TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

Extending the Order Preserving Submatrix: New patterns in datasets

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree in the School of Computer Science, Tel-Aviv University

by

Sagi Shporer

The research work for this thesis has been carried out at
Tel-Aviv University
under the supervision of Prof. Amos Fiat

May 2004

Abstract

This paper concerns in finding local patterns in gene expression datasets. We present new order relation patterns, and develop algorithms which finds those pattern. Our algorithms are the first algorithms to find the exact results for those patterns, yet in most cases they outperforms existing heuristical algorithm. Finally we present an algorithm for the broader problem of frequent itemset mining.

The patterns we investigate are order relations that are localized to subset G of the genes and subset T of the tissues. We introduce the following local order relation patterns: (a) patterns in data that contains errors, such as expected in probed data, (b) patterns between groups of tissues, (c) patterns in layered submatrices and (d) dynamic order relation patterns mining, without predefined pattern mask.

The above patterns are similar to patterns found by frequent itemset mining (FIM) algorithms. We present an algorithm for mining frequent itemsets, AIM-$\mathcal{F}$ (see also [FS03]). Past studies have proposed various algorithms and techniques for improving the efficiency of the mining task. We integrate a combination of these techniques into an algorithm which utilize those techniques dynamically according to the input dataset. The algorithm main features include depth first search with vertical compressed database, diffset, parent equivalence pruning, dynamic reordering and projection.

We introduce exact and efficient algorithms for mining the order relation patterns described, based on techniques and ideas from frequent itemset mining algorithms. The algorithms include features from AIM-$\mathcal{F}$ among other optimizations for order relation problems which we introduce.

We perform extensive experimental evaluation of our algorithm on serval datasets, and compare the patterns found verses a random matrix datasets. We also do experimental testing of the AIM-$\mathcal{F}$ algorithm suggesting that our algorithm and implementation significantly outperform existing algorithms/implementations.

Acknowledgments

I would like to express my gratitude to my supervisor Prof. Amos Fiat and to Prof. Benny Chor for their invaluable help during the work on this research project. Their determination at constantly improving the results achieved have led this work to truly new regions.

Contents

Chapter 1

Introduction

The advent of DNA microarray technologies has revolutionized the experimental study of gene expression. Those high throughput technologies produce vast amount of information. The analysis of those datasets poses computational and algorithmic challenges.

One of the usual goals is identifying groups of gene according to their expression. The most common approach so far is finding global patterns in the dataset. Most frequently used are clustering techniques, which were proved to be successful in many contexts. The novel paper of [ESBB98] adopted clustering techniques to the analysis of gene expression data. The paper was followed by much research, which is reviewed in [SES02, CYBD$^+$01, GB00, BEB99].

However clustering has its limitations. Clustering solution implies that each group of genes in a cluster has a single biological function, which may be an oversimplification of the biological system. Another drawback is the difficulty in identifying patterns that are common to only small portion of the data. Other clustering approaches described in [CC00, TSS02, GLD00, LO02].

[BDCKY02] describe a problem called *Order Preserving Submatrix* (OPSM). An OPSM is a submatrix created by a subset G of the genes, a subset T of the tissues and some ordering π of T such that for every gene the values are sorted in an increasing order in respect to the order of T. The problem is to find all the OPSMs in the dataset — with $|T|$ greater than some minimum threshold on $|G|$. Example of an OPSM is shown in figure 1.2 (Mined from the dataset in figure 1.1). To find those OPSMs [BDCKY02] use a greedy heuristic.

In this paper we extend the framework of OPSM to allow more patterns:

- OPSM with Errors - The gene expression data is a probed dataset and might contain errors. We present an extension of OPSM that allow errors in the pattern. We allow up to γ values of every gene in the submatrix to violate the increasing sequence. See example in figure 1.3 (based on the dataset in figure 1.1).

genes \ tissues	c_1	c_2	c_3	c_4	c_5	c_6	c_7
r_1	1	3	4	0	6	2	7
r_2	5	2	1	3	8	4	7
r_3	9	1	8	5	7	4	3
r_4	2	6	8	4	1	0	7
r_5	0	1	4	8	2	6	5
r_6	4	2	7	1	6	0	3
r_7	6	5	1	8	0	7	3
r_8	2	0	3	9	8	5	6
r_9	0	3	5	9	1	7	2

Figure 1.1: Sample gene expression dataset

row \ column	c_5	c_3	c_6	c_4
r_5	2	4	6	8
r_7	0	1	7	8
r_9	1	5	7	9

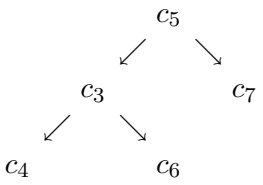
Figure 1.2: Example of OPSM for pattern (t_5, t_3, t_6, t_4) : The longest pattern with support 3

row \ column	c_2	c_3	c_7	c_6	c_4
r_3	1	**8**	3	4	5
r_5	1	4	5	6	8
r_7	**5**	1	3	7	8
r_8	0	3	6	**5**	9
r_9	3	5	**2**	7	9

Figure 1.3: Example of OPSM with Errors for pattern $(t_2, t_3, t_7, t_6, t_4)$, $\gamma = 1$: For each gene, the values are increasing with the tissues, errors are marked in bold. This is the longest pattern with support 5. Notice that for r_8 more than one error marking is possible

row \ column	$\{c_4$	$c_6\}$	$\{c_2\}$	$\{$ c_3	$c_7\}$
r_1	{ 0	2 }	{ 3 }	{ 4	7 }
r_4	{ 4	0 }	{ 6 }	{ 9	7 }
r_6	{ 1	0 }	{ 2 }	{ 7	3 }

Figure 1.4: Example of Increasing Groups Sequence of size 2 as most, pattern $(\{t_4, t_6\}, \{t_2\}, \{t_3, t_7\})$. The increasing order relation sequence is between the groups, and no order is implied inside the groups.



row \ column	c_5	c_7	c_3	c_4	c_6
r_5	2	5	4	8	6
r_7	0	3	1	8	7
r_8	1	2	5	9	7

Figure 1.5: Example of Connected DAG order relation. On the left is the graph of the pattern which implies the following order relation : $c_1 < t_7$, $c_1 < t_3$, $c_3 < t_4$ and $c_3 < t_6$. On the right we see the corresponding genes from the dataset that support this order relation.

- Increasing Groups Sequence - In the previous patterns, every tissue stand by itself. In this pattern the tissues are divided into groups and the order relation is between the groups. The size of each group is a given parameter, and the task is to find the tissues for every group, such that they are supported by $|G|$ genes or more. There is no order between the tissues in the group. See example in figure 1.4 (based on the dataset in figure 1.1).

- Layered submatrices - Similar to Increasing Groups Sequence, with a small difference, every tissue can be included only in a specified predefined group. The motivation is to divide the tissue according to some parameter, such as type of cancer, and try to find genes that create increasing sequence between the groups (layers) of the tissues. The patterns are the same as those in Increasing Groups Sequence however the search space is different.

- Connected Directed Acyclic Graph (Connected DAG) - Previous patterns are restricted to a certain order relation between the various tissues. For example in an OPSM every gene expression is larger than the gene expression in the tissue before. We present an attempt to search the space of patterns, to find significant patterns between the tissues. The pattern is best represented in a connected DAG where each edge represent an order relation. See example in figure 1.5 (based on the dataset in figure 1.1).

