidence. We believe that giving the user this choice makes the new exploratory model more flexible, redefining the definition of relationship in association mining to be any metric one sees fit to apply to possible rules.

Correlation is the measure of how closely related a set of variables are. We can extend this theory to association rules as well. More specifically, if we consider an association rule $S \Rightarrow T$ to be a pair of variables $S$ and $T$, then for each transaction, $S$ can be true and/or $T$ can be true; by true we mean that $S$ or $T$ is a subset of the current transaction, and false otherwise. We can then proceed to create a contingency table for each \{S, T\} pair. A contingency table is a table listing the number of occurrences where $S$ is true, $T$ is true, $S$ is not true, $T$ is not true, and combinations of $S$ and $T$ being true and not true. Table 3.3 shows the structure of contingency tables used in this implementation for candidate pairs.

<table>
<thead>
<tr>
<th></th>
<th>T is true</th>
<th>T is false</th>
<th>Sum of Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>S is true</td>
<td>s1t1</td>
<td>s1t0</td>
<td>s1</td>
</tr>
<tr>
<td>S is false</td>
<td>s0t1</td>
<td>s0t0</td>
<td>s0</td>
</tr>
<tr>
<td>Sum of Column</td>
<td>t1</td>
<td>t0</td>
<td>n</td>
</tr>
</tbody>
</table>

Table 3.3: Contingency Table for Association Rules
true, but not the support of \( S \) when \( T \) is not true, or \( T \) when \( S \) is not true, which is what we require.

We need the support value of \( s0t0 \) in order to complete the contingency table; instead of counting for \( s0t0 \), we can just calculate it by subtracting \( s1t1 \), \( s0t1 \), and \( s1t0 \) from the total number of transactions in the database, which is represented by \( n \) in the contingency table.

With contingency table values for each itemset pair, we can measure the significance of correlation of the antecedent and consequent by calculating the contingency table’s chi-squared value. More details of contingency tables, the chi-squared formula and how they are applicable to association rules can be found in [16].

The formula used in this implementation to calculate the chi-squared \( \chi^2 \) value of an \( \{S, T\} \) pair is:

\[
\chi^2 = \frac{(s1t1 - (s1 \times \frac{t1}{n}))^2}{s1 \times \frac{t1}{n}} + \frac{(s0t0 - (s0 \times \frac{t0}{n}))^2}{s0 \times \frac{t0}{n}} + \frac{(s1t0 - (s1 \times \frac{t0}{n}))^2}{s1 \times \frac{t0}{n}} + \frac{(s0t1 - (s0 \times \frac{t1}{n}))^2}{s0 \times \frac{t1}{n}}
\]

Each pair’s chi-squared value is then compared to the threshold. If the calculated figure is greater than the threshold, the pair along with its chi-squared value is displayed.

3.6.3 Dependence

The authors of [9] suggest that dependence could also be an appropriate metric for finding the degree of association between two sets; therefore, we decided to include this option into our relationship calculation module as well. Dependence is a measure of how dependent a variable is on another. The measure of dependence between two variables is calculated by the formula \( P(S \cup T)/(P(S) \times P(T)) \). A value greater than 1 indicates a dependency between the two variables. Again, like correlation, we can extend this theory to association rules. For each \( \{S, T\} \) pair, by finding the supports of \( S \cup T \), \( S \), and \( T \), we can calculate a pair’s dependency ratio. However, given that we already calculated the supports of \( S \) and \( T \) in Phase I, we
can input these supports directly from the GUI through the paired itemset file. As a result, we only need to count for the support of $S \cup T$. After this is done, we check that the calculated value is greater than or equal to the user-supplied threshold; if it is, the itemset pair and its dependency ratio are displayed.

### 3.7 Mining Between Sets of Different Domains

In all data mining programs implemented to date, the majority of them can only mine associations between sets of the same type. In fact, even in the original implementation of the CAP algorithm, all the data structures created assume that items being mined are item IDs. However, one of the most unique and prominent features of our prototype is its ability to mine associations between sets of different type, i.e. $\text{Type} \Rightarrow \text{ItemID}$. Consequently, this functionality was added on after the algorithms for Phase I and Phase II were implemented.

The first question asked was, "How do we implement mining on $\text{TYPE}$ as opposed to $\text{ITEMID}$?" To solve this problem, a number of functions were created to enumerate the domain of these non-$\text{ITEMID}$ fields into enumerated $\text{IDs}$. For example, if there were twelve different $\text{TYPES}$ in the item database, then the CAP algorithm would enumerate the types and assign them IDs from 1-12.

The constraint files passed to CAP tell the system what fields are going to be mined. The field currently being mined is stored in a global variable, $\text{class}$. If $\text{class}$ is $\text{ITEMID}$, then no enumeration is required. If $\text{class}$ is $\text{TYPE}$, $\text{PRICE}$ or $\text{ITEMNAME}$, the domain of the field values is enumerated and stored in an array $\text{enumarray}$ with the structure \{enumerated ID, field value\}. Suppose we were mining on the $\text{TYPE}$ attribute with a domain of five different types, then our $\text{enumarray}$ would look something like the following table.
The enumarray is kept global to enable efficient access by query and filter functions. These functions were also appropriately modified to handle querying and filtering on enumerated IDs.

Overall, it was not difficult to extend the implementation to handle non-ID fields. One small problem which did come about was the inefficiency of translating the ITEMID transaction database into enumerated databases. For example, consider the following transaction, given in ITEMIDs:

\{1, 3, 5, 7\}

If we were mining on TYPE instead of ITEMID, then, when reading the transaction database, we have to change each item's ITEMID into its corresponding enumerated TYPE ID. So, first we have to find each item's TYPE:

<table>
<thead>
<tr>
<th>ITEMID</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cheese</td>
</tr>
<tr>
<td>3</td>
<td>soda</td>
</tr>
<tr>
<td>5</td>
<td>cheese</td>
</tr>
<tr>
<td>7</td>
<td>juice</td>
</tr>
</tbody>
</table>

Then we have to find the enumerated ID of each item's type in the enumarray:

<table>
<thead>
<tr>
<th>Enumerated ID</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cheese</td>
</tr>
<tr>
<td>2</td>
<td>chocolate</td>
</tr>
<tr>
<td>3</td>
<td>juice</td>
</tr>
<tr>
<td>4</td>
<td>soup</td>
</tr>
<tr>
<td>5</td>
<td>soda</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enumerated ID</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cheese</td>
</tr>
<tr>
<td>2</td>
<td>chocolate</td>
</tr>
<tr>
<td>3</td>
<td>juice</td>
</tr>
<tr>
<td>4</td>
<td>soup</td>
</tr>
<tr>
<td>5</td>
<td>soda</td>
</tr>
<tr>
<td>TYPE</td>
<td>ENUMERATED ID</td>
</tr>
<tr>
<td>--------</td>
<td>---------------</td>
</tr>
<tr>
<td>cheese</td>
<td>1</td>
</tr>
<tr>
<td>chocolate</td>
<td>2</td>
</tr>
<tr>
<td>juice</td>
<td>3</td>
</tr>
<tr>
<td>soup</td>
<td>4</td>
</tr>
<tr>
<td>soda</td>
<td>5</td>
</tr>
</tbody>
</table>

Finally, we can translate the transaction \(\{1 \ 3 \ 5 \ 7\}\) into its enumerated counterpart \(\{1 \ 5 \ 1 \ 3\}\). As one can see, having to do this for all transactions during the counting process is both tedious and inefficient. Instead, we can generate an enumerated transaction file on the fly in CAP/Apriori before any counting is done; unfortunately, because we never know how large any one transaction file is, enumerated files created while running CAP can end up hogging valuable memory space. Therefore, it was decided that enumerated databases translated from the primary ITEMID transaction file be made ahead of time and stored on disk. When it comes time to call CAP/Apriori and read in an enumerated transaction, the corresponding transaction file is read instead of the ITEMID transaction file. The enumerated transaction files which have been created are `price10.bdat`, `price20.bdat`, `type10.bdat` and `type20.bdat`. The first few letters of each filename correspond to the attribute the transactions are enumerated on. The number in the filename indicates the number of transactions (in thousands) that are in the file.

