knowledge discovery:

efficient mining of frequent patterns from transactions

BENG CHIA HONG
Final Year Individual Project Report

Johann Ting
iksr08@doc.ic.ac.uk

Supervisor: Dr Moustafa Ghanem
Data Mining Group
Department of Computing
College of Science, Technology and Medicine
Queens Gate, London SW7 2BZ, UK

June 2001
acknowledgements

it is appropriate that this thesis should begin
with my acknowledgement of the people who
have given generously and selflessly to assist me
throughout the course of this project.

for going beyond the call of duty,
i am truly most grateful to my supervisor dr moustafa ghanem
for his continually encouraging me to stretch myself, his invaluable
insights, his tireless energy, and his willingness to share from his
experience and knowledge in our many lengthy encounters.

my second marker dr yike guo, christian
brennkmeijer, patrick wendel and other members of the
data mining group have also made many (often impromptu, but
always useful!) contributions to those encounters, and helped
extend the scope of discussion and uncover possibilities.
i extend my thanks to them as well.

mrs margaret cunningham offered a great deal during our one
discussion on data types, for which I am very grateful.

finally, i would like to acknowledge
the circle of family and friends who have
provided me with the immeasurable support
that has made both my project and my time
at imperial college as a whole both
educational and memorable.
abstract

the world around us is very much composed of entities
correlated with one another by some connection, sometimes
obvious, and sometimes less so. the value of being able to identify
these relationships is considerable in such areas as diverse and
contrasting as business and science, finance and medicine.

the availability of constantly increasing computing
power to manage and make some meaning of all this
information is encouraging today no industry can realistically
compete at a high level or advance satisfactorily without
taking full advantage of this technology.

however, the very same information age driven by this demand
has triggered an explosion providing data of such quantity that the
phrase “unable to see the wood for the trees” truly takes on new meaning!
data mining is the art (or science) of analysing this data to find conclusions
and relationships that would otherwise go unnoticed, thus making
the much better use of these vast quantities of data.

this thesis presents implementations
of highly efficient structures and algorithms
to represent and help identify frequent patterns
and extends these structures and algorithms to include
other types of related data groups and analyses.
a performance evaluation examines their
efficiency and explains the results.

finally, new structures are suggested
to represent the results of the mining for study. some
ideas are also outlined for taking the foundation laid by this
project and developing it further.
# table of contents

ACKNOWLEDGEMENTS ........................................................................................................... 1
ABSTRACT ............................................................................................................................ 2
TABLE OF CONTENTS ........................................................................................................ 3

1 INTRODUCTION........................................................................................................... 6
  1.1 Data Mining .............................................................................................................. 6
  1.2 The Supermarket Story ............................................................................................ 7
  1.3 My Contribution ....................................................................................................... 9

2 BACKGROUND .............................................................................................................. 10
  2.1 The Frequent Pattern Mining Problem ................................................................. 10
    2.1.1 Patterns ............................................................................................................. 10
    2.1.2 Sequences ......................................................................................................... 11
    2.1.3 Transactions .................................................................................................... 12
    2.1.4 The Data Set .................................................................................................... 12
    2.1.5 Frequent Patterns ........................................................................................... 13
    2.1.6 Associations ................................................................................................... 13
    2.1.7 Independence of Naming Conventions ........................................................... 13
  2.2 Candidate Generation ............................................................................................ 14
    2.2.1 The Explosive Candidate Generation Problem ............................................. 14
  2.3 The Brute-Force Heuristic ..................................................................................... 15
  2.4 The Apriori Heuristic ............................................................................................ 16
  2.5 Summary ................................................................................................................ 17
# 3 THE PATTERN TREE ALGORITHM

## 3.1 Representing the Data
- 3.1.1 The Dataset
- 3.1.2 The Tree Data Structure
- 3.1.3 The Frequent Pattern Tree
- 3.1.4 Optimising for Patterns
- 3.1.5 Handling Strict Sequences

## 3.2 Mining the Data

## 3.3 Summary

# 4 IMPLEMENTATION

## 4.1 Java

## 4.2 The Classes
- 4.2.1 In the beginning
- 4.2.2 Problems
- 4.2.3 Adaptation and Solution
- 4.2.4 Integrating Association Patterns and Sequences

## 4.3 The Pattern Analyser

# 5 PERFORMANCE ISSUES

## 5.1 Test Bench

## 5.2 Sequence Analysis
- 5.2.1 Overall Execution Time
- 5.2.2 Correlation to Pattern Quantity
- 5.2.3 Correlation to Threshold

## 5.3 Sequences: Breakdown Times
- 5.3.1 A Note on Pre-caching
- 5.3.2 Correlation to Pattern Quantity
- 5.3.3 Correlation to Threshold

## 5.4 Association Pattern Analysis
- 5.4.1 Correlation with Pattern Quantity
- 5.4.2 Correlation with Thresholds

## 5.5 Association Patterns: Breakdown Times
- 5.5.1 Comparison with Sequence Algorithm
1

introduction

1.1 Data Mining

Making deductions from available information is nothing new – for many years quantitative methods have been used to explain scientific phenomena, statistics to find hidden trends. Driven by both curiosity and the quest to put himself at a competitive advantage, man and woman have learned to use this knowledge to their benefit, but the limitations of the human brain always meant that the quantity of information which could be taken into account by manual computation could only be at best a selective sample, and at worst an unrepresentative fraction of the complete picture.

An increasing percentage of data stored and transferred worldwide today no longer exists in physically tangible form. Paper has made way for digital storage and digital transmission, mediums that are significantly less restricted in terms of accessibility and physical space occupied. Developments in technology providing much greater storage capacity and more instant availability lend themselves well to analysis, and have finally made analysis on very large (and increasingly so) datasets possible.

The field of data mining is the branch of computing that studies the issues and methods of dealing with these large amounts of data. The large data store can be likened to an underground mine, and mining to the often tedious process of locating the “gems” by sifting through the contents of the mine and filtering the irrelevant and unwanted.

One of the areas that attracts the most interest (and one of the areas with the most applications) deals with frequent pattern mining. Frequent pattern mining plays a vital role in mining associations, sequential patterns, correlations, causality, episodes, multi-dimensional patterns, max-patterns, partial periodicity, emerging patterns, and many other data mining tasks. [1]

Although they are all related in many ways, this project deals mainly with frequent pattern mining for mining associations and sequential patterns, and provides a foundation for analysis of some of the other derivations.
1.2 The Supermarket Story

Suppose you were the manager of a supermarket. Being the type of modern manager who believes in leadership by example, you choose to one day do a shift behind the cashier till.

Throughout your shift you face a continuous stream of customers, each with a shopping basket selection that reflects their very unique and individual tastes... or do they?

A transaction is a collection of items, and is so called because it usually represents some kind of real-world transaction, such as a shopping basket. Strictly speaking, it is not a set as there may be multiple occurrences of an item present within a transaction.

Since you are the cashier-manager of the supermarket, it is appropriate that we take a look at a typical transaction using our example of the shopping basket in Figure 1a.

![Figure 1a](image)

We can probably guess what the customer has planned for dinner!

In a busy supermarket this would be just one transaction of possibly thousands in a day. A savvy supermarket like yours might choose to keep records of each transaction carried out, and this collection of data would then be called a transaction database.

![Figure 1b](image)

Figures 1a and 1b show the relationship between the transactions and the transaction database.
As your shift passes and you serve more customers, you begin to notice certain trends in the shopping habits of customers who might otherwise seem very different! For example, you might notice that almost all those people buying crackers also bought cheese.

You begin to wonder whether there is some kind of pattern here, and if there might be some association between the two. Intrigued, you decide to take a closer look at what the transaction database and see what it might tell you.

We can generally define a **pattern** to be a set of items, and an **n-pattern** is a pattern with n items, and here we have an example of a 4-pattern:

![Diagram of pattern with items](image)

As you try to recall some of the trends you observed earlier (while the computer crawls through transactions), you notice something during your meticulous search for “white wine”. It seems that quite a number of shopping baskets with “white wine” in them seemed to have “fish” in them too. A closer look also reveals that chances are, there was some asparagus involved in the transaction as well.

Excited, you begin to imagine what this revelation could yield – by taking advantage of this apparent link, you could place a white wine stand near your fish section, as an experiment of course! A week later you check again, and as expected, there is an even higher proportion of people buying both fish and white wine.

