Mining Partial Periodicity in Large Time Series Databases using Max Sub-Pattern Tree

Matthew D. Boardman, #B00068260
Faculty of Computer Science, Dalhousie University

April 8, 2005

Abstract

In this paper, an Apriori-based algorithm for mining partially periodic patterns from very large sequential time series databases is implemented, and is applied to fifteen years of stock data from ten large technology corporations with daily resolution at two discretization levels, with the goal of finding common repeating patterns. The Max Subpattern Tree data structure, proposed in 1999 by J.Han, G.Dong and Y.Yin, is implemented to reduce the need for online memory storage and to increase the efficiency of the algorithm. A standard Apriori time series algorithm is also implemented for comparison purposes. We find that the algorithm including the Max Subpattern Tree requires only two scans of the database to deliver the exact same partially periodic repeating patterns as the Apriori-only algorithm, with the same confidence values. We also find strong correlations between patterns found in this technology sector, in particular the IBM, Intel and Microsoft stock data.

1 Introduction

The search for repeating patterns in sequential time series data has many practical applications, and is an interesting topic of research. Until recently, developments in this area have concentrated on full-cycle periodicity, in which every part of the cycle contributes in some part to the repeating pattern. For example, the days of the week form a fully periodic cycle, because all seven days contribute in part to the repeating pattern.

Partial periodicity, in which only some of the events within a pattern repeat, occurs often in real-world situations. For example, we may commute to work on weekday mornings between 8:30 and 9:00: this can be considered to be a partially periodic pattern, because it says nothing of our activity throughout the remainder of the day, or on the weekends. We might say that gasoline prices often go up on weekends and holidays, but this says nothing about the price fluctuations throughout the rest of the week.

The search for partial periodicity in a sequence of events in time related data is simplified by dividing the sequence into segments of equal length. This segment-wise periodicity greatly reduces the search space required for repeating patterns, since without this simplification a search would need to consider several offset thresholds from the start of each sequence. To illustrate this concept, consider the arbitrary start of a week: one person might assume the start of a week to be on a Monday, the first work day, whereas others might divide a calendar between Sunday and Saturday, with the work days in-between.

In this paper, we implement an Apriori-based approach to mining segment-wise partial periodicity in a discretized time series, using a novel data architecture proposed by Han et al [11]. This algorithm is ideal for large time series databases, which may contain millions or billions of rows, as it requires only two passes of the source data to determine all partially periodic patterns. The implementation is applied to stock market pricing data for ten technology stocks, to determine common partially repeating patterns which could be used to determine an ideal distribution for portfolio management.

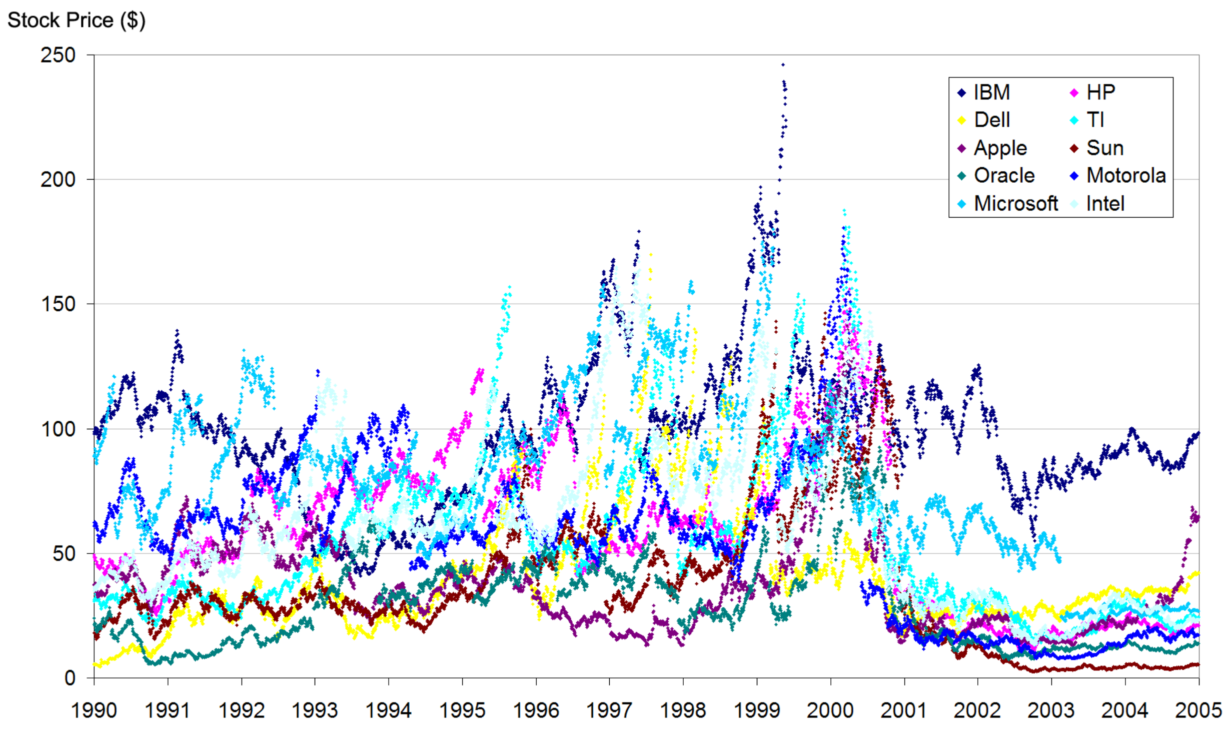


Figure 1: Daily closing price data from ten top technology stocks, downloaded from Yahoo Finance.

1.1 Related Work

There are several approaches to mining periodicity in a time series database. Statistical approaches are common, such as the Autoregressive Moving Average (ARMA) model [4] commonly used in many clustering methodologies. Another common mathematical approach is the application of Fast Fourier Transforms (FFT), although this requires full-cycle periodicity in order to measure the frequencies of occurrence of each element in the cycle.

Data mining approaches based on the Apriori technique for search space reduction [2] have been applied successfully to time series databases [12] and have been generally adopted. Several researchers have proposed data structures to increase the efficiency of these searches [5, 10, 11]. Most of these techniques require discretized data, but algorithms have also been proposed based on numerical curves [1] that also capture the efficiency of the Apriori principle.

Other techniques for the search for repeating patterns in time series data include the application of artificial neural networks (ANN), such as a self organizing map (SOM) or a continuous attractor neural network (CANN) [18], which can yield excellent noise insensitivity such as is required for biomedical applications [6, 7], or genetic algorithms, which evolve an initial population of pattern wavelets over several generations using crossovers and mutation to yield an elite population with high fitness values [16].

1.2 Application Scenario

The stock market is inherently quite random. The pricing fluctuations of a typical stock that is bought and sold by dozens or hundreds of people each day yield many complexities for economists, stock analysts and financial advisors as well as academic researchers.

In this paper, ten large, well-established technology stocks have been selected for analysis: Apple, Dell, Hewlett-

Packard, IBM, Intel, Microsoft, Motorola, Oracle, Sun and Texas Instruments. The daily closing price data for these stocks over a period of fifteen years (1990 through 2004) were downloaded from Yahoo! Finance [20]. The assumed goal of this scenario is to find frequent patterns common across these ten stocks, with a period length of 5 days (one business week) for the purpose of portfolio management.

1.3 Project Objective

In this paper, an Apriori-based approach is implemented using the Max Subpattern Tree data structure proposed in [11]. This algorithm was selected as it builds on a common technique (Apriori) for search space reduction, but also introduces a novel data structure to reduce memory requirements. This makes the algorithm ideal for inclusion in the very large databases used by portfolio management companies, for example, which may contain millions or billions of rows with decades of pricing data for hundreds of stocks and commodities. We will also implement a standard Apriori search, in order to compare the efficiency of these algorithms and verify that the results are correct.

1.4 Notation

The notation to be used in this paper to describe partially periodic patterns is similar to that used in [11].

