This is to certify that the
thesis entitled

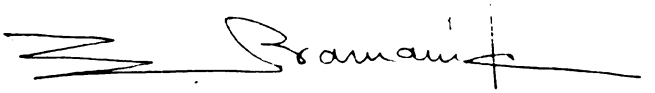
OPTIMIZATIONS AND APPLICATIONS OF
TRIE-TREE BASED FREQUENT PATTERN MINING

presented by

Stuart King

has been accepted towards fulfillment
of the requirements for the

M.S. degree in Computer Science



Major Professor's Signature

6 - 11 - 06

Date

OPTIMIZATIONS AND APPLICATIONS OF TRIE-TREE BASED FREQUENT PATTERN MINING

By

Stuart King

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTERS OF COMPUTER SCIENCE

Department of Computer Science and Engineering

2006

ABSTRACT

OPTIMIZATIONS AND APPLICATIONS OF
TRIE-TREE BASED FREQUENT PATTERN MINING

By

Stuart King

Frequent pattern mining is an active area of research in a field of artificial intelligence called knowledge discovery. With it, we are teaching computers to discover hitherto undiscovered information about ourselves and the world around us. Traditionally, discovering patterns has been a very resource-intensive endeavor, costing both a great deal of memory and processing time. In this research, we are going to introduce a mechanism to geometrically reduce memory requirements. This allows frequent pattern mining to be applied to new domains. To prove this, a new client application in PC Internet security is developed that takes advantage of this optimization. Additionally, research is presented on a CPU optimization for degenerate cases in the application. In combination, this thesis presents a road map for advancing frequent pattern mining in a new direction.

Acknowledgements

Masters theses in Computer Science are relatively rare. However, doing so gives a masters student an opportunity to not just learn science, but to contribute to the scientific community in a very real way. Computer Science is such a young field, that with the right encouragements and guidance, even minimal training can be leveraged to produce significant contributions. This work does exactly that, but it was not done alone.

Any thesis takes a team to make it happen, and this one is no exception. I wish to recognize and thank several key individuals in that team who helped make this thesis possible. First and foremost is Dr. Satki Pramanik who helped drive my desire to contribute to the scientific community and offered both perspective and direction as he led my thesis committee. In addition, Dr. Pang-Ning Tan provided encouragement, enlightenment, and the ever important critical challenges needed to produce strong research. Any work is always stronger when you have a partner and sounding board to bounce ideas off of. Li Jian and I had countless sessions, reviewing both good and bad ideas, and his efforts helped immensely. Another seed of help came from a simple challenge made by Dr. Matt Mutka. The challenge was to "make something practical", and Chapter 5 is the answer to that challenge.

There are many others, more than can be named here, but I wanted to make a special thank you to Linda Moore who helped me navigate the thesis process. Her experience and enlightenment helped the entire team to produce this work.

Thanks to one and all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOTE: Images in this thesis are presented in color.

CHAPTER 1: Introduction

Frequent pattern mining is a booming area of research on the process of discovery. In its simplest terms, it is finding patterns in large quantities of data. In a broader sense, it is a kind of artificial intelligence that reveals relationships in our complex world that we have never before seen. It took less than two years to get from the first practical algorithm for frequent pattern mining to the creation of an entire industry dedicated to the task. This thesis could not possibly cover in detail all of the various aspects of this research, but it does aim to introduce some important aspects of the field, and to present a new direction for it.

To accomplish that aim, the paper is broken down into two distinct parts. The first part introduces some of the concepts, objectives, and methods of frequent pattern mining, as well as some of the more common algorithms used. The second part introduces a new algorithm to geometrically reduce the memory requirements of a large genre of frequent pattern mining algorithms. As a proof-of-concept for the potential of the algorithm, a new client application in PC Internet security is developed that takes advantage of the optimization. As will be seen, this proves very successful. However, extending the application to do a more detailed analysis revealed a degenerate case in which CPU utilization dramatically increased. Additional research is presented on this degenerate case and how to more efficiently handle it. Combined, this thesis introduces a complete approach for a practical application of frequent pattern mining in a personal computing environment.

PART ONE

FREQUENT PATTERN MINING

CHAPTER 2: Definitions and Practices

2.1 What is "Frequent Pattern Mining"

To better understand what frequent pattern mining is (a.k.a. *mining*[1]), it helps to understand some of the applications it is being used for, and to know a few key terms. To start with, we will provide a definition of what a *frequent pattern* is:

Frequent Pattern:

Given a set of discrete items and a set of records composed from these items, a frequent pattern is a combination of items that occurs frequently within the records.

NOTE: Ideally, this means patterns that occur more often than can be explained by random chance; however other definitions can apply (see *minimum support* below).

Example:

Set of Discrete Items: "A", "B", "C", "D", "E"
Set of Records: "ABD", "BCD", "ABCDE", "BE"
Frequent Pattern: "BD" occurs in three of the four records

You might ask "So what, what is the big deal?". The answer is *association rules*. An association rule is a relationship between items in a dataset. Finding association rules between items helps to make predictions about the future. To illustrate, consider if the "Set of Records" from the example represents a database of purchase records where "B" stands for "Bread" being purchased, and "D" stands for "Hot Dogs" being purchased. Then someone buying "Bread" has a

[1] There are many meanings for the word *mining*, including the more general term *data mining* representing the discovery of any potentially useful knowledge from data. However, this is a *frequent pattern mining* thesis, so *mining* throughout the entire thesis will refer to "frequent pattern mining".

75% chance of also buying "Hot Dogs". This is an association rule and can be written "B => D[2] with 75% *confidence*".

Another Association Rule would be "AC => E with 100% confidence". However, this Association Rule needs to be tempered with the fact that the pattern "ACE" only occurs once in this database of four records. That is, the rule "AC => E" only has 25% *support*. Although both confidence and support are important, confidence can be calculated from support. That is, given the support of the rule's assertion, "ACE", and the antecedent, "AC", the confidence can be calculated as follows:

> Confidence = Support of the rule's assertions / Support of the antecedent
> Confidence(AC => E) = Support("ACE") / Support("AC") = 25% / 25% = 100%

As a result, a *minimum support* is frequently specified when mining for frequent patterns. Minimum support, like it sounds, is specifying the lowest support a pattern can have and still be considered a frequent pattern. Using the example above, a minimum support of 50% would remove the pattern "ACE" from consideration. A good value for minimum support is very subjective and data dependent. For example, a 10% success rate is small, but a 10% death rate is high.

What is not subjective is the definition of the terms "Support", "Confidence", and "Association Rule". Formally, the definitions are as follows:

> Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of literals called items. Let D be a set of transactions where each transaction T is a set of items such that $T \subseteq I$. Given an antecedent of A (where the items of $A \subset I$) and a consequence C (where the items of $C \subset I$) such that $A \cap C = \varnothing$, than an *Associate Rule* is an implication of the form A => C. As such, a transaction T is said to *contain* the antecedent A if

[2] "=>" stands for implication, so "AC => E" is saying "condition A and C implies condition D"

$A \subseteq T$. Similarly, a transaction T is said to *contain* the consequence C if $C \subseteq T$. The rule holds in the transaction set *D* with "*Confidence* of c%" if c% of the transactions in *D* that contain A, also contain C. The rule A => C has a "*Support* of s%" if s% of the transactions in *D* contain $A \cup C$.

Associations between items like "Hot Dogs & Bread" or "Beer & Diapers"[3] is very important for "market basket" analysis[4], but frequent pattern mining has many other uses. For example, it is currently being used to identify patterns of credit card and identity theft, intrusion detection over IP networks, and cancer detection. In addition, "Knowledge Discovery and Data mining" (KDD) is an active area of research for artificial intelligence systems, and as the field grows, I expect discoveries like the effects of "El Nino"[5] on global weather patterns and patterns in economic flux will be identified with mining techniques.

2.2 Metric Datasets and Dataset parameters

Since the introduction in 1993 of frequent pattern mining, many common practices have been adopted for categorizing data for analyzing the performance of algorithms. Just as different sorting algorithms work best with different kinds of data, so do different mining algorithms. Defining the different kinds of data an algorithm works best with involves defining key characteristics of the data that play a significant role in the mining process. Two types of data with known characteristics can assist in analyzing the performance of algorithms. One is a library of datasets with well known properties, and the other is a common algorithm for generating artificial datasets with desired characteristics.

---

[3] In the FIMI'04 data mining competition, the most frequent pattern was "Beer, Diapers".

[4] Argawal et al's research on finding simple patterns in market basket data has touched nearly every super market shopper with the advent of "Super Market 'Club' Cards" for tracking purchase patterns.

[5] The data mining technique Conical Correlation Analysis (CCA) has been used to help find the effects of El Nino, however, combining CCA with other data mining techniques may prove even more effective.

Four of the common attributes used for measuring the characteristics of datasets are:

1) The Number of Records in a Database
2) The Average Record Length
3) The Number of Items Within the Database
4) The Expected Pattern Length[6]

In addition, there is a qualitative attribute commonly used called *pattern density*. Pattern density is a measure of how many patterns a large number of records will produce[7]. If a relatively small number of records produce a large number of frequent patterns, the database is said to have a *high pattern density*. However, if a large number of diverse records only produce a small number of frequent patterns, then the database is said to have *low pattern density* (or its *pattern density is sparse*). Altering the minimum support frequently changes the density of a dataset. For example, a dataset may be dense with a minimum support of 5%, but sparse with a minimum support of 25%.

One of the sparsest datasets is random data, but it is not effective in testing mining algorithms. The very objective of frequent pattern mining is to find data patterns in seemingly random data. One of the biggest challenges of mining algorithms is to efficiently wade through and filter out random data to reveal the hidden patterns. As such, when random data is mined, it turns out that new patterns are found in only a narrow range of minimum support. That is, when minimum support is above this threshold, the random data is filtered out; below this threshold, there are very few combinations of items to be found that have not already been discovered.

---

[6] Many factors effect pattern length, especially minimum support. However, for FIMI datasets there is frequently a known set of useful patterns that can be used to define the expected pattern length.

[7] An alternate definition of a dense dataset is one in which, on average, most records contain at least half of the frequent items. Although most would agree this would produce a dense dataset, the term is frequently used in a broader sense.

So what kind of data is good for frequent pattern mining? Intuitively, it is data with frequent patterns. That is, data that has combinations of items that occur together far more often than can be explained by random chance. Usually such patterns are indicative of forces that we are either ignorant of, or do not fully understand. The real world is full of such forces that interplay in a myriad of fashions.

The FIMI (Frequent Itemset Mining Implementations) repository contains several well known datasets from the real world. These datasets have a variety of characteristics that make them ideal for testing the performance of different implementations of mining algorithms. Some examples of the kinds of datasets are:

Chess and Connect-4: A set of winning moves for the games chess and Connect-4
Mushroom: Mushroom classification data
Webdocs: Terms used in 1.7 million web documents
(and many others)

These datasets are real data from real problems. The reason data mining programs exist is because of datasets like these, and it is important that mining algorithms work well with this kind of data. However, the FIMI repository does not contain every possible real world dataset. As such, finding a spectrum of datasets to fully test and understand the limits, scalability, and capabilities of a mining program is not possible. As a result, FIMI has a method of generating *synthetic datasets*.

Synthetic datasets are artificial datasets that are guaranteed to contain frequent patterns. *Synthetic dataset generators* are programs designed to generate synthetic data and were developed to give the researcher more control of the characteristics of a dataset. As a result, a spectrum of datasets can be generated that allows the different components of an algorithm to be tested. These datasets not only give the developer the control to

incrementally vary the attributes of a dataset, but they are also guaranteed to contain frequent patterns[8]. One such generator was developed by IBM and creates data using the following process:

Procedure Generate_Synthetic_Data

1) The *data generator* takes many input parameters including:
   a) I: The number of items used by the artificial dataset
   b) T: The average length of records
   c) D: The number of records

   (and several other parameters)
2) A list of random patterns are generated[9]
3) Records are generated as follows:
   a) The size of the record is randomly chosen to be close to the desired length.
   b) A pattern is selected.
   c) One or more items are randomly chosen from the selected pattern.
   d) steps b & c are repeated until the record is the chosen length.
   e) Steps a-d are repeated until the desired number of records have been created.

Usually, when a synthetic dataset is generated, it is given a name similar to TxIyDz, where x is the average length of transactions, y is the number of items in the dataset (in 1000s) and z is the number of records in the database. So, a dataset called T10I10D100K would contain 100,000 records averaging 10 items in length and using about 10,000 items. Since each record was created from patterns, this forces the existence of frequent patterns within the dataset. Because of the ability to tightly control dataset attributes, synthetic datasets will be used later on in this work to evaluate the performance of algorithms.

[8] That is, data that has combinations of items that occur together far more often than can be explained by random chance.

[9] Since the items of records only come from pre-generated patterns, the actual number of items used in the synthetic dataset may be less than the desired number.

2.3 Trie-Tree data structure

Several effective algorithms in frequent pattern mining have adopted deviations on a structure called a Trie-Tree. Wikipedia defines a Trie as:

> "A **trie**, or **prefix tree**, is an ordered tree data structure that is used to store an associative array where the keys are strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of any one node have a common prefix of the string associated with that node, and the root is associated with the empty string."

For example, given the strings "ABC", "ABD", and "BCD", the associated Trie-Tree would look similar to Figure 3.1. In this example, "ABC" and "ABD" have a common prefix of "AB", so both strings share the use of the nodes "A:2" and "B:2". The number following the ":" is a common convention used by researchers to indicate the number of strings using the node. That is, the "1" of node "B:1" indicates only one string is using the node. As such, the original records of the dataset are captured within the Trie-Tree. For example, following up from the leaf node "C:1" of Figure 1, it can be seen the original dataset had a record "ABC".

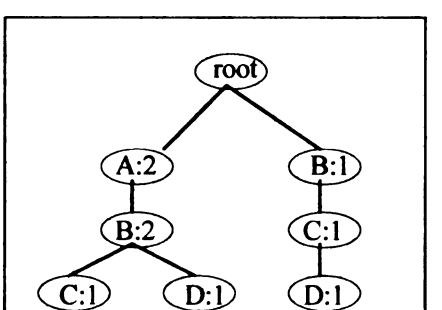


Figure 1: Trie-Tree

CHAPTER 3: Previous Work

3.1 SimpleMining: Is Not Easy To Do

Mining frequent patterns is a complex problem that is a very CPU and RAM intensive application. Given n unique items within a database, the SimpleMining algorithm below has a time complexity and space complexity of $O(2^n - 1)$. As such, even a small database with only 85 unique items[10] would require 38,685,626,227,668,133,590,597,631 patterns to be generated. Even though computers are constantly getting faster and bigger, this brute force method of mining is just not practical. As a result, the first practical algorithm for frequent pattern mining needed to introduce a mechanism for pruning pattern combinations that were not useful.

Procedure SimpleMining

1) Scan through the dataset and store each unique item.
2) Generate and store all possible patterns (combinations of items).
3) Scan the dataset again.
   a) Rearrange the unique items of each record into all possible combinations.
   b) For each combination, increment the counter of its associated pattern.
4) Scan through the patterns and print any that are above a given support.

3.2 LargeItemsets and Apriori: The First Practical Algorithms

In 1993, Agrawal (et. all.) wrote just such an algorithm. The *LargeItemsets*[11] algorithm (see below) was introduced in a paper called "Mining Association Rules Between Sets of Items in Large Databases". The primary method it used for pruning unneeded patterns involved the *Monotonicity Principle*. Basically, the Monotonicity Principle says:

[10] This would be the approximate number of items from the menu of your local fast food restaurant.
[11] This algorithm is sometimes called "AIS" after the authors of the paper (ie: "Agrawal Imielinski Swami")

If pattern B $\subseteq$.D, then the support(B) $\geq$ support(D)

Therefore, if the pattern B is infrequent, then all its supersets are also infrequent. As an example, if the pattern B = "AB", the pattern D = "ABCD", and the support(B)=10%, then the Monotonicity Principle says support(D) $\leq$ 10%.

LargeItemsets combined the Monotonicity Principle with the *Mutual Independence* method of estimating frequencies to greatly reduce the number of patterns that needed to be generated (see Section 5 in Chapter 5 for details on Mutual Independence). However, introducing "estimating" also introduced error. As a result, the LargeItemsets algorithm may "estimate" that a pattern is not frequent when it actually is. As a result, several passes through the database may be needed to eliminate such "estimating errors".

Procedure LargeItemsets

1) Scan through the dataset and count the frequency of each unique item.
2) Based on what is known, "estimate" which patterns are frequent, or might be almost frequent. Call this set the "candidate itemsets"
3) Scan the dataset again.
   a) Rearrange the unique items of each record into all possible combinations
   b) For each combination that is in the list of "candidate itemsets", increment the counter of its associated pattern.
4) If any pattern expected to be infrequent turns out to be frequent, go back to step 2.

Although LargeItemsets was a practical algorithm for mining, it still produced a large number of "candidate itemsets", however, some simple observations helped to suggest an improved algorithm. One observation was that most patterns were short (only three or four items), and the LargeItemsets algorithm usually took three or four scans through the database. As such, an algorithm that "grows" candidate itemsets might be more efficient.