To ease implementation, it was decided to write enumerated IDs, as opposed to the actual field values, into paired itemset files for Phase II. The enumeration is performed in the GUI before calling the relationship calculation algorithm.

The second problem we encountered was the succinct generation of candidate sets for sets whose domain comes from attributes which are not unique. Because many of the lemmas and solutions to problems regarding succinct constraints assume that the domains of the sets come from unique fields (i.e. ITEMID), it was difficult to extend these lemmas to non-distinct attributes. In such cases, the succinct constraints in
question are handled as non-succinct non-antimonotone constraints.

3.8 Running in the Classical Model

Another powerful feature of the CAP prototype is its ability to run in both the classical and the exploratory models. By default, the system runs under the exploratory model, calling CAP in Phase I and the relationship calculation algorithm in Phase II. However, the CAP/Apriori$^+$ algorithms have also been implemented to perform frequent set and relationship calculation together in one execution. We call this "no-breakpoint" mode. In no-breakpoint mode, the program finds all frequent sets for the antecedent and the consequent, pairs up all the antecedent sets with all the consequent sets, and performs relationship calculations on the pairs. The result is a list of frequent and significant \{antecedent, consequent\} sets satisfying one-variable and two-variable constraints.

3.9 Testing

After the algorithms for CAP and Apriori$^+$ were implemented, both versions had to be tested thoroughly for correctness. Since testing all combinations of constraints was not possible, the author chose a smaller comprehensive set of test cases to verify correctness. Not surprisingly enough, the prototype did indeed run faster with CAP than with Apriori$^+$.

There were two transaction databases used in the testing, one with 10000 transactions and one with 20000 transactions. There were also only about 50 distinct records in the transaction database; the rest of the database cycled and repeated these transactions. We decided against using real (non-cyclical) databases to test our prototype because it would be extremely difficult to verify the results of our tests.
3.9.1 Testing Individual Constraints

Each type of succinct, antimonotone, succinct antimonotone and non-succinct non-
antimonotone constraint was first tested individually in Apriori+ and CAP. The
results were proven correct by manual verification, and by comparing CAP and
Apriori+ results for the same constraints - since the output of both algorithms should
be the same. It was discovered that CAP was usually faster, but the actual speedup
depended on the type of constraint. Succinct antimonotone constraints seemed
to induce the most pruning in CAP, antimonotone constraints were second in line,
succinct constraints were third and non-antimonotone non-succinct constraints were
last. One interesting point observed was that succinct constraints were sometimes
only a fraction faster in CAP than they were in Apriori+. We believe that this
is probably due to some of the extra processing required when handling succinct
constraints, such as checking for repeated generation of candidate sets.

Sometimes the results swayed from the order given above, mainly due to different
constraint conditions such as the number of items in a comparison set, the strength of
pruning of a constraint, the support of the antecedent and consequent, etc. Because
of this, it is difficult to provide exact figures of speedup since all performance values
vary with respect to the constraint in consideration.

3.9.2 Testing Multiple Constraints

Next we had to test the correctness of CAP with multiple constraints. The first
type of multiple constraint testing was testing multiple constraints of the same type,
i.e. testing only succinct constraints, or testing only antimonotone constraints. In
this case, multiple succinct antimonotone constraints significantly improved perfor-
mance time in CAP. Multiple antimonotone constraints were second in performance
increase, and multiple succinct constraints were third.

It was more difficult to assess the performance of CAP over Apriori+ with mul-
tiple constraints of different types. In general, CAP was much faster than Apriori+.
when there were no succinct constraints. When there were succinct constraints, CAP was sometimes as much as eight times faster than Apriori+; at other times, it was only 0.2 of a second faster. As a result, it is very difficult to state, in exact numbers, how much more efficient CAP is than Apriori in the implementation. This is because there are so many different constraints and so many different combinations of constraints that make any sort of comparison difficult to make. However, it can be said that in general, CAP is definitely faster than Apriori+. But if the author had to state by how much, the answer could range anywhere from 0 to 80 times faster. More detailed statistics regarding the performance of CAP over Apriori+ can be found in [20].

3.10 Calling CAP and Apriori+

For reference, we include this section to describe how the implemented CAP and Apriori+ algorithms are run.

CAP and Apriori+ are called with the command line:

```
capapr proname Ssup Tsup Scfile Tcfile STcfile Strans Ttrans
```

Here, `capapr` is the name of the executable. The first argument, proname, is either "cap" or "apriori" and tells the `capapr` program to execute the corresponding algorithm. The second and third arguments are the minimum supports of the antecedent and consequent. The minimum support here is given in percentages. The fourth, fifth and sixth arguments, `Scfile`, `Tcfile`, and `STcfile` correspond to the files containing constraints for the antecedent, consequent and two-variable sets. The last two arguments are the names of the transaction database files for the antecedent and consequent. For example, the command line:

```
capapr apriori 3 5 s.cst t.cst st.cst id10.bdat type10.bdat
```
would imply that we are running Apriori$^+$ to find antecedent sets with a minimum support of 3% on the domain of ITEMID, and consequent sets with a minimum support of 5% on the domain of TYPE. For a detailed account of how Apriori$^+$ and CAP handle mining on non-ITEMID fields, the reader can refer to Section 3.7.

In the above, the constraints for the antecedent, consequent and two-variable sets are located in the files s.cst, t.cst and st.cst respectfully. To find only consequent set answers, users input “NoS” as the Scfile argument. Similarly, to find only antecedent set answers, “NoT” would be passed as the Tcfile parameter.
Chapter 4

Support for User Interaction

The goal of achieving human-centred exploration is at the core of the two-phase architecture, and this is accomplished in the system through our user interface, which guides the user through the process of two-phase association mining, and allows one to interactively perform a variety of exploratory tasks on the data generated and the data to be mined. Interestingly enough, the task of designing the prototype's interface was a difficult, yet fun task.

Since our goal is to promote user interaction and usability, the system had to be easy and obvious enough to use for the beginner. In particular, our unique prototype provides extreme ease of use in the basic functions of constrained frequent set mining, such as the creation, modification, and deletion of constraints. If a set of answers is found to be inadequate, queries can be edited in a matter of seconds, and the phases reiterated again to find the interesting data sets. User interaction is very prominent throughout the prototype. There are a variety of automated selection helpers which interact with the user to help one select specific candidate pairs to find relationships on. In addition, there are other features such as sorting and graphing which assist the user in displaying and visualizing candidates in more appropriate and intuitive ways. The following sections describe some of the aspects of the prototype which contribute to and exhibit the new concepts of human-centred exploration and control.
4.1 Using the Two-Phase Architecture

The two-phase architecture is naturally inherent in the prototype. One of the reasons for this is that Phase I of the interface was implemented first, and then Phase II was implemented afterwards. Therefore, since Phase I was implemented as one functional unit, it was not hard to impose the two-phase architecture into the system when Phase II was designed.

Our prototype fully supports the two-phase architecture by making it very easy to reiterate through the two phases. In Phase I, after constraints are created and the mining algorithm is executed, the antecedent/consequent sets and two-variable pairs are displayed in the prototype. At this stage, users can choose to modify the view of the sets and then move on to Phase II, or they can close the display containing the generated sets and redefine or modify their constraints. This latter action would be reiteration of Phase I.

Should they choose to move on to Phase II in the prototype, they would then select the sets they want to find relationships on, specify the relationship metric and thresholds and then run the corresponding algorithm. The associated pairs will be subsequently displayed in a separate window. At this point in the system, the user can choose to stop and save the associations, or can easily reiterate through Phase II again by closing the window and reselecting sets or redefining relationship variables.

4.2 Constraint Handling

The management of constraints was the first issue handled when considering how the user would interact with the prototype in Phase I. Since constraints are at the core of constrained frequent set mining in our prototype, the system had to make
it easy for users to create and manage constraints. To accomplish this task, the
author looked to some commercial database programs, such as Access and dBase,
to see how users would interact with their query management modules. Finally, all
the good points of the different designs were melded into one design that was used
for the prototype.