The experiment is a success, but now you face a problem. You would like to know all the popular items in your supermarket (say those appearing in 20% of all shopping baskets) and find the associations between them so you can make better decisions that you hadn’t thought of before (the 20% is known as your **threshold** value, as illustrated in the following diagram, **Figure 2d**).
Unfortunately, you're not sure if your system has this functionality. Equally dismal (or possibly worse), if the speed of your previous experiences are anything to go for, it could take years!

1.3 My Contribution

As the problem depicted in the “Supermarket Story” becomes increasingly commonplace, more sophisticated techniques have been developed to deal with this problem.

This project was established with the aim of providing the implementation of highly efficient algorithms to perform frequent pattern mining for sequence and associations analysis. As the course of the project progressed, the scope of the project developed alongside it, and my contributions are thus as follows:

- providing an efficient implementation in Java for the frequent pattern mining algorithms
- extending the algorithms to support strict sequences and provide basic rule mining
- providing a simple query tool for frequent pattern matching
- conducting detailed performance studies to evaluate the efficiency of each stage of the implementation as well as the overall performance
- devising methods for understanding / visualising the results
background

Given the many business-related applications to which the technique lends itself, and the growing number and availability of conveniently stored data collections suited to its purpose, it is hardly surprising that frequent pattern mining has drawn a considerable amount of attention in recent years.

This section describes some of the existing approaches to the frequent pattern mining problem, and addresses their relative merits and shortcomings. Before taking a closer look at some of the more technical aspects of these algorithms, it is worth ensuring an understanding of the problem they aim to solve.

2.1 The Frequent Pattern Mining Problem

Stripped to the basics, the basic problem of frequent pattern mining is: given a transaction database and a threshold frequency, extract from it patterns occurring frequently enough to lie above the threshold.

We have already gotten an appreciation of what these terms mean from the informal introduction, but we will now take a closer look at what they mean in the context of our problem.

2.1.1 Patterns

One general definition of a pattern is “a regular or logical form, order or arrangement of parts” [5]. Within the context of this thesis, the term pattern refers to item(s) of the same type grouped together in some kind of collection.

We have a domain itemset or set of items, which are eligible to appear in patterns, mathematically: [1]

\[ l = \{a_1, a_2, \ldots, a_m\} \]

where \( m \) is the number of distinct items in the dataset

\[ \text{Definition 2a} \]
Patterns are thus collections of items from the itemset, or more formally:

\[ Ti = \langle i_1, i_2, \ldots, i_k \rangle \]

where \( k \) is the transaction length and each \( i \in I \)

**Definition 2b**

Here are a few “real-world” examples of patterns:

**Figure 2a**

A sequence (alphabet)

**Figure 2b**

A transaction (shopping basket)

An association rule:

A sequence is a pattern with a sequential (or sometimes temporal) ordering, and may be likened to the List data structure. In other words, transactions to be used for sequence analysis have critical information on both the grouping and the ordering of the items, and multiple occurrences of an item in a pattern are also significant.

The earlier example **ABCDE** could be interpreted as a sequence, one item following the other as the arrows suggest. These could represent web pages, words, or courses of a meal, and the order they appear in can affect the results of the analysis.
A sequence is like an ordered pattern. In other words, there is a concept of precedence, whether temporal or otherwise. Mathematically, it could be represented as follows: [1]

\[ T_i = <i_1, i_2, \ldots, i_k> \]

where \( k \) is the transaction length and each \( i \in I \), and \( (i < j) \) by some ordering \( x \) if and only if \( (j < k) \)

**Definition 2c**

In our example **ABCDE**, for instance, the ordering happened to be alphabetical order.

This type of information is useful in analysing sequential data such as web click-streams (sequences of web pages visited on a web site), share prices and telephone bills, and is hence invaluable for finance and marketing applications.

In addition, sequence analysis plays a vital role in the fight against crime – fraud and money laundering detection are just two of the potential areas to benefit from research in the field.

### 2.1.3 Transactions

A transaction is simply a grouping of items, such as the shopping basket in the supermarket example, or a share’s quarter-hourly prices over a day.

The scope of what can be classified as a transaction is flexible and really depends on what the analyst is hoping to find information from. In *episodes* for example, a long stream of items (such as an itemised telephone bill) may be viewed using “windows” (such as a 24-hour span) to find patterns within that period.

Mathematically speaking, a transaction may be interpreted as being a complete pattern, or sequence, for example. There is no fundamental difference between the two – it is simply useful for us to be able to conceptually identify the complete patterns in a dataset as being transactions.

### 2.1.4 The Data Set

Conceptually we are already comfortable with the concept of a data set – it is the collection of the transactions (which as “data miners” we are hoping to analyse and gain something from).

For completeness however, here is a representation of the dataset formally, with respect to transactions. [1]

\[ DS = <T_1, T_2, \ldots, T_n> \]

where \( n \) is the number of transactions in the database

**Definition 2d**
2.1.5 Frequent Patterns

In the context of a dataset, a frequent pattern is simply a pattern that occurs sufficiently frequently in a dataset, by some quantifiable measure.

Common quantifiable ways of defining whether a pattern is considered frequent include specifying a minimum frequency (the number of times the pattern occurs), or a minimum support (the number of transactions in the database containing the pattern) – they differ in that a full pattern containing more than one occurrence of the pattern is only counted once for support. These are either specified as an absolute threshold (e.g. 5,000) or a percentage / fraction (e.g. 20% or 4/5).

2.1.6 Associations

Informally, an association rule \( X \rightarrow Y \) (where \( X \) and \( Y \) are sets of items) can be interpreted intuitively as “a transaction containing \( X \) is likely to contain \( Y \)” [4], and these are derived from the frequent patterns obtained from mining the dataset.

![Figure 2:]

For instance, if the given example “white wine, fish, asparagus” occurs often enough or in enough transactions in the dataset, it might be deduced that a transaction containing fish is likely to contain white wine (or possibly vice versa).

This type of information is clearly useful in business and marketing applications, but associations can also contribute to statistics in areas as diverse as gene analysis, psychology, or making political decisions.

2.1.7 Independence of Naming Conventions

From an analytical point of view, it is important to note that sequence and association analysis are independent of whatever names are given to the items, or in fact what the items are.

The immediately obvious advantage of this independence is that analysis can usually be conducted just as easily on one kind of items as on any other. Another advantage is that the analysis is therefore unbiased by nature and will likely give a more objective representation of the data at hand.
2.2 Candidate Generation

Most of the data mining algorithms in common use at present use candidate generation as a part of the pattern mining process.

In the context of pattern mining, candidate patterns are patterns which are possible contenders for that privileged position of Frequent Pattern. By looking at transactions and working out the combinations of the items contained within them, you can then determine whether they occur sufficiently frequently in the database to be worthy of the honour.

For example, given that the transactions in a dataset contained only the two items A and B in various combinations, you could generate candidates and count the number of transactions containing each combination A, B, AB, AA and so on until you have covered all possible combinations.

2.2.1 The Explosive Candidate Generation Problem

The main problem with looking for frequent patterns is the sheer number of possible patterns existing within the transaction database. Even within a single transaction, the number of possible patterns can be quite significant.

Consider the following transaction and its possible candidate patterns:

```
A B C D E
```

<table>
<thead>
<tr>
<th>A</th>
<th>A-B</th>
<th>B-C</th>
<th>A-B-C</th>
<th>A-D</th>
<th>A-D-E</th>
<th>A-B-C-D</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A-C</td>
<td>B-D</td>
<td>A-B-D</td>
<td>B-C</td>
<td>D A-B-C-E</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>A-D</td>
<td>B-E</td>
<td>A-B-E</td>
<td>B-C-E</td>
<td>A-B-D-E</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>A-E</td>
<td>C-D</td>
<td>A-C-D</td>
<td>B-D-E</td>
<td>A-C-D-E</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>C-E</td>
<td>D-E</td>
<td>A-C-E</td>
<td>C-D-E</td>
<td>B-C-D-E</td>
<td></td>
</tr>
</tbody>
</table>

All 31 candidate patterns - including the original transaction (greater than / equal to length 2)

*Figure 2d*
The number of candidates is obviously large, even for relatively small transactions. In fact, the total number of distinct candidate patterns for an $n$-pattern (or alternatively, a transaction with $n$ distinct items) given by this brief equation:

$$\text{candidates ( } n\text{-pattern } ) = 2^n - 1$$

**Definition 2e**

Wading through all the candidate patterns to find the useful ones is clearly going to be quite a task (especially once you start considering much larger patterns – the complexity increases quite dramatically). However, this is such an obvious (temporary) “make-do” solution, and so common, that it actually has a name.

