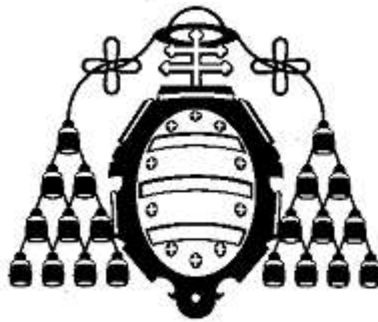
# UNIVERSIDAD DE OVIEDO

## MASTER IN SOFT COMPUTING
## AND INTELLIGENT DATA ANALYSIS

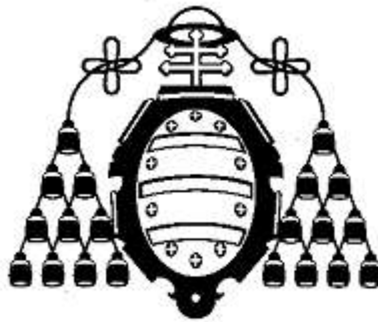
**PROYECTO FIN DE MASTER**
**MASTER PROJECT**

# Improving the IsTa algorithm by using Patricia trees


**Anaís Antonio Ruiz**

**July 2012**

# UNIVERSIDAD DE OVIEDO



# MASTER IN SOFT COMPUTING AND INTELLIGENT DATA ANALYSIS

## PROYECTO FIN DE MASTER
## MASTER PROJECT

**Improving the IsTa algorithm by using Patricia trees**

**Anaís Antonio Ruiz**

**July 2012**

**TUTOR / ADVISOR:**
**Dr. Christian Borgelt - European Centre for Soft Computing (ECSC)**
**Dr. Irene González Díaz – Universidad de Oviedo**

# ABSTRACT

DNA microarrays are a high throughput technology used to measure the expression levels of thousands of genes. Microarray datasets may contain up to thousands or tens of thousands of columns (genes) but only tens or hundreds of rows (samples). Discovering frequent patterns from microarray datasets is very important and useful to derive biological knowledge from gene expression data. However, the high-dimensionality and the complex relationships among genes impose great challenges for existing data mining methods. While there are a large number of algorithms that have been developed for frequent item set mining, their basic approaches are based on item enumeration in which combinations of items are tested systematically to search for frequent (closed) item sets.

In this project we propose an improvement over an algorithm that adopts an alternative approach, such as intersecting transactions. The IsTa algorithm works in a cumulative scheme, in which new transactions are intersected with a repository of already found closed item sets. IsTa has demonstrated to significantly outperform other approaches to frequent item set mining on specific data sets that occur particularly often in the area of gene expression analysis.

The original version of the IsTa relies on a prefix tree to store and compute the transactions. Our proposal to improve the IsTa algorithm is change the underlying data structure by using a compressed tree representation, which is the Patricia tree.

This document reports several experimental results on real datasets, which assess the effectiveness of the implementation. It is shown that the implementation with Patricia trees is more efficient in terms of memory usage, in fact, the number of nodes and memory occupied is smaller than the original version of IsTa with Prefix trees, although the best execution time was attained by the Prefix tree.

# CONTENTS

**1 Introduction** ........ 1

1.1. Data mining ........ 1

1.2. Motivation ........ 2

1.3. Document organization ........ 3

**2 Frequent item set mining** ........ 4

2.1. Setting the problem ........ 4

2.2. Types of frequent item sets ........ 5

2.3. Item Set Enumeration Algorithms ........ 6

2.4. Intersection Approach Algorithms ........ 8

**3 Ista Algorithm** ........ 9

3.1. Definition ........ 9

3.2. Prefix tree implementation ........ 10

**4 Improving IsTa** ........ 13

4.1. Patricia trees ........ 13

4.2. Construction of the Patricia tree ........ 14

4.2.1. Inserting transactions ........ 16

4.2.2. Intersecting transactions ........ 18

**5 Experimental Evaluation** ........ 22

5.1. Gene expression analysis ........ 22

5.2. Experimental results ........ 23

**6 Conclusions** ........ 27

**Bibliography** ........ 28

# Acknowledgments

This work was supported by the ECSC Scholarship Programme of the Master in Soft Computing and Intelligent Data Analysis.

# LIST OF FIGURES

Figure 1: An Example Database ........ 5
Figure 2 Maximal and Closed Item sets ........ 6
Figure 3: Structure of the Prefix tree nodes ........ 10
Figure 4: Code of the intersection function ........ 11
Figure 5: Structure of the Patricia tree node ........ 14
Figure 6: Patricia tree example ........ 15
Figure 7: Prefix tree example ........ 15
Figure 8: Insertion examples ........ 16
Figure 9: Code of the add function ........ 17
Figure 10: An example of intersection ........ 19
Figure 11: Function to add new transaction to tree ........ 20
Figure 12: Memory usage performance ........ 24
Figure 13: Runtime comparison of the data structures ........ 26

# CHAPTER 1
# INTRODUCTION

Many areas of science, business and other environments deal with a vast amount of data, which needs to be turned into something meaningful, knowledge. This is the aim of data mining, extraction of useful patterns and relationships from data. With the rapid growth of computational biology and e-commerce applications, extremely high dimensional data becomes more common. Thus, mining these kinds of data sets is a problem of great practical importance.

Studies of Frequent Item set (or pattern) Mining are acknowledged in the data mining field because of its broad applications in mining association rules, correlations, sequential patterns, and many other data mining tasks. This project takes on the challenge of extending one of the Frequent Item set algorithms, which is IsTa. The idea is to improve the algorithm by changing the underlying data structure through the use of Patricia trees, implement the extended algorithm, test it on gene expression data, and finally determine whether Patricia trees yield advantages over the current implementation.

## 1.1. DATA MINING

Data mining is an interdisciplinary field of computer science, [2, 3] is defined as the process of discovering new patterns in large data sets. It utilizes methods at the intersection of artificial intelligence, machine learning, statistics, and database systems [2]. The overall goal of the data mining process is to extract knowledge from an existing data set and transform it into a human-understandable structure for further use [2].

Simply stated, data mining is a method of extracting interesting knowledge, such as rules, patterns, regularities, or constraints, from data in large data sets. The extracted knowledge should be non-trivial, previously unknown, implicit, and potentially useful in that it may serve as an important input for making decisions. Some applications of data mining are target marketing, customer relation management, market basket analysis, cross selling, market segmentation, forecasting, quality control, fraud detection, and intelligent query answering.

Mainly, data mining involves six common classes of tasks: Anomaly detection, Association rule learning, Clustering, Classification, Regression and Summarization.

Frequent item set mining is an essential part of association rules mining, and its concept has also been extended for many other data mining tasks such as classification [3, 4], clustering [5], and sequential pattern discovery [6]. Thus, effective and efficient frequent pattern mining is an important and interesting research problem.

Among the best-known frequent set mining algorithms are Apriori [7, 8], Eclat [9] and FP-growth [10]. Most of these algorithms adopt an enumerate items approach that works "top-down", since they start at the one-element item sets and work their way downward by extending found frequent items sets by new items.

These algorithms can find frequent closed patterns when the dimensionality is low to moderate, but when the number of dimensions (items) is very high, e.g., greater than 100, the efficiency of these algorithms could be significantly impacted. CARPENTER [11] and IsTa [12] are some of algorithms proposed to solve this problem. They use a different solution approach; instead of enumerating items the approach adopted is the intersection of transactions. This approach works "bottom-up", because they start with large items sets, namely the transactions, which are reduced to smaller sets by intersecting them with other transactions.

## 1.2. MOTIVATION

The emergence of various new application domains stresses the need for analyzing extremely high dimensional data. Bioinformatics as an example, with the rapid advance of microarray technology, gene expression data are being generated in large throughput so that discovering frequent patterns from microarray datasets is very important and useful to extract biological knowledge.

However, these high-dimensional microarray datasets pose a great challenge for existing frequent pattern discovery algorithms. While there is a large number of algorithms that have been developed for frequent pattern discovery and closed pattern mining [13, 14, 15], their basic approaches are based on item enumeration in which combinations of items are tested systematically to search for frequent (closed) patterns. As a result, their running time increases exponentially with increasing average length of the records. The high dimensional microarray datasets render most of these algorithms impractical

The IsTa algorithm has proven to significantly outperform other frequent mining approaches on this particular type of datasets, in which there are very many dimensions or items (several thousand to tens of thousands), but fairly few transactions (several dozens to hundreds).