First of all, it was noted that the most frequent operation performed by CAP
users would be the creation of constraints. In addition, users had to be able to
create single variable constraints for the antecedent and consequent, and two variable
constraints between the antecedent and consequent. We therefore decided that three
different modes for constraint creation should be explicitly provided. Users can
erenter “antecedent constraint insertion” or “consequent constraint insertion” modes
to create single variable antecedent and consequent constraints, or they can enter
“two-variable constraint insertion” mode to create two variable constraints.

Entry points into these three modes were put right on the prototype’s main win-
dow for easy access. At a later point in time, we realized that users should also be
able to see on the main window all the constraints which have been specified so far.
This led us to design the main window of the prototype like Figure 4.1, with the
buttons “Antecedent Constraints”, “Consequent Constraints” and “Two-Variable
Constraints” displayed on the left side of the window, and a textbox displaying
the current constraints on the right side of the window. This textbox is called
a Constraints Display window. When a user presses on the “Antecedent Con-
straints” or “Consequent Constraints” button, he can enter antecedent or consequent
constraint insertion mode where he can specify single variable constraints. When
a user presses on the “Two-Variable Constraints” button, he enters two-variable
constraint insertion mode where he can create two variable constraints.
4.2.1 Creating Constraints

If we refer back to Figure 2.3, when inserting single variable antecedent or consequent constraints, the user should be able to create:

- frequency constraints
- class constraints
- domain constraints, and
- aggregate constraints

Consequently, both the Antecedent Constraints Creation Dialog and Consequent...
Constraints Creation Dialog windows should allow one to create any of these constraints. In addition, it was noted that one of the most important guidelines in UI design is that users should only have access to what they want and need to see. Therefore, given that a user can specify so many different types of constraints in each of the antecedent and consequent constraint insertion modes, it was decided that the best way to have the user interact with the Constraints Creation Dialogs is through notebook widgets. A widget is an element of a GUI, like a scrollbar or a textbox. A notebook widget is a widget which resembles a notebook containing tabs separating different sections. The Antecedent Constraints Creation Dialog illustrated in Figure 4.2 is implemented with a notebook widget. The dialog box opens when the user clicks on the “Antecedent Constraints” or “Consequent Constraints” buttons on the main window.

![Antecedent Constraints Creation Window](image)

Figure 4.2: Antecedent Constraints Creation Window

The figure above shows the Aggregate tab of the Antecedent Constraints Creation Dialog box displayed. In this tab, one can select variables, aggregates, attributes, and criteria from the drop-down boxes in the window to make a constraint. The constraint is then added to the current query by clicking the “Add” button. How-
ever, before users are able to create any constraints at all, they have to first create set variables for the antecedent and consequent in the Variable tab of the Antecedent Constraints Creation Dialog. Each variable can then have its own support value, domain, and constraints.

Creating constraints in this way is both easy and efficient. All users have to do in the prototype to create a constraint is open up a constraint creation dialog box, click on the appropriate tab to create a constraint of the corresponding type, select the required elements to construct the constraint, and click a button.

4.2.2 Modifying, Copying, Reordering and Deleting Constraints

Another important goal of the prototype is to allow for reiteration of the phases. That is, when a certain set of constraints does not return interesting rules, then the set should be changed. Therefore, facilities had to be created to allow users to easily modify, copy, reorder and delete constraints.

After constraints are added, they cannot be changed inside any of the Constraints Display windows. The underlying CAP algorithm expects them to be in a certain format; consequently, it was a design decision to prevent users from modifying constraints erroneously. Instead, specialized commands have been created allowing users to modify, copy, reorder and delete constraints. The user can interact with these commands through the Edit Popup Menu which is mapped by right-clicking inside any Constraints Display window or clicking on the “Edit” button inside Constraint Creation Dialog boxes. These commands are also listed under the “Edit” menu of the main window.

Before modifying, copying or deleting, users have to first select the constraint to edit by clicking on it in any Constraints Display window. This will highlight it and make it current. Figure 4.3 shows the accessibility of the popup menu containing the editing commands.

We wanted to make the modification module as intuitive as possible, and as
consistent as possible with how constraints were created. This led the author to make the decision that when a constraint is modified, a **Modify Constraint** window will pop up which will be almost identical to the window in which the constraint was originally created. This of course will be different depending on the type of constraint being edited. It is then almost trivial for the user to change a few values to modify the constraint. In Figure 4.4, we show the **Modify Constraint** window opened to modify a two-variable set constraint \( \text{SO.TYPE is subset of TO.TYPE} \).

The fields are already filled in once the window is opened. To change the constraint to \( \text{SO.TYPE is superset of TO.TYPE} \), all we have to do is click inside the “Constraint” field, select “superset” from the drop down box, and click “OK” to
Modify Two-Variable Set Constraints

Select the values for the set constraint and click Add to add:

Variable: S
Attribute: TYPE

Constraint
Criteria: subset

NOT:

Variable: T0
Attribute: TYPE

Figure 4.4: Modify Constraint Window

replace the old constraint with the new one. The newly modified constraint will then immediately appear at the top of all Constraints Display windows.

During a preliminary evaluation of the interactivity in our system, it was discovered that users should also be provided with a command to “copy” constraints. The command would not be used literally to add a copied constraint since that would be illegal. Instead, it would allow users to retrieve values from an existing constraint, change some of the fields, and then add the newly modified version. This type of “copying” can be extremely useful when, for example, we are creating a handful of aggregate constraints differing in only the aggregate field, or when we have to specify the same support value for multiple variables.

To ensure consistency again, the Copy Constraint dialog box was modelled after the Modify Constraint window. Therefore, when a constraint is highlighted and the copy command is invoked, a Copy Constraint window will open resembling the window the constraint was originally created in. A Copy Constraint window showing the constraint count(T0.PRICE) < 5 is shown in Figure 4.5.

If we wanted to create the constraint max(T0.PRICE) < 5, all we would have to do is change the aggregate from count to max in the Copy Constraint window.
and click the "Add" button.

In the current design of the system, constraints appear at the top of the display windows when they are added or modified. However, we have to account for cases when users may want to change this order. Consequently, the prototype provides access to such a command, allowing users to easily shift the order of constraints in the display windows.

Besides being able to modify, copy and reorder, users may also want to delete constraints which have narrowed down the focus in their answer set a little too much. The CAP prototype provides this functionality through the "Delete" or "Delete All..." commands in the "Edit" menus. When "Delete All" is selected, the system will ask whether or not the user really wants to proceed with the deletion of all constraints. The system has been designed to always double check with the user before executing any potentially disastrous tasks.

In constraint management modules, we interact with only the information we want to see. For example, when we want to create a domain constraint, we open up a constraint creation dialog box and click on the domain constraint tab. We do not have to interact with modules for the creation of other types of constraints. Likewise,
for the modification and copying of constraints, we only see the information relevant to the constraint itself. Our system therefore makes it very difficult for users to make mistakes of any kind; in addition, the consistency of the interface makes it simpler for users to jump from one command to the other, without having to learn new ways of doing things. We believe this - the ease with which constraint handling is designed and presented to the user - will be the key to our prototype's success.

4.3 Displaying Candidate Sets

After all constraints and options have been defined, a user can click on the “Run Algorithm” button on the main window to run CAP. The GUI will then ask the user to select the transaction database(s) to mine on. If the domain of the antecedent and consequent are the same, the GUI only asks the user to supply one database. If the domain is different, the GUI will ask the user to first supply a transaction database for the antecedent, and then one for the consequent.

Theoretically, the user should only ever have to declare one transaction database. For example, if the domain of the antecedent was ITEMID and the domain of the consequent was TYPE, the underlying algorithm should be able to automatically convert the transaction database of IDs for the antecedent to TYPES for the consequent. However, for efficiency sake, the current implementation of the CAP algorithm needs to know all transaction file names because it assumes that the transaction databases for non-ITEMID types are converted ahead of time by another program.

We also could have asked for the names of transaction databases to mine on when the .cap file was created, thereby eliminating the need to ask for names before each execution of the algorithm; however, the prototype also has to function in the case where users may want to perform mining with the same constraints on many different transaction databases. Consequently, database names are required each time the user wants to run CAP.
The next issue faced now was, "How can we efficiently and intuitively display candidate sets?" It was decided that in Phase I, after a user has created constraints and executed the mining algorithm, the prototype should display candidate sets as they are being generated. One of the goals of the author's research is to create a prototype giving the user good feedback. Having the prototype display candidate sets after they are generated does not give the user any feedback on what kind of information is being found. By allowing the user to see candidate sets as they are being generated, we give the user the option of quitting the execution and redefining the constraints if the sets found so far are not interesting.