### 2.3 The Brute-Force Heuristic

Also known as the “grab a large hammer” or the “caveman” approach, this way of tackling the problem is low on efficiency and requires minimal intelligence. Just for completeness, here is a pseudocode representation of the Brute-Force algorithm.

```plaintext
// pseudocode for Brute-Force algorithm
get_all_events();
get_number_of_patterns();
get_maximum_pattern_length();

k = 1;  // start with patterns of size 1

// consider all patterns from size 1 to maximum pattern length
while (k < maximum_pattern_length)
{  for (each k-pattern pat_k)
    {  for (each event e)
        {  add (pat_k + e) to list of pat_{k+1} patterns;
          if (pat_k + e is in database) && (frequency(pat_k + e) > threshold)
            {  add (pat_k + e) to list of frequent patterns;
            }
        }
      }
    }
  k++;  // go to next pattern size
}
```

**Pseudocode Segment 2a**

Like whacking an entire surface in the blind hope of hitting the nail, the brute-force method generates all possible candidate patterns and checks each of them, hoping to chance upon the right one (eventually). At each stage (each iteration of $k$), $n$ new candidate patterns are generated for each $k$-pattern (where there are $n$ distinct objects forming patterns), in other words, the number of candidates grows exponentially with each iteration.

Like a caveman, this approach requires raw power, raw (and senseless!) determination, relentless optimism and lots (and lots) of time. It is pretty much accepted that in the presence of almost any other alternatives, the brute force method is not a smart solution.
2.4 The Apriori Heuristic

**a priori** [Latin: from what is before]

**Definition 2f** [5]

It would seem that one obvious way, if possible, to improve the efficiency is to reduce the number of candidate patterns.

In their paper “Fast Algorithms for Mining Association Rules,” Srikant and Agarwal observed an important reductive property of frequent patterns. According to the *anti-monotone Apriori heuristic* [3]:

<table>
<thead>
<tr>
<th>The anti-monotone Apriori heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>If any length-k pattern is not frequent in the database,</td>
</tr>
<tr>
<td>Its length-(k+1) super-pattern can never be frequent.</td>
</tr>
</tbody>
</table>

**Definition 2g**

This is quite a significant observation, not as complicated as it possibly sounds, and well worth taking the time to understand. We shall take a closer look at a suitable example to reinforce this concept.

Say we were looking at a supermarket shopping basket, and we were wondering if the pattern below is frequent in the database (appearing, say, in more than 20% of the transactions):

![Bubble Gum → Fish → White Wine](image)

**Figure 2e**

Now if we know that bubble gum did not appear in 20% of the transactions, we can be absolutely certain that the pattern \{**bubble gum, fish**\} does not appear in the database, and by *induction* (the reductive or recursive principle of basing a property or result on the one immediately under it) using the **Apriori** heuristic, we also know that the entire pattern cannot appear in the database.

The basic idea is to generate the set of frequent candidate \((k+1)\text{-patterns}\) from the set of frequent \(k\text{-patterns}\) by checking their occurrence frequencies in the database, starting from \(k=1\), and continuing until all patterns have either been confirmed or eliminated as being frequent.

Sample pseudocode is given on the following page.
// pseudocode for Apriori algorithm
get_all_events();
get_number_of_patterns();
filter_frequent_events(threshold,patterns);

k=1;  // start with a pattern of size 1

while (there are still patterns left to be eliminated)
{  for (each k-pattern pat_k)
   {  for (each frequent event)
      {  if (pat_k + e) is frequent in database(threshold)
         {  add (pat_k + e) to database; }
         else
         {  don’t add (pat_k + e) to database; }
      }
   }
   k++;  // increase candidate pattern size
}

Pseudocode Segment 2b

The Apriori heuristic offers improved performance over brute-force by reducing the number of candidates at each step, possibly a great deal, and particularly with high thresholds.

It is not as effective with lower thresholds, long patterns or generally a large number of frequent patterns, for these reasons:

- handling a huge number of candidate sets is costly. If there are 10^7 frequent items, the Apriori algorithm must generate more than 10^7 candidate 2-patterns, and then test all their occurrence frequencies. Also, if a frequent 100-pattern is to be discovered (i.e. \{a_1, ..., a_{100}\}), more than 2^{100} candidates (that’s more than 10^{30} or one trillion million million!) have to be generated
- repeatedly scanning the database and checking a large set of candidates by pattern matching is a painful process (well, tedious and very time consuming anyway). This is especially true for long patterns. [1]

2.5 Summary

- a dataset is a collection of transactions, each of which is a collection of items. Each of these items are members of a domain itemset.
- a pattern is also a collection of items. A sequence is a type of pattern, which is under some kind of ordering.
- a frequent pattern is a pattern occurring frequently in a dataset. Support and frequency are two measures which may be used to determine whether a pattern is frequent.
- an association rule tells us that transactions containing certain items are likely to contain certain other items.
- for analytical purposes, the names of the items do not affect their analysis
- candidate generation is used to generate possible frequent patterns which are then evaluated.
- the Apriori method goes some way to reducing the number of candidates generated by the brute-force algorithm, but the efficiency of algorithms using candidate generation is still limited.
3 the pattern tree algorithm

The two central algorithms that form the basis of this project were designed by computer scientists from the School of Computing Science in Canada’s Simon Fraser University.

Jiawei Han, Jian Pei, Behzad Mortazavi-asl and Hua Zhu published their paper “Mining Access Patterns Efficiently from Web Logs” [2] detailing a clever method of sequence analysis.

Han and Pei also collaborated with Yiwen Yin to describe an algorithm for more generalised frequent pattern analysis in “Mining Frequent Patterns without Candidate Generation”, [1]

This section gives a description of the algorithms and the theoretical basis behind them. There are numerous references to the two papers throughout the chapter, and hence these will not be marked explicitly.

3.1 Representing the Data

According to these scientists, in order to mine a very large dataset as efficiently as possible, it is necessary to determine what information within the dataset is both sufficient and necessary for the mining. Here sufficient basically means that enough information is available to perform the mining without having to refer to anything else. Of course, there may be a lot of information which is not required, so the necessary criteria ensures that information which is not useful is not taking up valuable space.

A suitable data structure then has to be created to represent this sufficient and necessary information, and a closer examination of the data and a little thought will help to explain its design.

3.1.1 The Dataset

In the previous chapter, we established that a dataset is, of course, a collection of transactions, and each of these transactions is a collection of items. Further, each of these items can be considered to come from some set of items.

If we apply this to our supermarket example, the domain itemset consists of products for sale in the supermarket, the transactions are the shopping baskets, and the dataset is the database of those transactions.
In order to aid our understanding of the pattern tree algorithm, we will use the sample dataset below:

| a b d a c; 
| e a e b c a c; 
| b a b f a e c; 
| a f b a c f c; |

*Figure 3a*

Each line here is a transaction, consisting of a number of items (represented by letters of the alphabet here), and the dataset is the entire collection of transactions (a very small dataset in this case!). The semicolons (`;`) just show us where the end of each transaction is. Pretty much any pattern dataset can be mapped or converted into a form that is essentially mathematically equivalent to this.

While this structure is easy for us to understand, at least for smaller files, this representation is not as efficient a representation as is possible — there is likely to be a great deal of redundancy in an average dataset (that is, a lot of data may be duplicated unnecessarily).

One way we can reduce the redundancy in the representation without changing the meaning of the data is by choosing an appropriate data structure to represent the data.

### 3.1.2 The Tree Data Structure

One of the most common data structures used in Computing is the tree structure. The tree structure is just like a tree in real life — everything starts at the root node, and from this node there may be branches that point to other nodes, and so on.

In a general tree, a node may have any number of branches, or none at all (a node with no branches is sometimes referred to as a leaf). A number of nodes connected vertically by a branch form a path. This diagram shows a visual representation of a tree.

*Figure 3b*
In this particular tree, the root node is the one labelled A. It has two branches leading to the two nodes B (which has two further branches of its own) and D (which has none). Example paths include ABCD and AD.

You may have already noticed that the tree structure seems to lend itself well to representing the pattern information. One could imagine the root node lying at the top, and each transaction laid out vertically, tracing a path downwards. This is what we shall do and we shall call our data structure a pattern tree.