The patterns described above and the OPSM problem have similar characteristics to some classical data mining problems - Frequent Itemset Mining (FIM) [AIS93] and sequential patterns mining [AS95]. Those problems attracted much research in the last decade which we employ to introduce efficient mining algorithms for the patterns described. A key observation is that the patterns described can be grown from short patterns to larger ones, which is a fundamental principal in many of the frequent itemset and sequential pattern mining algorithms.

1.1 Organization of this thesis

The rest of this thesis is organized as follows. In rest of this chapter we describe our contribution and related work. Chapter 2 concerns in the formal problem definitions and NP-Completeness proofs. In chapter 3 we describe the building blocks of our algorithms. In chapter 4 we describe the mining algorithms for the order relations and analyze the complexity per output (which is polynomial). Chapter 5 concerns in the experimental evaluation of the various algorithmic building blocks and the AIM-$\mathcal{F}$ algorithm. In chapter 6 we describe the experimental evaluation of the order relation mining algorithms and analysis of the results.

1.2 Contributions of this thesis

This thesis has contribution in two distinct fields:

- **Frequent Itemset Mining** - We combine several pre-existing ideas in a fairly straightforward way and get a new frequent itemset mining algorithm. In particular, we combine the sparse vertical bit vector data structure along with the difference sets technique of [ZG03], thus reducing the computation time when compared with [ZG03]. We show how to combine diffsets and projection techniques, which were never used together before. The various techniques were put in use dynamically according to the input dataset, thus utilizing the advantages and avoiding the drawbacks of each technique.

  Experimental results suggest that for a given level of support, our algorithm/implementation is faster than the other algorithms with which we compare ourselves. This set includes the dEclat algorithm of [ZG03] which seems to be the faster algorithm amongst all others. See in Figure 1.6 run time comparisons. This figure describes running times as function of minimum support for various algorithms on a sample dataset. Our algorithms, AIM-$\mathcal{F}$, shows consistant good behavior. An extensive comparison is made in the chapter of the frequent itemset mining experiments of this thesis.

  This work has been published in [FS03].

- **Order Relation Patterns** - We introduce new types of local order relation patterns. We introduce pattern with errors, patterns in groups of columns, layered matrices and patterns with various order relation between the columns. We prove that the mining task of those patterns is hard (NP-Complete).

  We introduce and implement efficient algorithms, derived from our frequent itemset mining

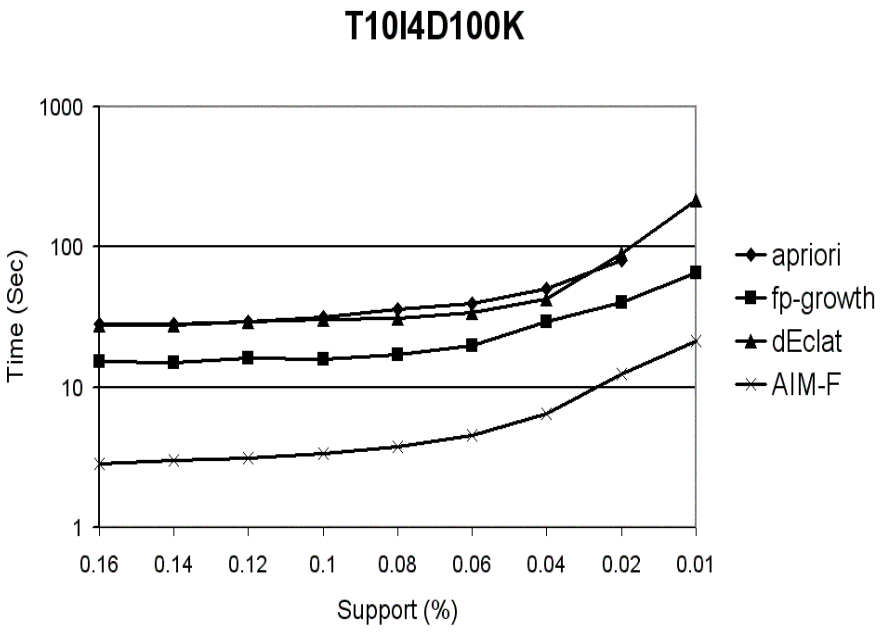


Figure 1.6: T10I4D100K dataset

algorithm, to mine order relation patterns. We introduce two new general tools, which increase the speed of the mining. These are:

- Order Relation-2 (OR2) matrix: a bit map matrix which enables fast comparison and support checking
- Groups pruning technique: Early pruning, without explicit support checking, for some type of patterns (groups)

The algorithms we introduce are exact algorithms that return all the patterns fitting to the pattern restrictions. The algorithms we introduce have polynomial complexity with the output size.

Problem	Complexity per output	Details
OPSM	$O(mn)$	
OPSM with Errors	$O(m^{\gamma+1}(\gamma+1)^2 n)$	γ - Number of Errors allowed
Increasing Groups	$O(mqn) \leq O(m^2 n)$	q - Maximal group size
Layered Submatrices	$O(mqn) \leq O(m^2 n)$	q - Maximal layer size
Connected DAG	$O(msn) \leq O(m^2 n)$	s - Longest pattern found

Extensive experimental evaluation of the algorithms on real gene expression datasets show their efficiency (even in comparison to previously reported results of heuristic algorithms).

1.3 Related Work

1.3.1 Frequent Itemset Mining

Since the introduction of the frequent itemset mining problem [AIS93] and the *Apriori* algorithm [AS94] many variants have been proposed to reduce time, I/O and memory.

The frequent itemset problem is defined as follows - Let $I = \{I_1, \ldots, I_m\}$ be a set of items. Let $D = \{t_1, \ldots, t_n\}$ be a transactional dataset such that $c_j \subseteq I$. Let minsupport some positive parameter. We call a set $\alpha \subseteq I$ an itemset. The support(α) is the number of transactions in D that contains α (*e.g.* $\alpha \subseteq t_j$). An itemset α is called frequent if support(α) $\geq$ minsupport. See figure 1.7.

TID	I			
1	A	B	C	D
2		B	C	
3	A		C	
4		B		D

support(B) $= 3$ (TIDs 1,2,4)
support(BC) $= 2$ (TIDs 1,2)
support(ABC) $= 1$ (TIDs 1)

Figure 1.7: Frequent Itemset Mining Example - On the left table is the dataset. On the right side are itemset examples and their support values.

Apriori uses breath-first search a bottom-up approach, to generate frequent itemsets. (*I.e.*, constructs $i + 1$ item frequent itemsets from i item frequent itemsets). The key observation behind Apriori is that all subsets of a frequent itemset must be frequent. This suggests a natural approach to generating frequent itemsets. The breakthrough with Apriori was that the number of itemsets explored was polynomial in the number of frequent itemsets. In fact, on a worst case basis, Apriori explores no more than m itemsets to output a frequent itemset, where m is the total number of items.