4.3.1 The Candidate Set Window

There are three different types of output generated for any execution of the CAP or Apriori+ algorithms. There are the antecedent sets, the consequent sets, and the candidate pairs created with the antecedent and consequent. We decided to provide the user with as much feedback as possible; therefore, we output all three types of answers in the prototype, even though users technically only have to interact with output containing the two-variable pairs, since relationships are only calculated on candidate pairs. However, since constraints can be applied on the antecedent and/or the consequent, we want to show the user how the constraints have affected their data or narrowed down the focus of the mining; allowing users to explore their data in this way is of course, one of the main features of the prototype.

As the algorithm is running, one should be able to see the constrained antecedent, consequent sets and two-variable pairs. To represent such feedback in the prototype, the Candidate Sets window is implemented with a notebook widget. The first tab in the window is the "1-Var" tab containing the one-variable antecedent and consequent frequent sets. The antecedent frequent sets are displayed in the left side of the Candidate Sets window, and the consequent sets are displayed on the right. The "2-Var" tab contains the pairs of sets formed by joining the one-variable
antecedent and consequent sets. Now, note that if two-variable constraints were applied to the \((S,T)\) pairs, the output in the "1-Var" tab will contain the projection of the original frequent sets. We can explain this with a simple example. Suppose there were antecedent frequent sets \{1\}, \{2\} and consequent frequent sets \{3\}, \{4\}. If there were no two-variable constraints, the resulting two-variable sets would be \{\{1\},\{3\}\}, \{\{1\},\{4\}\}, \{\{2\},\{3\}\}, \{\{2\},\{4\}\}. On the other hand, suppose there was a two-variable constraint stating that the sum of the items in the antecedent set plus the sum of the items in the consequent set have to be less than 5. The two-variable output then becomes the one pair \{\{1\},\{3\}\}. Now, the projected one-variable answers will be \{1\} for the antecedent and \{3\} for the consequent. We need to project one-variable sets in the prototype to ensure consistency between the one-variable and two-variable answers. Figure 4.6 shows frequent sets in the Candidate Sets window.

In the Candidate Sets window, there is also a progress bar which advances slowly as execution progresses. The progress bar does not keep track of how much longer the algorithm will take to complete; instead, it is an indication that the algorithm is still running and has not stalled. Therefore, after the meter in the progress bar reaches the end, it will begin again and continue to increment until the algorithm is finished executing. It might seem silly, then, to have a progress bar at all, seeing that it does not provide the user with any relevant information; however, what it does do is tell the user that the program is still running, and for an application that can take hours to complete, any indication that the algorithm has not stalled is a good sign.

### 4.3.2 Sorting Candidate Sets

To encourage user exploration and interaction, the prototype enables users to rearrange their data in different ways. The purpose of having breakpoints in the new framework is to permit the user to selectively choose which pairs of frequent sets
he/she wants to find relationships on. Not all candidate pairs may be important enough to justify the extra processing time. Thus, there needs to be a facility allowing one to rearrange or sort frequent sets and pairs, making it easier to view and choose the specific pairs of interest. Users may often want to select the pairs with the ten highest frequencies, or pairs which are "closest" in distance to a certain constraint.

Based on these requirements, the following has been done. The prototype allows a user to sort frequent sets based on the set's frequency or the set's distance from a certain constraint. One can also choose to view all sets or only maximal sets in ascending or descending sort order, or by the count of the number of items in the set. The window for specifying sort constraints is shown in Figure 4.7.
The design of the Sort Candidate Sets window is modelled after the standard sort dialog boxes used in most commercial applications. One can specify a maximum of three sort constraints in each of the 1-variable antecedent, 1-variable consequent, and 2-variable answer sets. In the Sort Candidate Sets dialog box, there are four sections. Sort constraints can be selected in the top three sections. One of the more popular sort criteria included is frequency. Other sort criteria that can be selected are constraint distances. Since our prototype is focused around the central notion of constraints, there should be an easy way for the user to view how "close" a frequent set or pair is from a particular constraint. This is what is meant by constraint distance.

The notion of distance is different for each type of constraint. For a one-variable domain constraint, the notion of distance is measured by the number of items in the set. For a two-variable set constraint, the notion of distance is the difference
between the number of items in the antecedent and the number of items in the consequent. Table 4.1 lists the distance value used in the prototype for each type of constraint. \( S \) and \( T \) represent sets.

One of the biggest problems faced while designing Phase I was the question of how to intuitively represent constraint distance criteria in the sort windows. At first, the actual constraints were used as the criteria; however, this confused users since it was not immediately apparent what it meant to sort by, i.e. \( \text{count}(T0.TYPE) \) is < 5. We then changed the prototype to show (as criteria) how the distances were calculated instead of the constraints themselves. While the notion of criteria is represented more intuitively by this solution, unfortunately, the mapping between constraints and constraint distance will not be as obvious to the novice user. To assist the user in making this mapping, the current set of constraints can be opened in a small pop-up window at the click of the “Query” button on the Candidate Sets window. This was implemented so that users could more easily see how the constraint distance sort criteria mapped to the constraints. The sets in Figure 4.8 are sorted by increasing frequency. The values of the sort criteria(s) on the sets are always shown in the left column of the output windows.

The actual implementation of the sort algorithm is accomplished by taking each

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Distance Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Var Domain</td>
<td>(</td>
</tr>
<tr>
<td>1-Var Aggregate</td>
<td>(agg(S)) where (agg) is the aggregate in the constraint</td>
</tr>
<tr>
<td>2-Var Set</td>
<td>(\text{abs}([</td>
</tr>
<tr>
<td>2-Var Relational</td>
<td>(</td>
</tr>
<tr>
<td>2-Var Aggregate</td>
<td>the greater of (aggS(S) - aggT(T)) or (aggT(T) - aggS(S)), where (aggS) is the aggregate for (S) and (aggT) is the aggregate for (T); if either (S) or (T) do not have an aggregate, apply (\max) to the set if all values in the set are less than the aggregate on the other set, otherwise apply (\min)</td>
</tr>
</tbody>
</table>

Table 4.1: Constraint Distance Calculation
### Figure 4.8: Sets sorted by Frequency

The CAP Candidate Sets tool allows users to sort data by calculating the sort value of the set as determined by the sort criteria (i.e., frequency or count). The set's "location" in the display along with its calculated sort value is stored in a list; the list is then sorted by Tcl according to the increasing/decreasing order chosen by the user. After the list is sorted, each set's new location in the output is its position in the newly sorted list.

#### 4.3.3 Viewing Candidate Sets

Besides allowing one to sort data, the prototype also encourages exploration and interaction by giving users the ability to change the way they view the data. This is important because users may not always want to view sets only by their sorted order.

<table>
<thead>
<tr>
<th>S Answers (projection)</th>
<th>T Answers (projection)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Frequent Sets Generated</strong></td>
<td><strong>Frequent Sets Generated</strong></td>
</tr>
<tr>
<td><strong>Sort Constraints:</strong></td>
<td><strong>Sort Constraints:</strong></td>
</tr>
<tr>
<td>1. Frequency</td>
<td>1. Frequency</td>
</tr>
<tr>
<td>16 (juice, alcohol)</td>
<td>10 (Shreddies)</td>
</tr>
<tr>
<td>18 (snacks, alcohol)</td>
<td>10 (Celery)</td>
</tr>
<tr>
<td>18 (soda, alcohol)</td>
<td>10 (Doritos, Mzzarella)</td>
</tr>
<tr>
<td>26 (snacks, juice)</td>
<td>10 (Tostitos, Bugles)</td>
</tr>
<tr>
<td>28 (snacks, soda, juice)</td>
<td>10 (Tostitos, Lecture)</td>
</tr>
<tr>
<td>32 (soda, juice)</td>
<td>10 (Doritos, CheetosBugles)</td>
</tr>
<tr>
<td>32 (alcohol)</td>
<td>10 (Cheetos, Tostitos, Bugles)</td>
</tr>
<tr>
<td>44 (juice)</td>
<td>12 (Tomato)</td>
</tr>
<tr>
<td>52 (snacks, soda)</td>
<td>12 (Doritos, Bugles)</td>
</tr>
<tr>
<td>60 (snacks)</td>
<td>12 (Cheetos, Mzzarella)</td>
</tr>
<tr>
<td>60 (soda)</td>
<td>12 (Tostitos, Cheddar)</td>
</tr>
<tr>
<td>14 (Chees, Tostitos)</td>
<td>12 (Mzzarella, Lecture)</td>
</tr>
<tr>
<td>14 (Cheetos, Bugles)</td>
<td>12 (Cheestos, Broccoli)</td>
</tr>
<tr>
<td>16 (Cauliflower, Lettuce)</td>
<td>12 (Mzzarella, Caes, Lettuce)</td>
</tr>
</tbody>
</table>