We describe a pattern by listing the events within the pattern in the order in which they occur, substituting an asterisk (*) for *don't care* events (ie. any event within the subset of defined events under consideration). For example, the partially periodic pattern [a b * * *] is a pattern of length 5, as it contains a sequence of five events: event *a* occurs, then event *b*, then any three other events. This we call a 2-pattern, since it includes two events that are not *don't care*.

We may also write events in which two or more of the events in the list may occur, but not any event; these optional events are written using braces. For example, the pattern [a $\{b_1, b_2\}$ * * *] indicates that event *a* occurs, then either b_1 or b_2, then any three events occur.

As a quantifiable measure of the value of a repeating pattern, we include both the *frequency count* and the *confidence* of each frequent pattern, where the count is the number of occurrences of this pattern in the segment-wise partitioning of the sequence of data, and the confidence is the count divided by the total number of segments. For example, in the sequence [a b c d e a b e e e d a d d c a b a b a d b], we find the segments [a b c d e], [a b e e e], [d a d d c] and [a b a b a]. A pattern [a b * * *] can be measured against these four segments with a count of 3 (the first, second and fourth segments) and a confidence of 75% (three of four).

It is important to note that left-over events that do not fit within this segment-wise partitioning are ignored (in the above example, the trailing [d b] are not measured as we do not consider partial segments). However, the maximum amount of data that can be ignored because of this is one less than the pattern length, which will be very small in relation to the total number of rows in the database. In our application scenario, we are lucky that the sequence of data for 1990 begins on a Monday, and the sequence of data for 2004 ends on a Friday, so in this case, no data is ignored.

As in [11], we also define S to be the complete time series of events under consideration, and D_i as the events within S, ie. $S = D_1, D_2, D_3, ..., D_n$ where n is the total number of elements in the sequence S. The list of available events under consideration is defined to be L, such that $\forall D_i \in L$. The number of events in L we denote as $|L|$, and we call this the cardinality. A repeating pattern is denoted by $s = s_1, s_2, s_3, ..., s_p$ where p is the length of the period (or segment) under consideration, also denoted as $|s|$. A repeating pattern must be non-empty (ie. a pattern [* * * * *] is not allowed). Finally, we define m to be the total number of segments in the series S, such that

$$confidence = \frac{frequency\ count}{m}$$

1.5 Simplification of Search Space

An important simplification is made in this implementation: the found patterns may include only definite events rather than optional events (ie. [a * * * *] is allowed as a search pattern, but [{a,b} * * * *] is allowed only as an interim step). This assumption significantly reduces the search space, and is appropriate in the context of this application scenario as it makes the resulting patterns easier to understand for the user.

We can prove this significant search space reduction as follows: with this simplification, each element in a pattern can be either one of the $|L|$ elements in L, or *, with the provision that a null pattern [* * * * *] is not allowed. The maximum possible search space with this simplification is therefore

$$(|L|+1)^p - 1$$

which for a period of 5 and a cardinality of 3 would be 1023, or for a period of 5 and a cardinality of 7 would be 32 767. These are the search spaces used later in this paper.

However, without this assumption, the potential search space becomes much more complex. Each element of a repeating pattern could be an event in L, or a combination of two or more events in L. We can write the total possible number of combinations as

$$C(|L|,k) = \frac{|L|!}{k!(|L|-k)!}$$

for a given k, where k ranges from 1 (a single element in $|L|$) to $|L|$ (the set of all events in $|L|$, or *). Again it is reasonable to exclude the null pattern [* * * * *] since it would match all patterns in the sequence S with a confidence of 100%. The maximum possible total search space, without the above assumption, is therefore

$$\left[\sum_{k=1}^{|L|} \frac{|L|!}{k!(|L|-k)!}\right]^p - 1$$

which for a period of 5 with cardinality 3 is $(3+3+1)^5 - 1 = 16\ 806$, or for a period of 5 with cardinality 7 is $(1+21+35+35+21+7+1)^5 - 1 = 25\ 937\ 424\ 600$, or nearly 26 billion possible combinations! If we were to consider a period length of two weeks with cardinality seven, as we consider in the Results section later in this paper, we would have a search space with over 6.7×10^{20} possibilities!

Obviously, the computing time that would be necessary to exhaustively search such a large space is not reasonable for practical applications such as this stock market scenario. Even with Apriori search space reduction, the space will still remain large due to the slow reduction of Apriori confidence values with increasing pattern length, as shown by example in the Apriori Principle section later in this paper.

2 Data Preparation

The first step in data preparation was to download the daily closing price data from Yahoo! Finance for each stock, copy-and-paste the results into a single Microsoft Excel spreadsheet, calculate the difference of each stock's closing price versus the previous day's closing price (hereafter referred to as the delta values), and save the results as a comma-delimited file. Note that the algorithm implementation in this paper accepts both comma-delimited and space-delimited (eg. fixed length) formats. This process is of course fairly straightforward, and results in the distributions shown in Figs. 1 and 2 which include a total of 39 150 points of data.

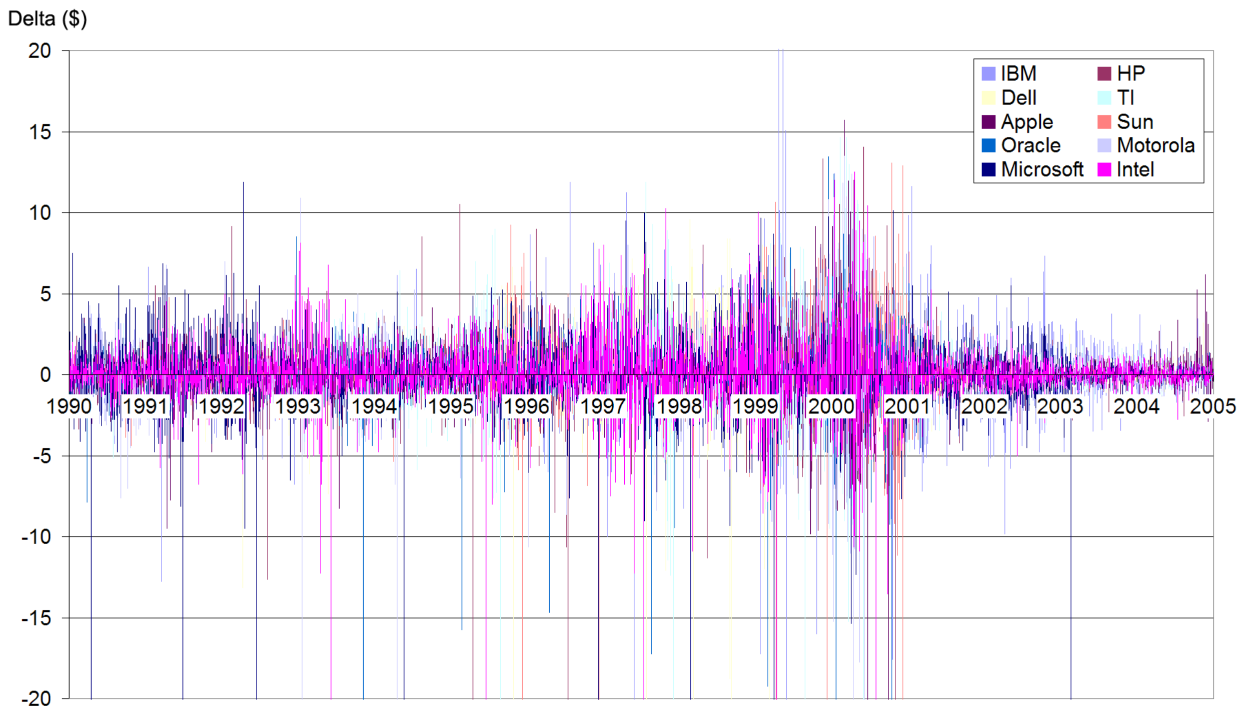


Figure 2: Daily increase or decrease in closing price from previous day for these same stocks, calculated in Excel as the first stage of data preparation.