That is, an algorithm would be more efficient if it generates "length 2" candidate itemsets before the second scan, "length 3" candidate itemsets before the third scan, etc. Another simple observation that helped is that most of the items within a database were not frequent. So, it was possible to significantly reduce the size of the database records by filtering out infrequent items. As such, encoding of the data could usually reduce the size of the database to a size that could fit in RAM.

This improved algorithm was introduced by Agrawal only one year after the LargeItemsets algorithm. It was called Apriori (see below), and it turned out to be significantly faster than LargeItemsets for most datasets. To this day, algorithms based upon the Apriori algorithm are commonly used for frequent pattern mining. The algorithm works as follows:

Procedure Apriori

1) Scan through the dataset and count the frequency of each unique item
2) Rescan the dataset and filter out infrequent items from each record and encode the data to create a "compressed dataset"[12]
3) Initialize K=1
4) Generate all "candidate itemsets" of length K+1
   a) Generate length K+1 patterns based upon the "length K candidate itemsets"
   b) For each new pattern, filter out patterns that can not be frequent
5) Scan the compressed dataset
   a) Rearrange the items of each record into length K+1 combinations
   b) For each combination that is in the list of "length K+1 candidate itemsets", increment the counter of its associated pattern
6) If any patterns in "length K+1 candidate itemsets" is frequent, Then increment K and repeat steps 4-6

[12] Technically, the first "encoding and compression" of Apriori was introduced in the same paper as Apriori, and was called AprioriTid. However, since then, many other mechanisms have been introduced.

The compression of the dataset (step 2) has been implemented in many ways, but there are two common features that are frequently used. One is that duplicate items are removed from each record. A more significant feature is that items are encoded into numbers. That is, each frequent item is assigned a number, and the numbers are stored in the "compressed dataset". For example, the record { "beer", "diaper", "Purple People Eater", "diaper" } could be encoded as { 1 2 } if "diaper"=1, "beer"=2, and "Purple People Eater" is not a frequent item. These two mechanisms are so common that a lot datasets targeted for mining will have these compression mechanisms applied as a pre-processing step to mining.

Step 4 of the Apriori algorithm[13] filters patterns using the Monotonicity Principle. Since one consequence of the Monotonicity Principle is that each frequent "length K pattern" could have supersets of "length K+1" that are frequent, step 4a generates its "length K+1 candidate itemsets" from its "length K candidate itemsets". Another consequence of the Monotonicity Principle is that a "length K+1 pattern" can only be frequent if all subsets of "length K patterns" are frequent. An example of Apriori filtering is:

Given:

K=3

The patterns L_3 = { {1 2 3}, {1 2 4}, {1 3 4}, {2 3 4}, {3 5 6}, {4 5 6} } are frequent

Then: (step 4a)[14]

{1 2 3} and {1 2 4} differ by one item, so {1 2 3 4} might be frequent

{1 2 4} and {1 3 4} differ by one item, also generates {1 2 3 4}

...

{3 5 6} and {4 5 6} differ by one item, so {3 4 5 6} might be frequent

[13] This application of the Monotinicity Principle is sometimes referred to as the *Apriori principle.*

[14] NOTE: There are many mechanisms to make step 4a more efficient; however, only a simple method is demonstrated here.

Filtering: (step 4b)

For the pattern {1 2 3 4} to be frequent, all "length 3" subsets must be frequent.

Since { {1 2 3}, {1 2 4}, {1 3 4}, {2 3 4} } is a subset of L_3, {1 2 3 4} $\in L_4$

For the pattern { 3 4 5 6 } to be frequent, all "length 3" subsets must be frequent.

Since the subset { 3 4 5 } $\notin L_3$, the pattern { 3 4 5 6 } $\notin L_4$

The final resulting "candidate length 4 itemsets" is L_4 = { {1 2 3 4} }

Scanning will need to be done to validate if any patterns in L_4 are actually frequent.

3.3 Trie-Tree Approaches

Apriori works very well when there are not too many items. However, if there are a lot of frequent items in the dataset, there is a drastic increase in the size of the "candidate itemsets". This has a significant impact on both the size and processing time of the algorithm. One common circumstance in which this occurs is when the support threshold is relatively small. When there are a lot of frequent items, there are Trie-Tree approaches that can help. One of the first such approaches is Tree Projection.

Tree Projection is a Trie-Tree frequent pattern mining algorithm that stores compressed and filtered copies of the dataset in the leaf nodes of the tree. It is a top down mining algorithm that grows the depth of the tree as it discovers longer frequent patterns. A key aspect of the algorithm is that it reorders the items within a record. For example, Tree Projection reorders the items within the records "ACB" and "ABCD" to make it easier to see the pattern "ABC". Tree Projection works as follows:

Procedure TreeProjection

1) Scan through the dataset and count the frequency of each item
2) Create a tree structure similar to Figure 2 for the frequent items in the dataset
3) Rescan the dataset. For each record scanned, do the following:

   Add_Records_To Nodes:

   a) Filter infrequent items from the record
   b) Order the items in the record
   c) Based on the first item in the record, add the record to a node of the tree
   d) Remove the first item from the record
   e) Repeat steps C & D until there are no more items in the record

4) For each set of records in a leaf node of the tree
   a) Count the frequency of items
   b) Add a branch to the leaf node for each item that is frequent in the node
   c) For each record of the node, use the "Add_Records_To_Nodes" procedure
5) Repeat step 4 until there are no more frequent items in any of the leaf nodes

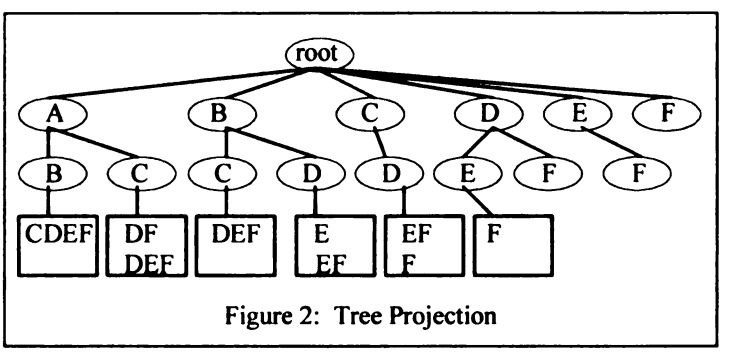


Figure 2: Tree Projection

Although Tree Projection was a big step forward in "growing" frequent patterns without generating "candidate itemsets", the algorithm had significant performance degradation with long records and dense datasets. A few observations in the mining process helps to suggest areas of improvement. First, a single record can contribute to many different patterns, so a record might be copied to many different leaf nodes. This has a significant impact for the initial Tree Projection Trie because the tree branches are short and the leaf

nodes contain longer records. An algorithm that uses pointers can significantly reduce this overhead. Another observation is that the number of frequent items within a branch of a Trie-Tree is drastically smaller then the number of frequent items within the entire Trie-Tree. As such, copying a branch to a new Trie-Tree with infrequent items removed has a small overhead and a large benefit.

An improved tree algorithm was developed by Han (et. all.) the year after Tree Projection was introduced. It was called FP-Growth (see below), and it worked by combining a Trie-Tree of records with an array of linked lists. Contrary to Tree Projection, FP-Growth starts with a complete Trie-Tree structure from the strings in the dataset. However, each node of the Trie-Tree has an additional pointer. The additional pointer is used to link records together. For example, all nodes of the Trie-Tree containing "A" would have an additional pointer linking them together, as such, the pointers create a linked list of records containing the letter "A". Similar lists are created for items "B", "C", etc. The headers for all these lists are stored in an array. Combined, this data structure is called an "FP-Tree". As the algorithm progresses, it creates and destroys additional FP-Trees from subsets of the data. The algorithm works as follows:

Procedure FP-Growth

1) IF this is the first scan of the dataset,
   THEN Scan through the dataset and count the frequency of each item
2) Create a linked list for each frequent item
   The array of linked lists will be called the "header table"
3) Rescan the dataset. For each record scanned, do the following:
   Procedure Add_Records_To_Tree:
   a) Filter infrequent items from the record
   b) Order the items in the record
   c) Add the record to the tree (See figure 3.2 for an example tree)

4) Add each node of the tree to one of the link list created in step 2
5) Create a subset of records by finding all records that contain the last item (ie: "F" in Figure 3). The last "link list" of the header table (the one for "F") points to the leaf nodes of these records
6) Scan up the subset of records to count the frequency of items within the subset
   Note: The list of frequent items in this subset represent patterns that begin with "F*"
7) Recurs the Procedure FP-Growth using the subset of records
   Note: The FP-Tree constructed from the filtered subset of records is called a *conditional fp-tree*
8) Repeat step 5-7 for each item of the header table (ie: E, D, C, B, A)

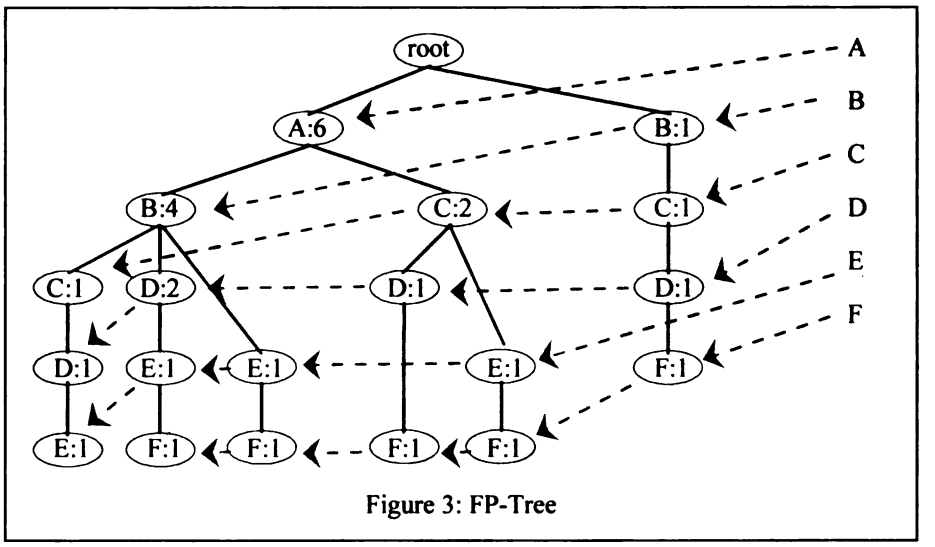


Figure 3: FP-Tree

In some cases, the most significant benefit of the FP-Growth approach is that longer patterns tend to produce conditional fp-trees with only one branch. For example, the removal of infrequent items might produce the three strings "BDF", "BDF", and "BDF". This would create a conditional Trie-Tree with one branch. As such, a very efficient method of producing all combinations of "BDF" can be used to generate patterns. Typically, the FP-Growth algorithm will re-order the strings of a subset from "most-frequent" to "least-frequent" to help promote this effect.

The algorithms presented here represent only a fraction of the many algorithms available. Frequent pattern mining is a very active area of research, and there are many, many more, certainly more than can be presented here. A large percentage of the algorithms are based upon the two approaches "Apriori" and "FP-Growth". An even larger percentage compare themselves against these two approaches. Many approaches are designed for specialized datasets such as *continuous data streams*, very dense datasets, etc. There is also a lot of research in special kinds of patterns such as *closed itemsets* and *constraint based rules* which are designed to reduce the number of frequent patterns returned by a *frequent pattern mining program*. However, the remainder of the paper will work on new approaches to optimize frequent pattern mining as well as exploring a new domain for the task.

PART TWO

OPTIMIZING MINING

CHAPTER 4: Using Compressed Tries in Frequent Pattern Mining

4.1 Introduction

The uses of frequent pattern mining is expanding, and with it, the size of datasets that are being analyzed are also increasing. Since a significant portion of the most commonly used frequent pattern mining algorithms use deviations of the Trie-Tree data structure, most programs need significantly more RAM than the original dataset. It is now to the point where even relatively small datasets can exceed the capacity of existing equipment. When this happens, the processing time it takes to mine data increases explosively. Although there are methods of sub-dividing the dataset to prevent this from happening (see "An efficient algorithm for mining association rules in large databases"), these methods require analyzing the data multiple times and thus require a great deal of time.

This section introduces a simple approach for significantly reducing the memory requirements during mining. This approach works well for a broad range of data and a large genre of frequent pattern mining algorithms. It literally reduces the memory requirements by an order of magnitude, and it does so with no discernable increase on processing time.

The approach works by modifying the Trie-Tree data structure. The Trie-Tree data structure can be used to create very time efficient mining algorithms. Because of this, there are even implementations of the Apriori algorithm that use Trie-Tree like structures to store data. However, like most trees, Trie-Trees require a great deal of overhead. One of the fastest Frequent pattern mining algorithms, FP-Growth, adds additional data structure complexity (ie: doubly linked nodes, usage counts, etc.) that increase this

requirement even further. For small datasets, this does not have a significant impact, however, for larger datasets, space requirements could be many times that of the original dataset.

Statistics can reveal quantitative results to seemingly random events. Analysis of the seemingly random distribution of data within a Trie-Tree used for frequent pattern mining shows a large number of *tails*. A tail is a branch of a Trie-Tree that is only used by one record, that is, each node has no siblings and all but the leaf node has one child. Knowing this pattern is prevalent, a *compressed trie*[15] is an obvious solution to significantly reducing space complexity. However, compressed tries do not lend them selves to the substring searches required for frequent pattern mining. This chapter introduces an algorithm to solve this challenge that not only has negligible overhead, but is occasionally slightly faster.

The organization of the rest of the paper is as follows: Sections 4.2-4.3 discusses Trie-Trees in frequent pattern mining. Section 4.4 and 4.5 gives a mathematical model for why tails exist. Next, section 4.6 presents an algorithm for efficiently storing and accessing data within tails, and section 4.7 gives experimental results. The last section presents conclusions and introduces future work.

4.2 Compressing Trie-Tree Tails

There are many deviations and customizations for building a Trie-Tree for frequent pattern mining. For efficiency, most mining algorithms using Trie-Trees will sort the items within a record. This usually increases the number of records with common

[15] Compressed tries is a method of storing a Trie-Tree that takes up less space. See section 4.4 for details.

prefixes, so it reduces the number of nodes within the tree, and lends itself to a methodical method of getting strings into and out of the tree. For simplicity, this chapter will sort the items within a record in a lexographical order (see Table 1).

Figure 4 is the Trie-Tree created from the records in Table 1. Even with just seven records, you can start to see some usage patterns. One obvious pattern is that nodes near the root of the tree have higher usage counts than nodes near the leaf nodes of the tree. A less obvious pattern is that it is not just leaf nodes that have a usage count of 1. Indeed, of the 20 nodes in the tree, 15 of them have a usage count of 1. Further, of the 7 records in the Tree, all of them have tails. For this paper, a tail is:

> A tail is a branch of a Trie-Tree that is only used by one record, that is, each node has no siblings and all but the leaf node has one child.

Record#	RECORD
1	ACDF
2	ACEF
3	ABDEF
4	ABEF
5	ABCDE
6	ABD
7	BCDF

Table 1: Sorted Items

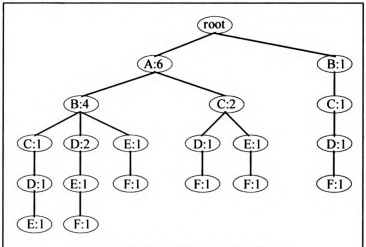


Figure 4: Sorted Items in a Trie

By this definition, node "B:1" is not part of a tail because it has sibling "A:6". However, the nodes below "B:1" do create the tail "C:1-D:1-F:1".

Trie-trees with a large number of tails compress very well. Trie-tree compression is a process of reducing the number of nodes in the tree. Although there are several methods to do compression[16], for this application, *level compression* alone will be very effective in reducing the number of nodes in the Trie-tree. In this case, level compression is performed by taking all of the child nodes of a tail and storing them as one string. For example, the tail "B:1-C:1-D:1-F:1" of Figure 14 can be stored as the single node B:1 with the string CDF attached. Note that such nodes will always have the following three characteristics: (1) they will always be leaf nodes; (2) they will not have siblings; and (3) they will only be used by one record. As such, compressing the Trie-tree of Figure 14 will produce a Compressed Trie-Tree that looks like Figure 5.

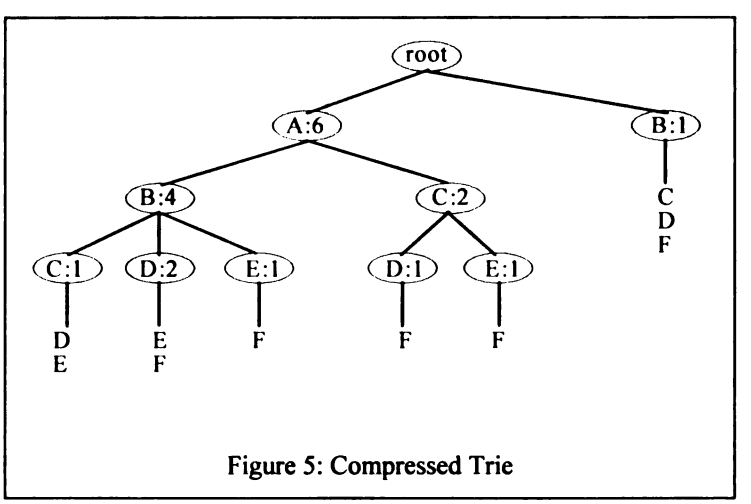


Figure 5: Compressed Trie

4.3 Mathematical Models

Tails are not just an anomaly, but a statistical probability. That is, the patterns seen in Figure 4 are common, and they are common for a reason. Section 4.3 and 4.4 present that reason using statistical analysis. However, before that can be done, you first need to see how the principles of Mutual Independence can be applied to Trie-Trees.