Subsequent to the publication of [AIS93, AS94], a great many variations and extensions were considered [BMUT97, LK98, Zak00]. In [BMUT97] the number of passes over the database was reduced. [LK98] tried to reduce the search space by combining bottom-up and top-down search – if a set is infrequent than so are supersets, and one can prune away infrequent itemsets found during the top-down search. [Zak00] uses equivalence classes to skip levels in the search space. A new mining technique, FP-Growth, proposed in [YC96], is based upon representing the dataset itself as a tree. [YC96] perform the mining from the tree representation.

In AIM-$\mathcal{F}$ [FS03] we build upon several ideas appearing in previous work for frequent itemset mining, a partial list of which is the following:

- Vertical Bit Vectors [SHS$^+$00, BCG01] - Instead of representing the dataset as a set of rows, the dataset is represented as a set of columns. Every column is stored as a bit vector (bit for every row) thus Vertical Bit Vector. Experimentally, this has been shown to be very effective.

- Projection [BCG01] - A technique to reduce the size of vertical bit vectors by trimming the bit vector to include only transaction relevant to the subtree currently being searched.

- Difference sets [ZG03] - Instead of holding the entire tidset at any given time, Diffsets suggest that only changes in the tidsets are needed to compute the support.

- Dynamic Reordering [Jr98] - A heuristical approach for reducing the search space - dynamically changing the order in which the search space is traversed. This attempts to rearrange the search space so that one can prune infrequent itemsets earlier rather than later.

- Parent Equivalence Pruning [BCG01, Zak00] - Skipping levels in the search space, when a certain item added to the itemset contributes no new information.

The algorithms we describe in this paper for the order relations are build on AIM-$\mathcal{F}$ basic algorithmic ideas (although some of them can not be used in that context).

We should add that there are a wide variety of other techniques introduced over time to find frequent itemsets, which we do not make use of. A partial list of these other ideas is

- Sampling - [Toi96] suggest searching over a sample of the dataset, and later validates the results using the entire dataset. This technique was shown to generate the vast majority of frequent itemsets.

- Adjusting support - [SK02] introduce SLPMiner, an algorithm which lowers the support as the itemsets grow larger during the search space. This attempts to avoid the problem of generating small itemsets which are unlikely to grow into large itemsets.

1.3.2 Sequential Mining

In parallel to the frequent itemset mining research, the sequential pattern mining [AS95] algorithms were developed. In sequential pattern mining problem, each transaction, besides itemset,

include a time stamp and user id. The task is to create a time increasing chain of itemsets that are common (frequent) in the dataset. While related to frequent itemset mining, it differs in the sense that frequent itemset mining concern in intra-transaction patterns (such as OPSM) where sequential patterns deals with inter-transaction relations.

Algorithms for mining sequential patterns employ techniques that are similar to those of frequent itemset mining. The first algorithm, introduced by [AS95] is based, partially, on techniques from [AS94]. Subsequent algorithms include SPADE [Zak01] (from the authors of dEclat [ZG03]) and SPAM [AGY$^+$02] (from the authors of MAFIA [BCG01]) which are the current state-of-the-art algorithms, based on frequent itemset mining research.

In general, the algorithms are based on two major approaches. One is dividing the task into sub-tasks, first finding frequent itemset in each transaction, than trying to build the chains ([AS95]). A second approach is building a search space, similar to the lexicographic tree, and mining over this search space ([Zak01, AGY$^+$02]). In this work we adopt an approach similar to the second one because the basic problem (OPSM) is based on a smaller search space.

Prior work regarding frequent itemset mining and sequential pattern mining have regarded the data as being without noise. Resent research had created a more flexible framework in which noise in the data is allowed. In [YWY00] a noise model was introduced to allow insertion and deletion of symbols in a repeating pattern and is further extended on [WYY01]. The algorithms proposed were simple extension of existing algorithms without any specific efficient techniques to accommodate noise.

1.3.3 Order Preserving Sub-Matrices (OPSM) Problem

[BDCKY02] introduced the OPSM pattern (described earlier in the introduction) and proved that finding OPSM is NP Complete. To mine OPSM [BDCKY02] introduced a greedy heuristic algorithm. They assume a random data model, and try to find hidden OPSM. This is done by growing patterns and evaluating the statistical probability for every partial pattern found of being part of the hidden OPSM.

The algorithm grows the patterns from both ends to the middle, until it reaches the needed length. It start by selecting first and last items in the pattern, then, iteratively, it adds items. This is done from both ends, thus keeping both end of the pattern balanced. Each iteration i starts with the ℓ patterns of length i found so far, and ends with the best ℓ patterns of length $i+1$ found in the i-th iteration. To evaluate the best patterns [BDCKY02] introduce a scoring model, based on the probability that the partial pattern found in the i-th iteration is in the hidden OPSM in the dataset.

The probability in which the algorithm discover hidden sub-matrix within otherwise random matrix was shown experimentally to be high. The success rate of the algorithm varies according to the mining dataset and the number of partial patterns stored in each step (ℓ). The complexity of the algorithm (total run time) is $O(nm^3\ell)$.

Chapter 2

Problem Definitions

In this chapter we present the the formal definitions for the order relation patterns. We also prove that the task of mining those patterns is hard (NP-Complete). For the definitions we regard the gene expression datasets as matrix, where the columns are the tissues and the rows are the genes.

2.1 Framework

Definition 2.1.1. Let D be data matrix of $n \times m$. The **Support Predicate**

$$T(r_i) \rightarrow \begin{cases} \text{true} \\ \text{false} \end{cases}$$

is a boolean function that describe the relationship between the values in a single row (r_i) of the data matrix D.

Definition 2.1.2. Let D be data matrix of dimensions $n \times m$. Let T be a Support Predicate. Let ***support***(D, T) be the number of rows $i \in \{1, \ldots, n\}$ in which $T(r_i)$ returns true. A pattern (defined by T) is called frequent if **support**$(D, T) \geq$ minsupport for some given parameter minsupport.

2.2 Support Predicates

From the above definitions we derive many problems with different support predicates. In this paper we shall focus on following variants:

Definition 2.2.1. Let $\pi = (\pi_1, \pi_2, \ldots, \pi_s)$ be a permutation of s columns of D. The **OPSM** pattern is defined by the following *support predicate*

$$T_\pi(r_i) = \text{true} \quad \Leftrightarrow D(r_i, \pi_1) < D(r_i, \pi_2) < \ldots < D(r_i, \pi_s).$$

An example of OPSM is shown in figure 2.2 (Mined from the dataset in figure 2.1)

row \ column	c_1	c_2	c_3	c_4	c_5	c_6	c_7
r_1	1	3	4	0	6	2	7
r_2	5	2	1	3	8	4	7
r_3	9	1	8	5	7	4	3
r_4	2	6	8	4	1	0	7
r_5	0	1	4	8	2	6	5
r_6	4	2	7	1	6	0	3
r_7	6	5	1	8	0	7	3
r_8	2	0	3	9	8	5	6
r_9	0	3	5	9	1	7	2

Figure 2.1: Sample gene expression dataset

row \ column	c_5	c_3	c_6	c_4
r_5	2	4	6	8
r_7	0	1	7	8
r_9	1	5	7	9

Figure 2.2: Example of OPSM for pattern (t_5, t_3, t_6, t_4) : The longest pattern with support 3

Definition 2.2.2. The **OPSM with Errors** pattern for γ errors is defined by the following *Support Predicate*

$$T_{\pi,\gamma}(r_i) = \text{true}$$
$$\Leftrightarrow \text{The sequence } D(r_i, \pi_1), D(r_i, \pi_2), \ldots, D(r_i, \pi_s)$$
$$\text{contains an increasing subsequence of length } s - \gamma$$

For example let $r_1 = \{3, 4, 2, 1\}$, $\gamma = 1$ and $\pi = \{c_1, c_2, c_3, c_4\}$. The support predicate $T_{\pi,1}(r_1) = false$ because the longest increasing sequence is of length 2, and no increasing subsequence of length 3 exists (*I.e.*, with one error).