2.1 Data Source

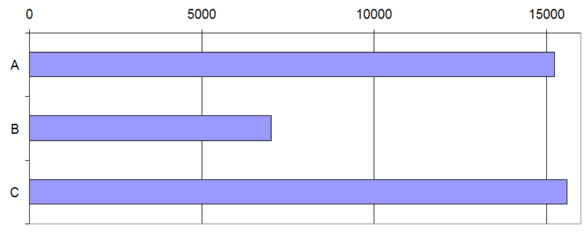
As shown in these figures, the selected data is inherently very random. The infamous "technology bubble" through the mid- to late-1990's is clearly visible in Fig. 1 for these stocks, a period of dramatic growth and a subsequent period of sudden decrease in technology stock values. However, the resulting delta values in Fig. 2 show little to no obvious pattern, resembling purely random noise.

It is important to note that stock splits, in which a given stock may have its value divided in half but its available shares doubled in order to keep the stock value within a target range, are not included in this data as we are interested only in the daily delta values. The effect of stock splits on this algorithm would be only to slightly lower the confidence values of the resulting repeating patterns, since stock splits (or reverse splits) occur only during a single day and are relatively rare in relation to the total number of data points.

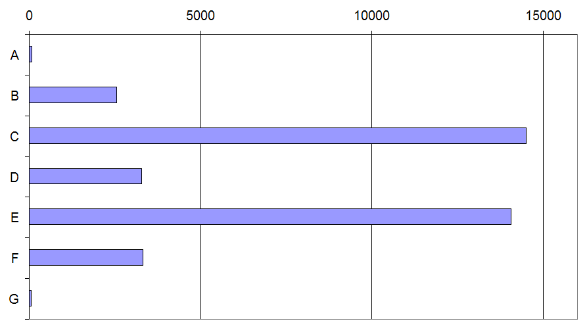
2.2 Discretization

Many techniques are available for the discretization of real-valued elements, such as the histogram equalization technique shown in [9], however given this application scenario it was decided to use arbitrary thresholds based on the meaning behind the data rather than a generalized solution.

On this basis, two discretization levels were decided upon: one with three class values (a cardinality of three), and a second level with seven class values (a cardinality of seven). The first set of threshold values were $\{-0.2, 0.2\}$ indicating a simple approach: if the stock's value goes down by a value of 20 cents compared to its previous day's value, the stock is considered to be going down (class A); if the stock's value goes up by a value of 20 cents compared to its previous day's value, the stock is considered to be going up (class C); otherwise, the stock is considered to



(a)



(b)

Figure 3: Distribution of class values for all ten stocks. a) Distribution of discretized class values {A,B,C} (a cardinality of three) using the threshold values $\{-0.2, 0.2\}$. b) Distribution of discretized class values {A,B,C,D,E,F,G} (a cardinality of seven) using the threshold values $\{-10, -2, -0.1, 0.1, 2, 10\}$.

be not moving (class B). The resulting histogram of class values for all ten stocks using these thresholds is shown in Fig. 3(a). Similarly, for cardinality seven, threshold values of $\{-10, -2, -0.1, 0.1, 2, 10\}$ were used to represent very low values indicating a possible stock split or reverse split, sharp decline, small decline, little movement, small increase, sharp increase and very high values respectively. The resulting histogram of class values for all ten stocks using these thresholds is shown in Fig. 3(b).

Discretization was performed using a separate program, to allow a high degree of flexibility while not forcing the user to discretize the data each time the periodicity program was run. The code for this simple, separate program is included in the file `process.c`. In addition to the resulting file, the output of this program shows equations for the resulting threshold values, as shown below:

```
Cardinality = 3:
                Class=[A]       --->              x  <   -0.2
                Class=[B]       --->    -0.2 <=   x  <=   0.2
                Class=[C]       --->     0.2 <    x
```

for the first set of class values, and

```
Cardinality = 7:
                Class=[A]       --->              x  <  -10.0
                Class=[B]       --->   -10.0 <=   x  <   -2.0
                Class=[C]       --->    -2.0 <=   x  <   -0.1
                Class=[D]       --->    -0.1 <=   x  <=   0.1
                Class=[E]       --->     0.1 <    x  <=   2.0
                Class=[F]       --->     2.0 <    x  <=  10.0
                Class=[G]       --->    10.0 <    x
```

for the second. This program also fills in missing days, such as holidays, with zero-delta values (ie. class B for the first set of thresholds, or class D for the second) in order to ensure that each element of the segment corresponds exactly with the Monday, Tuesday, Wednesday, Thursday and Friday of the particular week.

2.3 Metadata of Discretized File

An example of the results of this data preparation is shown in the following table:

Date	Day	IBM	HP	Dell	TI	Apple	Sun	Oracle	Motorola	Microsoft	Intel
2-Jan-90	1	D	D	D	D	D	D	D	D	D	D
2-Jan-90	2	F	F	E	C	E	E	E	F	E	E
3-Jan-90	3	E	C	C	E	E	D	B	E	E	C
4-Jan-90	4	E	C	C	E	E	C	E	E	F	E
5-Jan-90	5	C	C	E	D	E	D	C	C	B	C
8-Jan-90	1	E	E	E	C	E	E	E	E	E	E
9-Jan-90	2	C	C	C	E	C	C	C	E	C	E
10-Jan-90	3	C	C	C	E	C	C	C	C	B	C
11-Jan-90	4	E	C	D	C	C	E	C	E	C	D
12-Jan-90	5	C	C	D	C	D	C	C	C	C	C

In this excerpt from the data file, the first row has been added by the discretization program to fill in the sequence of days, since otherwise the segments would start on day 2 (Tuesday).

The first column contains the actual date included for reference, and in the case of added rows is repeated based on the subsequent row's value. The second column contains the day of the week, ie. 1 = Monday, 2 = Tuesday, ..., 5 = Friday. Subsequent columns contain the discretized closing price data for each of the ten stocks IBM, HP , Dell , Texas Instruments, Apple, Sun, Oracle, Motorola, Microsoft and Intel respectively.

Since the program reads only a single column from the input file in this implementation in order to generate the sequence S of discretized values, the input file could be as simple as a single, low-cardinality column: such a file was generated in order to test a sequence of all ten stock values at once, and is included in the `discretized_data` directory in the file `card7.big`.

3 Program Architecture

As mentioned in the previous section, the first step of data preparation (calculating the delta values) is performed in Excel, and the second step (discretization) is performed using a small, customized program separate from the main program. The remainder of the activities in this implementation are contained in the program `hitset`, which is comprised of the following nine files:

`apriori.c`	Provides functions that would be used in Apriori approach
`hitset.c`	Provides max-subpattern hit-set algorithm
`hitset.h`	Provides definitions, function prototypes and data structures used by all *.c files
`hitset_var.h`	Provides extern references to global variables used by all *.c files
`interface.c`	Provides program entry point main() and rudimentary user interface
`list.c`	Provides functions for linked list data structure for storing frequent patterns
`output.c`	Provides output of generated partially periodic patterns to the screen or file
`readdata.c`	Provides functions to read periodic data from an input file
`tree.c`	Provides interface to max-subpattern tree data structure

These source files are provided in the `source` directory. The discretization program source is provided in the `source_discrete` directory.

3.1 Building the Program

To build the program executables, change to either the `source` directory or `source_discrete` directory, then type the command `make`. The makefile in each source directory will assume that the `cc` compiler is installed on your system. The resulting executables will be named `hitset` and `process`, for the hit set algorithm implementation and discretization pre-processor respectively.

3.2 Running a Demo

A `demo` directory has been provided to run a quick demo of the features of these programs. To view the demo, switch to this directory, and type the command `./go`. The following commands will be run:

```
../source_discrete/process 1990.csv discrete.csv -s 2 -d 2 -10 -.5 .5 10
../source/hitset discrete.csv patterns.txt -s 5 -p 5 -c 0.1
cat patterns.txt
```

Note that the executables `hitset` and `process` are called from their respective source directories, so these must be built first using the supplied makefiles (see previous section).