[16] Patricia Trees are a method of doing "path compression", but, that method will not be used here.

4.3.1 Mutual Independence

Back in 1993, Agrawal et-al presented frequent pattern mining and some key mathematics used in the algorithm. The foundation behind the mathematics was the prediction of frequent patterns. That is, given the frequency of short patterns, it predicted the expected frequency of longer patterns. It did so by assuming a statistical property called "Mutual Independence". Mutual Independence basically says that if two (or more) events are independent of each other, then the probability that events will occur together can be calculated. That is, if event A and event B occur with a frequency of f(A) and f(B), then the frequency of both A and B occurring at the same time is f(A+B)=f(A)f(B). This implies that if event A happens frequently and event B happens frequently, then A+B is likely to occur often.

Agrawal at-al presented mutual independence as follows: (see Figure 6)

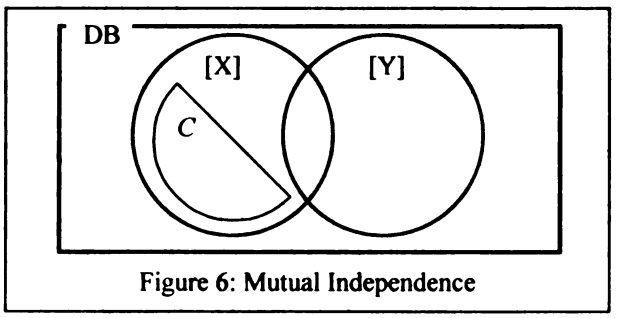


Figure 6: Mutual Independence

Given:

X is a pattern $x_1\ x_2\ x_3 \ldots x_n$

Y is a pattern such that $X \cap Y = \{\}$

Then:

S_{XY} is the frequency of the pattern X+Y

The formula for S_{XY} is:

F1) $S_{XY} = f(y_1)\ f(y_2) \ldots f(y_k)\ (x-c) / dbsize$

F2) $f(y_i)$ = support of item y_i within dataset DB

Where $y_i \in Y$

F3) x = the # of occurrences of X within DB

For this paper, [X] is a set of records containing X

If X = {}, then x = |DB| = dbsize = R

F4) c= | C | where C is a subset of [X] – [X+Y]

Note: (x-c)= Size of remaining portion of [X]

This is treating the occurrence of items within a string as events. The dataset DB represents R independent trials where each string is a combination of simultaneous events. The set [X] represents a set of strings that contain the simultaneous events X. [X] is subdivided into mutually exclusive subsets. Given the set [X+Y] is such a subset of [X], and knowing the support of X, S_{XY} predicts the support of the events X+Y. As each subset is processed, it is added to the set C (ie: C' = C + [X+Y]).

4.3.2 Mutual Independence for Trie-Trees

This same analogy can be applied to a Trie-Tree. Given a node X_n in a Trie-Tree (See Figure 7), then the path from the root to X_n represents the pattern $X = x_1 x_2 \ldots x_n$, and the strings that use node X_n would represent the set [X]. The child nodes of X_n represent mutually exclusive subsets of. [X]. Given the child nodes of X_n are $Y_1 Y_2 \ldots Y_k$, then Y can be represented as the single item y_j, and C would be represented as the subset of strings that use nodes $Y_1 \ldots Y_{j-1}$ As such, $S_{XY} = f(y_j)\ (x–c)/dbsize$, where x = |[X]| = the number of strings using x, and c = |C| = the number of strings using nodes Y_1 through Y_{j-1}.

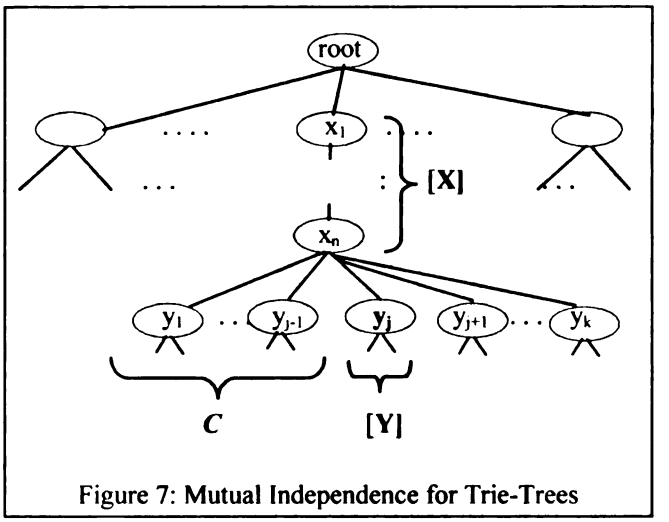


Figure 7: Mutual Independence for Trie-Trees

Figure 4 can be used to demonstrate how the analogy can be applied. Let node A:6 represent the pattern X, and item "B" be pattern Y to add to X. As a result, the pattern X+Y is represented by node B:4. The S_{XY} predicts how many records should use node B:4. So:

X= "A"

Y= "B"

dbsize = R = 7 records

f(Y) = f("B") / R = 5 / 7 = 0.71

x = 6

c = 0

S_{XY} = f(Y) (x – c) / R

= 0.71 (6 – 0) / 7

= 0.61 ≈ f(X+Y)

So, the predicted support of B:4 is S_{XY}=0.61, and the actual support is 0.57 (= f("AB") = 4 / 7). As in 1993, a model of Mutual Independence produces a fairly good estimate.

4.3.3 The Prediction of Trie-Tree Tails

So far, what has been presented is only a different perspective of Agrawal's formulas. However, the formula for S_{XY} can be generalized for all children of node X. For Trie-Trees, it is given Y will always represent one item from the set $i_{t+1}, i_{t+2}, \ldots, i_m$. As such, the support of the i_k child of X will be given by:

F5) $S_{XY} = f(i_k) (1 - \sum_{j = t+1 \ldots k-1} f(i_j)) x / R$

Further, the support for the i_p child of Y (a grand child of X) can be calculated as follows:

Given:

$Z = i_p$

$y = R S_{XY}$ = the size of [X + Y]

Then:

F6) $S_{XYZ} = f(i_p) (1 - \sum_{j = k+1 \ldots p-1} f(i_j)) y / R$

F7) $S_{XYZ} = f(i_p) (1 - \sum_{j = k+1 \ldots p-1} f(i_j))$
$f(i_k) (1 - \sum_{j = t+1 \ldots k-1} f(i_j)) x / R$

F8) $S_{XYZ} \leq f(i_p) f(i_k) x / R$

For the root, x=R, so the support of any node path $X = x_1 \ldots x_n$ will be bounded by:

F9) $S_X \leq \prod_{j = 1 \ldots n} f(x_j)$

As a result, the number of records that are expected to use a node X can be predicted with $R S_X$. If this is less than or equal to 1, then the node is predicted to be part of a tail. Assuming all items have equal support, the depth of non-tail nodes can be estimated with:

Given:

$f(x_1) = f(x_2) = \ldots = f(x_n) \leq 1 / n$ (where n = number of items in the dataset)

d is the depth of a node used by multiple records

R is the number of records in the dataset

Then:

F10) $1 < R\, S_X = \Pi_{j=1\ldots d}\ f(x_j)$

F11) $1 < R\,(1/n)^d$

F12) $0 < \ln(R) - d\ln(n)$

F13) $d < \ln(R) / \ln(n)$

So, for example, if the number of strings in a dataset DB is R=1,000,000, and the number of equally supported items in DB is n=20, then the expected depth where tails are expected to show up is less than d=4.6. In other words, in this example, tails are expected if the average number of frequent items within a string is more than 5.

To give an empirical test of the formula, the predicted depth was compared to the depth of random datasets. In this case, the random data did not have any frequent patterns; instead, each record contains 6 randomly selected items. As a result, the distribution of items was evenly distributed and the length of each record was 6. From each dataset, a Trie-Tree was constructed.

For Figure 8, nine datasets were generated with 100,000 records and the number of items within the dataset varied[17] from 20 to 100. For the datasets, the average depth of the associated Trie-Tree shared nodes was plotted on the graph. This was compared to the predicted average depth based on the formula $d < \ln(R) / \ln(n)$. As can be seen in Figure 8, the predicted depth does a very good job of bounding the average depth of shared nodes. Figure 9 does a similar analysis, only it varies the size of the database and keeps the number of items at 50. It also shows that the formula $d < \ln(R) / \ln(n)$ does well at bounding the expected average depth for random data.

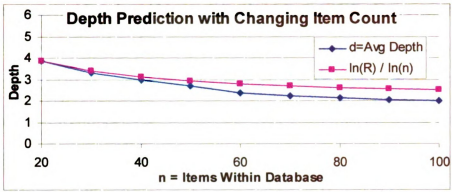


Figure 8: Predicting $d < \ln(R)/\ln(n)$ for Random Data
$R = 100{,}000$ and $n = 20$ thru100

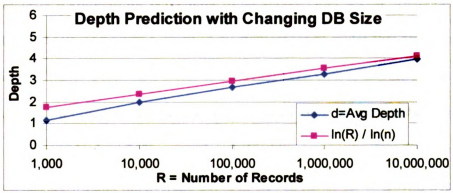


Figure 9: Predicting $d < \ln(R)/\ln(n)$ for Random Data
$n = 50$ and $R = 1{,}000$ thru $10{,}000{,}000$

4.3.4 The Observation of Trie-Tree Tails

The prediction of tails in random data is one thing, but do they occur in real data? After all, the mathematical model makes several simplifying assumptions such as mutual independence and evenly distributed items. To help answer that question, two distinct and diverse datasets were examined.

The two datasets were collected from the FIMI repository. They were chosen because of their diversity. The first dataset is a DB of Belgium Traffic Accident data. At 35 MB,

[17] Since each record was fixed at 6 items long, a minimum of 20 items was needed to create a sufficiently random set of records.

and an average record length of 45 items, it is relatively small DB. However, many of its 572 items are mutually exclusive[18], so it has a dense set of long patterns. Conversely, WebDocs is a 1.5 GB dataset with a relatively sparse set of frequent patterns. Each transaction of the WebDocs represents words of a document from the World Wide Web. As such, it has 1.5 million records, and over 5.3 million unique words. Because of its large size, it has a significant number of frequent patterns. Despite the diversity of the datasets, both have a large percentage of tails.

For both datasets, a graph of the percentage of the records that have at least one frequent item in it (Figure 10), as well as a graph of the percentage of nodes that are in tails (Figure 11) was made. From this, it can be seen that even with relatively high support, more than 90% of all records have at least one frequent item. It can also be seen that even with support as high as 25%, more than half of the nodes of a Trie-Tree are in tails. Observation has shown that the average depth of shared nodes for data with frequent patterns is deeper than random data. Despite this, the graphs show that datasets with infrequent items removed still have a lot of records, and it also shows their associated Trie-Trees have a high percentage of tails. So, there is a large number of tails in real data.

Since tails are so prevalent in the Trie-Trees of real data, what can be done to help? The answer is a more efficient method of handling tails.

[18] An example of mutually exclusive items would be "road condition = dry" and "road condition = wet".

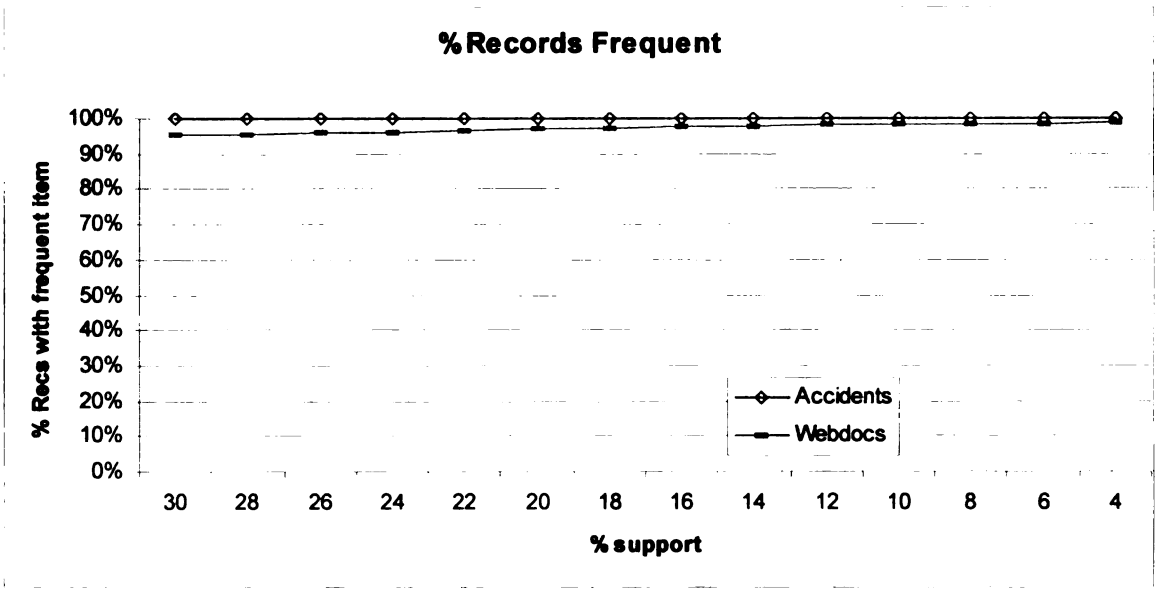


Figure 10: Most Records Have Frequent Items

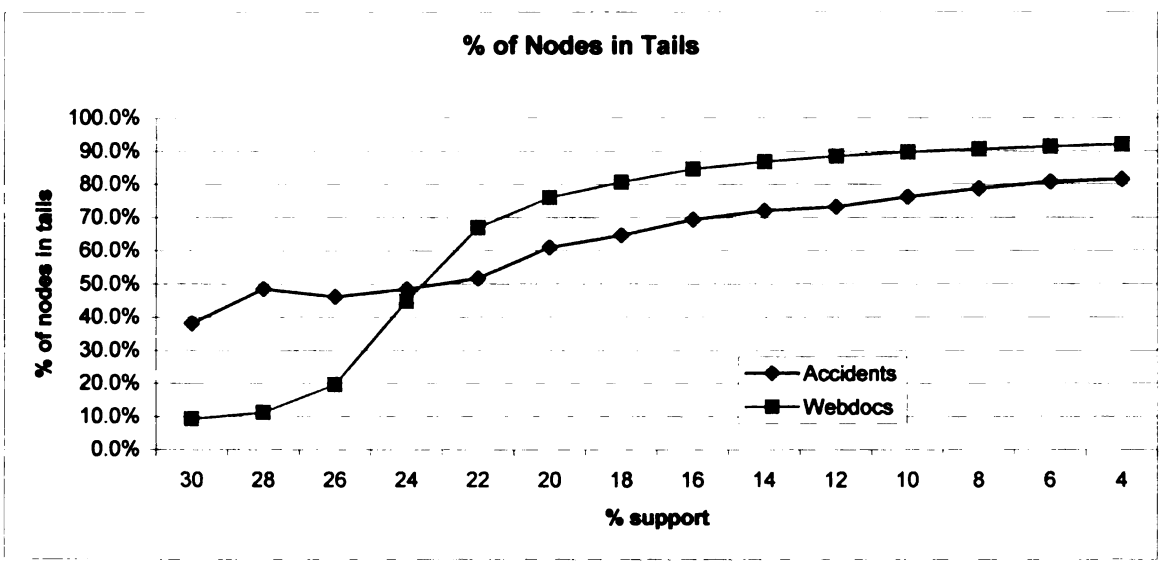


Figure 11: Most Nodes are in Tails

4.3.5 Exploiting Tails for Frequent Pattern Mining

Now that we have explained and justified that tails exist, our challenge is to exploit this characteristic in frequent pattern mining. Since several algorithms of mining use Tri-Tree type structures (see Section 3.3 for examples), they can be made to be more space efficient by using the compressed Trie-Tree structure presented in Section 4.2. However, some Tri-Tree based approaches add additional information and pointers to each node. Most of this can be reduced or eliminated with the realization that tails have no siblings and at most one child. Much of the rest can be eliminated by understanding the methodical nature of mining. The next section will explore how this can be done for the FP-Growth algorithm.

4.4 Implementing Tails

The most efficient method of storing a tail within a Trie-Tree is to store only the items of the tail. This can be easily done, but then the challenge for frequent pattern mining is identifying all strings containing a particular item without scanning the entire tree. This section will demonstrate how this can be done without such a scan, and without having a separate pointer for each item of the string. To understand better, it helps to explain how this has been done in the past.