---

"CAP Candidate Sets"
or their generated order. For example, a supermarket manager who applies data mining to his transaction database may only want to know the maximal associations generated from the program. In this case, our prototype can fully assist the manager in displaying only the maximal sets of the output. In addition, the prototype can also display sets by increasing or decreasing cardinality, and combine the two different views; that is, maximal sets sorted by frequency can be displayed according to decreasing cardinality. Such functionality is important to exploratory association mining because it allows one to explore and interact with the data at different levels and view it in more abstract or detailed ways.

In the prototype, toggling between showing all sets and showing only maximal sets can be easily achieved at the click of a button by changing the selection inside
the “Show” box of the Sort Frequent Sets window. Likewise, all a user has to do to change the order in which sets are viewed is to select a different viewing option (such as “Increasing Level” or “Sort Order”) from the “View” box of the Sort Frequent Sets window. Figure 4.9 shows only the maximal sets in Figure 4.8 and Figure 4.10 shows the antecedent sets in Figure 4.8 sorted by increasing level and the consequent sets sorted by decreasing level in the prototype.

Figure 4.10: Sets viewed by level

4.3.4 Graphing Sorted Sets

Graphical visualization is one of the most effective methods to further enhance the attractiveness and analytical capabilities of any program. Most would use Microsoft Windows any day over the text-based interaction with UNIX. This is the exact idea
which prompted the need to augment the prototype's exploratory capabilities with graphical visualization.

In particular, the prototype is capable of visually representing the results of sort criteria on sets and pairs through bar charts. Charts can be created with the sort results of the antecedent, the consequent, or the two-variable pairs. Figure 4.11 shows the graph of the antecedent set, sorted first by Frequency, and then by the constraint distance criteria \texttt{count(S0\.TYPE)}. The number of each frequent set is plotted on the x-axis of the graph, and the sort values of the frequent sets are plotted as the bars extending up the y-axis. In addition, users can click on any of the bars of each frequent set to display the set's contents. A numerized list of all the frequent sets and their sort values can also be brought up by clicking on the "Value" button at the bottom of any chart.

![Graph of Sort Results](image)

Figure 4.11: Chart of sort results for two criteria

Graphs produced by the prototype are useful for many reasons. The most important reason is probably the fact that viewing data on a chart is easier than viewing
it in text. In a demonstration of this prototype to some researchers, all of them agreed that the graphical representation of the sort values was more intuitive and easier to view than the textual representation.

To further enhance the visualization capabilities of the graphing package, the author added facilities allowing users to activate or deactivate bars representing the sorted values, and to activate only sets of a specific level. This idea came from reading about certain visualization techniques used to manage code in large software programs [10]. In the prototype, these visualization techniques can be applied by clicking on the "Edit" button at the bottom of any chart and going to the "Activate" tab of the Edit Chart window. The window is shown in figure 4.12.

![Edit Chart Window](image)

Figure 4.12: Edit Chart Window

The settings above will activate only the one-item sets in figure 4.11 and will also deactivate sort criteria 1 (left bars) of all sets in the graph. The results a user will see activated (shown in Figure 4.13) are all sort values of criteria 1 on one-item sets.
This type of "filtering" is necessary when users want to easily pick out which sets are closest to a constraint, or which sets have relatively high values in all their sort criteria. In the prototype, enabling users to interact with the sorted sets graphically makes the analysis task much simpler than only allowing them to view the data textually.

![Graph of S Sort Results](image)

Figure 4.13: Results of the activation/deactivation technique

### 4.4 Relationships

Designing how the user would interact with the system in Phase II involved quite a few steps. The first step was deciding how to make candidate set selection easier for the user. An execution of CAP or Apriori\(^+\) can output tens of thousands of candidates; it is therefore unthinkable to have the user manually search through all the sets in order to find those specific ones of interest. Allowing the user to sort the output makes the search easier. However, we still need to provide general selection...
functions to the user. For example, suppose we want to select all two-variable pairs of sets whose antecedent has only two items. The prototype should be designed with means of allowing one to easily make this selection. After considerable thought, an appropriate design was constructed, giving the prototype all the functionality necessary to automate the selection process for users.

4.4.1 Choosing Interesting Sets

One of the main features of the prototype is that it can significantly narrow down focus during the calculation of relationships by allowing the user to select only those pairs of sets which they want to find associations with. The design of this "selection" process took some time, but eventually, a solution became clear. After sets are generated by the mining algorithm, they are displayed in the Candidate Sets window where their views can be changed, and they can be sorted or graphed. The functionality of the prototype at this stage allows the user to modify the view of the sets before they are selected, making the future selection process easier for the user: When a user is finished rearranging the candidate pairs, he then has to enter "Selection Mode" in order to start the selection process. This mode can be entered by clicking on the "Selection Mode" button in the Candidate Sets window. This will "open" the Candidate Sets Selection window, which is technically just the two-variable tab of the Candidate Sets dialog, with the exception that users can now interact with the sets by selecting them, and the buttons on the right panel now activate selection functions. Only two-variable sets are shown in Selection Mode because relationships are calculated on pairs of sets. Figure 4.14 displays the Candidate Sets Selection window.

Manually selecting a set is accomplished by clicking anywhere inside the line representing the set; clicking an already highlighted set will deselect it. A selected set is highlighted in blue; this separates sets selected for Phase II from sets ordinarily highlighted with the touch of a mouse. Users can also select more than one pair at
Candidate Set Selection

(S, T) Answer Pairs

<table>
<thead>
<tr>
<th>Pairs selected: 11</th>
</tr>
</thead>
</table>

Figure 4.14: Candidate Sets Selection Window

a time by clicking on the first pair in the selection, and then dragging the mouse down to the last pair of the selection. Any sets which were already selected at the time will be deselected; any pairs not selected will be selected. One can always "undo" the last selection/deselection by clicking the "Undo Last" button on the window, or can clear the whole selection by clicking "Clear Selection". The number of pairs highlighted is always displayed in the status bar, but due to large memory requirements, the current version of CAP is hardcoded to work with at most 300 pairs of sets; therefore, when more than 300 pairs are selected, the prototype will report an error.
The Selection Helper

Since there can be as many as tens of thousands of candidate pairs, the selection task can sometimes get tedious if done manually. Therefore, to make the job easier for the user, the prototype provides a **Selection Helper** which can automate the selection process. With the **Selection Helper**, users can specify frequency, maximality, cardinality, attribute or numerical criteria to automatically highlight only those sets they are interested in, saving them the trouble of having to manually search through all sets.

The **Selection Helper** can be accessed by clicking on the “Select Helper” button in the **Candidate Sets Selection** screen. This pops open a notebook-style dialog box with the sections “Frequency”, “Maximality”, “Cardinality”, “Set Attribute”, “Numerical Attribute” and “Constraint Distance”. Inside each of these windows, users can specify corresponding criteria which can add all pairs satisfying the criteria to the selection, or filter the current selection with only valid pairs. For example, Figure 4.15 shows the **Maximality Selection** window.

![Selection Helper](image)

**Figure 4.15: Maximality Selection Window**

Currently, the **Maximal S and T Sets** option is selected. If a user clicks on the “Add to current selection” button, then all pairs containing a maximal an-
tecedent set and a maximal consequent set are highlighted in the Candidate Sets
Selection window.