The IsTa algorithm intersects transactions in accumulative approach, representing the dataset through a prefix tree, built on the already found transactions. The advantage of using a prefix tree is substantial for dense datasets because of the compression achieved by merging common prefixes, but in the worst case, when the dataset is highly sparse, the number of nodes may be close to the size N of the original dataset. Since our target are extremely high dimensional data sets, it has been proposed to change the data structure to a compressed tree, better known as Patricia tree, in order to improve space and time efficiency.

## 1.3. DOCUMENT ORGANIZATION

The outline of this project is as follows:

- Chapter 2 briefly discusses few essential topics in frequent set mining and some well-known algorithms.
- Chapter 3 introduces the IsTa algorithm and its current implementation.
- Chapter 4 presents our Patricia tree building.
- Chapter 5 shows the experimental results and the comparison against the original IsTa implementation.
- We conclude the document and describe future directions of research in Chapter 6.

# CHAPTER 2
# FREQUENT ITEM SET MINING

Frequent item set mining is a useful model for extracting salient features of the data. It was first proposed by [7, 8] for market basket analysis in the form of association rule mining. It analyses customer buying habits by finding associations between the different items that customers place in their "shopping baskets". For instance, if customers are buying milk, how likely are they going to also buy cereal (and what kind of cereal) on the same trip to the supermarket? Such information can lead to increased sales by helping retailers do selective marketing and arrange their shelf space.

A market basket data set is typically represented as a set of transactions. Each transaction contains a set of items from a finite vocabulary. The goal is to find the collection of item sets appearing in a large number of transactions

## 2.1. SETTING THE PROBLEM

The frequent pattern mining problem was first introduced by R. Agrawal, et al. in [7, 8] as mining association rules between sets of items.

Let $B$ be a set of distinct items, called the *item base*. The items could represent different products in a supermarket. A set of items is called an *item set*.

Each *transaction* is an item set and could represent some customer purchases of products from a supermarket. A database $T = \{t1, \dots, t_n\}$ is just a set of transactions.
The *cover* $K_T(I)$ of an item set $I \subseteq B$ w.r.t. this database is set of indices of transactions that contain it, that is, $K_T(I) = \{ k \in \{1, \dots, n\} \mid I \subseteq t_k\}$.

The *support* $S_T(I)$ of an item set $I \subseteq B$ is the size of tis cover, that is, the number of transactions in the database T it is contained in. There is a user-defined *minimum support* threshold $S_{min}$. An item set is *frequent* iff its support is above the minimum support. Otherwise, it is *infrequent*.

***Problem statement:*** Given a user-specified *support* threshold $S_{min}$, X is called a *frequent item set* or *frequent pattern* if $S_T(I) \geq S_{min}$. The problem of mining frequent item sets is

to find the complete set of frequent item sets in a transaction database $T$ with respect to a given support threshold $S_{min}$.

| Item base |
|---|
| a |
| b |
| c |
| d |
| e |

| Transaction Database | |
|---|---|
| 1 | {a,b,c,d,e} |
| 2 | {a,c} |
| 3 | {a,b} |
| 4 | {a,b,c,d} |

**FIGURE 1: AN EXAMPLE DATABASE**

***Example:*** Consider our example database in figure 1. There are five distinct items, from 'a' to 'e', in the item base. There are four transactions. If the minimum support is set to 2, then the frequent items are: {a}, {b}, {c}, {d}, {a, b}, {a, c}, {b, c}, {b, d}, {a, b, c}, {a, b, d}, {a, c, d}, and {a, b, c, d}, since they occur at least in 2 transactions. For instance, item set {a, b} is frequent, since three out of the four transactions (transactions 1, 3 and 4) contain the items 'a' and 'b'. On the other hand, item set {b, e} is infrequent, since only one out of the four transactions contains items 'b' and 'e'.

## 2.2. TYPES OF FREQUENT ITEM SETS

In order to find a frequent item set we have to go through all the sub-item sets which are frequent, so we unavoidably generate an exponential number of sub-patterns that we might not really need. In order to reduce the output the most basic approaches is to restrict the output to closed or maximal items frequent item sets.

A frequent set is *maximal* if it has no frequent superset; the set of maximal patterns is typically orders of magnitude smaller than all frequent patterns. Formally:

$$I \subseteq B \text{ is maximal } \Leftrightarrow S_T(I) \geq S_{min} \ \wedge \ \forall i \in B - I{:}\ S_T(I \cup \{i\}) < S_{min}$$

The only one downside of a maximal frequent item set is that, even though we know that all the sub-item sets are frequent, we don't know the actual support of those sub-item sets.

An item set is *closed* if it has no superset with the same support. Formally:

$$I \subseteq B \text{ is closed } \Leftrightarrow S_T(I) \geq S_{min} \ \wedge \ \forall i \in B - I{:}\ S_T(I \cup \{i\}) < S_T(I)$$

Closed sets are *lossless* in the sense that they uniquely determine the set of all frequent item sets and their support. At the same time closed sets can themselves be orders of magnitude smaller than all frequent sets.

***Example*** Consider the figure below (figure 2), corresponding to the example transaction database from figure 1. When the minimum support is set to 2, the individual items {a}, {b}, {c}, {d} are frequent item sets because their support is at least 2. But only {a} is closed because none of its immediate supersets has the same support, for instance the item {b} has the superset {a, b} having the same support. So it contradicts the definition. The item set {a, b, c, d} is frequent because it is present in at least 2 of the transactions, and it is maximal as well because its only superset {a, b, c, d, e} is not frequent.

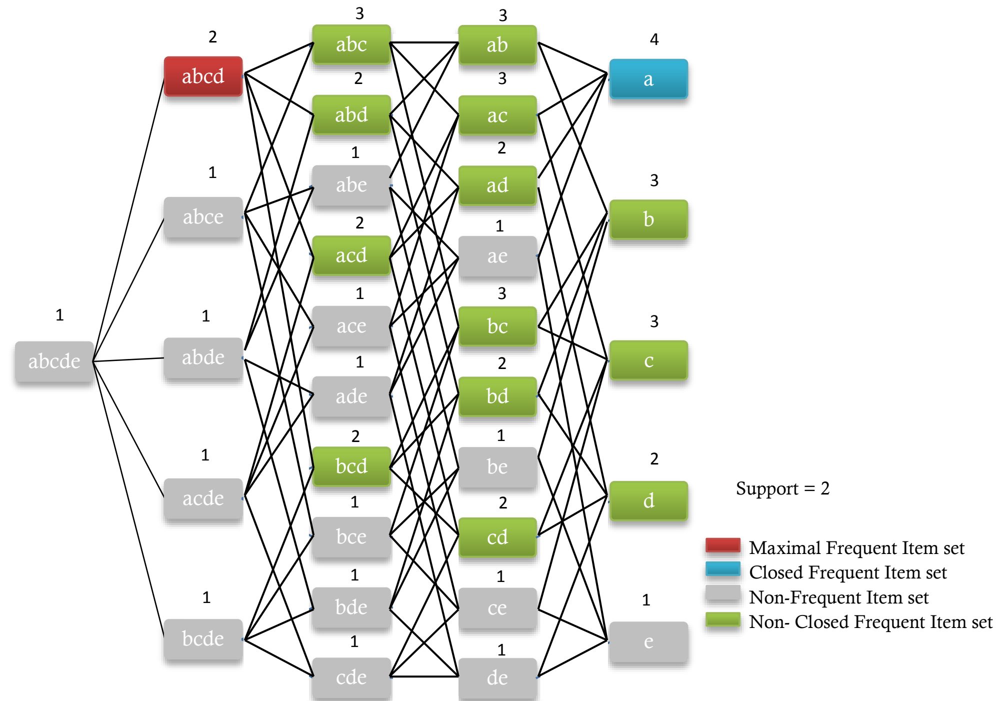



FIGURE 2 MAXIMAL AND CLOSED ITEM SETS

## 2.3. ITEM SET ENUMERATION ALGORITHMS

The first algorithm to generate all frequent item sets and confident association rules was the AIS algorithm by Agrawal et al. [7, 8], which was given together with the introduction of this mining problem. Shortly after that, the algorithm was improved and renamed Apriori by Agrawal et al., by exploiting the monotonicity property of the support of item sets and the confidence of association rules [8, 16]. The algorithm scans the transaction datasets several times. After the first scan the frequent 1-itemsets are found, and in general after the nth scan the frequent n-item sets are extracted. The method does not determine the support of every possible item set. In an attempt to narrow the domain to be searched, before every pass it generates candidate item sets and only the support of the candidates are determined. An item set becomes a candidate if all its proper subsets of are frequent. Due to the bottom-up

search, all frequent item sets of size smaller than the candidate are already determined; therefore it is possible to do the subset validations.