4.4.1 FP-Growth's Method of DB Scans

One of the core processes of FP-Growth is to scan all of the branches of its tree that contain a given item. It can quickly find all of these branches because it links the items of its branches when it builds its tree (see figure 12). With all of the links in place, FP-Growth can find all branches containing a particular item. For example, when FP-Growth wants to find all strings that contain "D", it starts with the D node for the string

"BCDF". This node points to the D node for the string "ACDF". The D node of "ACDF" points to the D node of "ABDEF". The last D node pointed to is for "ABCDE". By doing so, FP-Growth can methodically search for patterns that contain "D". FP-Growth will repeat this process for all items. A through F.

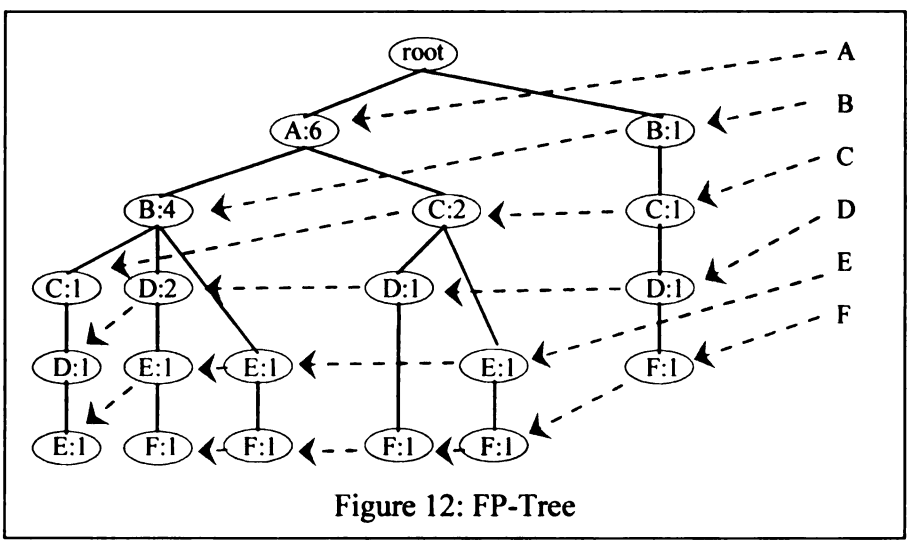


Figure 12: FP-Tree

4.4.2 Modify Scans to Scan Tails

The FP-Growth algorithm can be modified to scan tails by observing that the algorithm is methodical in its scans. That is, it will look at all strings containing "A" before it looks at strings containing "B"[19]. When it starts looking at the strings for "B", it will no longer need to link "A" strings. As a result, each tail needs to only have one link.

For example, Figure 13 shows the initial setup of the link lists. Notice that this is a kind of compressed Trie-Tree (ie: the compressed tail under B:1 is "CDF"). Notice also that the "CDF" tail is linked to the list of "C" strings, but not to the list of "D" or "F" strings.

As part of scanning the list of "C" strings, the tail "CDF" will be scanned. After the algorithm is done scanning the tail "CDF", "CDF" is linked to the list of "D" strings (see Figure 14). As such, the tail only needs one pointer for all of its items. This algorithm is outlined in Figure 15 as well as the algorithm for adding strings to the tree.

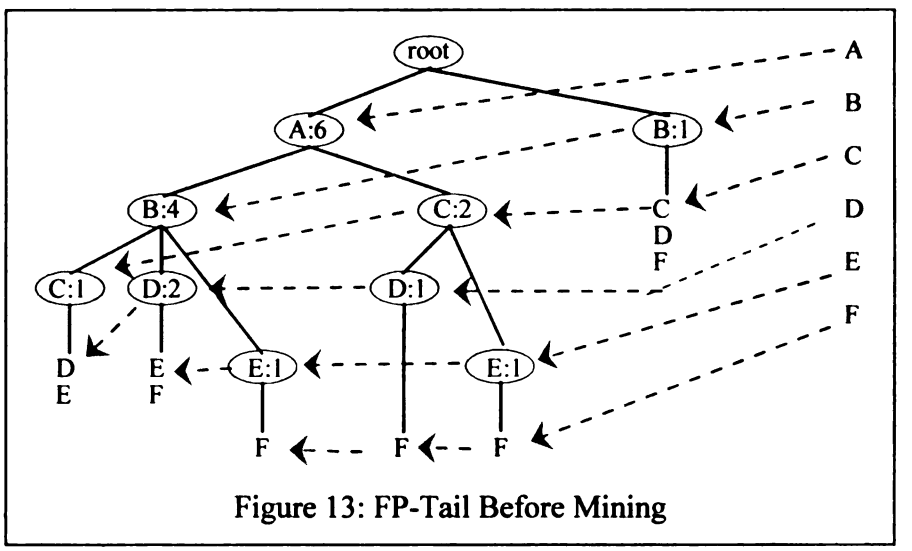


Figure 13: FP-Tail Before Mining

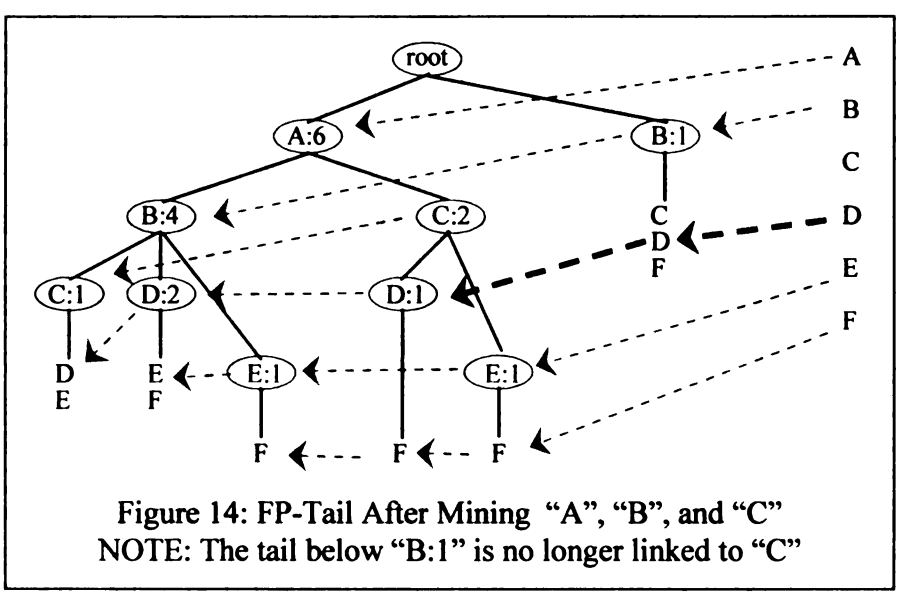


Figure 14: FP-Tail After Mining "A", "B", and "C"
NOTE: The tail below "B:1" is no longer linked to "C"

[19] The original FP-Growth algorithm was implemented with a bottom-up approach, however, there have been top-down approaches and the procedure presented here can be adapted for either top-down or bottom-up.

```
Procedure Add_String_To_Tree(String S, Tree T)
  Node = root of T
  For i = each item of S
   If Node has a tail
    make 1st item of tail a node
   end if
   If Node has children
     Merge i as a child of Node
     Node = i child of Node
   Else
     Tail = remainder of string S
     attach Tail to Node
     exit For loop
   end
  next i


Procedure Process_Tree(Tree T)
  For i = each item of I
   For s = each object of LinkListi
    Process_object(s)
    If (s is a tail)
     j= the item after i in string s
     add s to LinkListj
    end if
   next s
  next i
```

Figure 15: Tail Procedures

4.5 Experimental Evaluation

Since tails compose an increasingly greater percentage of a tree as the strings get longer, the advantage of compressing the tails becomes increasingly more significant. In the following experiments, the dense "accidents" and sparse "webdocs" datasets used from section 4.4 were analyzed using two versions of FP-Growth. The FP-Growth algorithm was used to demonstrate results because of its prevalence, however, other Trie-Tree based approaches should work equally well. The comparison of the results will help to demonstrate the algorithm is applicable for a wide range of datasets.

4.5.1 Experimental Setup

The experiments on the performance comparison between FP-tail and FP-growth were conducted on an AMD Atholon 2800+ PC with 1.2 GB RAM, running RedHat Linux 8.0. The programs were written in C++ and compiled with gcc using the "-O3" compiler optimization.

4.5.2 Experimental Results

Figure 16 is a graph of memory usage for analyzing Accidents and Webdocs data using the FP-Growth program and the FP-Tail program. The bottom two lines (AccFP and AccTail) represent the memory usage for analyzing the 35 MB Accidents data file, and the top two lines represent the memory usage for analyzing the 1.5 GB of WebDocs data (DocFP and DocTail). As the "% Support" goes down, memory usage goes up. This is expected since items with smaller support will need to be included in the analysis. What might be less intuitive is the rate of growth for RAM as support decreases.

Since Figure 25 is Log10, it can be seen decreasing support causes a geometric growth in RAM. Although this growth is bounded, it only reaches this bound when "% Support" reaches 0%. As a result, efficiently handling tails is not just a linear savings in space, but a geometric one. Since the nature of this geometric growth is beyond the scope if this chapter, Appendix A will expound upon its implications.

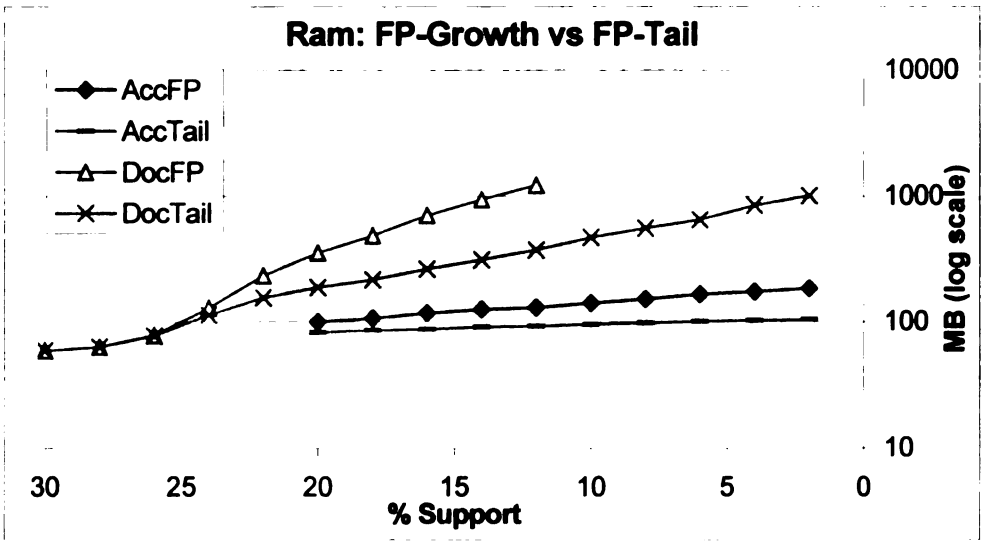


Figure 16: RAM for FP-Growth vs FP-Tail

Achieving this geometric savings in space can be achieved with no loss in performance. Figure 17 shows the amount of time it took to process each data file. Notice that the processing times for the accident data for both FP-Growth and FP-Tail (the bottom lines) are overlapping. That is, the amount of time it takes to implement the tails algorithm for the accidents data is negligible. Indeed, it produces a small time-savings because of the more efficient memory handling. The time savings for the analysis of the WebDocs data also overlaps until it reaches 12% support. At that support, using the standard FP-Growth algorithm takes over 5 hours instead of only 240 seconds. The reason this happens is because the algorithm has exceeded physical memory by less than 5%. Since frequent pattern mining requires analysis of the entire dataset on each pass, exceeding physical RAM carries a very heavy price. This helps emphasize that saving memory can directly lead to a significant savings in time.

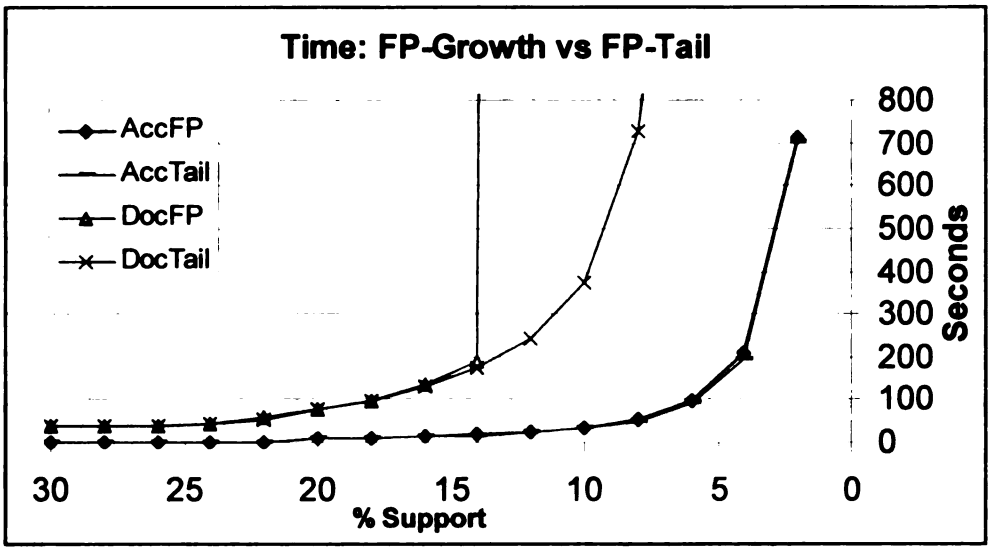


Figure 17: Time for FP-Growth vs FP-Tail

4.6 *Conclusions of Compressed Trie Mining*

Using compressed Trie-Trees for frequent pattern mining has proven extremely effective in saving space. The space savings is not just linear, but geometric; as such, it has changed the order of complexity of space requirements for mining. Further, this savings is not only effective for diverse datasets, but it becomes more effective as space demands increase. That is, the algorithm works best when it is needed the most, and it does not appear to have any deviant case with a noticeable cost in space and/or time.

Using this approach allows frequent pattern mining algorithms to be used in at least two new ways. First, with reduced space requirements, it becomes practical to analyze larger datasets. Although time will increase the capacity of systems, it will also increase the size of datasets, and this is a very effective approach to help bridge that gap. For example, 1.5 GB is not very large, but it is by far the largest dataset offered by the FIMI repository, and it is larger than a standard PC can mine using conventional methods. The

second way reduced memory can help is that it allows mining to enhance a system instead of overwhelming it. For example, pattern mining can be used to enhance hard drive usage, network security, or word processing preferences; and it can do so without having to size a PC to meet the high resource demands of frequent pattern mining. This point will be demonstrated in the next chapter with the development of an application for frequent pattern mining involving the enhancement of PC Internet security.

As a side note, this chapter also introduced several interesting observations from its experimental results. These observations can be seen in Figures 10, 11, and 16. Together, they help lead to a method of predicting the resources needed for mining. However, predicting resources is not the same as optimizing the use of resources. As such, this is out of the scope of this thesis, and the analysis of these observations has been added as appendix A.

CHAPTER 5: Enhancing PC Internet Security with Pattern Mining

5.1 Introduction

The previous chapter introduced a mechanism for significantly reducing the memory resources required for frequent pattern mining. As mentioned, this can be used to either mine bigger datasets, or to mine the same size datasets in a smaller environment. For example, mining is not just practical on a personal computer, but it is now no longer necessary to size your system to do some forms of mining. That is, mining can be implemented to improve everyday tasks without having to buy extra RAM to do so. But what task can be improved by mining? One of the contemporary problems plaguing personal computing today is Internet security.

In this case, we are talking about the long term leaching of information out of and/or into your computer. That is, network applications and applications enhanced with network connectivity are increasing the network traffic for personal computers to the point where it is getting more difficult to determine the network activity of your own system. Worms, viruses, spyware, and even helpful applications will regularly send undesirable information from your computer, or get unwanted data from the Internet. However, considering the volume of information involved, it would be impractical to monitor all of this traffic to determine what is, and is not, intended and necessary traffic. Frequent pattern mining can be used to analyze and classify this data to a manageable size. An application that finds patterns in Internet traffic can be used to enhance the effectiveness of PC Internet security programs. This chapter introduces the development of a new application called NetAppMine that does just that.

To understand why most systems need an application like NetAppMine, it is important to realize that almost every home/office computer today is connected to the Internet at least intermittently. Frequently, applications on your system will automatically check when you have connected to the Internet and silently transfer data. However, identifying the applications transferring this data can be challenging, and determining if the data transferred is *good* or *bad* is next to impossible, but NetAppMine can help.

There are several types of applications on your PC that might transfer bad data. For example, Adware will intermittently display advertisement on your screen[20]. Another more subtle example is Spyware. Spyware will discreetly collect information about you and your habits[21] for targeting purposes. Worse yet, Malware, Viruses, and Dialers can take control of your computer and do any number of malicious acts without your knowledge. Even purchased software will frequently have Spyware/Adware functions built-in.