Another example of OPSM with Errors is shown in figure 2.3 (Mined from the dataset in figure 2.1)

Definition 2.2.3. The **Increasing Groups Sequence** pattern is defined by the following *Support Predicate* - Let $\theta = \{\theta_1, \ldots, \theta_J\}$ be a *partition* of m such that $\sum_i \theta_i = m$. Let $G =$

row \ column	c_2	c_3	c_7	c_6	c_4
r_3	1	**8**	3	4	5
r_5	1	4	5	6	8
r_7	**5**	1	3	7	8
r_8	0	3	6	**5**	9
r_9	3	5	**2**	7	9

Figure 2.3: Example of OPSM with Errors for pattern $(t_2, t_3, t_7, t_6, t_4)$, $\gamma = 1$: For each gene, the values are increasing with the tissues, errors are marked in bold. This is the longest pattern with support 5. Notice that for r_8 more than one error marking is possible

$\{G_1, \ldots, G_J\}$ be a *column partition* of T with respect to θ (*I.e.*, $\forall_i |G_i| = \theta_i$) such that

$$\forall_{1 \leq \ell \leq J} G_\ell = \{\pi_k | \sum_{i=1}^{\ell-1} \theta_i < k \leq \sum_{p=1}^{\ell} \theta_p\}.$$

The *Support Predicate*

$$T_{\pi,\theta}(r_i) = \text{true}$$
$$\Leftrightarrow \forall \ell \forall \pi_p \in G_\ell \forall \pi_k \in G_{\ell+1} D(r_i, \pi_p) < D(r_i, \pi_k).$$

I.e., the permutation π is divided into J groups. The size of each group is set by θ_i. The support predicate is true iff for every group G_ℓ all the values $(D(r_i, \pi_p))$ in the group are smaller than all the values $(D(r_i, \pi_k))$ in the group $G_{\ell+1}$.

For example, let $r_1 = (3, 2, 4, 5)$, let $\theta = \{2, 2\}$, let $\pi = \{c_1, c_2, c_3, c_4\}$. The support predicate $T_{\pi,2,2}(r_1) = true$ because the following group division $\{3, 2\} \rightarrow \{4, 5\}$ defines increasing group sequences.

Another example of Increasing Groups Sequence is shown in figure 2.4 (Mined from the dataset in figure 2.1)

row \ column	$\{c_4$	$c_6\}$	$\{c_2\}$	$\{$ c_3	$c_7\}$
r_1	{ 0	2 }	{ 3 }	{ 4	7 }
r_4	{ 4	0 }	{ 6 }	{ 9	7 }
r_6	{ 1	0 }	{ 2 }	{ 7	3 }

Figure 2.4: Example of Increasing Groups Sequence of size 2 as most, pattern $(\{t_4, t_6\}, \{t_2\}, \{t_3, t_7\})$. The increasing order relation sequence is between the groups, and no order is implied inside the groups.

Definition 2.2.4. The **Layered Submatrices** pattern is defined as follows - Let ρ be a subset columns from D, $|\rho| = s$. Given partition $\Delta = \{\Delta_1, \ldots, \Delta_J\}$ of columns in D : $\forall c_i \exists \Delta_j$ s.t.

$c_i \in \Delta_j$ and $\forall i \neq j\ \Delta_j \bigcap \Delta_i = \emptyset$. The *Support Predicate*

$$T_{\rho,\Delta}(r_i) = \text{true}$$
$$\Leftrightarrow \forall \ell \forall \rho_p \in \Delta_\ell \forall \rho_q \in \Delta_{\ell+1} D(r_i, \rho_p) < D(r_i, \rho_q).$$

Notice that for this function the permutation π is the unit permutation as the order of the columns is implied by Δ.

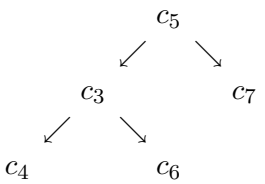
This is a special case of the Increasing Groups Sequence, where the partition is made on the original column order (and not on the column order in the permutation).

Definition 2.2.5. Connected DAG - Allowing any *Support Predicate*, and searching for the function that gives the highest support, with the following limitations -

- Every parameter in the function is in some order relation with a prior parameter (connected graph).
- For every column, at most two columns prior to the current in the relation are in order relation with the current one (graph in-degree of 2)
- For every column, the column can be in some order relation with up to two columns (graph out-degree of 2).

For example : $T_\pi(r_i) = D(r_i, \pi_1) < D(r_i, \pi_2), D(r_i, \pi_1) < D(r_i, \pi_3), D(r_i, \pi_4) < D(r_i, \pi_3)$. See also figure 1.5 for example of the graph representation of this order relation.

Another example of Connected DAG pattern is shown in figure 2.5 (Mined from the dataset in figure 2.1)



row \ column	c_5	c_7	c_3	c_4	c_6
r_5	2	5	4	8	6
r_7	0	3	1	8	7
r_8	1	2	5	9	7

Figure 2.5: Example of Connected DAG order relation. On the left is the graph of the pattern which implies the following order relation : $c_1 < t_7$, $c_1 < t_3$, $c_3 < t_4$ and $c_3 < t_6$. On the right we see the corresponding genes from the dataset that support this order relation.

2.3 Mining Problems

Definition 2.3.1. The *OPSM optimization problem*:

- Input: D, minSupport
- Output: $\pi = (\pi_1, \pi_2, \ldots, \pi_s)$ such that support$(D, T_\pi) \geq$ minSupport and s is maximal.

Definition 2.3.2. The *OPSM with Errors optimization problem*:

- Input: D, minSupport, γ
- Output: $\pi = (\pi_1, \pi_2, \ldots, \pi_s)$ such that support$(D, T_{\pi,\gamma}) \geq$ minSupport and s is maximal.

Definition 2.3.3. The *Increasing Groups Sequence optimization problem*:

- Input: D, minSupport, θ_{max}
- Output: $\pi = (\pi_1, \pi_2, \ldots, \pi_s)$, $\theta = \{\theta_1, \ldots, \theta_J\}$ such that support$(D, T_{\pi,\theta}) \geq$ minSupport, $\forall_j \theta_j \leq \theta_{max}$ and s is maximal.

Definition 2.3.4. The *Layered Submatrices optimization problem*:

- Input: D, minSupport, maxLayerDiff, $\Delta = \{\Delta_1, \ldots, \Delta_J\}$
- Output: ρ such that support$(D, T_{\rho,\Delta}) \geq$ minSupport,

$$\max(|\rho \bigcap \Delta_1|, \ldots, |\rho \bigcap \Delta_J|) - \min(|\rho \bigcap \Delta_1|, \ldots, |\rho \bigcap \Delta_J|) \leq \text{maxLayerDiff}$$

  and s (size of ρ) is maximal.