The Eclat algorithm by Zaki [17, 18] was the first algorithm proposed to generate all frequent item sets in a depth-first manner. It uses a vertical database layout i.e. instead of explicitly listing all transactions; each item is stored together with its cover (also called tidlist) and uses the intersection based approach to compute the support of an item set. In this way, the support of an item set X can be easily computed by simply intersecting the covers of any two subsets $Y, Z \subseteq X$, such that $Y \cup Z = X$. It states that, when the database is stored in the vertical layout, the support of a set can be counted much easier by simply intersecting the covers of two of its subsets that together give the set itself. In this algorithm each frequent item is added in the output set. After that, for every such frequent item i, the i-projected database $D_i$ is created. This is done by first finding every item j that frequently occurs together with i. The support of this set {i, j} is computed by intersecting the covers of both items. If {i, j} is frequent, then j is inserted into $D_i$ together with its cover. Then the algorithm is called recursively to find all frequent item sets in the new database $D_i$.

Later, several other depth-first algorithms have been proposed [19, 20, 21] of which the FP-growth algorithm by Han et al. [21, 10] is the most well-known. The main aim of this algorithm was to remove the bottlenecks of the Apriori algorithm in generating and testing candidate set. The problem of Apriori algorithm was dealt by introducing a compact data structure, called frequent pattern tree (FP-tree), then based on this structure a pattern fragment growth method was developed. FP-growth uses a combination of the vertical and horizontal database layout to store the database in main memory. Instead of storing the cover for every item in the database, it stores the actual transactions from the database in a tree structure and every item has a linked list going through all transactions that contain that item.

The SaM (Split and Merge) algorithm established by [22] is a simplification of the already fairly simple RElim (Recursive Elimination) algorithm. While RElim represents a (conditional) database by storing one transaction list for each item (partially vertical representation), the split and merge algorithm employs only a single transaction list (purely horizontal representation), stored as an array.

This array is processed with a simple split and merge scheme, which computes a conditional database, processes this conditional database recursively, and finally eliminates the split item from the original (conditional) database

## 2.4. INTERSECTION APPROACH ALGORITHMS

Basically, there are basically two techniques for implementing the intersection approach: enumerating transaction sets, as it is done in the Carpenter algorithm [11], and a cumulative scheme, adopted by the IsTa algorithm.

Pan et al. [11] proposed Carpenter, a method for finding closed patterns in high-dimensional biological datasets, which integrates the advantages of vertical data formats and pattern growth methods. By converting data into vertical data format {item: TID_set}, the TID_set can be viewed as row set and the FP-tree so constructed can be viewed as a row enumeration tree. Carpenter conducts a depth-first traversal of the row enumeration tree, and checks each row set corresponding to the node visited to see whether it is frequent and closed.

Technically, the task to enumerate all transaction index sets is split into two sub-tasks: (1) enumerate all transaction index sets that contain the index 1 and (2) enumerate all transaction index sets that do not contain the index 1. These sub-tasks are then further divided w.r.t. the transaction index 2: enumerate all transaction index sets containing (1.1) both indices 1 and 2, (1.2) index 1, but not index 2, (2.1) index 2, but not index 1, (2.2) neither index 1 nor index 2, and so on.

# CHAPTER 3
# ISTA ALGORITHM

The IsTa algorithm works with an intersection of transactions approach, in which all closed item sets are kept in repository that is updated each time a new transaction is intersected with the items in the repository.

The reported experimental results conclude that for data sets with few transactions and (very) many items, as they commonly occur in gene expression analysis, IsTa achieved good performance in terms of processing time, comparable with the Carpenter algorithm, but in some data sets IsTa clearly outperforms Carpenter.

## 3.1. DEFINITION

The idea of this approach was first outlined in [23], then improved and implemented as the IsTa algorithm [12].

The overall process starts with an empty set of closed item sets, and then the transactions are processed one by one. When a new transaction is processed, the set of closed item sets is updated by adding the new transaction itself and the additional closed item sets that result from intersecting the already known closed item sets with the new transaction. In addition, the support of already known closed item sets may have to be updated.

To justify this approach formally, consider the set of all closed frequent item set sets for Smin $= 1$, that is the set

$$C(T) = \{ I \subseteq B \mid \exists S \subseteq T : S \neq 0 \wedge I = \bigcap_{t \in S} t \}$$

The set $C(T)$ satisfies the following simple recursive relation:

$$C(0) = 0,$$
$$C(T \cup \{t\}) = C(T) \cup \{t\} \cup \{I \mid \exists s \in C(T) : I = s \cap t \}.$$

In practice it is not used the minimum support Smin $= 1$as it underlies $C(T)$. But this yields to removing transactions that could lead to find frequent items because they do not reach the specified minimum support. This problem is tackle by IsTa

doing an initial pass through the transaction database to determine the frequency of the in individual items. The obtained frequencies are updated with each processed transaction, so that they always represent the number of occurrences of each item in the unprocessed transactions.

## 3.2. PREFIX TREE IMPLEMENTATION

The prefix tree is the data structure used to store the closed item set in the algorithm, its main advantage is that it enables to compute the intersections between transactions quickly, as well as merging the result with them.

```
typedef struct node {          // a prefix tree node
   int step;                   // most recent update step
   int item;                   // assoc. item (last in set)
   int supp;                   // support of item set
   struct node *sibling;       // successor in sibling list
   struct node *children;      // list of child nodes
} NODE;
```

**FIGURE 3: STRUCTURE OF THE PREFIX TREE NODES**

Figure 3 shows the structure of a node within the tree. Each node represents an item set. The field children holds the head of the list of child nodes, which are linked by the field sibling, which thus implements a sibling list. The item set that is represented by a node consists of the item stored in it plus the items in the nodes on the path from the root. The support of this item set is stored in the field supp. In order to avoid duplicate representations of the same item set, all children of a node must refer to items with lower codes than the item referred to by the node itself. In addition, the items referred to by the nodes in a sibling list are in descending order w.r.t. their codes. Finally, the field step is used in the intersection process to indicate whether the support of a node has already been updated from an intersection or not.

The algorithm works on the prefix tree as follows:

1. Start with an empty tree (consisting only of a root node, which does not store any item and thus represents the empty set) is created.

2. Process the transactions one by one.

3. Each new transaction is first simply added to the prefix tree. Any new nodes created in this step are initialized with a support of zero.

4. Intersect new transactions with all sets represented by the current prefix tree. This is achieved with a recursive procedure that is basically a selective

depth-first traversal of the prefix tree and matches the items in the tree nodes with the items of the transaction.

```
void isect (NODE *node, NODE **ins)
{                                    // intersect with transaction
  int i;                             // buffer for current item
  NODE *d;                            // to allocate new nodes
  while (node) {                     // traverse the sibling list
    i = node->item;                   // get the current item
    if (trans[i]) {                   // if item is in intersection
      while ((d = *ins) && (d->item > i))
        ins = &d->sibling;            // find the insertion position
      if (d                          // if an intersection node with
        && (d->item == i)){          // the item already exists
          if (d->step >= step) d->supp--;
          if (d->supp < node->supp)
            d->supp = node->supp;
          d->supp++;                 // update intersection support
          d->step = step; }          // and set current update step
      else {                         // if there is no corresp. node
        d = malloc(sizeof(NODE));
        d->step = step;              // create a new node and
        d->item = i;                  // set item and support
        d->supp = node->supp+1;
        d->sibling = 0ns; *ins = d;
        d->children = NULL;
      }                              // insert node into the tree
      if (i <= imin) return;          // if beyond last item, abort
      isect(node!children, &d!children); }
    else {                            // if item is not in intersection
      if (i <= imin) return;          // if beyond last item, abort
      isect(node->children, ins);
    }                                // intersect with subtree
    node = node->sibling;            // go to the next sibling
  }                                  // end of while (node)
}                                    // isect()
```

**FIGURE 4: CODE OF THE INTERSECTION FUNCTION**

Figure 4 Illustrates the core steps of the intersection function (for a detailed version refer to the source code).