This does not mean all programs discretely connecting to the Internet are bad, indeed, some are helpful. A program can remove bugs via auto-update. Well targeted marketing can help you find products your interested in. A company that knows more about you can serve you better. The trick is to identify and control the data your sending and getting.

Several applications have been developed to help control the flow of Internet data from/to your PC. These include Pop Up Blockers, Spyware/Virus Detection, and Internet security programs. The problem with these solutions is that they only identify known

[20] Frequently, Adware will display ads on how to spend money you don't have.
[21] A common use of Spyware is to collect E-mail addresses so "spammers" can send you ads on how to enlarge or shrink something.

problems or have broad/stateless solutions. For instance, if an Internet security program lets a program access the Internet, the program can do anything it wants without further checks, including IP broadcasts in the middle of the night.

One solution is to add *fuzzy logic* to Internet security programs. That is, look for patterns in the data flow of information. Since frequent pattern mining is the process of looking for patterns in data, it is ideally suited for this task. The scope of NetAppMine is to find such patterns between network applications and network data. This information can be examined or post-processed to help identify problems.

The organization of the rest of this chapter is as follows: Section 5.2 presents previous work in Internet security and describes the unique niche NetAppMine will fill. Section 5.3 explains the methods used to collect and pre-process the data to make mining as effective and efficient as possible. Next, section 5.4 shows the results of analyzing the data from three environments to demonstrate the effectiveness of NetAppMine. This is followed by section 5.5, which demonstrates the efficiency NetAppMine. The last section presents conclusions and introduces future work.

5.2 Previous Work

Programs with dubious Internet communications have been around for many years. By the year 2000, Spyware was so prevalent that Gibson Research released a program dedicated to the removal of Spyware. Kazaa has been bundled with Spyware since at least Dec 2001 (see Table 2). SpywareGuide currently has 2041 Spyware programs listed, and this list is growing fast. A recent study found over 5% of the computers at the University of Washington were infected with Spyware, some of them for over 4 years.

Network viruses and worms have been around even longer than that. But what can be done?

Fundamental changes in the Internet, and how it is used, would be needed to preserve privacy. IPv6 would go a long way to helping out. This would allow for better packet route tracing and greatly reduce spoofing. In addition, IPv6 would allow the elimination of NAT and the addition of Kerberos authentication as needed. However, IPv6 would not be enough. Some organization would need to oversee network applications and certify Java Scripts, ActiveX applets, etc. to ensure their compliance with privacy standards. However, these are extreme measures that are not likely to happen soon, if at all.

Version	1.3.3	1.4	1.5	1.6	1.7	2.0	2.1	2.1.1	2.6
released	12/01	01/02	02/02	04/02	05/02	09/02	02/03	05/03	11/03
Gator									X
SaveNow	X	X	X	X	X	X	X	X	
Cydoor	X	X	X	X	X	X			X
BDE	X	X	X	X	X	X			
VX2	X	X							
New.net	X	X	X	X	X	X			
OnFlow	X	X						X	
D/L-Ware					X	X	X		
CmnName	X	X	X	X	X	X			X
PromulGate						X			
DirecTVIcon			X	X					
MySearch									X

Table 2: Spyware Bundled with Kazaa
Twelve different programs that were bundled with Kazaa at various points in time.

A more practical approach is for the individual to monitor their own Internet data. One way to do this monitoring is to have software tools that can find patterns in network traffic. IntelliGuard IT has just such a tool. It uses frequent pattern mining in their

architecture to create a self-learning firewall for an enterprise environment. It is this kind of process that NetAppMine will be using for enhancing the Internet security for the PC environment.

Mining can be used to find frequent patterns in network data.. That is, finding two or more network events that frequently occur together. For example, a DNS lookup of MSU.EDU may frequently occur with a PING of MSU.EDU and an FTP to MSU.EDU. This would result in a frequent pattern of (DNS, FTP, PING). Additional frequent patterns might be (DNS, FTP) or (FTP, PING). In fact, there are 7 possible combinations for the three items DNS, FTP, & PING. With 9 items, there would be 511 ($=2^9-1$) combinations. Finding all of the frequent patterns and filtering out the interesting ones is one of the challenges this research will tackle. After all, even with filtering, there is still lot of patterns to look at.

5.3 Methodology

The last section emphasized NetAppMine is a real application filling a real niche, this section will demonstrate that every effort was done to make NetAppMine as effective and efficient as possible. It will do so by describing the methods used to collect and pre-process data, as well as describing the kinds of results expected from the frequent pattern mining analysis. As will be seen in the experimental results (see Section 5.5), the optimization presented in Chapter 4 is an integral part of making NetAppMine practical due to the high volume of data.

5.3.1 Collecting Network and Application Data

The program collects and time stamps information from each network packet sent. The information that NetAppMine logs is Source IP, Source Port, Destination IP, Destination Port, and the protocol (ie: tcp, udp, icmp, or other). In addition, as packets of data are collected, the program also collects a snapshot of applications with active network connections. This is done up to four times a second. Lastly, all transactions are given a timestamp in milliseconds. However, without processing, the resulting logs quickly grow into many gigabytes of data.

5.3.2 Processing the Logs

Processing the logs prepares and reduces the amount of data that needs to be analyzed. Processing will include the filtering out of extraneous IPs, combining data to create items, and grouping items to create records. The objective here is to balance between having enough data to produce meaningful and useful patterns, while not having so much data that the time and space complexity needed for analysis is too costly to be practical.

Filtering extraneous data is the safest method of reducing the amount of data that needs to be analyzed. For example, the data was collected in promiscuous mode, so network traffic between other systems can be filtered out. This is done by removing any transactions that does not contain the IP address of the local machine, 127.0.0.1, or 0.0.0.0. Some space compression is also done at this stage by encoding the data. That is, IP addresses, application names, etc. are encoded as a single unique numbers. However, this still leaves tremendous amounts of data to be analyzed.

Combining data to make items is a key component to reducing the time complexity of analyzing the data. For example, one of the smaller experiments presented in Section 5.4 had 120 IPs, 700 Ports, 15 Applications, and 4 Protocols (ie: tcp, udp, icmp, and other).

Because of combinatorial explosion problem, this would mean 420 billion combinations to look for with thousands of them being frequent. To reduce this complexity, each transaction is converted to three items. These items are a combination of the following fields:

1) Application, Domain
2) Application, IP address
3) Application, IP address, Protocol, Destination Port, Send? [22]

Where "Domain" is the first two octets of the IP address.

Where "Send?" is FALSE if the source IP is a remote address.

As a result, the experiment with 420 billion combinations is reduced to less then 22 million; with less than 100 of them being frequent.

For example, event #1, event #2, and event#3 from Table 3 are all from a single transaction. As such, this transaction only creates three items. Otherwise, this same transaction would create the 6 items: "IE", "35.9", "35.9.20.20", "TCP", "Port=80", and "Send=True". However, with these definitions of items, we can still capture an application that frequently connects to the "35.9" domain even if it continually changes ports and/or IP addresses it is connecting to.

The last consolidation done by NetAppMine before analysis is the identification of records. In this case, a record is a collection of items. To reduce the number of records, the definition of an item is further refined to be an event. An event is the starting or ending of a virtual connection. For example, if IE starts sending data to 35.9.20.20 at 8:00 am, and continuously sends IP packets to port 80 until 9:00 am, that would generate

[22] Items of type three are similar to a *virtual circuit*. That is, a virtual connection between two applications that are communicating over the network.

6 events (see Table 3). Events that occur within the same time slice are grouped together. For these experiments, a time slice was defined as 10 seconds. As such, if the application PING and FTP frequently run within 10 seconds of each other, a frequent pattern will be identified that associates the running of PING with the running of FTP. It is this data that NetAppMine analyzes for frequent patterns, and these kinds of patterns that NetAppMine will find.

Event #1	08:00:00	(IE, 35.9)
Event #2	08:00:00	(IE, 35.9.20.20)
Event #3	08:00:00	(IE, 35.9.20.20, tcp, 80, True)
Event #4	09:00:00	(IE, 35.9)
Event #5	09:00:00	(IE, 35.9.20.20)
Event #6	09:00:00	(IE, 35.9.20.20, tcp, 80, True)
Record#1	08:00:00	(IE, 35.9) (IE, 35.9.20.20) (IE, 35.9.20.20, tcp, 80, True)
Record#2	09:00:00	(IE, 35.9) (IE, 35.9.20.20) (IE, 35.9.20.20, tcp, 80, True)

Table 3: Events and Records
This table shows 6 events and the two records created by the events.

5.3.3 Analyzing the Data

Frequent pattern mining can be done using several methods that produce several kinds of results. In this case, the algorithm was modified to reduce the number of patterns found without eliminating interesting patterns.

A pattern is a combination of items that occur together frequently. For example, (IE, 35.9) and (IE, 35.9.20.20) could occur together in 150 records. As a result, the pattern [(IE, 35.9), (IE, 35.9.20.20)] has a frequency of 150. Such a pattern would indicate that

a lot of the communication to the Domain "35.9" is to the same IP Address. For these experiments, a pattern needs to occur in at least 20% of the records to be considered a frequent pattern.

To reduce the number of patterns printed, usually only "maximal patterns" are printed. For example, if [(IE, 35.9), (IE, 35.9.20.20)] has a frequency of 150, then [(IE. 35.9)] and [(IE,35.9.20.20)] must have a frequency of at least 150. However, neither of these patterns will be printed unless one of them has a frequency significantly greater than the frequency of [(IE, 35.9), (IE, 35.9.20.20)].

In addition, to emphasize the role each application plays in network communications, mining was done on each application. For example, network communication by EXPLORE.EXE may indicate file sharing over the network. As a result, all records that reference EXPLORE.EXE are extracted from the logs and the dataset is mined separately. This is done for each application that connects to the network.

If a dataset only has a few records, it is not mined. If there are only 5 records in a dataset, then every record will be considered frequent. If two records contain several items that are the same, then hundreds of patterns will be generated from just a few records. As a result, any application dataset that contains 15 or fewer records is not mined. Instead, the high port (1000 or greater) items are removed from the records and each record is printed as a pattern.

5.4 Experiments

5.4.1 Experimental Environment

The network traffic of three computers were analyzed. All three environments were home PCs running different versions of Microsoft Windows. Each of the computers had a 2+ Mb/s Internet connection. Each of the computers had network applications other than browsers and were monitored during both active and idle times for at least 12 hours. The names of the three computers were TEST, WORK, and MEDIA.

The TEST computer was the computer that NetAppMine was developed and tested. Its network traffic was collected several times over a two week period. Some collections were under 10 minutes and others were over 12 hours. The Test PC was running Windows 2000 and was connected to the Internet via Cable through a D-Link Firewall Router. Some of the collection was made with Norton Internet Security installed, and some was not. In addition, the Spyware programs "iGetNet" and "BonziBuddy" were intentionally installed.

The Work computer's primary function was to connect to a remote office. Since it was a relatively new computer, it did not have many programs installed on it. It was running Windows XP and was connected to the Internet via a wireless hub. Data collection was done over a single 24 hour period.

The MEDIA computer's primary function was for entertainment. It had many applications installed on it, including two P2P applications (ie: Kazaa). It was running Windows XP and was connected to the Internet via Cable through a router. Data collection was done over a single 12 hour period.

The collection of network and application data was done with a program that uses the third party tools Ngsniff and Tcpvcon. These tools were chosen because they do not require any setup or installation. That is, they can be run on any computer.

5.4.2 Displaying Results Using a GUI

Although there may be fewer than a hundred patterns, each pattern might contain 30 or more items. To make it easier to review and recognize interesting patterns, NetAppMine uses a GUI to display the results. Figure 28 (at the end of this chapter) contains a GUI display of the pattern represented by the table below as well as summary columns:

Pattern #2 has a frequency of 6 with 13 simultaneously network events
for the application "C:\Program files\Internet Explorere\IEXPLORE.EXE"
The13 network events include:

Pat Type	Port	App #	IP	Protocol	Port	Direction	Name
1	1	0	-1				None
2	2001	0	-1				None
3	1	0	192.168.0.100	UDP	53	<	None
2	2043	0	216.177				None
1	43	0	216.177.73.139				None
3	43	0	216.177.73.139	TCP	80	<	None
2	2035	10	209.244				SymProxySvc.exe
2	2044	10	216.177				SymProxySvc.exe
1	44	10	216.177.73.139				SymProxySvc.exe
1	110	10	65.54.140.158				SymProxySvc.exe
1	57	12	127.0.0.1				IEXPLORE.EXE
3	57	12	127.0.0.1	TCP	1030	<	IEXPLORE.EXE
2	2008	12	205.161				IEXPLORE.EXE

Table 4: "Events" for a Single Pattern
NOTE: An "Event" does not mean a packet was sent,
but that a network connection was started or stopped

The format of the GUI display makes it easier to see information and patterns that can not be easily seen in the text. The GUI display lists all applications on the right side and all items of a pattern on the left. This makes it easier to see which applications are (and are not) participating in a pattern. Further, the current application dataset is in black and has an "*". This makes it easy to identify which application dataset is being looked at. In

addition, extra information is readily available about the current application dataset as well as other applications. This includes the number of records in other datasets as well as the number of patterns. The top line of the display shows the number of patterns in the dataset as well as the frequency count of the current pattern and the full path to the application. All of this data in Figure 18 makes it easier to see that Internet Explorer is using SymProxy to access the Internet for everything except the Domain 205.151. Further analysis of this pattern will be given in section 5.

5.4.3 Results of Analysis

In all, there were 89 patterns generated among the three computers. Figures 18, 19, and 20 demonstrate the results of the three of those patterns. This help to demonstrate that useful results can be gathered from this kind of analysis.

Figure 18 is a pattern from the analysis of the TEST computer. The TEST computer had a total of 45 patterns generated. The Local IP address of the TEST computer was 192.168.0.100. Figure 18 displays pattern #2 from the analysis of IE. It shows that IE primarily used SymProxy23 to access the Internet. This can be seen from the "127.0.0.1" communication while SymProxy was accessing 216.177.73.139 (IGnet Spyware web site). However, IE did have an exception. It directly accessed several computers in the Domain 205.161. Most likely, this is an indication that the IGnet software was using IE to access the web directly. However, it could also be normal communications by IE for tasks like downloading a program. Further collection of data would help to distinguish if this was an anomaly or just normal behavior.

[23] SymProxy is a Norton proxy program for Internet security.

The analysis of the WORK computer had 23 patterns as well as 27 records from programs that infrequently access the Internet. PokerStarsUpdate.exe is an example of just such a dataset. It helps to show what a pattern looks like for a program that infrequently accesses the Internet. Figure 19 is the first pattern of the PokerStarsUpdate.exe application. Since this program's dataset has fewer than 15 records, each record that contains PokerStarsUpdate.exe generates a pattern with a frequency of 1. This clearly demonstrates that PokerStarsUpdate.exe is accessing the web. In addition, it shows what else is going on while it is doing so. Based on this information, the PokerStarsUpdate.exe program could be blocked from accessing the Internet, or it could be blocked from the 66.212 Domain.

The MEDIA computer has an IP address of 192.168.2.35 and generated over 3,000 records in less then 12 hours. Most of these records were generated while the computer was not being used. During that time, it was generating network traffic at 6 times the rate of the other two computers. Figure 20 shows that a lot of these records were broadcasts. This is a likely indication that there is a misconfigured, poorly designed, or bad application on this computer. Further analysis would be needed to determine which application is causing the problem, but it is most likely the HP program "BackWeb" because of the high volume of network traffic it is generating.

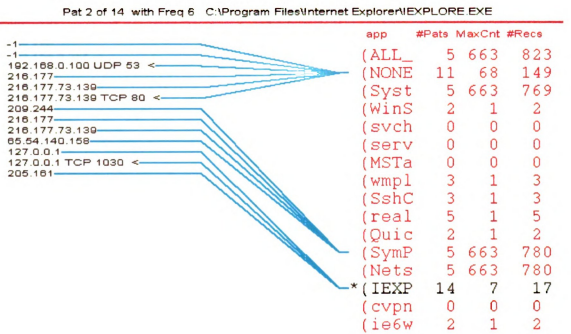


Figure 18: GUI Display of TEST Computer
Internet Explorer is using SymProxy (usually).

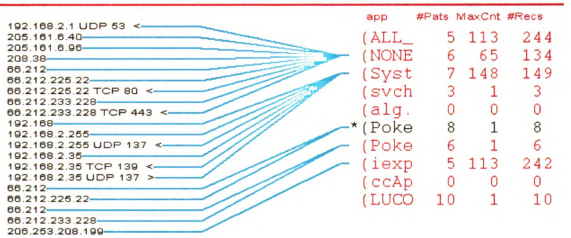


Figure 19: GUI Display of WORK Computer
PokerUpdate has fewer than 15 records in its dataset.