This demo reads in a subset of the stock data containing the first year of data only, discretizes the data using four thresholds $\{-10, -0.5, 0.5, 10\}$, then runs the main program on the fifth column (Dell) with a period length of five days and a minimum confidence threshold of 10%. Finally, the patterns generated are shown on the screen. Output from the demo will resemble the following:

```
Time Series Data Pre-processor:   Matt Boardman, Dalhousie University
------------------------------

Input filename:  1990.csv
Output filename: discrete.csv
Skipping first 2 column(s).
Will analyze day number in column 2.

Discrete levels: (5 total)
                Class=[A]       --->              x  <  -10.0
                Class=[B]       --->    -10.0 <=  x  <   -0.5
                Class=[C]       --->     -0.5 <=  x  <=   0.5
                Class=[D]       --->      0.5 <   x  <=  10.0
                Class=[E]       --->     10.0 <   x

Input file contains 12 columns.

Time Series Data Analysis:   Matt Boardman, Dalhousie University
-------------------------

Parameters:
                Data filename:      discrete.csv
                Output filename:    patterns.txt
                Minimum confidence: 10.00%
                Period length:      5
                Source column:      5
                Algorithm:          Max Hitset


35 partially periodic patterns generated to file patterns.txt (0.010 sec).
```

```
        [CCCCC]: L-length=5, count=28, confidence=53.85%
        [CCCC*]: L-length=4, count=32, confidence=61.54%
        [CCC*C]: L-length=4, count=32, confidence=61.54%
        [CC*CC]: L-length=4, count=29, confidence=55.77%
        [C*CCC]: L-length=4, count=30, confidence=57.69%
        [*CCCC]: L-length=4, count=32, confidence=61.54%
        [**CCC]: L-length=3, count=34, confidence=65.38%
        [*C*CC]: L-length=3, count=35, confidence=67.31%
        [C**CC]: L-length=3, count=31, confidence=59.62%
          ...
```

3.3 User Interface

Note that this demo uses the command line options for both programs, rather than the interactive mode. To use the interactive mode for the main program, simply enter `./hitset`. The program will then prompt the user for all needed variables:

```
Time Series Data Analysis:   Matt Boardman, Dalhousie University
-------------------------

Interactive mode:  Please enter each requested parameter.

Please enter input data filename: 1990.csv
Please enter source column number (1=first column): 5
Please enter output filename: patterns.txt
Please enter minimum confidence threshold: 15
Please enter period length: 5
Parameters:
Data filename:      1990.csv
Output filename:    patterns.txt
Minimum confidence: 15.00%
Period length:      5
Source column:      5
Algorithm:          Max Hitset

68 partially periodic patterns generated to file patterns.txt (0.020 sec).
```

However, using the command-line parameters gives the user additional flexibility. Using the command-line, the user must specify input and output files and a target column, but minimum confidence and period parameters have reasonable defaults of 10% and 5 days respectively if not specified.

For example, a verbose mode is included to give the user additional statistical information including a histogram of class values, and an Apriori-only algorithm is included which does not run the Hit Set algorithm in order to compare the generated patterns. For example, try the following command:

```
./hitset discrete.csv patterns.txt -s 5 -p 5 -c 0.1 -v
```

This will run the main program with the same parameters as the demo, but in verbose mode: using verbose mode, we can see that the root node of the tree C_{max} in this case is [C C C {D,C} {D,C}] (in the program, this is represented by the string "CCCD|CD|C") and that the tree contains 34 nodes. The generated 1-patterns and joined patterns are also shown, along with the following statistics:

```
Summary:
                Data filename:      discrete.csv
                Output filename:    patterns.txt
                Minimum confidence: 10.00%
                Period length:      5
                Source column:      5
                Algorithm:          Max Hitset

                Rows in source:     260
                Periods in source:  52
                File scans:         2
                Number of reads:    520
                Found patterns:     35
                Patterns explored:  55
                Neglected end rows: 0
                Highest L-length:   5
                Processing time:    0.010 sec

                Cmax:               [CCCD|CD|C]
                  L-length:         5
                  Level:            7
                  Tree Complexity:  34 nodes

                Source cardinality: 3
                   [B] count: 10
                   [C] count: 225
                   [D] count: 25
```

3.4 Overall Algorithm

The Max Hit Set algorithm is shown in Fig. 4. The data preparation steps are shown in blue. The result of this preparation is a discretized, comma-delimited file (or a space-delimited file, depending on the format of the original file) which is then input to the main program.

The first step in the algorithm is to scan the file to calculate and count the frequent 1-patterns. From these patterns, the maximal subpattern that can be formed using only these frequent 1-patterns is created: we call this pattern C_{max}. The discretized file is then scanned a second time, to build the Max Subpattern Tree. This data structure is then scanned in step 3 using an Apriori approach. These high-level steps are examined in more detail in the following sections.

3.5 Apriori Principle

The Apriori principle, first proposed by Agrawal and Srikant in [2], uses previously obtained knowledge to prune the future search space at each step in the search. For time-series data, the Apriori principle can be stated as, *if a k-pattern is frequent, all of the j-patterns for $j < k$ must be frequent as well* [11, 12].

Although efficient, in this case the principle means that at each level of the search space, the file must be scanned in order to eliminate those patterns that are not frequent. This can mean a costly search if the database is large enough that it cannot fit in the computer's main memory, for example if the database contains millions or billions of rows.

To illustrate, for a purely Apriori approach (implemented in the main program using the `-a` command-line option), the first step is to scan the file and count all possible 1-patterns, ie. those patterns which have exactly one event in the segment, with the remaining events in the segment set to *don't care* states, such as [a * * * *] or [* * b * *]. Once

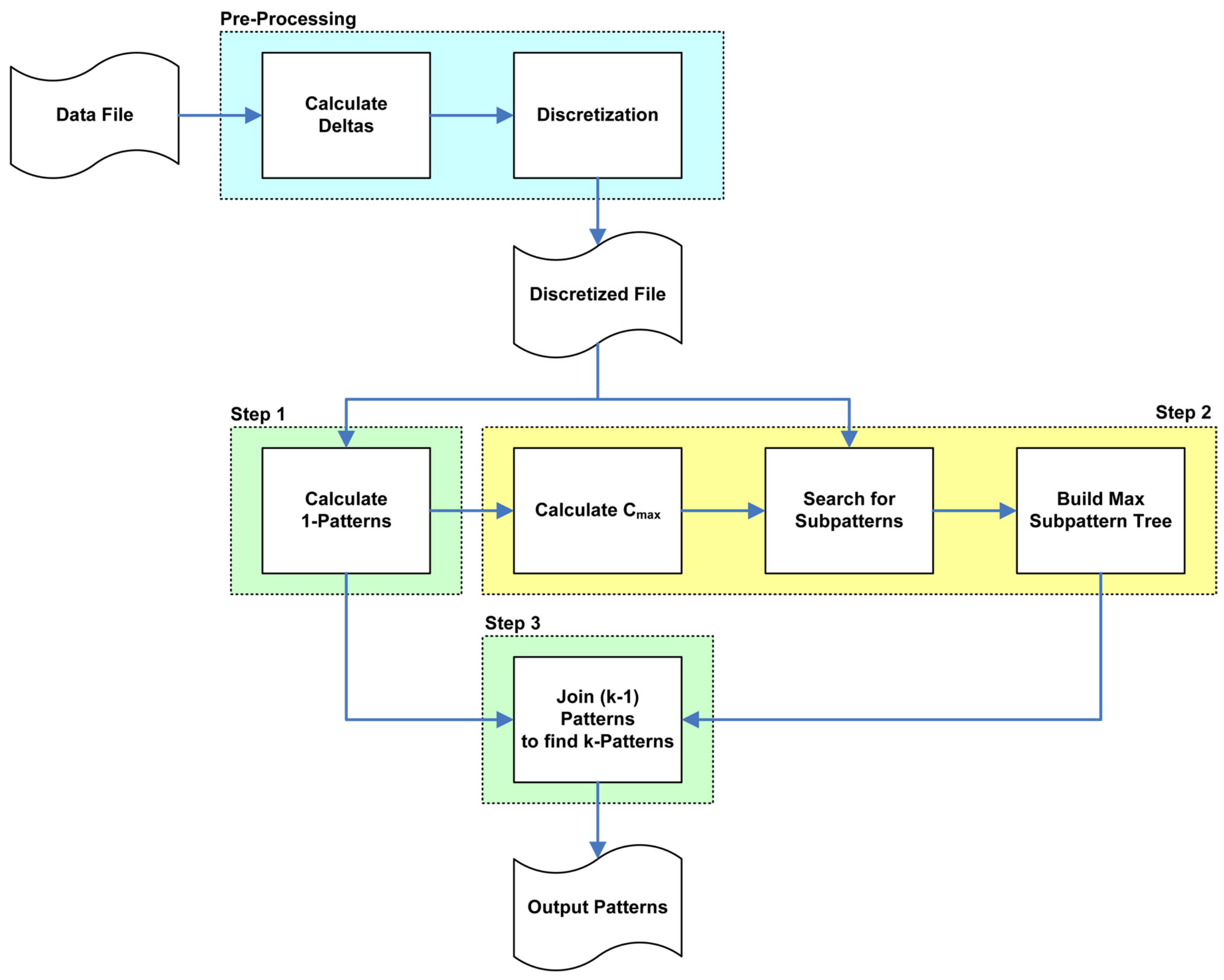


Figure 4: Overall system architecture, including data preparation steps and interim discretization file.