Definition 2.3.5. The *Connected DAG optimization problem*:

- Input: D, minSupport
- Output: $\pi = (\pi_1, \pi_2, \ldots, \pi_s)$ and T_π connected DAG support predicate such that support$(D, T_\pi) \geq$ minSupport and s is maximal.

2.4 Complexity

The complexity of the OPSM problem was shown to be NP-Complete in [BDCKY02].

Lemma 2.4.1. *The following order relation mining tasks are NP-Complete:*

- *OPSM with Errors*
- *Increasing Groups Sequence*
- *Layered Submatrices*

- *Connected DAG*

We will now prove the Lemma for every case.

Claim. *The OPSM with Errors is NP-Complete*

Proof. For $\gamma = 0$ the Partially Increasing Sequence restricts to the OPSM problem. □

Claim. *The Increasing Groups Sequence problem is NP-Complete*

Proof. For partition $\theta = \{\theta_1, \ldots, \theta_J\}$, $\forall i \theta_i = 1$ the problem restricts to the OPSM problem. □

Claim. *The Layered Submatrices problem is NP-Complete*

Proof. **LS: Decision Problem** Let D matrix, support threshold minsupport, partition Δ of the columns in D, and integer k. Is there a Layered Submatrices submatrix in D ? That is, is there set of columns ρ, —ρ=k+1— and set of rows G such that

$$\forall r_i \in G \forall \ell \forall \rho_p \in \Delta_\ell \forall \rho_q \in \Delta_{\ell+1} D(r_i, \rho_p) < D(r_i, \rho_q)$$

We prove this by reduction from the *balanced complete bipartite subgraph* problem that is known to be NP-Complete [Joh87] (See also [GJ79] page 196).

BG: Decision Problem Given a bipartite graph $G = (V, U, E)$ and a positive integer k, are there two disjoined subsets $X \subseteq V$,$Y \subseteq U$ such that $|X| = |Y| = k$, and for all $x \in X$, and $y \in Y$, $(x, y) \in E$?

The Reduction $BG \propto LS$ Given a bipartite graph $G = (V, U, E)$ and a positive integer k, the matrix $D = \{D_{ij}\}$ is defined as follows: if $(v_i, u_j) \in E$, then $D_{i,j} = j$ otherwise, $D_{i,j} = -1$. We add a first column to D containing '-1' entries. The partition Δ is defined $\Delta_j = \{c_j\}$ and minimum support minsupport $= k$.

G contains a balanced complete bipartite subgraph of size k iff the matrix D contains layered submatrices of size $k \times (k+1)$:

$\Longrightarrow$ If G contains a balanced complete bipartite subgraph of size k, than at the same indices we have a layered submatrix of size $k \times k$. Note that we can extend the submatrix by the '-1' column to get an $k \times (k+1)$ order preserving submatrix.

$\Longleftarrow$ Lets assume D contains a layered submatrices B of size $k \times (k+1)$. Note that at most one column of B can contain a '-1' entry (first column), otherwise we contradict the order of the layered submatrices order relation (every layer is of one column). Thus, D contains a $k \times k$ single columns layered submatrices consisting of only positive numbers. This matrix corresponds to a balanced complete bipartite graph in G (on the same indices). □

For the Connected DAG problem, we prove NP-Completeness for two instances:

- General case - As described above, connected DAG, in/out degree 2.
- Directed Binary Tree graph - Later in this work we address some specific variants of the general case, specifically directed binary tree graph (*e.g.* A single root to the graph).

Claim. *The General Connected DAG problem is NP-Complete*

Proof. The proof is similar to the proof that OPSM is NP-Complete.

GCD: Decision Problem Let D matrix, support threshold minsupport, and integer $|\pi| = k$. Is there a Connected DAG (in degree 2, out degree 2) pattern submatrix in D ?

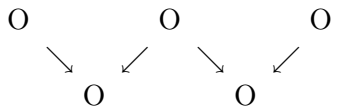
We prove by reduction from the *balanced complete bipartite subgraph* problem that is known to be NP-Complete [Joh87] (See also [GJ79] page 196).

BG: Decision Problem Given a bipartite graph $G = (V, U, E)$ and a positive integer k, are there two disjoined subsets $X \subseteq V$,$Y \subseteq U$ such that $|X| = |Y| = k$, and for all $x \in X$, and $y \in Y$, $(x, y) \in E$?

The Reduction $BG \propto GCD$ Given a bipartite graph $G = (V, U, E)$ and a positive integer k define the matrix $D = \{D_{ij}\}$ as follows: if $(v_i, u_j) \in E$, then $D_{i,j} = j$ otherwise, $D_{i,j} = -1$. We add a $k + 1$ columns to D containing '-1' entries. Minimum support minsupport $= k$, $|\pi| = 2k + 1$.

We show now that G contains a balanced complete bipartite subgraph of size k iff the matrix D contains a Connected DAG submatrix of size $k \times (2k + 1)$.

$\Longrightarrow$ If G contains a balanced complete bipartite subgraph of size k, than at the same indices we have a Connected DAG of size $k \times k$ (a single trail). Note that we can extend the submatrix with the $k+1$ '-1' columns to get an $k \times (2k+1)$ connected DAG submatrix. This gives a zigzag graph pattern with $k + 1$ columns of '-1' and k columns with positive values (see below):



This DAG is supported by k rows (same indices as the balanced complete bipartite subgraph).

$\Longleftarrow$ Lets assume D contains a connected DAG submatrix B of size $k \times (2k + 1)$. Note that at most $k + 1$ columns of B can contain '-1' entries, otherwise we contradict the order of the connected DAG submatrix property (the zigzag graph is the one with the most '-1' columns).

Thus, D contains a $k \times k$ single trail connected DAG submatrix consisting of only positive numbers. This matrix corresponds to a balanced complete bipartite graph in G (at the same indices). □

Claim. *The DAG Binary Tree problem is NP-Complete*

Proof. The proof is similar to the proof that OPSM and Layered Submatrices are NP-Complete. The main observation is that '-1' can only be in the root of the graph ('-1' column), and the rest of the graph forms a biclique.

BT: Decision Problem Let D matrix, support threshold minsupport and integer k. Is there a DAG Binary Tree in D ? That is, is there a permutation π, $|\pi| = k+1$, a DAG binary tree support predicate T_π and set of rows G such that $\forall r_i \in G\ T_\pi(r_i) = True$.

We prove this by reduction from the *balanced complete bipartite subgraph* problem that is known to be NP-Complete [Joh87] (See also [GJ79] page 196).

BG: Decision Problem Given a bipartite graph $G = (V, U, E)$ and a positive integer k, are there two disjoined subsets $X \subseteq V$,$Y \subseteq U$ such that $|X| = |Y| = k$, and for all $x \in X$, and $y \in Y$, $(x, y) \in E$?

The Reduction $BG \propto BT$ Given a bipartite graph $G = (V, U, E)$ and a positive integer k, the matrix $D = \{D_{ij}\}$ is defined as follows: if $(v_i, u_j) \in E$, then $D_{i,j} = j$ otherwise, $D_{i,j} = -1$. We add a first column to D containing '-1' entries. Minimum support minsupport $= k$, $|\pi| = k+1$.