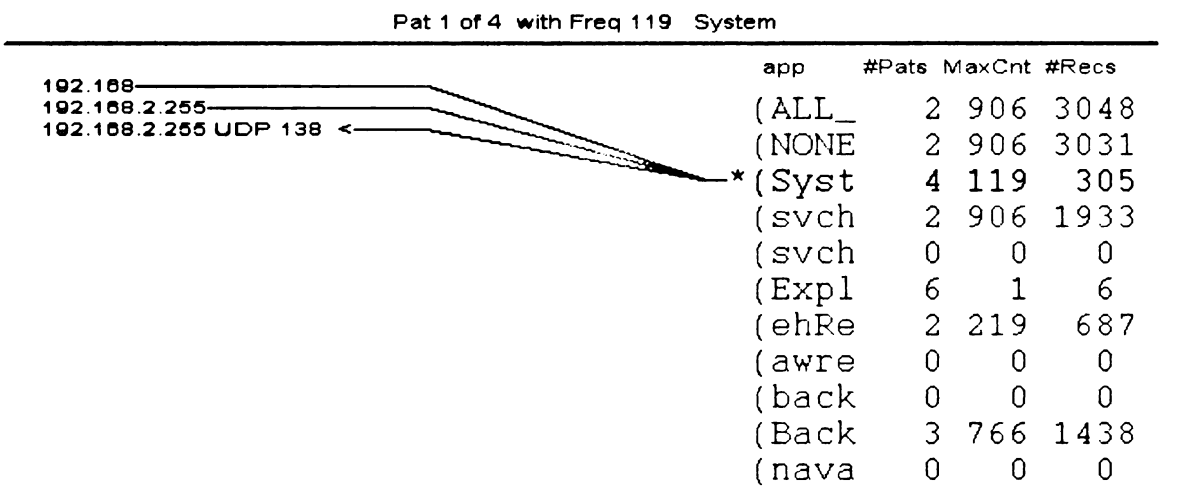


Figure 20: GUI Display of MEDIA Computer
Over a 12 hour duration, this computer was very "chatty" despite it was idle.

5.5 Benefits of Optimizations

Section 5.3 presented the measures taken to filter and process the data so that the analysis done by NetAppMine is as space and time efficient as it could be. Section 5.4 demonstrated that the analysis of this data still produced meaningful and useful results. This section analyzes the performance of NetAppMine.

Because of the high volume of network traffic that a PC can generate, NetAppMine can have large datafiles. Figure 21 is a graph of the RAM usage of NetAppMine for analyzing data. In this case, each dataset represents a 24 hour period of traffic and is ordered from the smallest dataset to the largest. As you can see from the figure, the amount of RAM required to mine patterns grows geometrically with the size of the

dataset. However, implementing space optimizations has a geometric reduction of RAM requirements, and it does so without increasing processing time (see Figure 22).

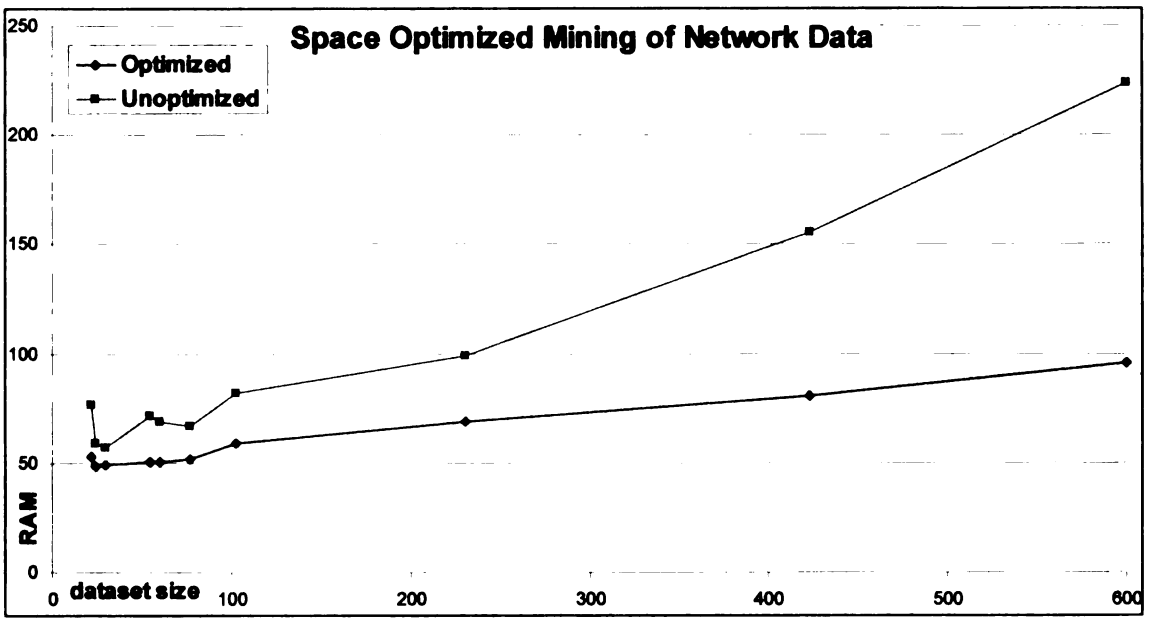


Figure 21: Space Savings Due to Optimizing

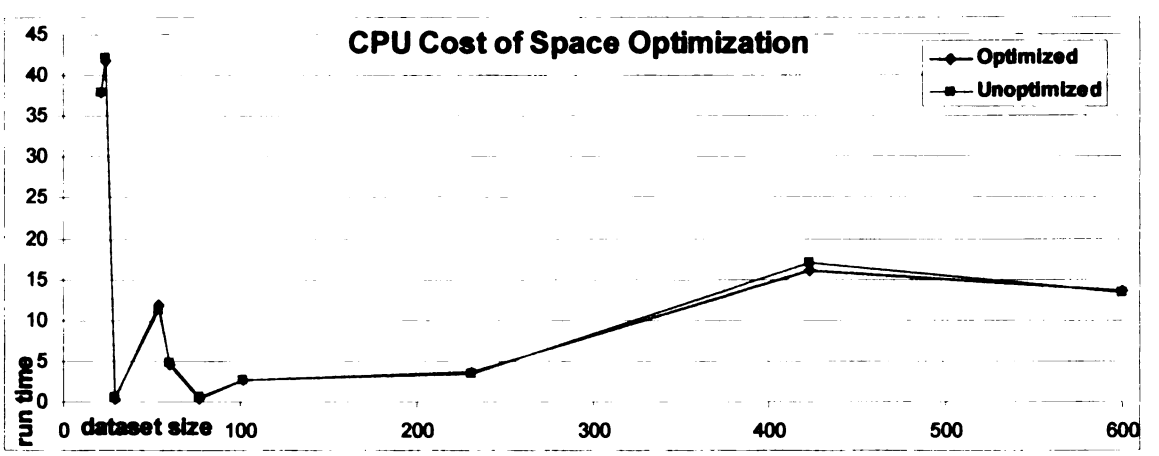


Figure 22: CPU Cost of Space Saving from Optimizing

You may notice that although there appears to be a good relationship between the size of the dataset and the amount of RAM it takes to mine, a similar relationship between the dataset size and runtime does not appear to exist. A lot of parameters affect the runtime of mining, and the size of the dataset is not necessarily the most important. Indeed, while

doing performance analysis, the worst dataset was one of the smaller datasets. Figure 23 is a graph of the processing time needed to do detailed analysis of the network traffic running through the firewall from the experiment done on the TEST computer. From this graph, it can be seen that decreasing the support by a small amount has a dramatic effect on the processing time. The next chapter will look at why this is occurring, and what can be done about it.

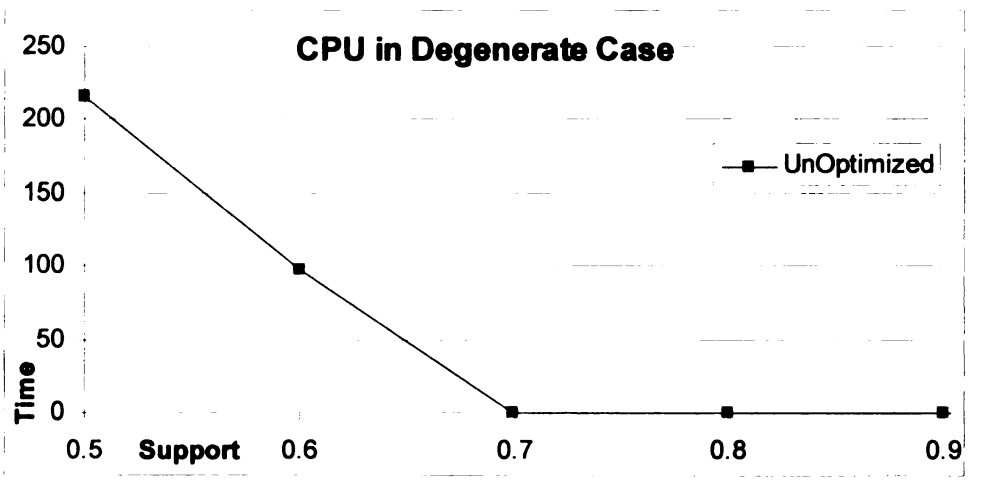


Figure 23: CPU usage with a Dense Data

5.6 Conclusions Drawn From the Application

NetAppMine works, and it does so efficiently. The results of analysis proved useful in determining applications with network problems. Further, the combination of reducing the quantity of data analyzed and implementing the RAM optimization technique has made NetAppMine a practical application that should not overwhelm standard PC.

NetAppMine could be significantly improved. Further studies in normal network activity and a database of acceptable patterns could be added to help automate the process of identifying unusual network traffic. In addition, the application could keep track of user

responses to help automate the future distinction between acceptable and unusual patterns. NetAppMine can be further improved by increasing the kinds of data it collects. For example, it could collect packet sizes, or classifying the kinds of TCP data that is being transferred. This would enhance the analysis and widen the kinds of problems that the NetAppMine could detect. However, degenerate datasets do exist that make such analysis difficult. The next chapter will look at how such datasets might be handled.

CHAPTER 6: Reducing CPU Requirements For Dense Datasets

6.1 Introduction

The last chapter introduced a frequent pattern mining application; however, this application discovered a degenerate datasets that was difficult to mine. The reason for this drastic degeneration of performance was due to the high number of patterns produced by the small dataset. In other words, it was an extremely dense dataset. However, it is common that the performance of mining algorithms degrade as pattern density increases. Such degradation occurs when you are trying to get past common knowledge (ie: "it is dry in the summer") to more interesting knowledge (ie: conditions for "drought in the summer"). This chapter presents an algorithm to increase the efficiency of such analysis.

Recent work on frequent pattern mining focused on using pattern growth algorithms such as FP-Growth. This approach works well when the patterns are longer or there are a large number of patterns. Although Trie-Tree like data structures have proven effective for such algorithms, the construction and the traversal of the trees are a bottlenecks of these approaches. In this chapter, we propose a method that uses a self modifying prefix-tree structure that mines the patterns in a predefined order. The method is called Fri, it works through a process of extracting frequent patterns from a Trie-tree structure in a depth first fashion. Our experiments show that when minimum support is low, or when the pattern density is high, Fri significantly outperforms previous approaches.

FP-tree, the Trie-Tree data structure of FP-growth, has a complex structure of pointers. As such, the cost of construction and traversal of the conditional FP-tree is not trivial, and it adversely affects the performance of FP-growth under certain circumstances. Further,

the bottom-up approach produces results in a very erratic fashion. Indeed, the only pattern it tends towards is the production of the least supported patterns first. In this work, we propose the following techniques to reduce the cost: First, the prefix-tree structure, Trie-tree, is constructed to store the critical information about frequent patterns; Second, a depth-first pattern growth method is proposed to mine the frequent patterns incrementally, as such, the Trie-tree grows during the mining process without having to create conditional trees. The organization of the following sections is as follows. We start by giving a brief description of frequent pattern generation and association rule mining in section 2. Then the general idea of mining frequent patterns from the Trie structure is introduced in section 3. Section 4 discusses the Fri algorithm as well as several optimizations that could be applied. Section 5 is the presentation of the performance study. Section 6 summarizes the study and discusses future work.

6.2 Trie and Frequent Patterns

The data structure Trie was first introduced by Fredkin. The construction procedure of Trie guarantees the sufficiency of this compressed data structure, i.e., it contains all the information in the original dataset. For detailed construction, see section 2.3 or see the article "An efficient Implementation of TRI Structures" by Aoe (et. al).

Figure 24 illustrates a Trie-tree of an example dataset. We denote the subtree with a root of A:6 under Root as subTree(A:6, Root); so the Trie in Figure 24 has 3 subtrees under Root: subTree(A:6, Root), subTree(B:2, Root), and subTree(D:1, Root). It can be seen that the subTree(A:6, Root) represents all the transactions that contain "A", so subTree(A:6, Root) contains the exact information that is needed to generate frequent patterns containing "A"; the union of subTree(B:4, A:6) and subTree(B:2, Root) contains

the exact information to generate frequent patterns containing "B". Due to the structure of the prefix tree, subTree(B:4, A:6) and subTree(B:2, Root) could be merged to form a new Trie, subTree(B:6, Root). The Fri process is a pattern growth algorithm like FP-growth. However, its data structure is a standard prefix-tree (Trie-tree). Further, the frequent patterns are mined in a depth first approach. As such, patterns are created in a very predictable fashion. Further, the mining process is a recursive algorithm that can easily be modified for parallel processing. That is, after the first iteration of mining patterns that begin with "A", the mining of the patterns that begin with "B" can be started. The ordering of the pattern generation can be in any predefined order such as alphabetical or numerical. The patterns are generated in this lexographical order. As a result, if the patterns are added to a tree, the tree would always be adding leaf nodes. Such a tree would be very similar to the tree produced by the Tree Projection algorithm and can be compactly stored and quickly reread for future reference. The Fri algorithm is a recursive algorithm with each iteration producing one new pattern. When mining the "A" branch, the first step is to copy the branches under "A" to other parts of the tree. After that has completed, the "B" branch can be mined independently of the "A" branch. Similarly, after the first iteration of the "B" branch, the "C" branch can be mined independently of "A" and "B". Further, once the mining of a branch has started, the nodes under the branch can be copied to another system and mined independently. This partitioning can be applied to each level of the tree. That is, once the mining of the "AB" branch has started, it too can be copied to another system and mined independently.

DATASET

AB
ABC
ABC
AC
AC
B
BCD
D

TRIE

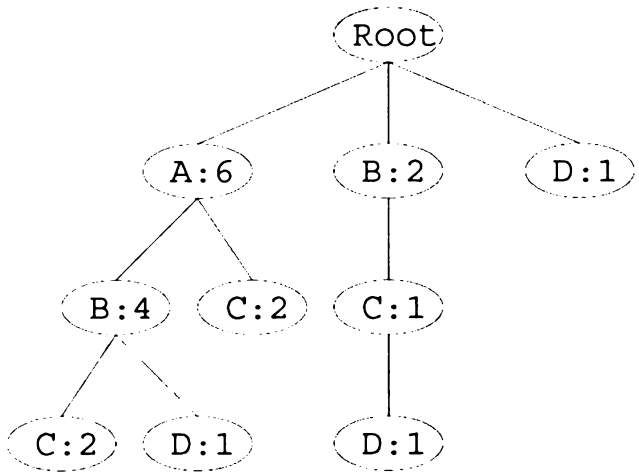


Figure 24: A sample dataset and its Trie

6.3 The Fri Algorithm

The simplified algorithm for producing patterns using a tree is:

Procedure SimpleFri(Branch)

(1) For(each Limb below Branch)

(2) CopyMerge the Limb up one level

(3) Output Pattern

(4) For(each Limb below Branch)

(5) SimpleFri(Limb)

The technique behind pattern growth algorithms is to generate combinations of various patterns in an orderly fashion. Since our algorithm searches for frequent patterns using a Trie-tree, we call the algorithm Fri. The basic approach is a depth first recursive procedure of copying branches up one level. This means the tree will be modifying itself as the patterns are generated. For example, the tree in Figure 24 has subTree(B:4, A:6) which contains the nodes(C:2 & D:1). This subtree would be copied up one level to subTree(B:2, Root). The resulting tree would look like Figure 25. In some cases, the

result will be to change nodes(i.e.: B:2 becomes B:6). Other times, new nodes will need to be created(i.e.: D:1 under B:6 is created).