In Figure 4.16, the Constraint Distance Selection window is shown. In the
window, it specifies to select all pairs of sets whose minimum price of items in the
consequent minus the maximum price of items in the antecedent is less than 5. If we
clicked on “Filter current selection”, then only those pairs satisfying the constraint
within the current selection are highlighted. The pairs in the current selection not
satisfying the constraint are deselected.

Figure 4.16: Constraint Distance Selection Window

With the “Add” and “Filter” commands in the selection helper, any disjunction
or conjunction of selection constraints can be created. Given certain constraints in
mind, this feature of the prototype allows users to easily narrow down the number
of pairs to find relationships on, without the hassle of having to manually find sets
satisfying the constraints. The prominence of user interaction in the process of
selecting candidate pairs is a prime example of the prototype’s support for human-
centred exploration.
4.4.2 Selecting Different Metrics

In the classical model, relationships were only measured through confidence. Our prototype does better than this, and gives the user a choice of three different relationship metrics to choose from instead of just one, as proposed of the exploratory model. This is desirable because different metrics will return different association rules, and sometimes users may want to find all rules using all metrics, or reinforce the strength of a rule by testing it with another metric. Therefore, in the prototype, relationships can be calculated using confidence, correlation (chi-squared value) or dependence. In addition, the user can very easily interact with the prototype to choose suitable metrics and to supply threshold values for the metrics chosen. Many revisions were done on the interface for selecting relationship metrics before finally coming up with the design in Figure 4.17.

![Specify Relationship Window](image)

**Figure 4.17: Specify Relationship Window**

The **Specify Relationship** window can be opened by clicking on the “Phase 2 >>” button in the bottom of the **Candidate Sets Selection** window or by clicking “Edit...Specify Relationship” under the main menu. The default relationship metric is “confidence”, with a minimum threshold of 60% and a minimum support of 0%. When running under “no breakpoints” mode (the classical model), users will have to open the **Specify Relationship** window from the main menu before running CAP if they want to change the relationship metric and threshold.
In the Specify Relationship window, one can define relationships using confidence, correlation or dependence by clicking on the corresponding radio button. This type of interface makes it immediately intuitive how the user should go about supplying the threshold values, which are entered in on the same line. Clicking "OK" in the Specify Relationship window opened from the main menu will save the relationship settings into the system. Clicking "OK" in the dialog box when it is opened from the Candidate Sets Selection window will immediately run the algorithm to find the related pairs of sets. When the algorithm is finished executing, the Related Sets window will pop up, displaying the related antecedent, consequent pairs (rules) and their significance values. This window is shown in Figure 4.18.

<table>
<thead>
<tr>
<th>SET</th>
<th>SIGNIFICANCE MEASURES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Metric: Confidence</td>
</tr>
<tr>
<td></td>
<td>Threshold: 20%</td>
</tr>
<tr>
<td></td>
<td>Support: 3%</td>
</tr>
<tr>
<td>Conf</td>
<td>Set</td>
</tr>
<tr>
<td>46.7</td>
<td>{ snacks }, { juice }</td>
</tr>
<tr>
<td>30.0</td>
<td>{ snacks }, { cheese juice }</td>
</tr>
<tr>
<td>20.0</td>
<td>{ soda }, { cheese alcohol }</td>
</tr>
</tbody>
</table>

![Related Set Window](image)

Figure 4.18: Related Sets Window

In the Related Sets window, the user can save the association rules into a text file by clicking on the "Save..." button. One can also strengthen or relax relationships by reiterating through Phase II again, redefining the relationship variables. This is easily accomplished by closing the Related Sets window and opening the
Specify Relationship dialog box again.

### 4.4.3 Mining between sets of different types

One of the faults of the classical model was the rigid notion of relationship. The model did not account for cases where users may want to mine between sets of different types. Our prototype fixes this problem by allowing users to create class constraints which specify what domains the antecedent and consequent come from. These domains can be of the same or of different type; for example, we can use the class constraint window in Figure 4.19 to specify that the antecedent set (represented by the set variable S0) is in the domain of the TYPE attribute.

![Class Constraint Creation Window](image)

Figure 4.19: Class Constraint Creation Window

Later, we can specify with the same window that the consequent set is in the domain of the ITEMID attribute. The prototype is fully capable of handling this type of "hetero-domain" mining, as are the underlying algorithms.
4.5 Selecting Different Algorithm Options

There are two options the user can set in the prototype which will change how the underlying mining algorithm runs. The options are found in the Options dialog box. The window, shown in Figure 4.20, is opened by selecting “View...Options” from the main menu.

![Options Window](image)

Figure 4.20: Options Window

4.5.1 The Classical vs. The New Model

The prototype displays all the improved characteristics of the new model proposed in [17]. However, a well-rounded prototype should be able to run in the new two-phase exploratory model as well as the classical model (where there are no breakpoints between Phase I and Phase II). The CAP prototype is capable of doing just this, and the option can be selected by checking the “Suppress Breakpoints” checkbox in the Options window.

If this box is selected, there will be no breakpoints in the execution. The CAP or Apriori+ algorithms will run and find all frequent sets. The prototype will then assume that the user wants to find relationships between all pairs of antecedent and consequent frequent sets, and proceed to execute the Phase II algorithm. The
metrics and thresholds for Phase II have to be selected ahead of time by supplying them in the Specify Relationship window, which can be opened from the main menu under "Edit...Specify Relationship". If the "Suppress Breakpoints" checkbox is not selected, it means that the CAP or Apriori\(^+\) algorithms will find all frequent sets and stop. The user will then be able to manually select only those pairs he wants to find relationships on, or redefine his constraints if appropriate.

The decision to offer this first option is made based on the discussion in [17]. The system should give the user the choice to run under the classical Apriori framework, where the concept of "breakpoints" is not included, or the option to use the new framework, which encourages exploration and gives the user more control. The prototype can do both.

### 4.5.2 Apriori\(^+\) vs. CAP

The second option displayed in the Options window is to run CAP or to run Apriori\(^+\). If the "Run Apriori" checkbox is selected, the Apriori\(^+\) algorithm will be executed. If the checkbox is deselected, the CAP algorithm will run. Ideally, this option need not exist since CAP is faster than Apriori\(^+\) in most cases, thereby making it inefficient to ever choose execution with Apriori\(^+\). Apriori\(^+\) was originally included to interactively test the correctness and efficiency of CAP through the system; however, given that this project is a prototype used to compare the classical model with the new exploratory model, including Apriori\(^+\) in the comparison and as one of the features is completely justified.

In the system, any combination of the two types of options is a mode. The default mode runs CAP with breakpoints. Since users can only adjust and see the current options through the Options window, it was decided to display the current mode at all times in the bottom right hand frame of the main window.
4.6 Database Management

The prototype provides many functions for the user to explore and interact with data generated from the mining algorithm. However, to fully promote exploration, the prototype should also allow the user to view the contents of the item and transaction databases.

4.6.1 Item Database Issues

One of the most important issues encountered during the design of the user interaction aspect of the system was to decide on the format of the databases that would hook up to the prototype, and how the user would specify these databases. Would they be Access, dBase, Oracle databases, or just simple datafiles? We wanted our system to be as general as possible, so we did not want to restrict it to work with only one type of database. Contrary to this goal, we were not able to explore the idea of implementing compatibility with all types of databases due to time constraints.

Therefore, simple text files were used as “databases”, and querying was done from these text files. The item database, dbfile.sql is semi-hardcoded into the CAP GUI and algorithm. The format of a record in this database is:

<table>
<thead>
<tr>
<th>ITEMID</th>
<th>ITEMNAME</th>
<th>TYPE</th>
<th>PRICE</th>
</tr>
</thead>
</table>

An item with ID 5, name “Doritos”, type “chips” and price 3 would consequently have the following record:

| 5   | Doritos | chips | 3    |

After deciding on the formats of the item database, next came the question of how to import the metadata of the item database into the GUI. It is necessary for the prototype to provide metadata to both the GUI and the underlying CAP algorithm. This is a form of good user feedback, and the GUI also needs this information to know the correct fieldnames to put into the “Attribute” lists in the constraint
creation dialog boxes and to assist in error handling. The CAP algorithm itself needs the metadata information to perform mining on the correct fields. In [17], it was suggested to create a general wrapper which would be able to take in any database and supply the GUI with the necessary data. However, some of the issues encountered when designing this wrapper were:

- support for different types of databases (Oracle, Sybase, Access)
- support for different query languages
- support for different types of connections between the GUI and database

Nevertheless, a wrapper has been implemented, although it does not contain all the generality which was originally proposed. The wrapper implemented is basically a text file parser. This text file parser parses SQL files containing the CREATE TABLE statements which define the item database. The text file parser retrieves table names, attribute names and datatypes, and passes the information on to the CAP GUI. Querying an item database (i.e. to find $S.Type$, or $T.Price$) is done by a simple Tcl script which does a linear search through the file.