these 1-patterns have been found, the patterns with a confidence lower than the minimum confidence threshold are deleted. The next step is to join these patterns together as detailed below, in order to form a candidate set of 2-patterns. The file is then scanned again, in order to prune those 2-patterns in the candidate set that do not meet the minimum confidence requirement.

This means that using a purely Apriori approach, the file would have to be scanned for each level of the search, which for a pattern search with period p, means p scans of the database. This is compounded by the slow Apriori reduction in search space as illustrated by the following example from [11]: suppose that we have two frequent 1-patterns [a *] and [* b], both with confidence 0.9 (90%). Since all segments in the sequence that match the pattern [a b] must also match both [a *] and [* b], we know from the Apriori principle that the confidence of [a b] must be less than 0.9. If we were to scan the database for those segments that did not match [a *], we know that this would be $1 - 0.9 = 0.1$. Similarly, if we were to scan the database for those segments that did not match [* b], we know that this would also be $1 - 0.9 = 0.1$. The confidence of pattern [a b] must therefore be greater than $1 - 0.1 - 0.1 = 0.8$. This slow reduction in the search space of the Apriori algorithm for time series data can have a significant effect on the efficiency of the algorithm, as we shall investigate later in this paper by comparing the performance of these two algorithms.

3.6 Joining Patterns to Find Candidate Patterns

In order to join two time series patterns to form the candidate patterns for the next Apriori search, each element of both patterns is examined and compared in turn. If one pattern has a *don't care* for the position being examined, and the other pattern contains an event, the second pattern's event is added to the joined pattern. If both patterns contain a *don't care*, then a *don't care* is added to the joined pattern. If both events contain an event, then we consider these patterns to be not joinable due to our simplification as stated earlier: the set of frequent patterns may not contain optional events.

The exception to this simplification is when joining patterns in order to form maximal subpatterns to create the Max Subpattern Tree, as detailed below: in this case, all events in all 1-patterns are considered and contribute to the creation of C_{max}. The only difference is that in the case where both of the patterns to be joined contain an event, both of the options are included in the join as an optional event. This is denoted by braces, such as in the pattern [a $\{b, c\}$ * d *] in which the second position can contain either event b or event c. In the implementation of this algorithm used in this paper, these patterns are stored as a variable-length string such as "AB|C*D*", where the logical OR symbol ("|") is used to represent optional events. In this case, event 2 of 5 may be either B or C.

For example, the 1-patterns [a *] and [* b] used in the example in the previous section are joinable, since each position in the patterns contains a *don't care* for either one pattern or the other. When joined, these two patterns therefore form the candidate pattern [a b].

3.7 Building the Max Subpattern Tree

As stated earlier, the first step in generating the Max Subpattern Tree data structure is to calculate the maximal pattern that can be generated from the set of 1-patterns found during the first scan of the database. To illustrate, consider the set of 1-patterns: [a * *], [* b_1 *], [* b_2 *]. Performing a join of all events in all three of these patterns, we obtain $C_{max} =$ [a $\{b_1$,$b_2\}$ *]. This then becomes the root node of the Max Subpattern Tree.

To build the rest of the tree, we scan the file a second time. For each segment in the file, the maximal subpattern that can be created by taking away the minimum number of events from C_{max} is generated (we call this the maximal subpattern of C_{max} that is "hit" in this segment), and is added to the tree if it does not already exist: if it does exist, the count of that pattern is increased by one. Other nodes in the tree may also be created as a result of this insertion.

To illustrate, consider Fig. 5. If we are to add the pattern [* b_1 * d *] to this structure, starting from the root node C_{max} =[a $\{b_1, b_2\}$ * d *], we first follow the path $\bar{a}$ ("not a", shown in red in the figure) as it does not appear in the pattern to be inserted, and arrive at the node [* $\{b_1, b_2\}$ * d *]. Since b_1 does appear in the pattern to be inserted, the next missing event is b_2: we therefore follow the path $\bar{b_2}$ and arrive at the target node [* b_1 * d *]. Since this node already exists, we increase its count by one.

It is important to note that since we follow these "not" paths in the order in which they appear in the pattern, the resulting graph will not be fully connected. These paths that have not been connected through the insertion process are shown as dotted lines in Fig. 5 (paths connected by the insertion process are shown as solid lines).

It is also important to note that nodes which do not appear in the database are not created: such a node is shown in gray in Fig. 5. This means that although it would be possible to create all nodes immediately without scanning the file, and then scan the file once the tree was built in order to obtain the counts for each pattern, performing the algorithm in this way would create many redundant nodes with a count of zero. The author first tried this approach as a possible improvement on the original algorithm, however in practical tests on the target data it was found that creating these redundant nodes with a count of zero added significantly to the processing time.

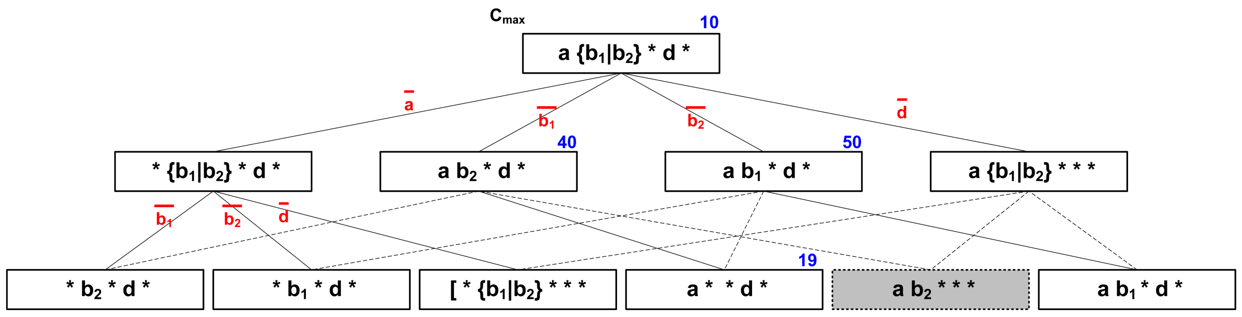


Figure 5: An example of the max sub-pattern tree structure, used to store the partially periodic patterns in the max hit set.

3.8 Calculate Frequent Patterns from the Max Subpattern Tree

In order to determine the set of frequent patterns from the Max Subpattern Tree, an Apriori approach is again used to form the candidate k-patterns as detailed above, however, rather than scan the file to determine the frequency count and confidence of each pattern, the tree data structure can give the frequency count directly by employing a simple algorithm.

To determine the frequency count for a given candidate pattern, the pattern is first compared with C_{max} to determine the set of events in C_{max} that are missing from the pattern. This set of missing events is then used as the search path to find the set of "reachable ancestors" of this node. The frequency count is then the sum of the count of the particular node, and its set of reachable ancestors including C_{max}.

To illustrate, consider the candidate pattern [a * * d *] as shown in Fig. 5. Comparing this pattern with C_{max} =[a $\{b_1, b_2\}$ * d *], we find its set of missing events to be [* $\{b_1, b_2\}$ * * *]. Following these two paths b_1 and b_2 from C_{max}, we obtain the set of four reachable ancestors (shown in the figure with blue counts to the upper right of these four nodes). The frequency count for the pattern is obtained by summing the frequency counts of the subpattern and its reachable ancestors, ie. $19 + 40 + 50 + 10 = 119$.