How does this reduce redundancy then? Even in the example tree, we can see that the transaction ABE and ABCD share the same A and B nodes, and all transactions share the A node. In a larger tree we might have lots more sharing, and we can make this even more efficient by allowing repeated sections of transactions to keep count of how many transactions pass through them, thus avoiding having to produce duplicate branches.

On the right is a tree representing our dataset.

Admittedly, at this point with this example, we are not saving much space and perhaps with this smaller example it may even be less efficient due to the keeping count of the nodes, but it’s not difficult to imagine a case where the counts are several hundred or thousand and much space is being saved!

3.1.3 The Frequent Pattern Tree

In its current state, the tree doesn’t really tell us anything we didn’t already know. Remembering that we are only interested in frequent patterns, and making use of the Apriori heuristic, we can restrict our consideration to objects which are considered frequent. By the Apriori heuristic, none of the patterns eliminated by omitting non-frequent items would have been frequent patterns anyway.

Consider the elements in our dataset, filtered using a support threshold of 75%. In other words, we will consider items appearing in 75% of the transactions (3 transactions in this case). Notice that, for example, a has a support of 4 despite appearing 8 times:

<table>
<thead>
<tr>
<th>Item</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Frequent?</td>
<td>Yes</td>
<td>yes</td>
<td>Yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3a

Now we can filter the items in the patterns to obtain the frequent subsequences.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Frequent Subsequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b d a c</td>
<td>a b a c</td>
</tr>
<tr>
<td>e a c b a c</td>
<td>a b c a c</td>
</tr>
<tr>
<td>b a b a c</td>
<td>b a b a c</td>
</tr>
<tr>
<td>a b a c f c</td>
<td>a b a c c</td>
</tr>
</tbody>
</table>

Table 3b
If we now build the results of these sequences into a tree, we get the tree in the figure to the left.

We are now close to the final incarnation of the pattern tree. In order to assist traversal of the tree for the mining process, the functionality of being able to locate all the nodes with a particular label is important, and relatively “inexpensive” to implement.

This is accomplished by using a header table, containing a list of pointers (or references) to each node for each frequent item. Given a particular item, it is then simple to obtain the list of pointers and iteratively trace each node in the tree.

Figure 3d We now have our sufficient and necessary data structure, holding a minimal amount of information:

- a frequent pattern tree holds the frequent sequence information. Each node represents an item in one or more of the transactions in the database, and consists of a node label and node count (represented as label_count).
- a header table containing a list of pointers to each node for each frequent item.

We can now present our finished frequent pattern tree, displayed proudly in Figure 3e. A brief description of how we constructed the tree is as follows:

```java
public PatternTree getPatternTree(dataset)
    • we scan the dataset once to find frequent events
    • we scan the dataset again, and by iteratively inserting each frequent transaction subsequence, construct the pattern tree
return patterntree
```

Pseudocode 3a

### 3.1.4 Optimising for Patterns

While the basic structure and construction of the pattern tree are common to both regular pattern mining and sequence mining, there is a subtle optimisation which may be performed on patterns that should be noted.

Unlike sequences, in general (unordered) patterns, rearrangement or reordering of transactions does not change the meaning of the data. As if happens, this is advantageous as this means we can arrange the data to give us the most compact form of the tree without compromising the data representation.

A transaction may be divided into two parts. The sub-pattern in the first part (in the tree, the part closer to the root node) is known as the prefix, and the second part is then called the suffix (concatenated they reform the original transaction).
To make the tree as compact as possible, we want to have as many patterns as possible share common prefixes, but without too much cost incurred when trying to work out how to arrange the transactions to achieve this.

A relatively simple hypothesis is that more frequent items are more likely to be common to different patterns (from simple statistics), and so arranging the transactions so that more frequent items move towards the prefix (front) end of the transaction makes sense. Thus we modify our construction algorithm slightly to the following:

```java
public PatternTree getPatternTree(dataset)
• we scan the dataset once to find frequent items
• we sort the list of frequent items putting the most frequent at the front
• we scan the dataset again, and by iteratively sorting each transaction according to its order in the list of frequent items and then inserting each frequent transaction subsequence, construct the pattern tree
return patternTree
```

**Pseudocode 3b**

Applying this to our earlier example, we sort the list of frequent items a, b and c by frequency (a:8, b:5, c:3) and coincidentally obtain the items in the same order: a b c

Filtering and sorting these transactions now gives us these sorted frequent subsequences:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Frequent Subsequence</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>abdac</td>
<td>abac</td>
<td>abac</td>
</tr>
<tr>
<td>eabcac</td>
<td>ab cac</td>
<td>ab c c</td>
</tr>
<tr>
<td>babace</td>
<td>babac</td>
<td>babc</td>
</tr>
<tr>
<td>abacfc</td>
<td>ab acfc</td>
<td>ab c c</td>
</tr>
</tbody>
</table>

**Table 3c**

The resulting tree is displayed in **Figure 3e** on the right.

This generally gives us a more compact tree structure (as evidenced by this example) fairly easily and with relatively low additional cost.

3.1.5 Handling Strict Sequences

Strict sequences are sequences where adjacent items are also adjacent in the transaction they come from. When mining frequent strict sequences, we can consider a transaction to be divided by non-frequent items, producing two or more sequences. For instance using our example:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Frequent Strict Subsequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>abdac</td>
<td>ab, ac</td>
</tr>
<tr>
<td>eabcac</td>
<td>a, b cac</td>
</tr>
<tr>
<td>babace</td>
<td>bab, a, c</td>
</tr>
<tr>
<td>abacfc</td>
<td>a, b ac, c</td>
</tr>
</tbody>
</table>

**Table 3d**

The resulting frequent strict subsequences are then used to construct the tree in the same way.
3.2 Mining the Data

We now have the opportunity to put this organised, efficiently represented data to good use, and examine the algorithm that mines all frequent patterns from the database (or now, the tree).

First we consider the case of a tree with a single path (as in the picture on the left). What are all the patterns in this one-path tree? The patterns are simply the various combinations of patterns that can be generated. In this case, that means a, b and ab. (ba is a pattern but in the general pattern case it is a duplicate of ab, and in the sequence case, ba is not a valid sequence from the given tree as a precedes b).

Now that we have handled the simple case, we must handle the more complicated general case. We start by initialising the set of frequent patterns to an empty set {}.

First, we already know that the frequent items are going to be frequent patterns, so we add those to the set.

To mine the rest of the frequent patterns, we can take advantage of the header table produced earlier. We choose a frequent item, f, and obtain the list of f-nodes from the header table.

Now we construct a conditional pattern base for f. A conditional pattern base is basically the collection of all (and only all) prefixes of f in the dataset (tree). We can build this by going along the list of nodes and tracing each of the prefixes to the root node.

Once we have this conditional pattern base, we then build a conditional pattern tree for f from the collection of prefixes in the same way we described earlier in the chapter. We then mine this conditional sequence base recursively in the same way we did to this one. Once it has returned its collection of frequent patterns, we add f to the end of each of them, and return the entire collection.

Summarised into a pseudocode-like format (and painting in very broad strokes):

```java
public Collection getPatterns(PatternTree patternTree, HeaderTable headTable)
{
    // check if single path
    if (patternTree.singlePath())
    {
        // single path case
        Collection += patternTree.getCombinations();
    }
    else
    {
        // general case
        for each frequent item f in headTable
        {
            // add item
            Collection += f;
            temporaryCollection = headTable.nodes(f).prefixes(); // get prefixes
            temporaryPatternTree = makePatternTree(temporaryCollection); // build tree
            Collection += (temporaryPatternTree.getPatterns() + f); // recursive call
        }
    }
    return Collection;
}
```

Pseudocode 3c
Like most of the other descriptions, it becomes a lot clearer with an example. We will use the unsorted frequent pattern tree from earlier in the chapter, and use c and then ac to demonstrate the recursive mining.

Starting with the full tree as on the left, we can see it is not a single-path tree, so we select c as our sample frequent item. We first add all the frequent items in the tree (by referring to the header table) to the collection before proceeding.