At the beginning of each CAP session, the user will have to import the item database's metadata into the program. This is done by clicking on the "Open Database..." button in the "File" menu of the main window. When this button is pressed, a dialog box will open, prompting the user to enter or choose an SQL file containing the metadata information (the parser specifically searches for CREATE TABLE statements in this file). After the file is chosen and the "Open Database" dialog box closed, the View Metadata window will open, displaying the attributes and datatypes of all tables in the database. This window is shown in Figure 4.21.

Users can click on the name of the relation in the left frame of the View Metadata window to display the relation's attributes in the right listbox. Furthermore, the primary key for the relation is always shown centered in the bottom of the window. When users are finished viewing the metadata, they can close the window to begin their CAP session. However, since one might need to look at the metadata again
during the mining process, the View Metadata window can be opened again at any time during a session by choosing "View...Metadata" from the main menu, or by clicking on the View Metadata button on the button bar.

During the preliminary evaluation of the GUI, one comment given was that users may want to view not only the metadata but also the actual records in the item database. This is consistent with exploration; therefore, the prototype also contains a command which displays these contents in a pop-up window when the "View...Relation" command from the main menu is invoked.
4.6.2 Viewing the Transaction Database

The GUI and underlying CAP algorithm assume that a transaction in a transaction database has the format:

<table>
<thead>
<tr>
<th># Items in transaction</th>
<th>Transaction ID</th>
<th>Item_1</th>
<th>Item_2</th>
<th>...</th>
<th>Item_n</th>
</tr>
</thead>
</table>

Transaction databases are stored in binary files to save space. Unfortunately, because the files are in binary, users cannot open databases to view what transactions are in the file. Such functionality is definitely required in a data mining program; consequently, the prototype enables users to view the contents of transaction databases by providing the View Transaction File window which can be opened by clicking “View...Transaction File” in the main menu. This window is shown in Figure 4.22
The first line in the output tells the user where the transaction database is located on the file system. The rest of the output displays the transactions in the format:

| Transaction ID | Transaction |

Ideally, transactions should only consist of item ids. When the domain of the antecedent and consequent sets are of a type other than ITEMID, then CAP should look to the transaction database of IDs and query the item database to find frequent sets of another domain. However, due to the efficiency issues of this conversion, it was decided to convert the transaction database of ITEMIDs into the corresponding enumerated transaction files of TYPES, PRICES, ITEMNAMEs, etc. ahead of time. For example, the transaction database “id10.bdat” is a file of 10000 transactions where each transaction is made up of item ids. The transaction database “type10.bdat” is also a file of 10000 transactions, but each transaction is made up of the (enumerated) types of the ITEMIDs in “id10.bdat”. This is discussed in more detail in Section 3.7.

At the end of the output in the View Transaction File window is the total number of transactions in the file. If the number of transactions shown in the output is different from the actual number of transactions in the database, then it is assumed that the ones shown are cyclically repeated until the count of transactions equals the displayed value. The decision to not output all instances of repeating transactions was made on the notion that doing so would be both unnecessary and confusing to the user. Transaction databases generated to test the CAP GUI will contain cyclically repeated transactions; real transaction databases do not.

4.7 File Operations

In addition to all the features mentioned so far, the prototype also provides a file management module. This module has made the prototype even more flexible,
since users now have the ability to try out different settings and/or combinations of constraints in sessions, and then save these sessions into files to be opened up again at a later time. This saves users the effort of having to redefine their constraints each time they open the program. The author feels that this is an important aspect integrated with the new association mining framework which was not mentioned in [17].

In the prototype, users can easily create, open, save and print constraint files. Whenever a new database is imported, a new file is automatically created. In this file, a user can then proceed to add new constraints and set different options, such as running Apriori \(^+\) or running CAP. One can later save all the constraints and options of the session into a file by selecting “File...Save” from the main menu. The constraints and settings will then be saved in a file with the extension \texttt{.cap}. The “New”, “Open”, “Save”, “Print” and “Close” buttons are located under “File” in the main menu and on the button bar for convenience.

A \texttt{.cap} file is saved as a text file. This file contains variable instantiation commands which are sourced by the Tcl/Tk interpreter when the file is opened. Therefore, \texttt{.cap} files only contain information necessary to initiate a \texttt{.cap} session; they do not contain Phase I or Phase II output. We decided against this because there could be tens of thousands of frequent sets outputted in a single execution of CAP, and automatically saving all the frequent sets into the \texttt{.cap} file would result in overwhelming file sizes. There are, however, options for the user to save output from both Phase I and Phase II into normal text files, but these actions are executed only at the user’s request.

4.8 Other Features

Besides all the functionality described above, the prototype is also enhanced with other features which support user interaction and usability. These features are discussed in the following sections.
4.8.1 Display Options

Fonts in the Display

The prototype's GUI was implemented with Tcl/Tk to work across all platforms. Unfortunately, fonts look different on different operating systems. The author once ran one version of the prototype on a Windows machine, and then later on a machine running UNIX. In the Windows machine the fonts were oversized, while on the machine running UNIX one could barely read the text on the prototype. Consequently, to enhance both usability and flexibility of the prototype, a font selection mechanism
was created, enabling users to choose fonts and sizes for each type of widget. This is easily done by opening the Font Selection window under “View...Font Selection” from the main menu, which will pop up the dialog box shown in Figure 4.24. The top of the window contains a dropdown box listing names of widgets whose fonts can be changed. The listboxes in the middle of the window contain font names and font sizes.

![Font Preferences Window](image)

**Figure 4.24: Font Selection Window**

To view the font name and size of a particular widget, the user clicks on the widget name from the “Widget” drop-down box; the font name and font size of the widget will then appear in the entry boxes in the left and right frames. To change the widget’s font name and/or size, one can choose a different font name and/or size from the corresponding listboxes, and click “OK” to apply the new font and pitch size to all widgets of the chosen type.
Chart Display Options

When a chart is produced by the prototype, it is displayed in a default set of colors and fonts. The system allows one to easily change these colors and fonts through the "Edit Chart" window shown in Figure 4.12. In this window, users can change the color of the bars, background, axes, legend, tick marks, values and headings. In addition, they can also change the font and text of all labels on the chart. Flexibility of the prototype is definitely enhanced by allowing users to change the look of their charts, since these charts are at the core of data analysis and can be saved as postscript files to be printed out and analysed at a later time.

4.8.2 Error Checking

The prototype also includes a considerable amount of error checking, which is crucial to increasing the usability of any program. Users working with a program for the first time can become severely frustrated with using it if they keep making mistakes because the system is not catching any errors, since the result of this would be nasty and incomprehensible messages or system crashes. Given this, a fair amount of error checking has been implemented into the prototype, and a number of design decisions have taken into account the possibility of user errors.

It was decided that the constraints in the Constraints Display windows be uneditable by the user because constraints have to be in a certain format when passed to the underlying CAP program. It seemed all too easy for users to disrupt this format by changing a bracket here, or a comma there. As a result, users are forbidden from manually editing constraints in Constraints Display windows. Instead, modify, copy and delete commands were implemented to easily help the user change their constraints, but keep them in the correct format for the system.

When reading in the metadata from a particular database, the GUI extracts and stores the type information of all the attributes. The type information is later used to verify that the corresponding attribute is valid for constraints the user is
creating. For example, suppose the user wants to specify the aggregate constraint $\text{sum}(S0.\text{TYPE}) > 5$. In this case, the GUI will respond with the error message given in Figure 4.25.