Each call of the function isect traverses a sibling list, the start of which is passed as the parameter node. The parameter ins points to the location in the prefix tree that represents the item set that resulted from intersecting the already processed part of the current transaction with the item set represented by node. Hence it indicates the location where new nodes have to be inserted in order to represent new closed item sets that result from intersections of the subtree rooted at node with (the unprocessed part of) the current transaction.

The procedure works as follows:

1. Whenever the item in the current sibling equals an item in the transaction, it is checked whether the sibling list starting at *ins contains a node with this item.
2. If such a node exists, its support is updated. The step field determines whether the current transaction was already counted for the support in the node or not.
3. If a node with the intersection item does not exist, a new node is allocated and inserted into the tree at the location indicated by ins.

4. The subtree is processed recursively, unless the item of the current node is not larger than the minimum item in the current transaction. In this case no later siblings can produce any intersection, because they and their descendants refer to items with lower codes.

# CHAPTER 4
# IMPROVING ISTA

Crucial to the efficiency of the main strategy presented by the IsTa algorithm is the choice of the data structure employed to represent the item sets. On previous section we discuss the current implementation based on a prefix tree to represent the dataset built on the set of transactions, with items sorted by decreasing support.

The advantage of using the prefix tree is substantial for dense datasets because of the compression achieved by merging common prefixes, but in the worst case, when the dataset is highly sparse, the number of nodes may be close to the size N of the original dataset (i.e., the sum of all transaction lengths). Since each node of the tree stores an item, a count value, which indicates the number of transactions sharing the prefix found along the path from the node to the root, plus other information needed for navigating the tree (e.g., pointers to the children and/or to the sibling), the overall space taken by the tree may turn out to be $\alpha N$, where $\alpha$ is a constant greater than 1. For these reasons, it has been suggested to store the item sets to a compressed tree, a Patricia tree [8].

## 4.1. PATRICIA TREES

Patricia stands for "Practical Algorithm to Retrieve Information Coded in Alphanumeric"; this data structure was introduced by Donald R. Morrison, 1968 [24].

A Patricia tree is a compressed tree that uses common sequences of characters as a starting point for collapsing tree branches. Each node with only one child is merged with its child; as a consequence every internal node has at least two children. Unlike in regular tries, edges can be labeled with sequences of characters as well as single characters.

As a result of this compression, Patricia tries are "denser" than regular tries. That is specifically the reason why operations on the structure, e.g. addition, removal, lookup, prefix lookup, and its memory consumption are more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes.

.

## 4.2. Construction of the Patricia tree

The overall construction strategy is as follows:

1. Starts from an initial empty tree.
2. Insert one transaction at a time in it. Initially set the item support as zero.
3. To insert a transaction *t*, the current tree is traversed downwards along the path that corresponds to the prefix shared by t with previously inserted transactions.
4. Intersect to update the support count, and also insert the item sets generated during the insertion.

```
typedef struct patnode {          // a PATRICIA tree node
   int size;                      // number of items
   struct patnode *sibling;       // successor node in sibling list
   struct patnode *children;      //  list of child nodes
   PATDATA data[1];               // associated items and support
} PATNODE

typedef struct {
   int item;          // assoc. item
   SUPP_T supp;       // support of item
   int step;          // most recent update step
}PATDATA;
```

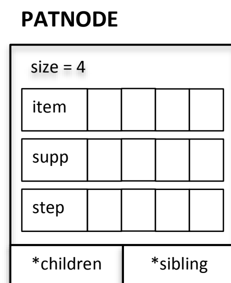



**Figure 5: Structure of the Patricia tree node**

Figure 5 shows the structure of a patricia node (patnode). Each node represents a sequence of items in an item set. The patnode can hold multiple or single items according with the prefix shared by transaction inserted previously, the number of the items at a particular node is stored in the field *size*. The field *children* holds the head of the list of child nodes, which are linked by the field *sibling*, which thus implements a sibling list. The individual information about the items represented in the node is defined by the *PATDATA* structure that contains the item, its associated support stored in the field *supp*. And the field *step* is used in the intersection process to indicate whether the support of a node has already been updated from an intersection or not.

The item set that is represented by a node consists of the items stored in it plus the items in the nodes on the path from the root. In order to avoid duplicate representations of the same item set, all children of a node must refer to items with

lower codes than the item referred to by the node itself. In addition, the items referred to by the nodes in a sibling list are in descending order w.r.t. their codes.

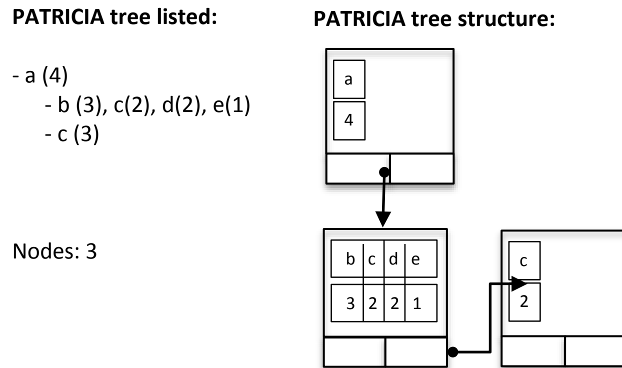



**FIGURE 6: PATRICIA TREE EXAMPLE**

The patricia tree on figure 6 was built using the transaction database from figure 1 with a minimum support equal to 1, if we recall, it consisted in 4 different transactions from an item base of 5 elements. The resulting representation consists in 3 nodes.

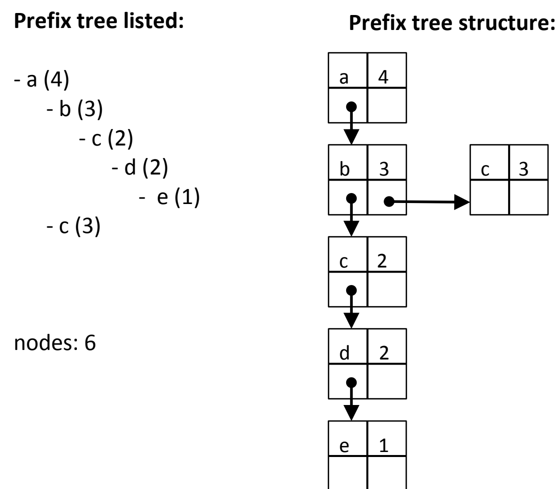



**FIGURE 7: PREFIX TREE EXAMPLE**

Figure 7 shows the representation of the same transaction database but using a prefix tree. The prefix tree has 6 nodes, each for a distinct item in the path. As we can observe the main difference between these representations is that at a patricia tree none of its nodes can have a single child, thus yielding the compression of the nodes {b}, {c}, {d}, {e} into a single node {b, c, d, e}.

### 4.2.1. INSERTING TRANSACTIONS

In the insertion function new transactions are added to the tree by recursively traversing nodes from the tree. But before describing the code, it is important to review some of the cases that can be found when inserting a transaction.