Walking along the nodes we get this conditional pattern base:

<table>
<thead>
<tr>
<th>Frequent Item</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>aba</td>
<td>2</td>
</tr>
<tr>
<td>ab</td>
<td>1</td>
</tr>
<tr>
<td>abca</td>
<td>1</td>
</tr>
<tr>
<td>ab:-1</td>
<td>1</td>
</tr>
<tr>
<td>abac</td>
<td>1</td>
</tr>
<tr>
<td>ab:-1</td>
<td>1</td>
</tr>
<tr>
<td>baba</td>
<td>1</td>
</tr>
</tbody>
</table>

What does the –1 mean? If you follow the list of nodes from the header table entry for c, you will notice that the first c records a prefix of aba, then the second has a prefix of ab.

The third c records a prefix of abca, but this includes an ab that was counted earlier (from another c node in this c node’s prefix). To avoid counting the ab twice, ab is recorded again with a count of –1, which will later effectively remove the overlap (wait and see!).

The next c does the same thing — recording a prefix of abac, it compensates for the earlier count by adding an entry for ab (the prefix recorded by the first c) with a count of –1. Note that if the count of this node was 2 instead of 1, the compensation would likewise have been –2. The final c node contributes baba with a count of 1.

Now that the conditional pattern base has been built, the corresponding conditional pattern tree is constructed. Since c is not a conditional frequent pattern (its support of 2 is less than the threshold of 75%, 3). The resulting tree resembles the one shown on the right. Notice how the patterns with a –1 count have effectively compensated for overlap and the tree is produced correctly.

The tree is then mined in the same way. Again it is not a single-path tree, so we add all frequent events (a and b) to a new collection and this time we will demonstrate mining with a (effectively finding prefixes for ac). Following the nodes for a, we get these prefixes:

In the same fashion, the tree is constructed (see figure on left), and this time the tree is a single-path tree! So we return all the combinations (which are a, b and ab). The items a and c are then added to the end of each of these producing aac, bac and abac.

This process is repeated for each of the other frequent items, and is also carried out recursively.
3.3 Summary

- The frequent pattern tree approach to mining frequent patterns has a twofold strategy for doing this efficiently:
  - it uses a highly compressed data structure that holds only the information sufficient and necessary for the frequent pattern mining task, and
  - it uses algorithms that take advantage of this data structure to mine frequent patterns without generating candidates.
- The data structure consists of a tree representing the patterns and their counts, and a header table which contains a list of nodes for each frequent item.
- The efficiency of the tree may be further improved for unordered patterns by sorting items according to their frequency in the dataset before storing them in the tree.
- The algorithm may be extended to provide support for strict sequences.
4  implementation

The data structures used in the pattern tree algorithm were nicely defined mathematically, so the approach I took was to start with the structures first, and then gradually work my way towards a complete implementation of the algorithm.

The initial decisions I took during the implementation were not always the best ones – by the end I had changed a great deal of my initial implementation, as I understood more about the algorithm and how it worked. As with many learning experiences, in this case the journey was a significant part of the reward.

4.1 Java

I chose to do my implementation in Java for a few reasons. In particular, I felt that:

- the data structures involved would be best implemented using classes and objects.
- because the data mining group is in continual development of a data mining suite written in Java (Kensington), it would then be possible to integrate my implementation into this application at some point in the future without having to translate the code.
- my previous familiarity with Java meant less time spent learning a new language and more time available to improve and extend my implementation.
- Java’s inter-platform compatibility meant that my implementation could be used (and developed!) on different platforms without having to compile platform-specific binaries.

4.2 The Classes

This section on the classes used and their relationships to one another may be slightly unusual for a report, but I felt that the mistakes I made and the lessons I learned (painfully!) were worth mentioning, and hopefully a reader will find them insightful.

4.2.1 In the beginning...

My first exposure to the project was the paper “Mining Access Patterns Efficiently from Web Logs” [2], which described a sequence mining algorithm for mining web clickstreams from web server logs.

When I first started my implementation, I mapped out a diagram of classes pretty much as I interpreted the data structures from the description of the algorithm. The structures I chose to represent and the classes that resulted were:

- the “WAP-Tree” (Web Access Pattern Tree) became the SequenceTree class
- the “Header Table” became the QueueList class
- the “sequence data set” became the SequenceDatabase class

In addition to those classes I also created the following classes for my convenience:

- the Event class. This was just a holding class with a String label, which I used to represent items such as “a” or “fish”.
- the CountedEvent class. This was basically a holder for Event which also kept a count for it.
- The ObjectList data structure. This was basically the list of nodes which was kept for each item in the QueueList.

It should be noted that the name QueueList is in fact a misnomer. It came about because I imagined the list of nodes for each item to be a queue, and the header table to be a list of these queues (I have since realised what I should have realised earlier, that a queue is in fact a first-in-first-out data structure similar to the line of people at a cash till).

The Unified Modelling Language (UML) diagram below shows the relationships between these early classes graphically.

<table>
<thead>
<tr>
<th>CountedEvent</th>
<th>SequenceTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructors: CountedEvent(Event)</td>
<td>Constructors: SequenceTree()</td>
</tr>
<tr>
<td>Methods:</td>
<td>Methods:</td>
</tr>
<tr>
<td>boolean equals(Event)</td>
<td>void createChild(Object)</td>
</tr>
<tr>
<td>int counter()</td>
<td>int count()</td>
</tr>
<tr>
<td>Event getObject()</td>
<td>Object getObject()</td>
</tr>
<tr>
<td>void increment</td>
<td>int getChildren()</td>
</tr>
<tr>
<td>String toString()</td>
<td>SequenceTree getChild(int)</td>
</tr>
<tr>
<td>Question List</td>
<td>SequenceTree getChild(Object)</td>
</tr>
<tr>
<td>Event(String)</td>
<td>int getIndexOfChild(Object)</td>
</tr>
<tr>
<td>Methods:</td>
<td>SequenceTree getParent()</td>
</tr>
<tr>
<td>boolean equals(Event)</td>
<td>Void increment()</td>
</tr>
<tr>
<td>boolean label()</td>
<td>Boolean hasChild(Object)</td>
</tr>
<tr>
<td>String toString()</td>
<td>List prefix()</td>
</tr>
<tr>
<td>Question Tree</td>
<td>SequenceTree root()</td>
</tr>
<tr>
<td>Question Event</td>
<td>String toString()</td>
</tr>
<tr>
<td>Question String</td>
<td>String toFullString()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructors: Event(String)</td>
</tr>
<tr>
<td>Methods:</td>
</tr>
<tr>
<td>boolean equals(Event)</td>
</tr>
<tr>
<td>String label()</td>
</tr>
<tr>
<td>String toString()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QueueList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructors: QueueList()</td>
</tr>
<tr>
<td>Methods:</td>
</tr>
<tr>
<td>ObjectList getList(Object)</td>
</tr>
<tr>
<td>void insert(Object, SequenceTree)</td>
</tr>
<tr>
<td>String toString()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ObjectList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructors: ObjectList(Object)</td>
</tr>
<tr>
<td>Methods:</td>
</tr>
<tr>
<td>SequenceTree getNode(int)</td>
</tr>
<tr>
<td>Object getObject()</td>
</tr>
<tr>
<td>void insert(SequenceTree)</td>
</tr>
<tr>
<td>int size()</td>
</tr>
<tr>
<td>String toString()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SequenceDatabase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields:</td>
</tr>
<tr>
<td>static int GENERAL, STRICT, ASSOCIATIONS</td>
</tr>
<tr>
<td>Constructors:</td>
</tr>
<tr>
<td>SequenceDatabase()</td>
</tr>
<tr>
<td>Methods:</td>
</tr>
<tr>
<td>void createSequenceTree(List, SequenceTree,</td>
</tr>
<tr>
<td>List)</td>
</tr>
<tr>
<td>int sequences()</td>
</tr>
<tr>
<td>List filterObjects(threshold)</td>
</tr>
<tr>
<td>SequenceDatabase next()</td>
</tr>
<tr>
<td>List getObjects()</td>
</tr>
<tr>
<td>List getSequence()</td>
</tr>
<tr>
<td>int getSequences()</td>
</tr>
<tr>
<td>void setMode(int)</td>
</tr>
<tr>
<td>void setNext(SequenceDatabase)</td>
</tr>
<tr>
<td>void setMode(int)</td>
</tr>
<tr>
<td>void setSequence(List)</td>
</tr>
</tbody>
</table>

Figure 4a
UML Diagram – Initial Implementation
The `Event` class was a placeholder class which I used to represent the items in the transactions. The `CountedEvent` class was a holder for the `Event` class, and as the name suggests, kept count of it as well.