![Error Window](image)

Figure 4.25: Error Window

The system realizes that the TYPE attribute is a VARCHAR, and not a number. Since the sum aggregate can only be applied to numbers, the system returns an error. The only aggregate that can be applied to both string and number-typed attributes is the count aggregate. We believe that this type checking is crucial in the prototype, because users can frequently make these kinds of mistakes, and not catching these mistakes will just frustrate the user and discourage them from using the system.

Another example of error checking lies in the Aggregate tab of the Constraint Creation windows. In the Aggregate window, the program does not allow the user to enter anything except numbers into the “Value” field, since aggregation is purely a numerical operation. Error checking of this type is called “gagging”, where the system prevents the user from continuing upon encountering an error. This type of error checking is very effective because it lets the user know immediately that an error has occurred.

There are several other instances of error checking in the prototype. However, the author will refrain from describing every occurrence, and instead encourage the reader to experience for himself the usability of our system.
4.8.3 Constraints Viewer

Another feature of the prototype supporting good user feedback is the Constraints Display section of the main window, which contains all the constraints created in the current session. Users can turn to the Constraints Display window whenever they want to find out what constraints they have created. However, one specifies constraints inside the Constraint Creation Dialog boxes which pop up on top of the main window. This in effect becomes a problem because the dialog boxes end up obscuring the view of the constraints. Subsequently, users have to constantly move dialog boxes out of the way in order to see the main window. This of course is not very efficient; and as a result, it was decided to include a Constraints Display window which could be easily mapped and unmapped inside the constraint creation dialog boxes.

Pushing the “Constraints <<” button in Figure 4.26 will unmap the Constraints Display window shown beneath the Variable tab. The text inside the “Constraints <<” button will then change to “Constraints >>”, indicating that pressing the button now will expand the dialog box, mapping the window again and allowing the user to view all the constraints in the current session.

4.9 Interface Implementation Issues

4.9.1 Programming Language

One key issue that arose in the first phase of the project was “What language should be used to create the user interface of the prototype?” We wanted to use a language that was easy to program with and portable to different platforms; therefore, we decided on Tcl/Tk. Programming in Tcl/Tk is very simple, and the resulting code can be used on Unix, Windows and Macintosh systems. With some modifications, Tcl/Tk code can also be run on the web, with any browser configured with the Tcl/Tk plugin. There are also many helper applications written for the toolkit, such
as tcl2c, which creates binary executables from Tcl/Tk code in order to prevent end users from accessing the source. With the ambition of having our prototype go commercial, using a portable toolkit with all these characteristics was a very viable solution.

4.9.2 Tcl/Tk problems

Unfortunately, the use of Tcl/Tk introduced a number of problems during implementation. In the UNIX version of Tcl/Tk, all the functions used to implement the GUI work perfectly. However, in Windows, some of the same commands are not working as well. For example, to track algorithm execution, the progress bar uses...
the "fileevent" command in Tcl/Tk to update its display. The progress bar works perfectly in the UNIX system; on the other hand, in Windows, because of a bug in the "fileevent" command, the progress bar does not remap properly. While the best thing to do is to wait for the next release of the scripting language (hoping by then that Scriptics will have fixed the bug), recompiling Tcl/Tk for Windows under a special patch can temporarily solve the problem. Another problem encountered was selection in text boxes. In UNIX, when a text box is marked as uneditable, a user is still able to perform selections within the text box, and is able to see the selection. In Windows, the system can tell when a selection has occurred, but does not display the selection. Currently, this has been fixed by manually implementing a selection function. Besides these few problems however, Tcl/Tk has worked very well and deserves much mention for the speed and efficiency with which the prototype was produced.
Chapter 5

Conclusion

Association mining has been, for a long time, hard to control and understand. It is precisely this reason that has prevented data mining programs from entering into and successfully existing in the commercial market. Data mining programs will not be used if systems are inflexible and the framework does not allow for human-centred exploration and control. Most of the research prototypes created so far are modeled after the black box. Transactions are passed into the system, minimum support and confidence values are supplied, and the program is put to work. Then, at the end of the processing, which can take anywhere from 10 seconds to a day, association rules are displayed. This lack of interaction with the user makes such a system unusable; it is no wonder why data mining, be it so useful in all realms of life, is not yet widely embraced by society.

In this work, the author has created the first fully functional prototype modelled after a new association mining framework. In the prototype, users have complete control of the mining process because the interface is modelled after a two-phase architecture which is fully backward compatible. The prototype completely supports exploration, and fixes the earlier problems of a “closed black box view of the system”, a “lack of focus” and a “rigid notion of relationship” inherent in the classical association mining model. The fact that users can modify how they want to
view frequent sets, and choose only interesting sets to find relationships on, is a
demonstration of opening up the black box. Frequent sets can be sorted by different
criteria, and their displays can be changed to show all sets or only maximal sets
in increasing or decreasing order. Graphical visualization of the sort results has
also been incorporated into the prototype to assist users in analyzing their data.
Furthermore, in classical Apriori, all candidate pairs were tested for significance.
In the CAP system, only interesting pairs are selected by the user and checked for
association. This significantly improves execution time in the prototype, because
now the system only has to find relationships on a narrow set of pairs.

The CAP prototype is also the only program out there to date which allows
users to specify focus through constrained frequent set queries and incorporates the
optimized CAP algorithm to handle the constrained frequent set mining. Other
data mining programs such as IBM’s Intelligent Miner and Simon Fraser University’s
DBMiner can impressively perform a variety of mining tasks, but they fail to
solve the “lack of focus” problem that is at the core of classical association mining.
Constraints created in our prototype can be used to significantly narrow down the
search space, and return only the frequent sets a user wants to see. In addition,
constraints can always be refined to provide even greater pruning power. We have
made this refining easy to do by providing all the necessary functions to manage
constraints effectively.

The usability of the CAP prototype is enhanced further by the fact that it allows
users to specify different notions of relationship. Relationships can be specified in
terms of confidence, correlation and dependence, and the system allows users to
easily mine between sets coming from different domains. We can safely say that no
other association mining prototype or program to date has been able to achieve this
kind of flexibility.

Our prototype has definitely proved its usability by providing all of the enhance-
ments of the new association mining framework. It is fast, efficient, interactive and
user-friendly; furthermore, it contains many nice additional features such as graphing, sorting, and several display and selection utilities. We therefore believe that novice users will find data mining both easy to do and useful in our prototype. The work developed here is meant to be a preliminary demonstration of where data mining programs should be headed - specifically, towards the incorporation of more user interaction. It has been said that data mining may one day assist in finding the cure for AIDS. However, it is difficult for society to adapt the technology and make it useful in any way with the classical framework. Microsoft was not successful until they introduced the Windows system, making computers interactive and usable by everyone. The same can be applied to data mining. We believe that the key to popularizing data mining is making the process understandable and encouraging human-centred exploration; our prototype is the first of its kind to show how this can be done. Hopefully, it will also soon become the first of a new generation of data mining programs aimed at expanding the data mining community to include not only researchers, but the most common entrepreneurs as well.

5.1 Future Work

The CAP system implemented here is only a prototype. Therefore, there is still much room for improvement. One area for future work is to create the general wrapper which was first described in [17]. This wrapper would be able to read metadata and records from any type of database and pass the data onto the system. To complement this generality, the CAP algorithm can also be modified to mine on databases with any number of fields of any type; this will eliminate the current limitation of mining from only the dbfile.sql file.

Another future research topic is investigating how to more efficiently handle multiple succinct constraints in CAP. The current implementation of the CAP algorithm can perform optimizations on multiple succinct constraints whose MGFs, when joined, produce only one "required" set; the multiple MGF combination lemma
was not general enough to allow us to do otherwise. Concentrating on correcting this lemma should eliminate this restriction and allow us to optimize run time in all cases where multiple succinct constraints exist.

The GUI is currently designed to be general enough to support multiple variables for the antecedent and the consequent. Future work in this direction with the algorithm is also desirable. Another improvement that can be made to the system is to expand the charting tools and possibly research other visualization techniques which will be useful for constrained frequent set mining.

Single-variable constraints have antimonotone and succinctness properties which enable us to speed up the mining process. The classification of two-variable constraints has already been presented in [18]. The next logical step in the direction of future work is to modify both the GUI and the algorithm to take advantage of the pruning capabilities of two-variable constraints.
Bibliography