G contains a balanced complete bipartite subgraph of size k iff the matrix D contains DAG binary tree submatrix of size $k \times (k+1)$:

$\Longrightarrow$ If G contains a balanced complete bipartite subgraph of size k, than at the same indices we have a DAG binary tree submatrix (single trail graph) of size $k \times k$. Note that we can extend the submatrix by the '-1' column to get an $k \times (k+1)$ order preserving submatrix.

$\Longleftarrow$ Lets assume D contains a DAG binary tree submatrix B of size $k \times (k+1)$. Note that at most one column of B can contain a '-1' entry (first column, root of the graph), otherwise we contradict the order of the DAG binary tree order relation. Thus, D contains a $k \times k$ submatrix consisting of only positive numbers. This submatrix corresponds to a balanced complete bipartite graph in G (on the same indices). □

Chapter 3

Algorithm Building Blocks

In this section we describe the building blocks that make up the mining algorithms.

3.1 Search Space and Data Representation

3.1.1 Lexicographic Tree

Let $<$ be some lexicographic order of the items in I such that for every two items i and j, $i \neq j$: $i < j$ or $i > j$. Every node n of the lexicographic tree has two fields, n.head which is the itemset node n represent, and n.tail which is a list of items, possible extensions to n.head. A node of the lexicographic tree has a $l$evel. Itemsets for nodes at level k nodes contain k items. We also say that such itemsets have length k. The root (level 0) node n.head is empty, and n.tail $= I$.

The items in the node's tail are smaller lexicographic ($i < j$) from any item in the node's head. This can be created, for example, by the pseudo code in figure 3.1. Figure 3.2 is an example of lexicographic tree for 3 items.

```
(1)  newTail = sort n.tail in lexicographic order
(2)  while (newTail ≠ ∅)
(3)      remove α from newTail
(4)      α = the first item in newTail
(5)      n'.head = n.head ⋃ α
(6)      n'.tail = newTail
```

Figure 3.1: Simple lexicographic tree node creation procedure

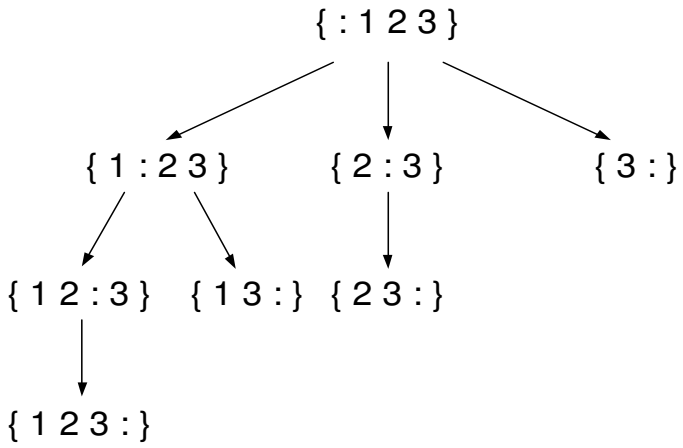


Figure 3.2: Full lexicographic tree of 3 items

3.1.2 Prefix Tree

In most frequent itemset mining algorithms a *Lexicographic Tree* is used to describe the conceptual framework of the itemset lattice. However this framework may be used only if the order of the items in the itemset has no influence. In order relation mining the ordering of the items in the pattern changes the pattern, and therefore all combinations of items ordering must be checked, thus we should use a *Prefix Tree*.

For example in frequent itemset mining the pattern ABC equals to ACB, however in order relation mining ABC dose not equal to ACB.

Every node n of the prefix tree has two fields, n.head which is the itemset node n represent, and n.tail which is a list of items, possible extensions to n.head. However unlike Lexicographic Tree, items in the tail can be small than those in the head. Pseudo code in Figure 3.3 describes the node creation.

For example see figure 3.4.

(1) foreach α in n.tail
(2) $\quad n'$.head $= n$.head $\bigcup \alpha$
(3) $\quad n'$.tail $= n$.tail $- \alpha$

Figure 3.3: Simple prefix tree node creation procedure

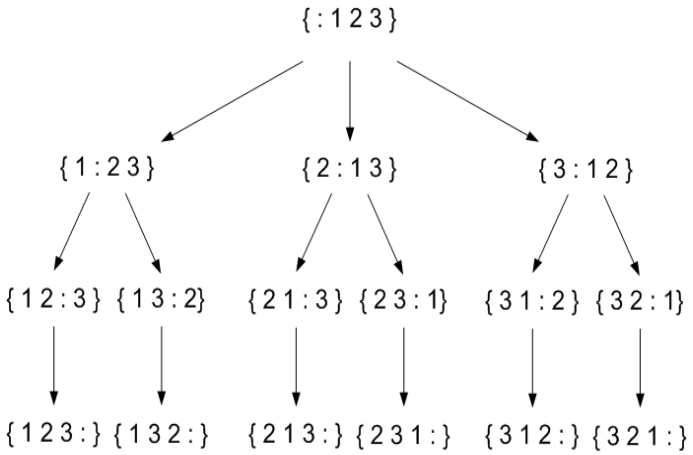


Figure 3.4: Full prefix tree of 3 items

Project(p : vector, v : vector)
/* p - vector to be projected *upon*
v - vector being projected */
(1) t = Empty Vector
(2) $i = 0$
(3) for each nonzero bit in p, at offset j, in
ascending order of offsets:
(4) Set i'th bit of target vector t to be the
j'th bit of v.
(5) $i = i + 1$
(6) return t

Figure 3.5: Projection

3.1.3 Depth First Search Traversal

In the course of the algorithm we traverse the lexicographic tree (and prefix tree) in a depth-first order. In lexicographic tree at node n, for every element α in the node's tail, a new node n' is generated such that $n'.\text{head} = n.\text{head} \bigcup \alpha$ and $n'.\text{tail} = n.\text{tail} - \alpha$. After the generation and traversal of n', α is removed from n.tail, as it is no longer needed (see Figure 3.6). In prefix tree traversal α is not removed from n.tail, those creating the full prefix tree (see Figure 3.7).

Several pruning techniques, on which we elaborate later, are used in order to speed up this process.

DFS(n : node,)
(1) $t = n$.tail
(2) while $t \neq \emptyset$
(3) Let α be the first item in t
(4) remove α from t
(5) n'.head $= n$.head $\bigcup \alpha$
(6) n'.tail $= t$
(7) *DFS*(n')

Figure 3.6: Simple DFS - Lexicographic Tree

DFS-Prefix(n : node,)
(1) $t = n$.tail
(2) foreach α in t
(3) n'.head $= n$.head $\bigcup \alpha$
(4) n'.tail $= t - \alpha$
(5) *DFS-Prefix*(n')

Figure 3.7: Simple DFS - Prefix Tree

3.1.4 Vertical Sparse Bit-Vectors

Comparison between horizontal and vertical database representations done in [SHS+00] shows that the representation of the database has high impact on the performance of the mining algorithm. In a vertical database the data is represented as a list of items, where every item holds a list of transactions in which it appears.