3.9 Notes on Code Design

For storing the frequent patterns, a doubly-linked list was employed, with each node containing both forward and backward pointers. This allowed new nodes to be stored at the top of the list rather than the end, which improved performance of the join by effectively sorting the resulting patterns in order of creation. Each node also contains the pattern stored as a variable-length, dynamically allocated string, and the pattern's frequency count and confidence value.

The Max Subpattern Tree was stored in a series of nodes, each of which contains a dynamically created array of pointers to its child nodes and a dynamically allocated, variable length string in which to store the subpattern. Each node also contained the subpattern frequency count, L-length and the level of the node in the tree (the distance from C_{max}).

Several additional measures were taken to improve the speed of the design, including using statically declared arrays rather than dynamic allocation for functions that are called frequently, and storing the length of a string used in a loop in a variable, rather than calculating the length each time through the loop as part of the termination criteria.

In verbose mode, the program also gathers frequency information about the set of possible events L, while reading the sequence from the database. This information is stored in two simple arrays: an array of chars for the value and an array of integers for the count. If the program is not in verbose mode, this information is not gathered nor displayed to the user.

3.10 Code Structure

This implementation is written in C language. Including the discretization process, the two programs include 2800 lines of code in ten files. The program structure is as follows:

```
main()
  PrintUsage()            - print usage and exit
  Find1Patterns()         - find frequent 1-patterns in data set
    GetNextPattern()      - get the next pattern from the input file
      strip()             - strip a line of extraneous characters and convert commas to spaces
      GetSourceColumn()   - get the nth column's value from the input buffer
  PrintPatterns()         - print the list of frequent patterns with stats
  DetermineCmax()         - derive the Cmax (maximal subpattern possible from F1)
  CreateTree()            - derive the rest of the tree by scanning the data file a second time
    GetHitMaxPattern()    - calculate the maximal subpattern of Cmax that is 'hit' by this sequence
    InsertMaxPattern()    - insert a hit max pattern into the tree
    RemovePosition()      - remove a character from a pattern, checking for ORs
  PrintTree()             - recursively print out the max subpattern tree (for debug purposes)
  FindKPatterns()         - find frequent k-patterns in max sub-pattern tree
    JoinPatterns()        - join two patterns if they are joinable
      FindPosition()      - find the position in pat1 corresponding to the (pos)th position in pat2
    CountPattern()        - count the occurrences of this pattern in the tree
      CountTree()         - search the tree, following the paths down these missing characters
    AprioriCount()        - find patterns using Apriori algorithm instead of hitset algorithm
  SortUniqueRows()        - sort the unique values encountered in the sequence
```

Additional supporting memory functions are used to implement the linked list used to store frequent patterns and the Max Subpattern Tree structure:

`AddListNode()`	- create a new node and insert into linked list
`ListNodeExists()`	- search for an existing linked list node
`DeleteListNode()`	- delete a node from the linked list from pattern
`DeleteListNodeP()`	- delete a node from the linked list from pointer
`FreeAllListNodes()`	- delete all linked list nodes
`CreateTreeNode()`	- create a single tree node for insertion into the tree
`AddTreeNode()`	- add a tree node as a child of another tree node
`DeleteTree()`	- recursively delete the entire tree

For additional details on the input and output variables for each function, please see the function prototypes in *hitset.h*. Further details can be found in the comments above each function in the code.

4 Experimental Results

Experiments were performed on the two discretized sets of data with cardinality three and seven, in order to compare the final results to find common repeating partially periodic patterns, and to compare the accuracy and performance of the Hit Set algorithm to the strictly Apriori approach.

4.1 Interpretation of Results

The patterns resulting from analyzing each of the ten source columns in the cardinality three and cardinality seven discretized files, using a period of five days and a minimum confidence of five percent, are given in the `results` directory.

For example, considering the third column only (IBM) for all 3915 rows, we obtain a total of 165 partially periodic patterns in the input file discretized to cardinality three. Following are the top ten patterns, ordered by length (period minus the number of *don't care* events) and confidence:

Pattern	Length	Frequency	Confidence
`[CAAA⋆]`	4	43	5.49%
`[CCAC⋆]`	4	40	5.11%
`[CA⋆A⋆]`	3	85	10.86%
`[CAA⋆⋆]`	3	81	10.34%
`[CA⋆⋆A]`	3	81	10.34%
`[C⋆A⋆A]`	3	80	10.22%
`[C⋆A⋆C]`	3	78	9.96%
`[CA⋆⋆C]`	3	77	9.83%
`[C⋆AC⋆]`	3	77	9.83%
`[CAC⋆⋆]`	3	77	9.83%

Comparing these results to those for the eleventh column (Microsoft):

Pattern	Length	Frequency	Confidence
`[⋆ACA⋆]`	3	78	9.96%
`[⋆AC⋆C]`	3	77	9.83%
`[CAC⋆⋆]`	3	74	9.45%
`[⋆⋆CAA]`	3	74	9.45%
`[⋆CC⋆A]`	3	71	9.07%
`[CC⋆A⋆]`	3	69	8.81%
`[⋆ACC⋆]`	3	69	8.81%
`[⋆AC⋆A]`	3	69	8.81%
`[C⋆C⋆A]`	3	69	8.81%
`[C⋆CA⋆]`	3	69	8.81%

There are many differences between these sets of patterns, for example two patterns of length four are frequent in the IBM set but not in the Microsoft set.

However, some patterns are repeating at roughly the same confidence values: for example, the pattern [CAC**] (indicating growth on the Monday, decrease on the Tuesday, and growth on the Wednesday of these time segments) is seen in both files, at confidence values of 9.83% and 9.45% respectively. Comparing these results with those of the other files, we see this same pattern appearing in all the files, with similar confidence values:

Pattern:	[CAC**]
Stock	Confidence
IBM	9.83%
Intel	9.83%
Microsoft	9.45%
Dell	8.05%
HP	7.92%
Sun	6.39%
Apple	6.39%
Motorola	5.75%
TI	5.62%
Oracle	5.11%

We might interpret this result as meaning that the stocks for IBM, Intel and Microsoft, and to a somewhat lesser extent Dell and HP, are closely correlated within this technology sector, at least in terms of this particular partially periodic pattern. This makes intuitive sense considering the nature of these companies, however additional domain knowledge would be required to confirm this to be true.