The `SequenceDatabase` class represented the dataset. This was a basic implementation of a `List`, with each `transaction` being held in its own object and a reference made to the next one in the dataset.

As the general `SequenceTree` was a `multiway-tree` (with a dynamically changeable number of children), I used a `List` within the `SequenceTree` structure to manage the child or branch nodes. Access methods were available to handle these, and among the others were provisions for referring to the represented item (a.k.a object), the count of the node, its parent and its prefix.

The (mismarked) `QueueList` was basically a `List` which contained the `Lists` of all the nodes of each frequent item in the `SequenceTree`. These `Lists` were represented by the `ObjectList` class. Both of these classes were simple and had a minimal number of necessary access methods.

The construction of the `SequenceTree` followed the description of the algorithm closely. In choosing to represent the data set with the `SequenceDatabase` class, I had one additional step of reading the data set into a file into the `SequenceDatabase` object. Once the data set had been read into memory, the object scan and tree construction as described in 3.1 were performed using that data alone. The construction returned a `SequenceTree` that could then be explored and analysed.

### 4.2.2 Problems

While this implementation was correct and worked, some major shortcomings were soon discovered when testing and performance evaluation began.

The performance of the algorithm was not nearly as good as was expected. Batch testing and visualising the results showed that while the algorithm was supposed to be running in linear time with respect to the number of patterns in the dataset, it was at least quadratic and possibly higher, as shown in the graph below:

![Graph showing Processing Time against Number of Patterns (by threshold %)](image)

*Graph 4a*
Mentally, I began running through the different calls in the main loop of algorithm trying to figure out which parts of the algorithm could be in quadratic time.

Worse, when in the final stage of building the pattern tree from the database, the Java Virtual Machine would crash with an out of memory exception. Again, this didn’t make sense as the tree appeared to be a compact, efficient data structure, on paper at least.

As if the execution times and running out of memory weren’t bad enough, the algorithm would often crash with stack overflow exceptions when reading even moderately small files (around 50kB) into memory. The ceiling for these files was around 2,000 patterns (notice in Graph 4a there are no results for datasets with more than 2,000 patterns), which is tiny in a field where datasets may contain hundreds of thousands or even millions of transactions.

In the midst of the ensuing panic, I learned a great deal (about the aspect of troubleshooting that went beyond fixing bugs) from what followed.

### 4.2.3 Adaptation and Solution

There were two distinct issues that needed to be resolved:

- the performance was not satisfactory — linear time was the expected performance of this algorithm, and somewhere in the main loop of my algorithm (or possibly in several places) there was an bottleneck operation that was hindering its performance.
- there was an issue with memory and stack overflows which were not resolved by increasing memory and stack size. I suspected that the problem lay not with the pattern tree taking up too much space, but rather with the representation of the data set I was holding in memory.

I approached the performance problem first, by breaking the algorithm down into stages and timing the individual stages. This, I hoped, would help give me some idea of where the problem lay.

I divided the algorithm into load time, object scan time, object filter time and model building time. tabulated the benchmark results and got this graph.

![Graph 4b](image-url)
The fact that the filter time was unexpectedly long led me to discover that I was inadvertently scanning for frequent Objects twice. Apart from that, it was not obvious that a single stage of the algorithm was holding it up with its performance - they were all exhibiting a similar order of performance relative to the number of patterns. It was time to look at each stage individually and trace each call.

When this proved unsuccessful, my next guess was that there was some standard Java component I was using that operated in quadratic time. As Lists featured heavily in my implementation and as the List type is iterative by nature, it became my primary suspect, and to check this I replaced Lists whenever possible with Hashables. Upon recompilation and re-execution, the improvement in performance was obvious:

![Graph showing time against number of patterns](image)

*Graph 4c*

It should be noted that the size of the dataset the implementation could cope with probably increased in the second performance evaluation because I discovered how to increase the memory and stack sizes allocated to the Java Virtual Machine. This improvement was still relatively modest however, and a more drastic solution would be required to resolve the memory crisis that still loomed.

I decided in the end that I would have to implement a different representation of the data set. I created the SequenceFile class and performed the Object scans and built the model directly from the data in the file.

This turned out to be the breakthrough required, both in terms of volume handling capacity and performance. A detailed analysis of the performance evaluation of this final implementation is offered in the next chapter.

### 4.2.4 Integrating Association Patterns and Sequences

As the general (association) pattern and sequence algorithms are very similar, it made sense for them to share common fields and methods using inheritance. To achieve this, basic skeleton classes were defined for PatternFile and PatternTree, and SequenceFile, AssociationFile, SequenceTree and AssociationTree were set to extend them with any specific implementation differences.

An extension was implemented on the SequenceFile class to deal with strict sequences as described in the previous chapter.
An AccessModel class was formed to unite the PatternTree and ListTable. To handle the creation of the AccessModel, the concept of a ModelCreator class was also introduced. The AccessModel would then be responsible for dealing with major access functions for the PatternTree and ListTable, such as mining the set of all patterns.

**Figure 4b**
UML Diagram - Current Implementation

### AccessModel

**Fields:**
- static int GENERAL, STRICT, ASSOCIATIONS

** Constructors:**
- SequenceDatabase()

**Methods:**
- allPatterns()  
- conditionalPatternTree(Object, PatternTree)  
- generateAll()  
- getLists()  
- getMode()  
- getPatternTree()  
- getLabel()  

### ListTable

** Constructors:**
- ListTable()

**Methods:**
- ObjectList getList(Object)  
- insert(Object, SequenceTree)  
- toString()  

### PatternTree

** Constructors:**
- PatternTree()  
- PatternTree(Object, PatternTree)

**Methods:**
- createChild(Object)  
- count()  
- getChildren()  
- getChild(int)  
- getChild(Object)  
- getChildCount(Object)  
- getIndexOfChild(Object)  
- getParent()  
- increment()  
- hasChild(Object)  
- prefix()  
- root()  
- toString()  
- toFullString()  

### ObjectList

** Constructors:**
- ObjectList(Object)

**Methods:**
- SequenceTree getNode(int)  
- Object getObject()  
- insert(SequenceTree)  
- size()  
- toString()  

### SequenceTree

** Constructors:**
- SequenceTree()
- SequenceTree(Object, SequenceTree)

**Methods:**
- void addTreeModelListener(TreeModelListener)  
- removeTreeModelListener(TreeModelListener)  
- valueForPathChanged(TreePath, Object)  

### AssociationTree

** Constructors:**
- AssociationTree()  
- AssociationTree(Object, AssociationTree)

**Methods:**
- void addTreeModelListener(TreeModelListener)  
- removeTreeModelListener(TreeModelListener)  
- valueForPathChanged(TreePath, Object)
### PatternFile

<table>
<thead>
<tr>
<th>Fields:</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int</td>
</tr>
<tr>
<td>GENERAL</td>
</tr>
<tr>
<td>static int</td>
</tr>
<tr>
<td>STRICT</td>
</tr>
<tr>
<td>static int</td>
</tr>
<tr>
<td>ASSOCIATIONS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructors:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatternFile()</td>
</tr>
<tr>
<td>PatternFile(String)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccessModel</td>
</tr>
<tr>
<td>createModel(Hashtable,PatternTree, int)</td>
</tr>
<tr>
<td>AccessModel</td>
</tr>
<tr>
<td>createModel(List,PatternTree, int)</td>
</tr>
<tr>
<td>Hashtable</td>
</tr>
<tr>
<td>getObjects()</td>
</tr>
<tr>
<td>StreamTokenizer</td>
</tr>
<tr>
<td>getTokenizer()</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>getPatterns()</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>precache()</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>setFilename()</td>
</tr>
</tbody>
</table>

### CountedObject

| CountedEvent(Event) |

<table>
<thead>
<tr>
<th>Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Event)</td>
</tr>
<tr>
<td>int counter()</td>
</tr>
<tr>
<td>Event getObjectId()</td>
</tr>
<tr>
<td>void increment()</td>
</tr>
<tr>
<td>String toString()</td>
</tr>
</tbody>
</table>

### Event

| Event(String) |

<table>
<thead>
<tr>
<th>Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Event)</td>
</tr>
<tr>
<td>String label()</td>
</tr>
<tr>
<td>String toString()</td>
</tr>
</tbody>
</table>