The list of transactions held by every item can be represented in many ways. In [Zak00] the list is a tid-list, while [SHS+00, BCG01] use vertical bit vectors. Because the data tends to be sparse, vertical bit vectors hold many "0" entries for every "1", thus wasting memory and CPU for processing the information. In [SHS+00] the vertical bit vector is compressed using an encoding called *skinning* which shrinks the size of the vector.

We choose to use a sparse vertical bit vector. Every such bit vector is built from two arrays - one for values, and one for indexes. The index array gives the position in the vertical bit vector, and the value array is the value of the position, see Figure 3.12. The index array is sorted to allow fast AND operations between two sparse bit vectors in a similar manner to the INTERSECT operation between the tid-lists. Empty values are thrown away during the AND

```
Apriori(n : node, minsupport : integer)
(1)  t = n.tail
(2)  while t ≠ ∅
(3)      Let α be the first item in t
(4)      remove α from t
(5)      n'.head = n.head ⋃ α
(6)      n'.tail = t
(7)      if (support(n'.head) ≥ minsupport)
(8)          Report n'.head as frequent itemset
(9)          Apriori(n')
```

Figure 3.8: Apriori

operation, save space and computation time.

Counting and support

To count the number of ones within a sparse bit vector, one can hold a translation table of 2^w values, where w is the word length. To count the number of ones in a word requires only one memory access to the translation table. This idea first appeared in the context of frequent itemsets in [BCG01].

3.2 Pruning and Optimization

To improve performance of the algorithms some pruning and optimization techniques are used. Not all the techniques can be used for both Prefix trees and Lexicographic trees, if so it is stated in which search space the technique works.

3.2.1 Previous Techniques

Apriori

Proposed by [AS94] the *Apriori* pruning technique is based on the monotonicity property of support: support(P) $\geq$ support(PX) as PX is contained in less transactions than P. Therefore if for an itemset P, support(P) $<$ minsupport, the support of any extension of P is also lower than minsupport, and the subtree rooted at P can be pruned from the search space. See Figure 3.8 for pseudo code (demonstrated on Lexicographic Tree)..

```
PEP(n : node, minsupport : integer)
(1)  t = n.tail
(2)  while t ≠ ∅
(3)      Let α be the first item in t
(4)      remove α from t
(5)      n'.head = n.head ⋃ α
(6)      n'.tail = t
(7)      if (support(n'.head) = support(n.head))
(8)          add α to the list of items removed by
             PEP
(9)      else if (support(n'.head) ≥ minsupport)
(10)         Report n'.head ⋃{All subsets of items
             removed by PEP} as frequent itemsets
(11)         PEP(n')
```

Figure 3.9: PEP

Dynamic Reordering

To increase the chance of early pruning, nodes are traversed, not in lexicographic order, but in order determined by support. This technique was introduced by [Jr98].

Instead of lexicographic order we reorder the children of a node as follows. At node n, for all α in the tail, we compute $s_\alpha = \text{support}(t.\text{head} \bigcup \alpha)$, and sort the nodes by s_α in an increasing order. Items α in n.tail for which $\text{support}(t.\text{head} \bigcup \alpha) < \text{minsupport}$ are trimmed away. This way, the rest of the sub-tree benefits from a shortened tail. Items with smaller support, which are heuristically "likely" to be pruned earlier, are traversed first. See Figure 3.10 for pseudo code.

Dynamic reordering can be used only with Lexicographic Tree. In Prefix Tree the depth of each node is the same and dynamic reordering have no influence in terms of early pruning.

3.2.2 New General Tools introduced here

Groups Pruning

We introduce this pruning step in the group sequence algorithm. Consider a sequence $s = (s, \{i_\ell\})$ at some node and suppose it sequence-extends include $s_a = (s', \{i_\ell, i_k\})$ and $s_b = (s', \{i_\ell\}, \{i_k\})$. If s_a is not frequent then s_b is not frequent.

This pruning step is specific to the group sequence problem and can not be used for the sequential patterns problem in the manner we introduced it.

DynamicReordering(n : node, minsupport : integer)
(1) $t = n$.tail
(2) for each α in t
(3) Compute $s_\alpha = \text{support}(n.\text{head} \bigcup \alpha)$
(4) Sort items α in t by s_α in ascending order.
(5) while $t \neq \emptyset$
(6) Let α be the first item in t
(7) remove α from t
(8) $n'.\text{head} = n.\text{head} \bigcup \alpha$
(9) $n'.tail = t$
(10) if (support($n'.head$) $\geq$ minsupport)
(11) Report n'.head as frequent itemset
(12) *DynamicReordering*(n')

Figure 3.10: Dynamic Reordering

Preprocessing ($OR2$ Matrix)

Fast checking of the support is crucial for a successful generate-and-test algorithm. To enable fast support checking we create $m \times m$ matrix of all 2-items **O**rder **R**elation ($OR2$ Matrix). For every entry (c_i, c_j) we create a bit vector L of length n as follows :

$$L_g^{c_i,c_j} = \begin{cases} 0 & D_{g,i} \leq D_{g,j} \\ 1 & otherwise \end{cases}$$

Therefore (for simple increasing sequence) $support(c_i, c_j)$ = number of '1's in L^{c_i,c_j}. We use this matrix to generate patterns and check their support by holding a vertical bit vector for each pattern, representing the genes supporting the pattern, and to add new items to the pattern we use the relevant bit vector.

3.3 AIM-$\mathcal{F}$ Specific Optimizations

This section describes specific techniques used only for AIM-$\mathcal{F}$ and not relevant for any of the other algorithms. The AIM-$\mathcal{F}$ pseudo code is described in figure 3.11.

```
AIM-F(n : node, minsupport : integer)
/* Uses DFS traversal of prefixlexicographic itemset tree
   Fast computation of small frequent itemsets
   for sparse datasets
   Uses difference sets to compute support
   Uses projection and bit vector compression
   Makes use of parent equivalence pruning
   Uses dynamic reordering    */
(1)  t = n.tail
(2)  for each α in t
(3)       Compute s_α = support(n.head ⋃ α)
(4)       if (s_α = support(n.head))
(5)            add α to the list of items removed by PEP
(6)            remove α from t
(7)       else if (s_α < minsupport)
(8)            remove α from t
(9)  Sort items in t by s_α in ascending order.
(10) While t ≠ ∅
(11)      Let α be the first item in t
(12)      remove α from t
(13)      n'.head = n.head ⋃ α
(14)      n'.tail = t
(15)      Report n'.head ⋃ {All subsets of items
             removed by PEP} as frequent itemsets
(16)      AIM-F(n')
```

Figure 3.11: AIM-$\mathcal{F}$

3.3.1 Previous Techniques

Bit-vector projection

In [BCG01], a technique called projection was introduced. Projection is a sparse bit vector compression technique specifically useful in the context of mining frequent itemsets. The idea is to eliminate redundant zeros in the bit-vector - for itemset P, all the transactions which does not include P are removed, leaving a vertical bit vector containing only 1s. For every itemset generated from P (a superset of P), PX, all the transactions removed from P are also removed. This way all the extraneous zeros are eliminated.