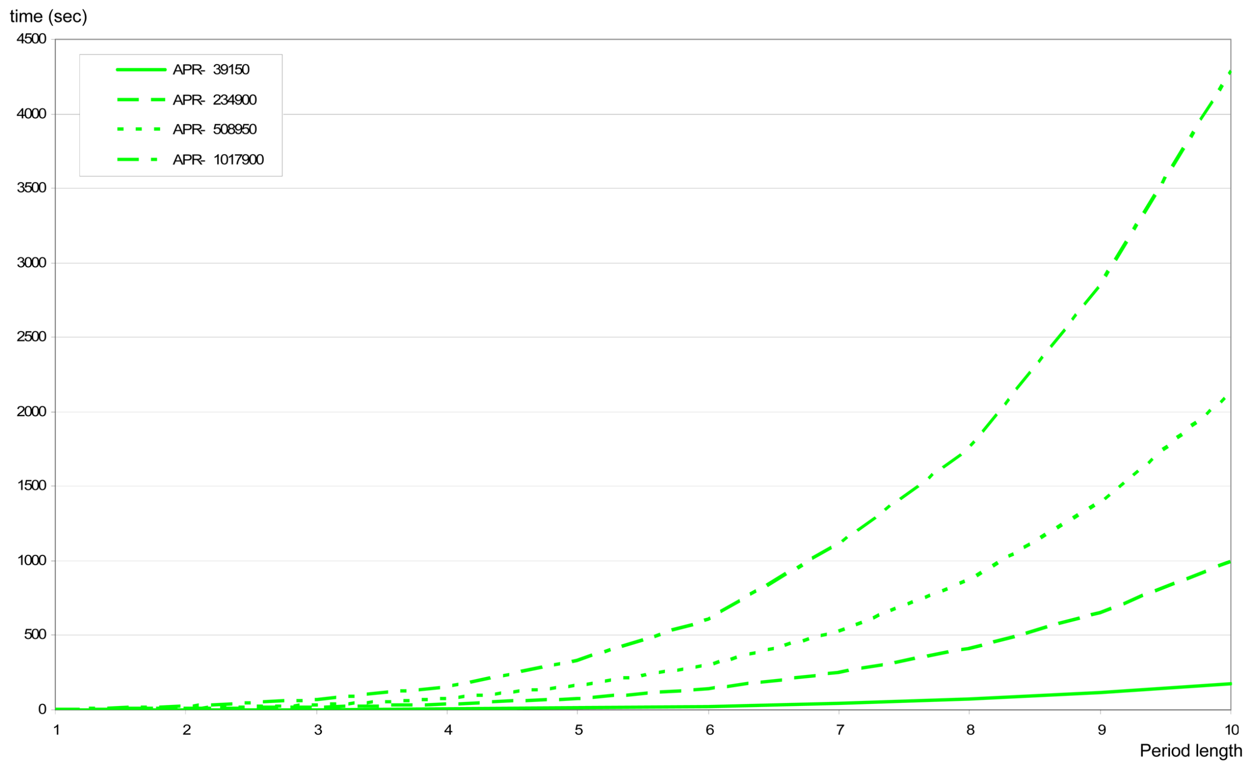
4.2 Higher Period Lengths

For interest, the program was also run using a period of ten days with the cardinality seven discretized input file, using all ten stocks at once (39 150 rows of data). A total of 1069 repeating patterns were found at a minimum confidence threshold of five percent, including the following top ten rows:

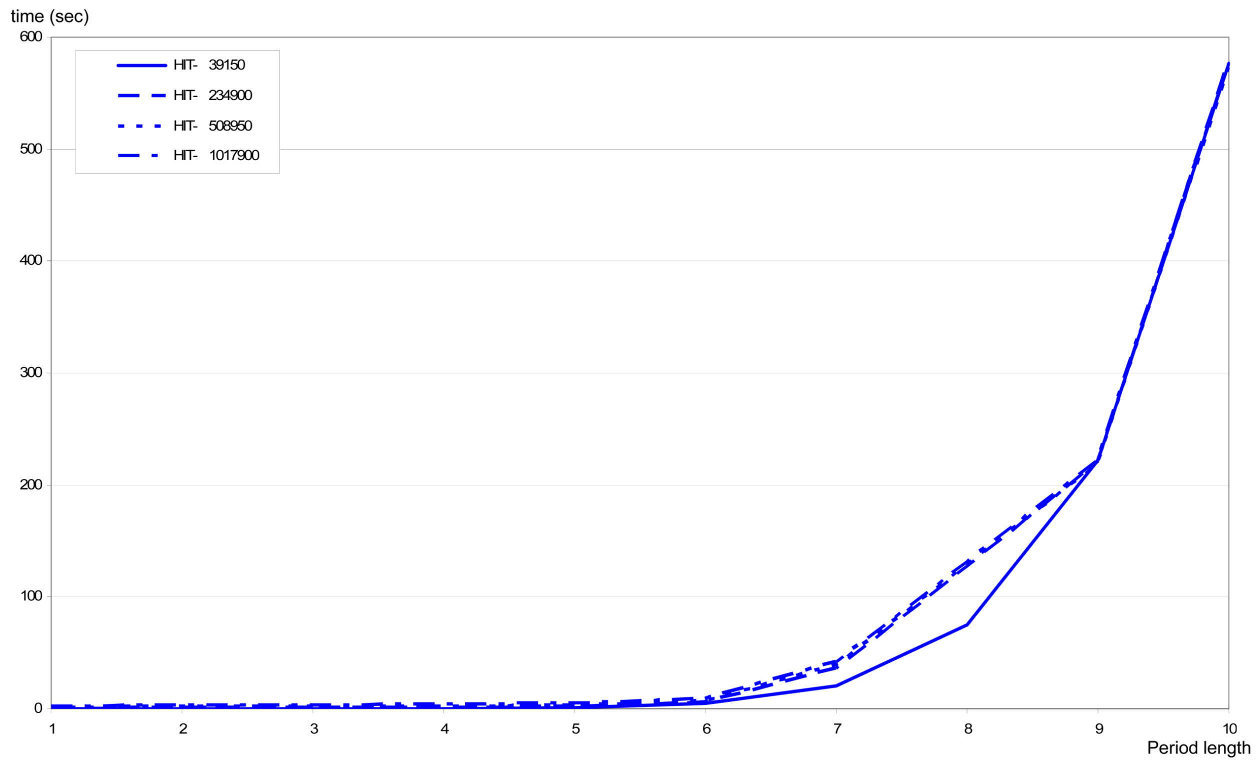
Pattern	Length	Frequency	Confidence
`[⋆⋆E⋆C⋆⋆E⋆⋆]`	3	283	7.23%
`[⋆⋆E⋆⋆⋆CE⋆⋆]`	3	276	7.05%
`[⋆⋆E⋆C⋆C⋆⋆⋆]`	3	276	7.05%
`[⋆CE⋆⋆⋆⋆E⋆⋆]`	3	275	7.02%
`[⋆CE⋆C⋆⋆⋆⋆⋆]`	3	275	7.02%
`[⋆⋆⋆⋆⋆⋆E⋆CC]`	3	274	7.00%
`[⋆⋆⋆CC⋆⋆E⋆⋆]`	3	270	6.90%
`[C⋆E⋆C⋆⋆⋆⋆⋆]`	3	270	6.90%
`[⋆⋆⋆C⋆⋆E⋆C⋆]`	3	267	6.82%
`[⋆⋆ECC⋆⋆⋆⋆⋆]`	3	267	6.82%

These patterns can be found in the `results` folder, in the file `period10.patterns`.

The program was also run for periods of 65 (one fiscal quarter) and 260 (one fiscal year) to ensure that the implementation was capable of analyzing patterns of this length, however no meaningful patterns were generated even with low minimum confidence values. In order to obtain meaningful results, it would be more effective to lower the daily resolution to weekly or monthly resolution, and discretize the increase or decrease for each stock within that time period.



(a) Apriori-only algorithm.



(b) Max Hit Set algorithm.

Figure 6: Comparison of execution time for the Max Hit Set algorithm and Apriori-only algorithm, for four files containing 39 150, 234 900, 508 950 and 1 017 900 rows respectively, with a minimum confidence of 5% and varying target period lengths from one to ten. The Max Hit Set algorithm achieves nearly an order of magnitude better performance when processing the file containing over one million records.

4.3 Performance Results

In Fig. 6 we see a comparison of the performance results for patterns of varying length, for the set of all ten stocks with 39 150 rows. To reduce the effects of disk caching, ten additional redundant columns were added. The number of rows in each file was then increased by appending the file to itself several times, in order to form larger files with 234 900, 508 950 and 1 017 900 rows. In this way, we increase the size of the file to approximately one quarter million rows, one half million rows and one million rows respectively, but since the data points are all reproduced the same number of times, we affect only the frequency counts of all repeating patterns and not the confidence values.

This allows us to verify the accuracy of the results for both algorithms at each targeted period length: in all four cases, both algorithms gave the exact same resulting patterns at each period length from one (1-patterns only) through ten. These tests were run on a 3.4 GHz Pentium IV processor with 1 GB of DDR2 dual-channel memory, running SuSE Linux 9.2 Professional with kernel 2.6.8. The results are detailed in the `results` directory, in the file `performance.results`.

As expected, increasing the pattern length increases the complexity of the resulting patterns and therefore the processing time is approximately proportional to the square of the pattern length. However, whereas the Apriori-only algorithm dramatically increases processing time in proportion to the number of rows, the Max Hit Set algorithm does not: increasing the number of rows for this algorithm has little effect on the total processing time as only the repeating subpatterns are stored in online memory.