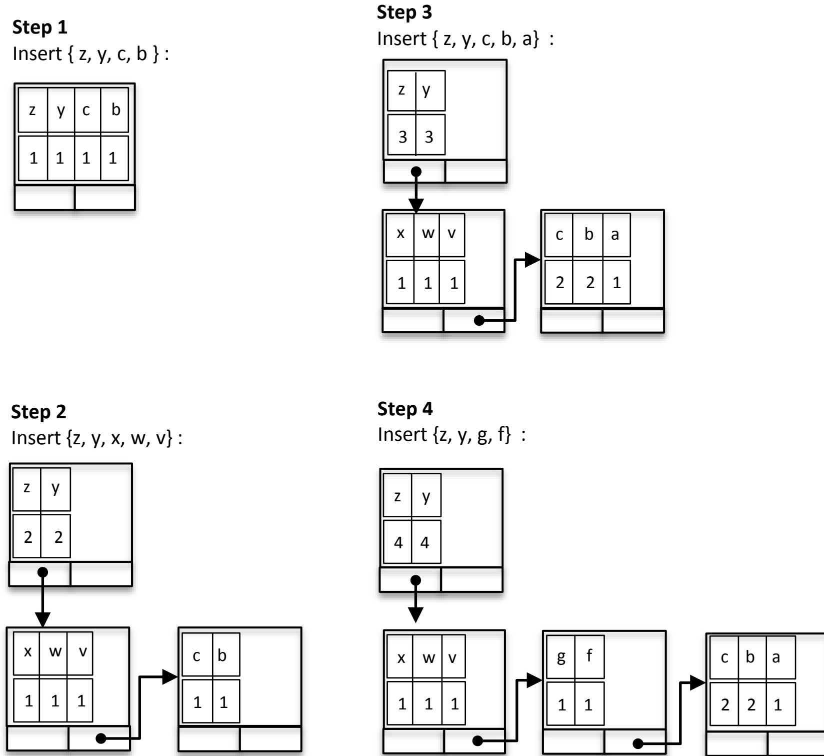



**FIGURE 8: INSERTION EXAMPLES**

Figure 8 illustrate the result of inserting new transactions. The Step 1 is the basic case in which the tree is empty; a node is created with all of the items contained in the new transaction. In the Step 2, the new items partially match the items in the node, as a consequence the items shared are kept in the node, and then 2 new nodes are created; one node containing the remaining items in the new transaction, and the second node containing the not matched items at the node. The nodes are ordered by decreasing order according with the first item at the nodes. Finally the node at first place is set as the child node of the existing node, and the second node is set as the sibling of the first.

At Step 3 the items in the new transaction match a path within the tree except for the last item; which is not contained in the path. In this case the missing item is simply added to the node.

In Step 4 the items in the new transaction partially match a path within the tree; resulting in the creation of a new node with the not matched items. The new node is set at the children level of the node containing the matched items, and then the position at sibling list is determined following the decreasing order at the first item.

```
void add (PATNODE *patn, int *items, int n)
{                                            /* --- add item set to patricia tree */
 PATNODE *splitNode, *diffNode, *node        /* pointers to insertion position */
 int j;                                      /* */
 node = patn->children;                      /* Point node to children of patn */
 if (!node){                                 /* If node doesn't exist*/
  diffNode = (PATNODE*)malloc(sizeof(PATNODE) +(n-1)*sizeof(PATDATA));
  for (j=0; j<n; j++){                       /* Create a new PATNODE diffNode*/
   store each item and its support }         /* and store the new items*/
  patn->children = diffNode;}                /* */
 else{                                       /* If node exists */
  while (node->sibling &&                    /* Traverse the list of siblings until */
   node->sibling->data[0].item >= items[0]){/* find the approppiate insertion position */
   node = node->sibling;}                    /* */
  if (node->data[0].item != items[0]){       /* If the item is unique at sibling list*/
   diffNode = malloc(sizeof(PATNODE));       /* Create a new PATNODE diffNode */
   for (j=0; j<n; j++){                      /* and store the new items */
    store each item and its support          /* Determine the insertion position*/
   }                                         /* (rigth/left)with respect to node according */
  }                                          /* with the decreasing order in sibling list*/
  else{                                      /* Some items at node might match with new items */
   firstDiff( &diff, &foundDiff, node, items, n); /* Compare items at nodes vs. the new items and */
   if (foundDiff == 1)                       /* return if they're different and at which position*/
          {                                  /* If found difference*/
    if (node->size >= diff){                 /* Process the left items at node */
     splitNode = malloc(sizeof(PATNODE));    /* Create a new PATNODE spltNode*/
     for (j=0; j < (node->size - diff); j++){ /* and store the not matched items a node*/
      store each item and its support}}      /* */
    if (n >= diff){                          /* Process the left items at insert node */
     diffNode = malloc(sizeof(PATNODE));     /* Create a new PATNODE diffNode,*/
     for (j=0; j< (n - diff); j++){          /* store the not matched new items*/
      store each item and its support}       /* and determine the insertion position*/
     }                                       /* at sibling list */
    node->size = diff;                       /* Modify the node size with number of matched */
    node = (realloc(sizeof(PATNODE));        /* elements and reduce the space allocated */
   else{                                     /* If NOT found difference*/
    if (n != diff){                          /* and there are still new items to process*/
     if (node->size == diff){                /* */
      if (node->children){                   /* if the node has children*/
       add(pat, node, items+diff, n-diff, supp, step);}/* process the left items recursively*/
      else{                                  /* if the node does't have children*/
       node = (PATNODE*)realloc(node, sizeof(PATNODE) +(n)*sizeof(PATDATA));
       for (j = diff; j<n; j++){             /* allocate more memory for new items */
        store each item and its suppor}      /* and store the new items in the node*/
       node->size = n;}                      /* update the number of items at node*/
} } } } }
```

**FIGURE 9: CODE OF THE ADD FUNCTION**

Figure 9. Illustrates the core steps of the add function (for a detailed version refer to the source code) . The function is fed with a new transaction, which items are kept in *items and the item count in 'n'; and the node at which we want to start the search is patn. 'node' points to the children of patn. So we actually start searching at children level of patn.
If there isn't a node at that level then we allocate all the items of the new transaction in diffNode, and finally, set diffNode as child of patn, and the function is over.

In case there is a node at children level, that means we need to compare the items in the transaction against the items in 'node'. First it is determined if there is a node in the sibling list whose first item equal to the first item in the new transaction, so we traverse the sibling list. The first item at both lists serves as an index that indicates the right position to insert the items.

If the first item didn't matched, all items in the new transaction are allocated in diffNode, next, we determine the insertion position of diffNode within the sibling list and the respective pointers are updated.

In case there the first item matched, it means that the items from both lists partially matched, and we need to continue the search in the current node. The items are compared one by one until we find a difference; this is calculated by the firstDiff function. From there, we have two cases:

- If there are items that didn't matched at both lists; then we need to split 'node'. Keep in 'node' the matched items, then create two nodes and allocate the left items at 'node' and in the transaction. And update the respective pointers.

- If there are items that didn't matched only at transaction list; Process the not matched items recursively if the node has children, else, just add the missing items to 'node'.

### 4.2.2. INTERSECTING TRANSACTIONS

The intersection function is the key step in IsTa. In this step the support of the frequent items are updated and also new frequent items sets are discovered. We'll review step by step how this function works through the following example, shown in the figure 10.