### SequenceFile

<table>
<thead>
<tr>
<th>SequenceFile()</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>List sort(List, List)</td>
</tr>
</tbody>
</table>

### AssociationFile

<table>
<thead>
<tr>
<th>AssociationFile()</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>List sort(List, List)</td>
</tr>
</tbody>
</table>

### FormattedTime

<table>
<thead>
<tr>
<th>FormattedTime()</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getHours()</td>
</tr>
<tr>
<td>int getMinutes()</td>
</tr>
<tr>
<td>int getSeconds()</td>
</tr>
<tr>
<td>float toFloat()</td>
</tr>
<tr>
<td>long toLong()</td>
</tr>
<tr>
<td>String toString()</td>
</tr>
</tbody>
</table>

### ModelCreator

<table>
<thead>
<tr>
<th>ModelCreator()</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashtable filterObjects(int, Hashtable)</td>
</tr>
<tr>
<td>String getFilename()</td>
</tr>
<tr>
<td>int getModel()</td>
</tr>
<tr>
<td>Hashtable getObjects()</td>
</tr>
<tr>
<td>int getThreshold()</td>
</tr>
<tr>
<td>void init(int, String)</td>
</tr>
<tr>
<td>int patterns()</td>
</tr>
<tr>
<td>void preCache()</td>
</tr>
<tr>
<td>void setMode(int)</td>
</tr>
<tr>
<td>List sort(Hashtable)</td>
</tr>
</tbody>
</table>

### LogFile

<table>
<thead>
<tr>
<th>LogFile()</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void close()</td>
</tr>
<tr>
<td>void setFilename(String)</td>
</tr>
<tr>
<td>void write(String)</td>
</tr>
<tr>
<td>void writeln(String)</td>
</tr>
</tbody>
</table>
4.3 The Pattern Analyser

As part of my implementation I built a graphical-user-interface (GUI) for my implementation.

The main frame (above!) is the first thing the user sees when the Pattern Analyser is started. At the top of the window is a text box for specifying the input file (alternatively the file chooser to its right may be used). The toolbar to the right allows you to choose options or perform operations on the file, such as choosing the mode of analysis (Association, General, Strict), whether to pre-cache the file and what threshold to use.

The Load File button builds an AccesModel (containing a PatternTree and a ListTable) from the file according to the desired options and displays it in the main window area:

![Figure 4c](image)

![Figure 4d](image)
Once the AccessModel has been created, operations can be performed on it, such as to query the model. Pressing the Query Model button brings up a simple query tool which can search for the nodes of a particular item in the dataset and retrieve prefixes.

This file contains sequences of English words, so a query on the letter A will tell us something about other letters appearing in words with the letter A.
Perhaps more useful to the analyst is the ability to generated conditional pattern trees from the model. Returning to the main frame we can generate the conditional pattern tree for \( i \) by right-clicking on it and selecting the appropriate menu option.

![Figure 4g](image1)

The frame is updated with the resulting conditional pattern tree for \( i \).

![Figure 4h](image2)
We can now find all the patterns in this tree by using the **View All Patterns** button (slightly obscured, in the bottom right corner). This mines the patterns from the tree and opens the built-in file-editor to display the results:

![Figure 4i](image)

The source code, executable code and Javadoc documentation of this implementation has been made available at this web page:

```
http://www.doc.ic.ac.uk/~jkst98/project/
```

You may also contact the Data Mining Group in the Department of Computing, Imperial College.
5

performance issues

In this chapter, the performance of this implementation is examined in some detail, including the relationships between thresholds and execution times and breaking it down to show its performance in the different stages of execution. There are also comparisons between the two algorithms for basic (association) patterns and sequences, and at each stage interpretations and conclusions are suggested.

I made use of benchmarking features I built into my implementation to give an indication of the behaviour of the implementation as datasets of differing sizes and characteristics were thrown at it.

5.1 Test Bench

I used the run-line version of my implementation, firstly because it presented a more accurate representation of the algorithm (graphical interfaces have a detrimental effect on performance in Java), and secondly because it made batch jobs more convenient. Suitable output was generated to ensure that the implementation was returning accurate results, and times and details were outputted to log files in a format suitable for importing into a spreadsheet.

The results were obtained on one of the voxel machines in the Department's Teaching Laboratory. It was equipped with an Athlon microprocessor, 256MB of RAM, and was running Microsoft Windows NT.

The Java Virtual Machine returned the following information when “java --version” was run:

```
java version "1.3.0"
Java™ 2 Runtime Environment, Standard Edition (build 1.3.0-C)
Java HotSpot™ Client VM (build 1.3.0-C, mixed mode)
```

5.2 Sequence Analysis

We start by taking a look at the results from benchmarking of the sequence mining algorithm.

The sequence analyser was run in general sequence mode unless otherwise noted.
Overall execution time here is defined as the time taken from input (specifying datafile, threshold, association / sequence mode) to the time when the complete queryable model has been produced. The following graph shows how the overall execution time correlates with the minimum threshold value and the number of patterns in the dataset.

**Graph 5a**

### 5.2.2 Correlation to Pattern Quantity

*Graph 7a* shows that execution time increases linearly with the number of patterns in the dataset. The following cross section of this graph (taking the section where threshold is 0%) show this relationship quite clearly, and this is reflected at any given cross-section.

**Graph 5b**
5.2.3 Correlation to Threshold

What is perhaps more interesting is the unusual shape of the cross section in the other direction – given a particular number of patterns, the execution time also changes shape depending on the minimum threshold value. This graph shows such a cross-section, taking it at 100,000 patterns.

Graph 5c

Between 0 and 20%, and again between 30 and 100%, the execution times are about the same. Why is there a sudden drop between threshold values of 20 and 30%? A quick examination of output reveals that the increase in threshold causes the number of frequent events to decrease dramatically between 20% and 30%. A more detailed analysis of the change between these two values confirms this:

Graph 5d

For the test data represented by these graphs, practically all the objects forming patterns had frequencies lying between the two values of 20 and 30%. We know the execution time as a whole is decreased, but which part of the algorithm is responsible for this?
5.3 Sequences: Breakdown Times

By examining the breakdown times – i.e. the time taken to complete each stage of the algorithm – we can see how the individual stages react to changes in threshold and the amount of pattern data.

The four stages defined in this implementation are as follows:

- **Pre-cache Time**  
  The time taken for the file to be read through once to take full advantage of any disk caching.

- **Object Scan Time**  
  The time taken to scan through the data file and obtain a list of the objects forming the patterns within it, as well as their frequencies.

- **Filter Time**  
  The time taken to examine the objects and get the list of frequent events (by filtering out those below the threshold).

- **Model Build Time**  
  The time taken to build the frequent pattern tree from the data file and the list of frequent events.

### 5.3.1 A Note on Pre-caching

The pre-caching step described above is basically a single pass of reading through the file. This both allows the algorithm to take advantage of any caching in the disk subsystem and hence potentially improve performance, and also equalises different executions in a batch test by making sure none has an advantage over any other because of caching.

The pre-caching itself did not produce a marked improvement in performance, but it was never worse off with pre-caching on, and it certainly evened out batch runs and made the results more reliable.

### 5.3.2 Correlation to Pattern Quantity

Here is a graph giving the breakdown times across a range of pattern sizes for the fixed thresholds of 0%:

![Graph 5e](Image)

*Graph 5e*
From the graph, the **pre-cache time** does not appear to be influenced greatly by the number of patterns. In fact, examination of the results used to form the table reveal that in fact there is no real apparent correlation – as the time is so small, it is probably affected more by the timing of the disk subsystem.

The **event scan time** is, predictably, linearly related to the number of patterns (the algorithm involves iterating through each pattern in the file and identifying / counting the distinct elements after all).

The **filter time** is very small (it recorded itself as 0 or 0.1 in my benchmarks) and is not influenced by the number of patterns (it makes more sense that it be correlated to the number of distinct objects, as the filtering is simply a pass through the object list).

Like the **event scan**, the **model build time** is also linearly related to the number of patterns, as this part of the algorithm also involves iterating through each pattern in the file as it builds the tree.

As the **total execution time** is the sum of the other execution times, and as these times are either close to zero or are linearly related to the quantity of patterns, the total time also has a linear correlation with the pattern number.

### 5.3.3 Correlation to Threshold

We recall from earlier in the chapter that a change of threshold caused execution times to decrease in the 20 - 30% range. We can now analyse the breakdown to see which stages were affected (the most), and once again we’ll fix the pattern quantity at 100,000.