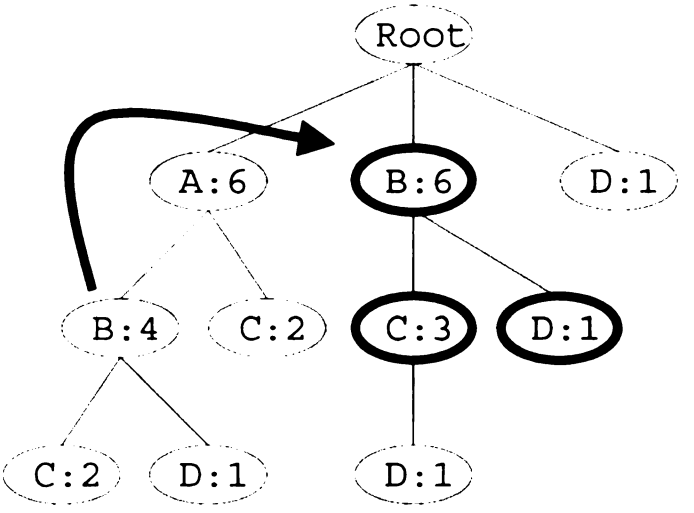


Figure 25: The Trie after applying CopyMerge of subTree(B:4, A:6)

Since subTree(A:6, Root) also has C:2 under it, C:2 would also be copied up one level. This would complete the processing of the "A:6" node. The effect of doing these copies is to create new combinations of items, This is like removing all of the "A"s from the records that contain "A"("AB", "ABC", "ABC", "ABD", "AC", and "AC") and adding them to the Tri-tree. Now the tree would look like Figure 26. Since this is a depth first algorithm, the next subtree to be processed is subTree(B:4, A:6). Copying C:2 and D:1 up one level would have the effect of adding the combinations "ABC", "ABC", and "ABD" without the"B" and without modifying the count of A:6. Processing this new tree would process nodes C:2, D:1 of subTree(B:4, A:6) and nodes C:4 and D:1 of subTree(A:6, Root). However, since these are leaf nodes, they have no subtrees beneath them. As a result, Figure 27 is how the tree would look like after processing all of the nodes in subTree(A:6, Root).

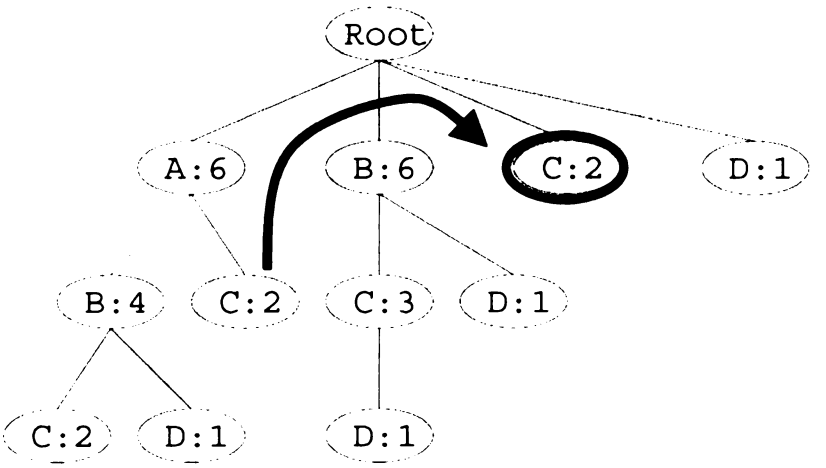


Figure 26: The Trie after applying CopyMerge of subTree(C:2, A:6)

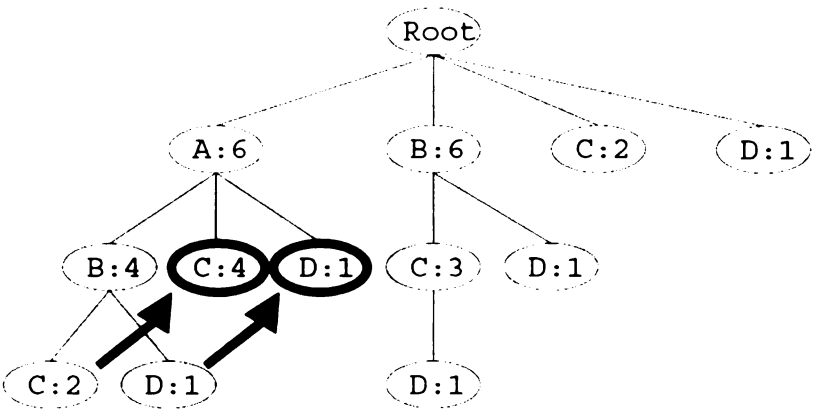


Figure 27: The Trie after applying CopyMerge of two leaf nodes of subTree(B:4, A:6)

After all of the nodes in subTree(A:6, Root) have been processed, each node in subTree(A:6, Root) will represent a pattern. For example, the "A" branch represents the patterns "A:6", "AB:4", "ABC:2", "ABD:1", "AC:4", and "AD:1". When all of the nodes are processed, the resulting tree will have a similar structure to the tree produced by the "Tree Projection" algorithm. Figure 28 is the fully-fledged Trie-tree, each node whose item has a frequency no less than min sup represents a pattern generated by Fri, and the

frequency of the pattern is equal to that of the item in the current node, due to the fact that nodes at a higher level will have a higher frequency.

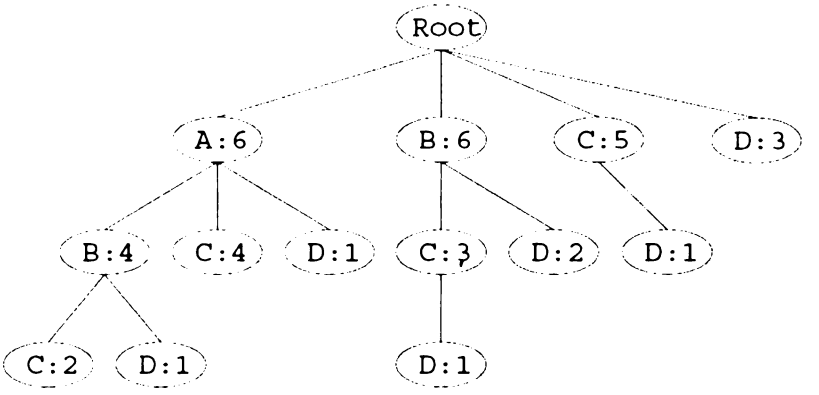


Figure 28: The fully-expanded Trie

6.4 Optimizations

The SimpleFri algorithm is very efficient at generating all patterns for all items in all the transactions. However, when min sup is bigger than one transaction, there are a few optimizations that will significantly increase performance. The motivation for these optimizations is driven by the general structure of a database stored in a Trie-tree. That is, leaf nodes of a Trie-tree tend to be at the end of a long string of nodes with only one child. An example of this is in the branch created by the transaction "BCD" in Figure 24. These "tails" are frequent because long records tend to have unique "tails". For example, two records "BCDE" and "BDE" both have the same prefix "B" but have different postfix items(i.e., "CDE" and "DE"). Because of this, they will create different "tails" in a Trie-tree. Copying these tails up one branch is costly and will usually not generate a frequent pattern. As a result, it is a big optimization to allow multiple references to the same "tail" whenever possible. Another optimization can be done when the algorithm is performing its depth first search and it comes across a node that is not frequent. Since the nodes of

the Tri-tree are ordered (i.e., alphabetically, a node will only have children that are alphabetically bigger than it). This means that when the children of a node are copied up, they can never add to the current node. It can greatly reduce the copying of infrequent nodes if the branch is pruned of infrequent items. Since the siblings of a node were copied up before traversing the branch, pruning will not eliminate any possible frequent patterns of parent nodes. Similarly, after a node has been processed, it can be deleted to reclaim any resources it is using. The last optimization is a result of the pruning process. When infrequent items are removed from a branch, it frequently leaves a tail of nothing but frequent items. This is especially true if there is a high confidence between the items. As a result, a more efficient algorithm for printing all combinations of all nodes can be utilized. The new optimized algorithm works very well for processing nodes in memory. The optimized algorithm is:

```
Procedure Fri(Branch)
    (1)   For (each Limb below Branch)
    (2)       CopyMerge theLimb up one level
    (3)       If (Branch Support < min sup)
    (4)           Return
    (5)       Output Pattern
    (6)   CheckForRef(Branch)
    (7)   For (each Node below Branch)
    (8)       Limb = subTree(Node, Branch)
    (9)       If(Node Support < min sup)
    (10)          Prune(Limb)
    (11)      Else
    (12)          Fri(Limb)
    (13)      Delete Node
```

The function CheckForRef() will ensure that the branch is not multiply referenced. Limb is a reference to a subtree, not a copy of the subtree.

6.5 Experiments

The experiments on the performance comparison between Fri and FP-growth were conducted on an AMD Atholon 2800+ PC with 768MB RAM, running RedHat Linux 8.0. The programs were written in C++ and compiled with the "-O3" compiler optimization.

We used synthetic datasets for our experiments. The procedure of generating the datasets is described in. In the first experiment, min sup is varied from 0.07% to 0.7% with the dataset D1(T20I7D500K) where average transaction size is 20, the average maximal potentially frequent itemset size is 7, there are 500K transactions, and the number of different items is 20K. In the second experiment, the dataset size is varied from 100K transactions to 2 million transactions with a min sup of 1%, 2%, and 3% and the D2 datasets (T20I6D*K) where the average transaction size is 20, the average maximal potentially frequent itemset size is 6 and the number of different items is 20K. In the third experiment, the potentially frequent pattern length (PatLen) varies from 1 (random data) to 13 (very long maximal patterns) with the D3 datasets (T20I*D500K) where the average maximal potentially frequent itemset size is from 1 through 14 and the number of items is 20K. The last experiment varied the number of predefined frequent patterns used to generate the artificial datasets. This would be the "NPats" parameter of the "gen" program. NPats was varied from 40 patterns to 200 patterns for the D4 datasets (T20I7D500K) with 20K items.

The first experiment (Figure 29) showed that Fri outperforms FP-growth significantly when min sup is less than 0.2%, This would be because of the high density of patterns when min sup is small. The second experiment (Figure 30) showed that Fri is up to 5 times faster than FP-growth when the dataset has 200K transactions. Further, Fri is consistently faster than FP-growth for datasets of at least 2 million transactions. FP-growth does appear to scale better, but the dataset would have to be significantly bigger than 2 million transactions to outperform Fri. Further, because Trie-Trees have a lot of "tails", and Fri does not prune tails until it needs to, Fri's runtime is much more predictable for similar datasets.

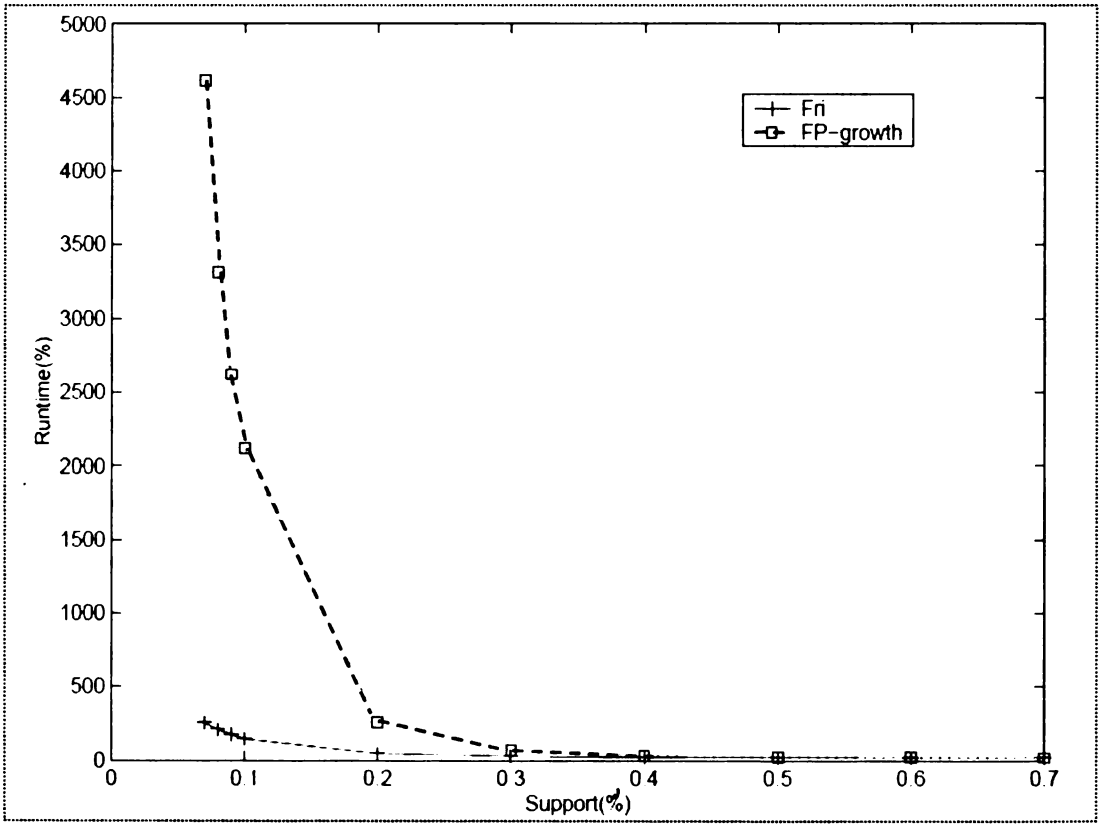


Figure 29: Performance comparison on T20I7D500K

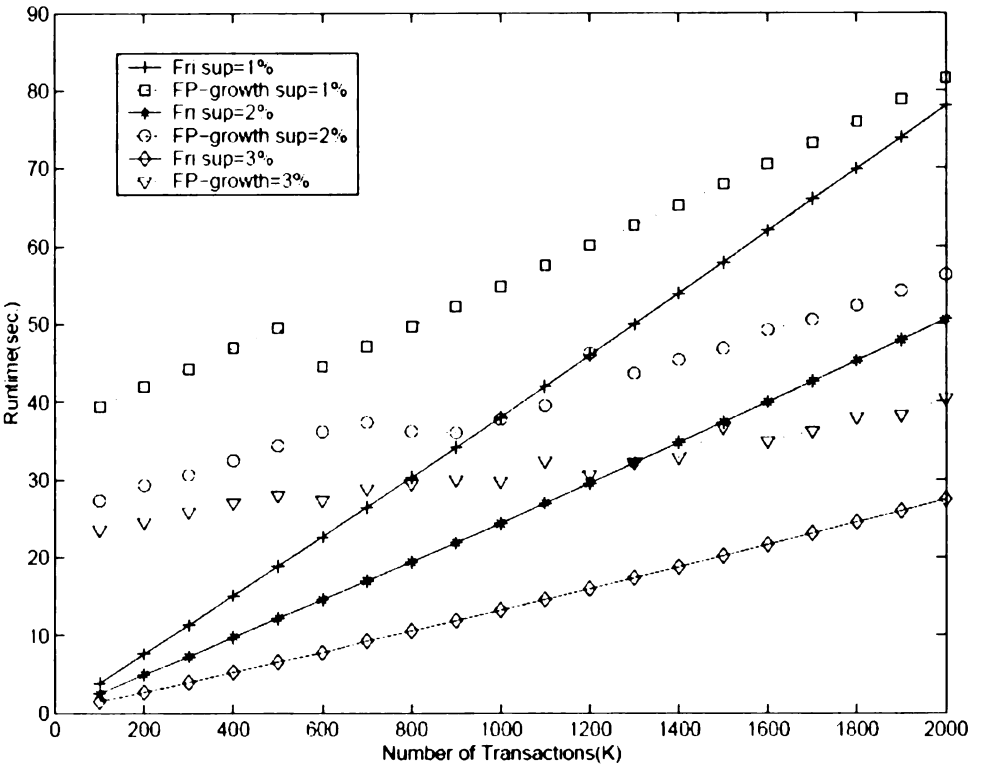


Figure 30: Performance comparison on T20I6D*00K

The third experiment (Figure 31) showed that Fri works better with longer average maximal pattern lengths. Fri outperforms FP-growth as pattern lengths are greater than 6. Further, as the pattern lengths get longer, the difference becomes very significant. When the pattern length is 1, there is essentially a random or orthogonal relationship between the items in the dataset. Under such circumstances Fri does not perform as well as FP-growth, but neither algorithm will find many patterns because there are not many patterns to be found. This helps demonstrate that Fri works best when frequent patterns are dense.

The last experiment (Figure 32) showed Fri performs well when there is a large number of shorter patterns. Since the procedure described in draws its items for the generated dataset from a set of predefined potential frequent patterns, a larger number of such predefined patterns will decrease the occurrence of any one potential pattern. This has

the effect of reducing the density of frequent patterns. With a reduced pattern density, Fri and FP-growth perform about the same, but when the density gets higher, Fri significantly outperforms FP-growth.