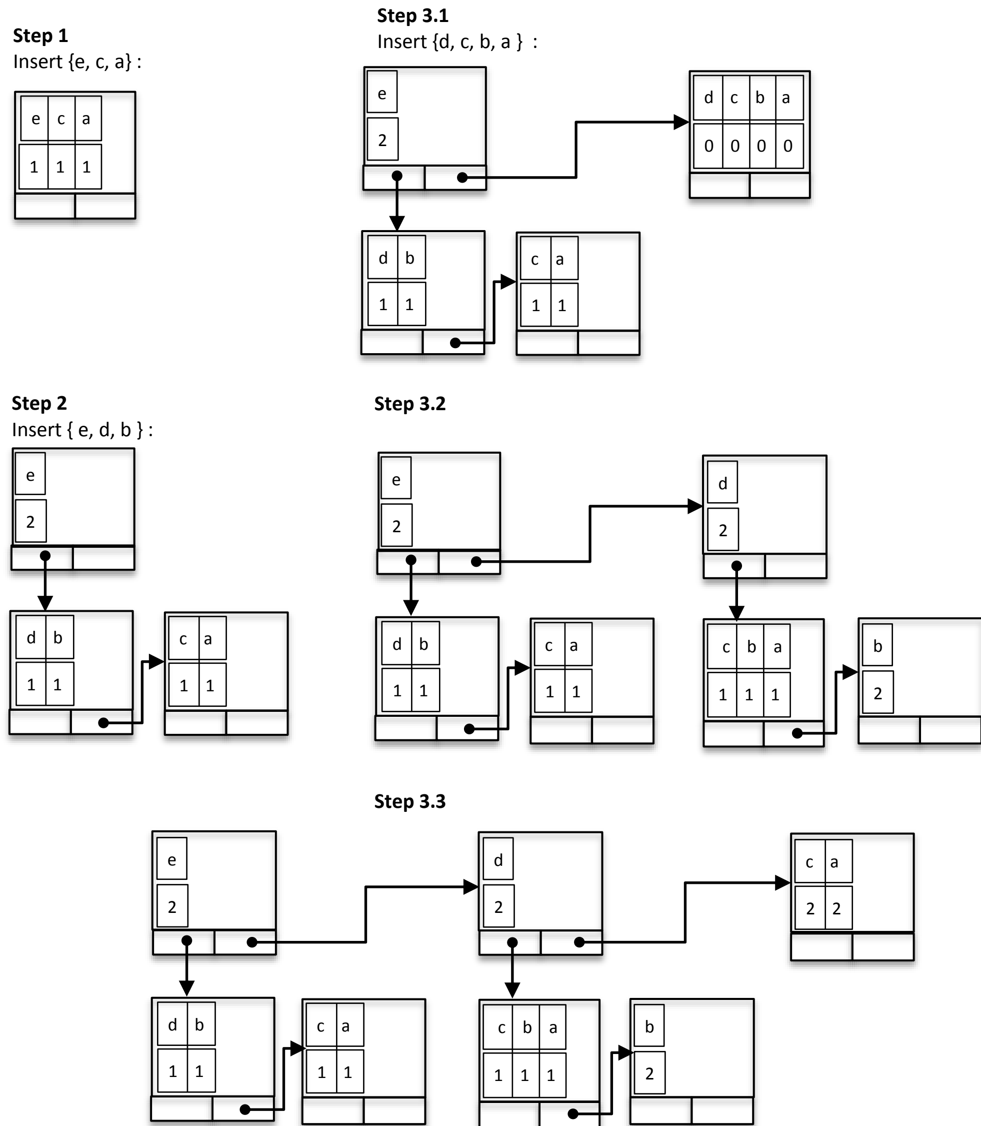



**FIGURE 10: AN EXAMPLE OF INTERSECTION**

Initially the tree is empty, then in step 1 the transaction {e, c, a} is added to the tree. Since it was empty before, no new intersections have to be taken care of, so no further processing is carried out. In step 2 the transaction {e, d, b} is added. This is done in two steps: first the two nodes on the bottom left are added to represent the transaction. Then, in the recursion, it is discovered that the new transaction overlaps with the one that was already present on the item e. As a consequence, the support value of the node that represents the item set {e} is incremented to 2. In the third step the transaction {d, c, b, a} is processed. Since things are more complicated now, we split the processing into three steps. In step 3.1 the new transaction is added to the tree, with new nodes initialized to support 0. Steps 3.2 and 3.3 show how the tree is extended by the two intersections {d, b} = {e, d, b}∩{d, c, b, a} and {c, a} = {e, c,

a}∩{d, c, b, a}. Note that the increment of the counters in the nodes for the full transaction {d, c, b, a} is not shown as a separate step.

```
int isect(PATNODE *patn, int Nlower, int Nupper, int *items, int Ilower, int Iupper, PATNODE ins, int
*isectItems)
{
 PATNODE *node;
 if (Ilower== Iupper){                          // If there are no more items to intersect
  if (setIns == 1 && pos>0){                    // Insert to tree new elements found at intersection
   isectAdd (pat, ins,isectItems, pos, isectSupp, step, supp);} //
 }                                              // Else if there are more items to intersect then
 else if (Nlower == Nupper){                    // If all elements at *patn have been compared
  if (patn->children){                          // Intersect recursively the left *items with patn children
   isect(pat, patn->children, 0, patn->children->size, items, Ilower, Iupper, ins, isectItems);
   for (node= patn->children->sibling; node; node= node->sibling){ // Traverse the sibling list and intersect
    isect(pat, node, 0, node->size, items, Ilower, Iupper, ins, isectItems);} }// recursively the missing items
  else{                                         // If there are not children at *patn, then insert to tree
    isectAdd (pat, ins,isectItems, pos, isectSupp, step, supp);}}// new elements found at intersection
 }
 else{                                          // If there are still elements to proces at *patn
  if (patn->data[Nlower].supp < pat->mins[patn->data[Nlower].item]) {  // If item to intersect doesn't
   isect(pat, patn, Nlower, Nupper, items, Ilower+1, Iupper, ins, isectItems)} // have the minimum
  else {                                        // support, skip the item
   if (patn->data[Nlower].item == items[Ilower]){ // If the item to intersect match an item in the tree,
    isectItems[pos]= items[Ilower];             // Save the item
    if (patn->data[Nlower].step != step){// Verify that the support hasn't being updated at current step
      patn->data[Nlower].supp = patn->data[Nlower].supp + supp;// update its support
      patn->data[Nlower].step = step;}          // and set last modification to current step
    isect(pat, patn, Nlower+1, Nupper, items, Ilower+1, Iupper, ins, isectItems);
   }                                            //Intersect recursively the missing *items
   else if (patn->data[Nlower].item < items[Ilower])
   {                                            // Skip the element at *items since it is > than item at *patn
    isect(pat, patn, Nlower, Nupper, items, Ilower+1, Iupper, ins, isectItems);
   }                                            //Intersect recursively the missing *items
   else if (patn->data[Nlower].item > items[Ilower])
   {                                            // Skip the element at *patn since it is > than item at *items
    isect(pat, patn, Nlower+1, Nupper, items, Ilower, Iupper, ins, isectItems);
   }                                            //Intersect recursively the missing *items
  }
 }
 return 0;}
```

**FIGURE 11: FUNCTION TO ADD NEW TRANSACTION TO TREE**

Figure 11. Illustrates the core steps of the intersection function (for a detailed version, refer to the source code). In this function, the nodes and their item list are traverse recursively in order to be intersected with the items in a new transaction. The parameters introduced are the node to intersect (patn), the index variables (Nlower, Nupper) that help to travel over the items, the new transaction (*items) and its index variables (Ilower, Iupper), *ins that will point to an insertion position, and finally, *isectItems that stores items that result from the intersection.

First it is verified if all transaction items have been intersected, if so, the items that result from the intersection are added to the tree.

Else, if there are still transaction items that have to be intersected, we proceed to verify if all the items at 'patn' have been processed. If it is the case, the intersection process continues recursively with the children nodes of patn.

Else, we know that there still items, in both lists, which have to be intersected. If the node item is equal to the transaction item (at current indexes); the item support is increased if it hasn't been updated in current step. The matched item is stored in isectItems to be later added to the tree.

In the next part, we take advantage that the items are sorted in decreasing order. Hence, if the node item is greater than the transaction item, the node item is skipped, and the next node item is intersected recursively.

If the node item is lower than the transaction item, the transaction item is skipped, and the following transaction item is processed recursively.

# CHAPTER 5
# EXPERIMENTAL EVALUATION

In this section we present the experimental evaluation of our IsTa implementation with Patricia trees and compare its result against the original version built with Prefix trees. We assess the algorithms performance with three different datasets.

For the experimental analysis, the performance measure is the execution time of the algorithms with different minimum support thresholds. In addition, we studied the memory usage and the number of nodes in the resulting trees.

## 5.1. GENE EXPRESSION ANALYSIS

Gene expression is the process of transcribing a gene's DNA sequence into mRNA sequences, which are later translated into amino acid sequences of proteins.

Microarray technologies provide the opportunity to measure the expression levels of tens of thousands of genes in cells simultaneously which is correlated with the corresponding protein made either under different conditions or during different time spots. Gene expression profiles generated by microarrays can help us understand the cellular mechanism of biological process. For instance, it provides information about the cancerous mutation of cells: which genes are most responsible for the mutation, how they are regulated, and how experimental conditions can affect cellular function. With these advantages, microarray technology has been widely used in post-genome cancer research studies.