**Graph 5f**

The graph indicates that the **object scan time** is not affected by the threshold, while the **model build time** is affected by the threshold. We can explain this by remembering that the **object scan** is always the same for a dataset no matter what the threshold is – the threshold only starts to affect the algorithm after the **filter** stage. Once non-frequent items have been filtered, there are less items to insert into the tree when building the model so performance improves.
5.4 Association Pattern Analysis

Using the same datasets, testing was also performed using the unordered pattern algorithm and tabulated into graphs so the performance could be compared and contrasted with the sequence results. This graph shows how the overall execution time correlates with the minimum threshold value and the number of patterns in the dataset.

Graph 5g

The shape of the graph is immediately familiar. Compared to the matching sequence graph, Graph 5a, the times are slightly higher (136.6 vs 127.9 seconds at threshold 0 for 100,000 patterns), and this can be attributed to the sorting process that takes place for each pattern using this algorithm.

5.4.1 Correlation with Pattern Quantity

A quick look at the graph above shows that for any given threshold, the overall execution time is, like the sequence algorithm, linearly dependent on the number of patterns.

5.4.2 Correlation with Thresholds

Again, for a given pattern quantity, the relationship between overall execution time and the threshold is identical to that of the sequence algorithm.
5.5 Association Patterns: Breakdown Times

The breakdown time graphs were also produced from the benchmark data:

![Graph 5h]

### 5.5.1 Comparison with Sequence Algorithm

Across the board, there are no major differences between the performance characteristics of the two algorithms. The algorithms themselves are not dissimilar and this correlation is hence not a surprise.

Perhaps one observation is worth making about Graph 5h above – the object scan time is absolutely the same as the matching object scan time for the sequence algorithm (unsurprising since that stage of the algorithm is identical). The source of the slight difference in their times lies in the model building stage, almost certainly confirming the hypothesis that the pattern sorting was causing this algorithm to trail slightly in performance.

### 5.6 Conclusions

In the flesh the algorithms confirm their theoretical potential. The algorithm scales linearly and manages large volumes of data well. Both the sequence and associations algorithms, being very similar, exhibit similar performance characteristics, with the associations algorithm hindered slightly its pattern sorting.
visualisation

This project began as a study of click-stream analysis, a specific real-world application of sequence analysis on the web pages requested by visitors to a website. As the Internet domain becomes increasingly driven by commercial requirements, it is becoming necessary for a savvy web-based business to understand the browsing patterns of their customers in order to remain competitive.

One of the most important questions click-stream analysis aims to answer is this:

Given two web pages a and b on a web site, what can we tell about how visitors click their way from a to b?

One potential application for this question is how, for instance, most users get from the home page to the purchase page. This information could enable the analyst to better influence visitors and direct them where he would like them to go (i.e. to buy something!).

Although visualisation is beyond the scope of my project, this chapter has some suggestions and proposals on how the PatternTree could be used or adapted for visualisation purposes to aid an analyst graphically.

6.1 Path between Two Nodes

Let us say we have a collection of paths between a and b as follows:

- a b;
- a c b;
- a c b d b;
- a d a b;

We would like to represent the collection of paths starting at a and ending at b. The most intuitive way of doing this on paper is by using a connected directional graph with nodes to represent each web page, and each edge between nodes lies on a path between a and b.

Here's a possible way of representing these paths:

![Figure 6a](image.png)
For the purpose of seeing what the paths are between A and B, this representation is excellent. It is easy to understand and relatively simple to construct using a pen and paper.

How would we represent this data using a data structure? If we visually rotate the graph on its side we find that it is actually a tree, and some of the nodes of the tree point to our destination node B, as shown in the figure to the right.

One might recall that it looks very similar to the conditional pattern tree described in Chapter 3. In fact the tree to the right is a conditional pattern tree for B, but constructed with all its prefixes that begin with a. Hence we only require a small modification to the conditional pattern tree construction algorithm (considering prefixes starting with a, or locating a within the prefix and considering the prefix starting from a) to create a suitable representation for visualising the sequence from a to b. By varying the threshold it is possible to view the sequence data to the level of detail desired.

### 6.2 The Reversed Tree

A tree structure is hierarchical, and hence allows a user to view its top level contents first before deciding to trace further down along the branch. In the case of the special conditional pattern tree just mentioned, this means pages closer to the start (a) are at the top level (refer Figure 6b).

Since an analyst may be more interested in the pages visited immediately before a purchase page for example (items before b in transactions rather than those after a), a proposed modification may be made to the conditional pattern tree construction algorithm which inserts sequences backwards. This creates, in effect, a reversed tree which can then be browsed hierarchically from the destination node.

A reversed tree for the paths from a to b is shown to the right.

### 6.3 The DoubleTree Structure

One idea for another representation of these sequences is what I call a DoubleTree. This is in effect a double-ended tree with the two extreme nodes on both ends and branches between these two start and end nodes:

Like a tree this is a recursive structure and grows dynamically. Each dotted box in the figure indicates a node in this structure and each arrow indicates a branch. In the top-level node, A is the start and B is the end of the DoubleTree.
Here is an example of a pseudocode skeleton class representing a DoubleTree in Java.

```java
public class DoubleTree
{
    // node of DoubleTree
    int count; // the count of the DoubleTree
    Object start, end; // the start and end Objects
    List branches; // a List containing the child DoubleTrees

    public DoubleTree(Object newstart, newend) { count = 1;
        branches = new LinkedList(); start = newstart; end = newend; }
    public int count() { return count; }
    public Object start() { return start; }
    public Object end() { return end; }
    public void addbranch(DoubleTree tree) { branches.add(tree); }
    public int branches() { return branches.size(); }
    public DoubleTree branch(int index) { return ((DoubleTree) branches.get(index)); }
}
```

**Pseudocode 6a**

Further functionality that could be added to the DoubleTree are exploding (being able to selectively explode branches to get a more detailed view), construction from lists, being able to intelligently allocate new branches or combine branches efficiently and so on.

### 6.4 Summary

- The PatternTree is not just useful for mining patterns from a dataset, it can also be used, with a little coaxing, to aid visualisation of data.
- The DoubleTree is a structure for representing sequences sharing the same start and end node which could possibly be examined in greater detail in future for aiding visualisation of such sequences.
conclusions

7

7.1 Summary

In conclusion, this project has fulfilled and exceeded the original specifications and intentions set out at its inception, and has produced a starting point which other projects will hopefully expand upon and develop in the future.

The Pattern Tree method has proved itself to be a fast, scalable algorithm for mining frequent patterns and sequences from datasets. It has also been found to be a flexible structure for performing queries and it is believed that it will prove useful in other areas of data mining such as visualisation.

7.2 Beyond this project

Here are some ideas for possible ways to continue to extend or develop this project:

- Association rule mining – now that the frequent patterns have been located, it would be useful to develop it into association rule mining.
- Integration into the Kensington data mining suite – a component could be developed for Kensington based on this implementation, or modifications made to the implementation to allow this integration.
- Visualiser – visualisation tools are another way to extend this project’s functionality
- Performance enhancements – this implementation has proven efficient, but there is always the possibility of improving on its efficiency, as well as extending its scaling capabilities by using disk-based methods, for instance.

7.3 Further reading

The publications listed in the references (see appendix) are a good starting point to find out more about frequent pattern mining and other related areas in data mining (with the possible exception of the Concise Oxford Dictionary, which is probably best left as a dataset!).

Knowledge Discovery - Johann Ting, BEng Computing 47
References:

[1] Jiawei Han, Jian Pei and Yiwen Yin
   “Mining Frequent Patterns without Candidate Generation”]

[2] Jiawei Han, Jian Pei, Behzad Mortazavi-asl and Hua Zhu
   “Mining Access Patterns Efficiently from Web Logs”]

   “Fast Algorithms for Mining Association Rules”]

   “Algorithms for Association Rule Mining – A General Survey and Comparison”]


Other references:
These publications were read or referred to during research, but were not quoted in this
thesis.

- The Dictionary of Algorithms, Data Structures and Problems
  http://hissanist.gov/dads
- Jochen Hipp, Andreas Myka, Rudiger Wirth and Ulrich Guntzer
  “A New Algorithm for Faster Mining of Generalized Association Rules”
- Hannu Toivonen
  “Discovery of Frequent Patterns in Large Data Collections”