We note that the curve for the Max Hit Set algorithm with 39 150 rows is slightly below the remaining three curves. It seems likely that this is caused by the fewer rows allowing the machine's disk caching algorithm to read ahead for these rows, whereas with a higher number of rows this effect is diminished.

Importantly, it can be seen that the Max Hit Set algorithm actually takes *longer* to process the file with 39 150 rows for higher pattern lengths in comparison to the Apriori-only algorithm. Although at first surprising, this again makes sense in the context of the Unix disk caching mechanism allowing the Apriori-only algorithm to (effectively) read the smaller file only from memory. This underscores the intention of the Max Hit Set algorithm to be used only for very large databases, where the database cannot fit entirely into main memory.

5 Evaluation of Algorithm

In this section, we list some of the advantages and disadvantages of this algorithm that were found through the course of this implementation and subsequent testing on the discretized stock market data.

5.1 Algorithm Advantages

The Max Hit Set algorithm has been found to be very efficient for large database sizes in comparison with the Apriori-only approach, since the online memory is used only to store the repeating patterns, rather than the input file itself. This means that the Max Hit Set algorithm requires only two passes of the database in order to find all repeating patterns, which significantly reduces the I/O overhead required for the Apriori-only algorithm for larger database sizes. However, this also means that with decreased randomness in the file, ie. with greater repetition, the Max Hit Set algorithm will use less online memory since it needs to create a less complex Max Subpattern Tree data structure. In other words, the more frequently the patterns repeat, the less memory is used by the algorithm and the greater the performance.

Another advantage is the flexibility of this algorithm: it can be easily applied to a wide variety of realistic application scenarios. In fact, it seems likely that the algorithm can efficiently be applied to any large, sequential time series database, if the column values to be analyzed have low cardinality and a high degree of partially periodic repetition.

5.2 Algorithm Limitations

As with the Apriori-only approach, the Max Hit Set algorithm does require low cardinality in the columns in order to be efficient. In this implementation, the cardinality is arbitrarily limited to 255 values as a single char is used to represent each event, however a file with cardinality this high is likely to have very few partially periodic patterns, so this limitation is a reasonable assumption to make — for our stated application scenario, we are looking at files with cardinalities of 3 and 7, significantly lower than the artificial limit. However, this need for low cardinality means that real-valued columns must be discretized, an additional step which is not required for some similar algorithms such as [1].

Another disadvantage was shown in the performance testing in the previous section: for files with a smaller number of rows and higher periods, where the database can be fitted within the computer's disk caching mechanism, the Apriori-only approach yields higher performance.

Lastly, segment-wise periodicity means that there may be some data points at the end of a sequence that are not considered, where the number of remaining data points is not sufficient to form a full segment. This implementation reports the number of rows that are ignored in this fashion in the "Neglected End Rows" statistic when in verbose mode. However, since the number of neglected rows must by definition be fewer than the period length, and since the period lengths must be very small in comparison to the number of rows for this algorithm to be efficient as we have seen, this limitation is somewhat negligible.

6 Conclusion

The Max Hit Set algorithm, employing the Apriori principle for search space reduction, and a Max Subpattern Tree interim data structure to reduce the required number of database scans, has been shown in this paper to be very efficient for large database sizes containing hundreds of thousands of rows and larger.

In comparing the performance to an Apriori-only approach, we find that the processing time required for the Max Hit Set algorithm is proportional only to the amount of randomness in the file, and is virtually insensitive to the total number of rows in the file. This means that given two files of the same length, the file with more frequent repetition of values will require less online memory and will complete faster using this algorithm, as it would have a less complex Max Subpattern Tree with fewer nodes, whereas with an Apriori-only approach, both files would take roughly the same amount of time regardless of the amount of randomness in the file. This characteristic makes the algorithm particular attractive for very large databases with low cardinality columns, although an additional discretization step would be necessary to apply the algorithm to real-valued columns, as was done in this paper.

In the particular application scenario for analyzing pricing data for ten technology stocks from 1990 to 2004, we find strong correlations in the repeating patterns for IBM, Intel and Microsoft as illustrated by the pattern [CAC**], which was found to be a frequent partially periodic pattern common to all ten technology stocks, but had higher confidence values for these particular three stocks.

Future improvements to this implementation could be made by allowing optional events at the cost of dramatically increasing the search space, or simultaneous mining of multiple periods through the use of multiple C_{max} structures as suggested by [11]. Independent Component Analysis (ICA) [17] could be added to allow n-dimensional vector data from multiple columns to be discretized as the event set, and a fuzzy threshold could be added as a disturbance to the start of each retrieved segment to allow some insensitivity to inter-segment shifts [21].

References

[1] R.Agrawal, R.Srikant (1995), “Mining Sequential Patterns,” Proceedings of the 1995 International Conference on Data Engineering, pp. 3–14.

[2] R.Agrawal, R.Srikant (1994), “Fast Algorithms for Mining Association Rules,” Proceedings of the 1994 International Conference on Very Large Databases.

[3] W.G.Aref, M.G.Elfeky, A.K.Elmagarmid (2004), “Incremental, Online and Merge Mining of Partial Periodic Patterns in Time Series Databases,” IEEE Transactions on Knowledge and Data Engineering, 16(3).

[4] A.J.Bagnall, G.J.Janacek (2004), “Clustering Time Series from ARMA Models with Clipped Data,” ACM-KDD.

[5] R.J.Bayardo Jr. (1998), “Efficiently Mining Long Patterns from Databases,” Proceedings of. 1998 ACM-SIGMOD, pp. 85–93.

[6] A.Cichocki, A.K.Barros (1999), “Robust Batch Algorithm for Sequential Blind Extraction of Noisy Biomedical Signals,” ISSPA.

[7] A.Flexer, H.Bauer (2003), “Monitoring Human Information Processing via Intelligent Data Analysis of EEG Recordings,” Intelligent Data Analysis, 4, pp. 113–128.

[8] D.Garrett, D.A.Peterson, C.W.Anderson, M.H.Thaut, (2003), “Comparison of Linear, Non-Linear, and Feature Selection Methods for EEG Signal Classification,” IEEE Transactions on Neural Systems and Rehabilitation Engineering, 11(3).

[9] J. Han, M. Kamber (2001), “Data Mining: Concepts and Techniques,” Morgan Kaufmann.

[10] J.Han, J.Pei, Y.Yin (2000), “Mining Frequent Patterns without Candidate Generation,” Proceedings of the 2000 ACM SIGMOD International Conference.

[11] J.Han, G.Dong, Y.Yin (1999), “Efficient Mining of Partial Periodic Patterns in Time Series Database,” ICDE.

[12] J.Han, W.Gong, Y.Yin (1998), “Mining Segment-Wise Periodic Patterns in Time-Related Databases,” Proceedings of the 1998 ACM-KDD.

[13] E.Keough, S.Chu, D.Hart, M.Pazzani (2001), “An Online Algorithm for Segmenting Time Series,” IEEE.

[14] K.B.Pratt, E.Fink (2002), “Search for Patterns in Compressed Time Series,” International Journal of Image and Graphics.

[15] R.J.Povinelli, X.Feng (2003), “A New Temporal Identification Method for Characterization and Prediction of Complex Time Series Events,” IEEE Transactions on Knowledge and Data Engineering, 15(2).

[16] R.J.Povinelli, X.Feng (1998), “Temporal Pattern Identification of Time Series Data using Pattern Wavelets and Genetic Algorithms,” Artificial Neural Networks in Engineering.

[17] A.D.Back, T.P.Trappenberg (1999), “Input Variable Selection Using Independent Component Analysis,” Proceedings of the 1999 Annual Joint Conference on Neural Networks.

[18] T.P.Trappenberg (2002), “Fundamentals of Computational Neuroscience,” Oxford University Press.

[19] Y.Yamada, E.Suzuki, H.Yokoi, K. Takabayashi (2003), “Decision Tree Induction from Time Series Data Based on a Standard Example Split Test,” ICML.

[20] Yahoo! Finance, Commodity Systems, Inc., [http://finance.yahoo.com].

[21] J.Yang, W.Wang, P.S.Yu (2003), “Mining Asynchronous Periodic Patterns in Time Series Data,” IEEE Transactions on Knowledge and Data Engineering, 15(3).