A key task to derive biological knowledge from gene expression data is to detect the presence of sets of genes that share similar expression patterns and common biological properties. Frequent item set mining has proved to be very efficient for the integrative analysis of such data [26, 27].

In this project we have used three different datasets. A gene expression profiling of skin from 15 people that comprises of 37,777 distinct features.

The yeast dataset provided in [28] which contains expression profiles for 9,948 transcripts corresponding to 300 diverse mutations and chemical treatments in Saccharomyces cerevisiae (baker's yeast).

The Webview-1 data set, which was derived from click streams of the online shop of a leg-care company that no longer exists and has been used in the KDD cup 2000 [29]. We used the transposed form to obtain a data set with many items and few transactions, as this is the type of data set the algorithms presented here are designed for. At the end, it comprises 497 items and 59,602 transactions.

## 5.2. EXPERIMENTAL RESULTS

In this section, we discuss results of several experimentations of our proposed data structure over different datasets. The goal of these experimentations is to find out the extent of the advantages thrown by the Patricia tree over the current implementation.

The experiments were performed on a 2.1GHz Intel Core i3 virtual machine with 1003.6MB of memory, running Fedora Linux. Our version of Patricia tree was implemented in C, as well as the Prefix tree version coded by Christian Borgelt [12].

The first experiment is about the memory usage and the number of nodes in the trees. For this experiment, the measures are based on the resulting trees from both implementations with different minimum support thresholds. The memory usage is calculated according to the number of nodes. In the case of the Prefix tree the size for each node is fixed; each node is 20 bytes. The node size in a Patricia tree varies according to the number of items held. An item is assumed to fit in 12 bytes, plus an overhead per node of 15 bytes, which are needed to store the item count and the pointers to the children and sibling. Finally, for both of the trees we add 4 bytes per node for the administrative overhead of the memory management.

The results are shown in Figure 12. In this figure, there are six charts; three of them correspond to the memory usage (in megabytes) with respect to each minimum support for both algorithms in the three datasets (skin, yeast and webview1). And the other three describe the number of nodes in the final tree for each minimum support.

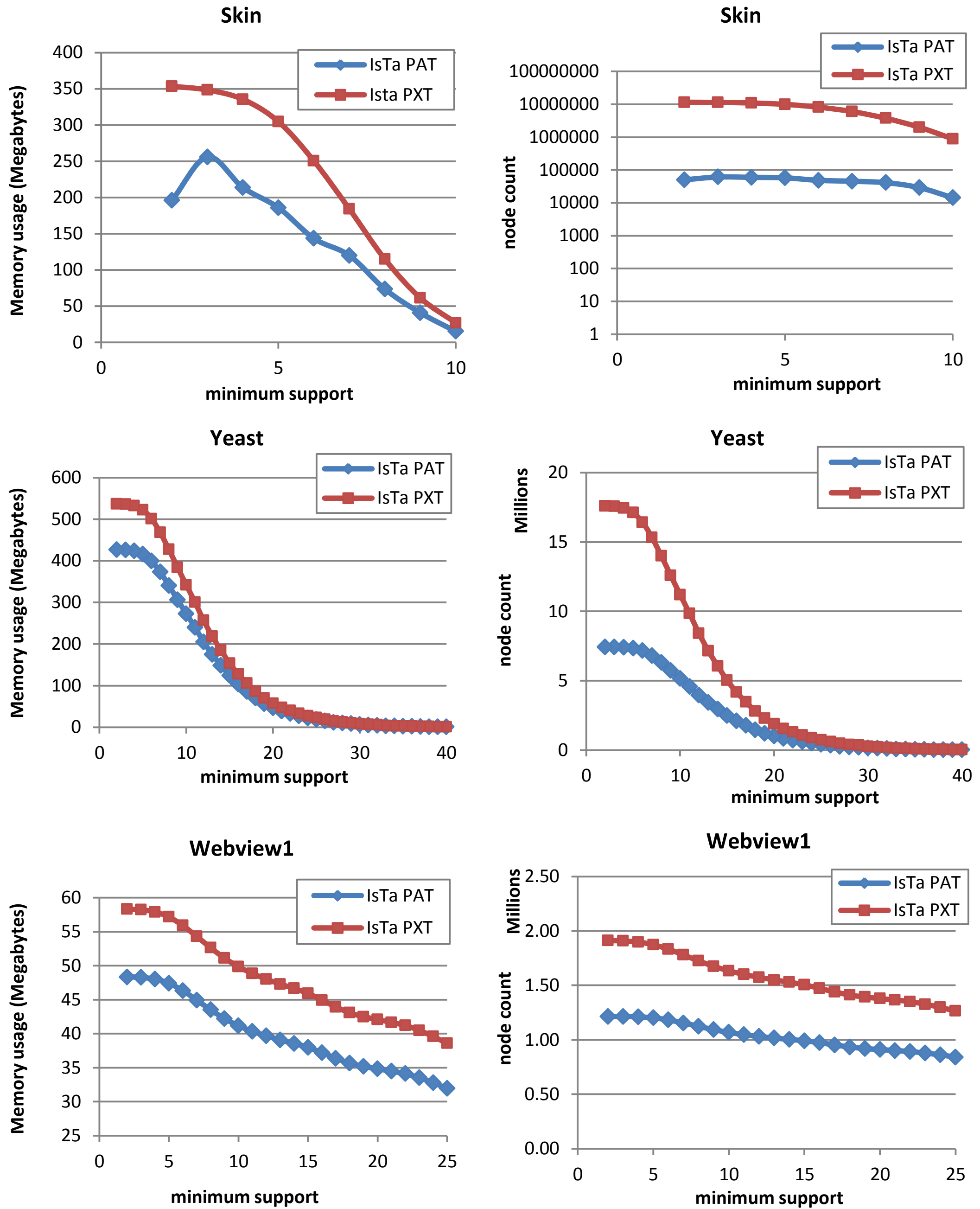


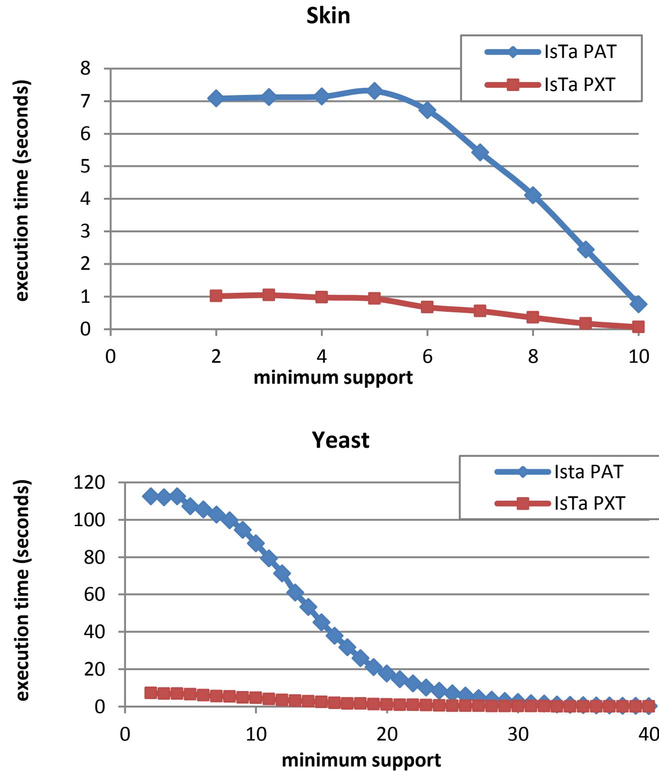

FIGURE 12: MEMORY USAGE PERFORMANCE

As shown in Figure 12, memory usage of the Patricia tree is lower than the Prefix tree representation, especially when the minimum support is small. This is true for all datasets and it is due to the more compact data structure of the Patricia tree. The Patricia tree has far smaller number of nodes and thus has better memory usage. Therefore, the Patricia tree outperforms the Prefix tree in terms of memory usage.

It is important to remark that even for sparse datasets, which exhibit a moderate sharing of prefixes among transactions, the total number of items stored in the tree may turn out much less than N, and if the number of transactions is M<< N, as is the case of the three datasets, the Patricia tree becomes very space efficient.