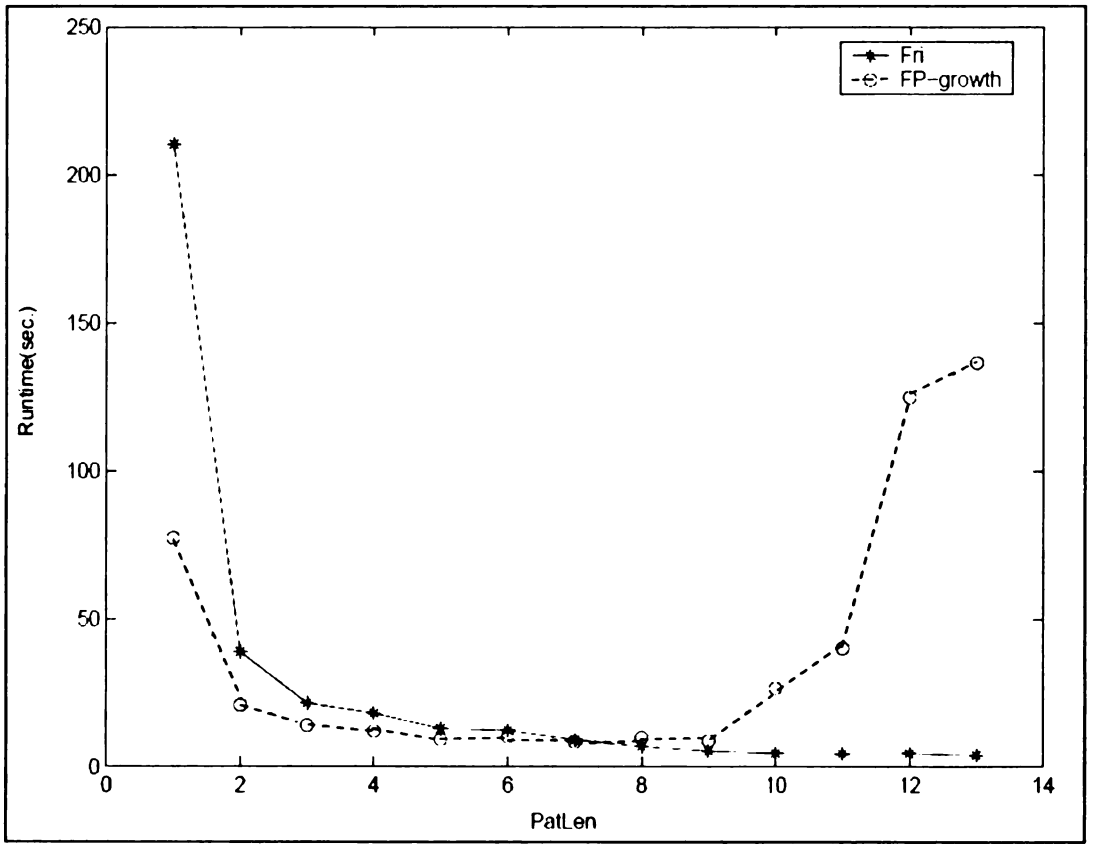


Figure 31: Performance comparison on T20I*D500K

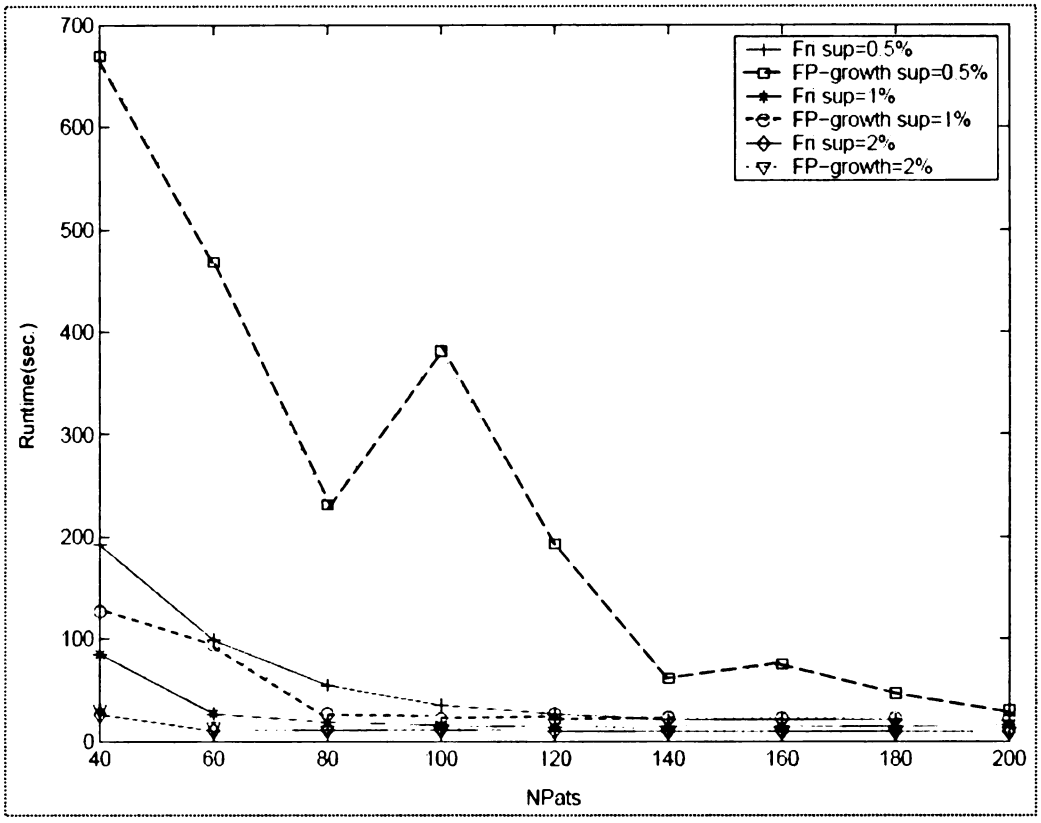


Figure 32: Performance comparison on T20I7D500K with varying NPats

6.6 Conclusions of Reducing CPU Requirements

In this chapter, we have used a Trie structure to store the database in a compressed format without information loss, and we then proposed that a pattern growth algorithm, Fri, be used for frequent pattern mining in databases with dense frequent patterns. Fri outperforms other algorithms as follows: (1) the temporary data structure for storing the compressed dataset is very compact; (2) the pattern growth method not only avoids candidate set generation and test, but also significantly reduces the cost of node count re-computation when the pattern density is high.

However, the real question is, does it help with the problem discovered in the previous chapter. The figure below is a comparison between mining without the CPU optimization and mining with Fri. As you can see, it cut the run time in half. This is not to say Fri is best in all situations, but it clearly helped in this one.

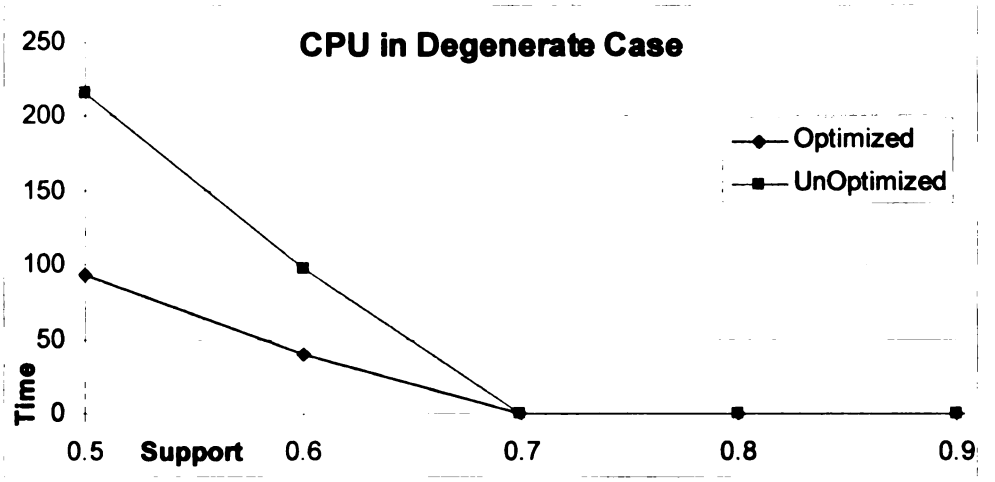


Figure 33: CPU Optimization of Degenerate Case

Further research issues related to Trie and Fri are caching and disk-based algorithms. For example, when the dataset is very large and the resulting Trie-tree cannot fit into memory, we can put Trie onto disk. The locality of the Fri algorithm makes it possible to efficiently generate patterns with a caching strategy, thus very large databases could be mined. Initial studies have shown that data copied forward in the tree accounts for less than 1% of the data processing needed to calculate frequent patterns. That is, the algorithm has better than a 99% cache hit rate as it performs it's mining. An algorithm to reduce the cost of cache hit/miss calculations is still being worked on.

The current algorithms repeat a lot of work. The same transactions are repeatedly searched if they contain the items of two or more frequent patterns. As a result, they will need to be repeatedly pruned and/or copied. One approach we are looking into is to store

the results of a transaction search. The obvious disadvantage of such an approach is the high RAM requirements. Of course, the technique presented in Chapter 4 will help deal with that.

CHAPTER 7: Conclusions and Future Work

This thesis has presented a road map for expanding frequent pattern mining into a new environment. The new environment is pattern mining as a secondary application. Under this new role, mining needed to be optimized to avoid having to size the resources of a system for a secondary application. Since mining sometimes has high processing costs, chapter 4 demonstrated how to implement frequent pattern mining to reduce processing requirements. Since mining often has high memory requirements, chapter 5 introduced an approach to significantly reduce the memory usage of pattern mining. To demonstrate that mining can have practical uses as a secondary application, chapter 6 implemented enhancements to PC Internet security using frequent pattern mining. Together, these chapters provide a practical map for optimizing and applying frequent pattern mining to a new area of research.

Further research of frequent pattern mining in secondary application can lead to an enhanced computing environment. For instance, realize that the document you are reading now was created with a program that continually tried to "autocorrect" the formatting, but kept "guessing" wrong. Reconfiguring the program could have been done, but formatting preferences changed as the context of the document changed. Frequent pattern mining could be used to adapt to the pattern usages of the user and enhance the word processing experience. This is just a second example of how frequent pattern mining could enhance the computing environment: it could also be used to enhance disk sage; adjust the display of icons, windows, and colors; reorder search results for files and web pages to match our usage patterns; and many, many others

secondary applications. But it all depends upon mining being optimized enough to handle the increasing data load without overwhelming the system it is trying to enhance.

Frequent pattern mining is a very active area of research; some of this research can be used to enhance this work. For instance, because of the nature of permutations, there are often too many frequent patterns to be analyzed. To assist, there is a lot of research in reducing this number to a more manageable size. For instance, there are frequently groups of patterns that use the same items. Analyzing these groups instead of analyzing the individual patterns reduces the complexity of analysis to something more practical. In addition, having a preset criterion of interesting patterns helps to limit analysis to more useful patterns. Also, analyzing patterns that are missing helps to find missing data. Missing patterns could be an indication of preferences people do not want to see, or forces that keep patterns from occurring. All of this research, and much more, can be incorporated with this work to enhance the computing environment and make applications more useful.

APPENDIX A: Support vs Space

The focus of the thesis has been on the optimizing of mining for PC applications. However, as is often the case, while researching one topic, new and interesting topics arise. Figure 16 of section 4.6 demonstrated a relationship between RAM and minimum support. In this appendix, we have the opportunity to expound on the implications of this relationship. Especially the rather surprisingly predictable relationship between minimum support and RAM requirements.

Since the diverse datasets used in section 4.6 were from unrelated sources, it was surprising to see such a strong and predictable relationship arise in Figure 16. Figures 34 and 35 repeat the memory requirements for WebDocs and Accidents, as well as adding a 160 MB dataset of census data from 2000, and a 500 MB synthetic dataset. Although Figure 34 shows curved lines, Figure 35 demonstrates how straight the lines become when comparing support with the log of RAM. Interestingly enough, this logarithmic relationship holds strongest for the real world datasets, and such datasets do not normally exhibit such nice linear results without a strong overriding principle governing it. Although the principle governing the relationship between support and RAM is not strict, it has a dominating effect for minimum support as high as 20%. Working from the $y = ax + b$ formula of a straight line, the principle (hereafter called the *support-space principle*) would be as follows:

$$\ln(D') = -a\,s + \ln(D)$$

where:

- s = the minimum support
- D = the space needed for storing the entire dataset
- D' = the space needed for a given support
- a = a constant dependent upon the dataset and the algorithm used to mine it.

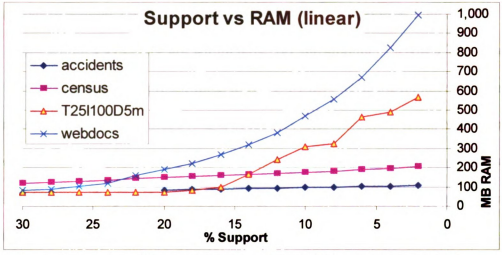


Figure 34: Support vs RAM (linear)
Note: The Linear Graph Produces Curved Lines

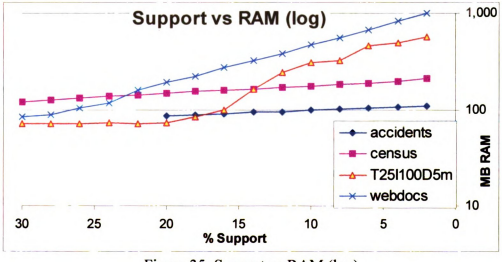


Figure 35: Support vs RAM (log)
Note: The Log Graph Has Straight Lines

To understand the implications of this principle, it helps to look at some of the observations of section 5.6. That section observed that for support as high as 20%, most of the nodes within a Trie-Tree are within tails. So, the amount of RAM a Trie-Tree is going to need is going to be proportional to R * L', where R is the number of records in the dataset, and L' is the average length of records after infrequent items have been filtered out. However, section 5.6 also demonstrates that most records within a dataset contain at least one frequent item. As a result, the number of records within a dataset remains relatively constant with respect to the minimum support.

The consequence of the observations of section 5.6 and the support-space principle is that there is a strong relationship between L' and minimum support. Hereafter, this relationship will be called the *support-length principle*, and it can be deduced from the support-space principle as follows:

Given:

$\ln(D') = -a\,s + \ln(D)$ (where a is a constant and s is the minimum support)

$D' = R * L'$ (where L' is the average length of filtered records)

L = Average length of unfiltered records in the dataset

Then:

$\ln(L' * R) = -a\,s + \ln(L * R)$

$\ln(L' * R) = \ln(L * R * e^{-as})$

$L' * R = L * R * e^{-as}$

$L' / L = e^{-as}$

Using the support-space and the support-length principles, it is possible to estimate the space needs and the average lengths of filtered records. This appears to work best for larger real world datasets with longer records. Fortunately, it is just such datasets that need good estimates. To do so would only require some basic knowledge of the

characteristics of the data, and a way to calculate "a". Calculating "a" can be done by either measuring the space requirements at a relatively high minimum support, or measuring the average length of records after filtering out infrequent items.

In any case, these principles are a beginning to getting a handle on analyzing resource requirements for frequent pattern mining. Further research would need to be done to define the environments best suited for these principles, but it is clear that they do occur in the real world. Perhaps further analysis may reveal why these relationships exists; but, until then, the results have an immediate and practical application of estimating resource needs.

BIBLIOGRAPHY

Definition of "trie" http://en.wikipedia.org/wiki/Trie

R. Agarwal, T. Imielinski, and A. Swami. (1993) Mining association rules between sets of items in large databases In SIGMOD '93, pp. 207-216.

R. Agarwal and R. Srikant (1994) Fast Algorithms for Mining Association Rules In VLDB '94 pp. 487-499.

R. Agarwal, C. Aggarwal, V.V.V. Prasad (1999) A Tree Projection Algorithm for Generation of Frequent Itemsets. In Proc. of High Performance Data Mining Workshop, 1999.

J. Han, J. Pei, and Y. Yin (2000) Mining frequent patterns without candidate generation. In SIGMOD '00, pp. 1-12.

J. Park, M. Chen and P. Yu. (1995) An Effective Hash-based Algorithm for Mining Association Rules. In SIGMOD '95, pp. 175-186.

J. Pei, J. Han, etc. (2001) H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. In ICDM '01.

E. Fredkin. Trie memory, Commun. ACM, 3,(9), 490-500, 1960.

J. Aoe, K. Morimoto and T. Sato. (1992) An Efficient Implementation of TRI Structures. Software--Practice and Experience, 22, 695-721, 1992.

J. Han and M. Kamber (2000) Data Mining: Concepts and Techniques. Morgan Kaufmann, 2000.

Spyware Wikipedia: (2004) http://en.wikipedia.org/wiki/Spyware (Nov 14, 2004)

Stefan Saroiu, Steven D. Gribble, and Henry M. Levy. (2004) Measurement and Analysis of Spyware in a University_ Environment. (Feb 2004)

SpywareGuide, (2004) http://www.spywareguide.com/product_list_full.php (Nov 2004)

Brad Rosen. (2003) Infrastructural Requirements for a Privacy Preserving Internet, (Fall 2003)

Stephen Withers, ZDNet Australia.
http://www.zdnet.com.au/news/security/0,2000061744,20269428,00.htm, (Oct 28, 2002)

Markus Hegland. Algorithms for Association Rules, (2002)

Robert F. Erbacher and Deborah Frincke. Visual Behavior Characterization for Intrusion and Misuse Detection (2001)

Ngsniff of Ngsec. http://www.ngsec.com/ngresearch/ngtools (Jun 9, 2004 version 1.2)

Mark Russinovich. tcpvcon of Sysinternals. http://www.sysinternals.com/ntw2k/source/tcpview.shtml (Aug 9, 2004 version 2.34)

A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In 21st VLDB Conference, 1995.

Geurts, Wets, Brijs, and Vanhoof. "Profiling High Frequency Accident Locations Using Association Rules" Limburg University: Data Analysis & Modelling Group

Lucchese, Orlando, Perego, and Silvestri. "WebDocs: a real-life huge transactional dataset" Dipartimento di Informatica, Universit`a Ca' Foscari di Venezia, Venezia, Italy