The second experiment compares the run time of Patricia tree against the Prefix tree. Figure 13 shows the result of this experiment for different minimum support thresholds. In this Figure, for the three charts, horizontal axis show minimum support values and vertical axis shows the run time measured in seconds.

As shown in figure 13, the Prefix tree implementation is more efficient than the Patricia tree implementation for the three datasets. As the minimum support decreased, the efficiency of the Prefix tree implementation is more revealed and the performance gap becomes more significant.



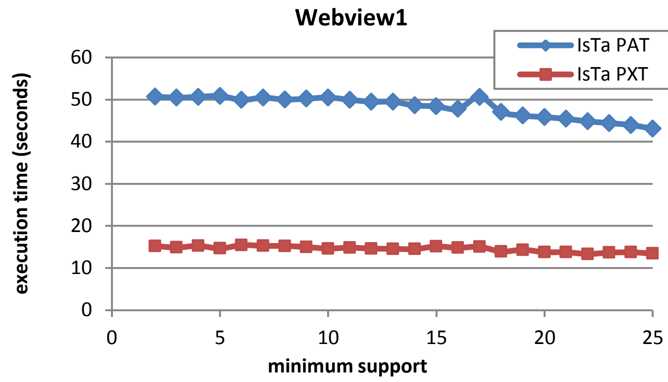



**FIGURE 13: RUNTIME COMPARISON OF THE DATA STRUCTURES**

Figure 13 shows that the Prefix tree is faster than the Patricia tree implementation even for low support thresholds where the number of generated frequent item sets is high.

Although the Patricia tree provides a space efficient data structure for representing the item set, its actual construction may be rather costly, thus influencing the overall performance of the algorithm.

# CHAPTER 6
# CONCLUSIONS

As world is now in its digital era, huge amounts of data are generated every day, and a real challenge is to discover knowledge from it. High-dimensional data, such as images, handwriting and gene expression profiles, are becoming common, and thus analyzing and handling such kinds of data have become an issue of great practical importance. Revealing the hidden patterns, efficiently and effectively, in high-dimensional data imposes a greater challenge on data mining.

In this project, we have proposed a modified version of IsTa, which was developed by [12]. In our version, we use Patricia trees as the underlying data structure to replace the Prefix tree used in the original version. The objective was to test this new implementation with a gene expression dataset and determine whether Patricia trees yield advantages over the current implementation. We found that although the Patricia tree provides a space efficient data structure for representing the item set, its actual construction may be rather costly, thus influencing the overall performance of the algorithm. Still there are areas of opportunity for performance improvements on the side of the code as well as the overall Patricia tree design.

While we focus on gene expression data, this data mining techniques can be applied to other kinds of high-dimensional data with similar characteristics.

# BIBLIOGRAPHY

[1] Usama Fayyad and Gregory Piatetsky-shapiro and Padhraic Smyth, From Data Mining to Knowledge Discovery in Databases, AI Magazine, 1996, vol 17 pages {37-54}

[2] Data Mining Curriculum. ACM SIGKDD. 2006-04-30. Retrieved 2011-10-28.

[3] R.R. Bouckaert; E. Frank; M.A. Hall; G. Holmes; B. Pfahringer; P. Reutemann; I.H. Witten (2010). "WEKA Experiences with a Java open-source project". Journal of Machine Learning Research 11: 2533–2541.

[4] Kantardzic, Mehmed (2003). Data Mining: Concepts, Models, Methods, and Algorithms. John Wiley & Sons. ISBN 0-471-22852-4. OCLC 50055336.

[5] Proceedings, International Conferences on Knowledge Discovery and Data Mining, ACM, New York.

[6] SIGKDD Explorations, ACM, New York.

[7] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. In \U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, Advances in Knowledge Discovery and Data Mining, pages 307{328. AAAI Press / MIT Press, Cambridge, CA, USA, 1996.

[8] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In Proc. 20th Int. Conf. on Very Large Databases (VLDB 1994, Santiago de Chile), pages 487{499, San Mateo, CA, USA, 1994. Morgan Kaufmann.

[9] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD'97, Newport Beach, CA), pages 283-296, Menlo Park, CA, USA, 1997. AAAI Press.

[10] J. Han, H. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In Proc. Conf. on the Management of Data (SIGMOD'00, Dallas, TX), pages 1{12, New York, NY, USA, 2000. ACM Press.

[11] F. Pan, G. Cong, A. Tung, J. Yang, and M. Zaki. Carpenter: Finding closed patterns in long biological datasets. In Proc. 9th ACM SIGKDD Int. Conf. on

Knowledge Discovery and Data Mining (KDD 2003, Washington, DC), pages 637-642, New York, NY, USA, 2003. ACM Press.

[12] Christian Borgelt, Xiaoyuan Yang, Ruben Nogales-Cadenas, Pedro Carmona-Saez, and Alberto Pascual-Montano. 2011. Finding closed frequent item sets by intersecting transactions. InProceedings of the 14th International Conference on Extending Database Technology(EDBT/ICDT '11)

[13] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In Proc. 7th Int'l Conf. Database Theory (ICDT), 1999.

[14] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In Proc. ACMSIGMOD Int'l Workshop Data Mining and Knowledge Discovery (DMKD), 2000.

[15] M. J. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed association rule mining. In Proc. SIAM Int'l Conf. on Data Mining (SDM), 2002.

[16] R. Srikant, R. Agrawal: "Mining Generalized Association Rules", Proc. of the 21st Int'l Conference on Very Large Databases, Zurich, Switzerland, September 1995. Expanded version available as IBM Research Report RJ 9963, June 1995.

[17] M.J. Zaki. Scalable algorithms for association mining. IEEE Transactions on Knowledge and Data Engineering, 12(3):372–390, May/June 2000.

[18] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In D. Heckerman, H. Mannila, and D. Pregibon, editors, Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, pages 283–286. AAAI Press, 1997.

[19] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad. Depth first generation of long patterns. In Ramakrishnan et al. [32], pages 108–118.

[20] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad. A tree projection algorithm for generation of frequent itemsets. Journal of Parallel and Distributed Computing, 61(3):350–371, March 2001.

[21] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Mining and Knowledge Discovery, 2003.

[22] C. Borgelt and X. Wang, SaM: A Split and Merge Algorithm for Fuzzy Frequent Item Set Mining. ;In Proceedings of IFSA/EUSFLAT Conf.. 2009, 968-973.

[23] T. Mielikainen. Intersecting data to closed sets with constraints. In Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL), Aachen, Germany, 2003. CEUR Workshop Proceedings 90.

[24] D. R. Morrison. PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric. Jrnl. of the ACM, 15(4) pp514-534, Oct 1968.

[25] G. H. Gonnet. Handbook of Algorithms and Data Structures. Addison-Wesley, International Computer Series, pp 109, 1984

[26] P. Carmona-Saez, M. Chagoyen, A. Rodriguez, O. Trelles, J. M. Carazo, and A. Pascual-Montano. Integrated analysis of gene expression by association rules discovery. BMC Bioinformatics, 7:54, 2006.

[27] C. Ceighton and S. Hanash. Mining gene expression databases for association

[28] T. Hughes, M. Marton, A. Jones, C. Roberts, R. Stoughton, C. Armour, H. Bennett, E. Coffey, H. Dai, Y. He, M. Kidd, A. King, M. Meyer, D. Slade, P. Lum, S. Stepaniants, D. Shoemaker, D. Gachotte, K. Chakraburtty, J. Simon, M. Bard, and S. Friend. Functional discovery via a compendium of expression profiles. Cell, 102:109-126, 2000.

[29] R. Kohavi, C. Bradley, B. Frasca, L. Mason, and Z. Zheng. Kdd-cup 2000 organizers' report: Peeling the onion. SIGKDD Exploration, 2:86-93, 2000.

[30] A. Pietracaprina and D. Zandolin, "Mining Frequent Itemsets using Patricia Tries," in Proc. 1st FIMI Workshop. Frequent Itemset Mining Implementations, Florida, 2003.

[31] Bart Goethals. Survey on frequent pattern mining. Technical report, Helsinki Institute for Information Technology, 2003.