The projection done directly from the vertical bit representation. At initialization a two dimensional matrix of 2^w by 2^w is created, where w is the word length or some smaller value that we choose to work with. Every entry (i,j) is calculated to be the projection of j on i (thus

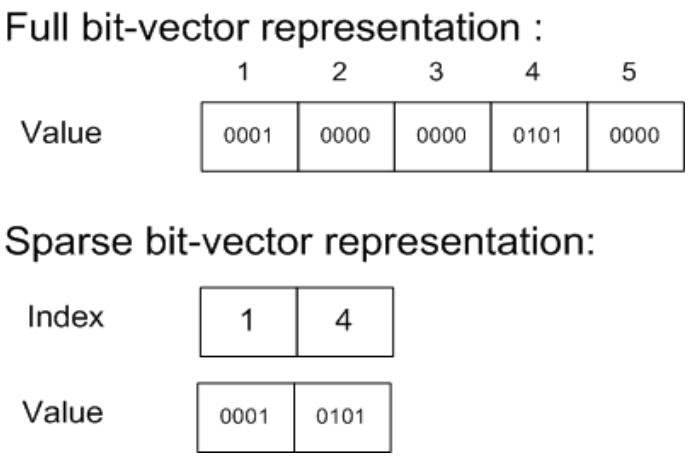


Figure 3.12: Sparse Bit-Vector data structure

covering all possible projections of single word). For every row of the matrix, the number of bits being projected is constant (a row represents the word being projected upon).

Projection is done by traversing both the vector to project upon, p, and the vector to be projected, v. For every word index we compute the projection by table lookup, the resulting bits are then concatenated together. Thus, computing the projection takes no longer than the AND operation between two compressed vertical bit lists.

In [BCG01] projection is used whenever a *rebuilding threshold* was reached. Our tests show that because we're using sparse bit vectors anyway, the gain from projection is smaller, and the highest gains are when we use projection only when calculating the 2-itemsets from 1-itemsets. This is also because of the penalty of using projection with diffsets, as described later, for large k-itemsets. Even so, projection is used only if the sparse bit vector shrunk significantly - as a threshold we set 10% - if the sparse bit vector contains less than 10% of '1's it will be projected.

Diffsets

Difference sets (*Diffsets*), proposed in [ZG03], are a technique to reduce the size of the intermediate information needed in the traversal using a vertical database. Using Diffsets, only the differences between the candidate and its generating itemsets is calculated and stored (if necessary). Using this method the intermediate vertical bit-vectors in every step of the DFS traversal are shorter, this results in faster intersections between those vectors.

Let $t(P)$ be the tidset of P. The Diffset $d(PX)$ is the tidset of tids that are in $t(P)$ but not in $t(PX)$, formally : $d(PX) = t(P) - t(PX) = t(P) - t(X)$. By definition support$(PXY)$ = support$(PX) - |d(PXY)|$, so only $d(PXY)$ should be calculated. However $d(PXY) = d(PY) -$

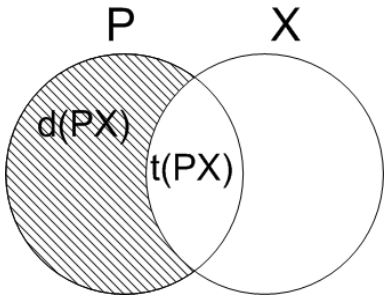


Figure 3.13: Diffset threshold

$d(PX)$ so the Diffset for every candidate can be calculated from its generating itemsets.

For the sequence mining problems Diffsets can not be used, as the basic is now $d(PX) = t(P) - t(PX) \neq t(P) - t(X)$, and therefore $d(PXY) \neq d(PY) - d(PX)$ (as the order relation of Y and X not considered).

Diffsets have one major drawback - in datasets, where the support drops rapidly between k-itemset to k+1-itemset then the size of $d(PX)$ can be larger than the size of $t(PX)$ (For example see figure 3.13). In such cases the use of diffsets should be delayed (in the depth of the DFS traversal) to such k-itemset where the support stops the rapid drop. Theoretically the break even point is 50%: $\frac{t(PX)}{t(P)} = 0.5$, where the size of $d(PX)$ equals to $t(PX)$, however experiments shows small differences for any value between 10% to 50%. For this algorithm we used 50%.

Parent Equivalence Pruning (PEP)

This is a pruning method based on the following property : If
support(P) = *support*(PX) then all the transactions that contain P also contain PX. Thus, X can be moved from the tail to the head, thus saving traversal of P and skipping to PX. This method was described by [BCG01, Zak00]. Later when the frequent items are generated the items which were moved from head to tail should be taken into account when listing all frequent itemsets. For example, if k items were pruned using PEP during the DFS traversal of frequent itemset X then the all 2^k subsets of those k items can be added to X without reducing the support. This gives creating 2^k new frequent itemsets. See Figure 3.9 for pseudo code.

This can not be used for sequence mining as the relation between the extending item must be checked. For example support(P) = support(PX) and yet PY can be frequent and PXY not, and the relation of XY must be checked.

Optimized Initialization ($F2$ Matrix)

In sparse datasets computing frequent 2-itemsets can be done more efficiently than than by performing $\binom{m}{2}$ itemset intersections. We use a method similar to the one described in [Zak00]: as a preprocessing step, for every transaction in the database, all 2-itemsets are counted and stored in an upper-matrix of dimensions $m \times m$. This step may take up to $O(m^2)$ operations per transaction. However, as this is done only for sparse datasets, experimentally one sees that the number of operations is small. After this initialization step, we are left with frequent 2 item itemsets from which we can start the DFS procedure.

3.3.2 New General Tools introduced here

Diffsets and Projection

Never before Diffsets and Projection were used together. As $d(PXY)$ is not a subset of $d(PX)$, Diffsets cannot be used directly for projection. Instead, we notice that $d(PXY) \subseteq t(PX)$ and $t(PX) = t(P) - d(PX)$. However $d(PX)$ is known, and $t(P)$ can be calculated in the same way. For example $t(ABCD) = t(ABC) - d(ABCD)$, $t(ABC) = t(AB) - d(ABC)$, $t(AB) = t(A) - d(AB)$ thus $t(ABCD) = t(A) - d(AB) - d(ABC) - d(ABCD)$. Using this formula the $t(PX)$ can be calculated using the intermediate data along the DFS trail. As the DFS goes deeper, the penalty of calculating the projection is higher.

Chapter 4

Mining Algorithms for Order Relations

In this chapter we show how to combine building blocks described in the previous chapter to create the order relation mining algorithms. Later we analyze the complexity per output of every algorithm.

4.1 Problem Algorithms

4.1.1 OPSM

For the OPSM algorithm we used the prefix tree DFS traversal, as the order of traversal create different patterns (*e.g.* the pattern (c_1, c_2, c_3) differs from (c_1, c_3, c_2)). Given a pattern (s, c_i) and a new column c_j the support is calculated by performing bitwise AND : $L^{s,c_i,c_j} = L^{s,c_i} \bigcap L^{c_i,c_j}$, and count the '1's in the new vector. The DFS traversal is pruned by the Apriori pruning technique.

The mining algorithm is described in the pseudo code in figure 4.1

4.1.2 OPSM with Errors

The support predicate T_γ returns true if the given pattern contains an increasing sequence of length $s-\gamma$. This task can be achieved by searching the longest increasing subsequence (LIS) in the given pattern, and check that the longest increasing subsequence is at least $s-\gamma$ in length.

An important observation is that the apriori property holds in OPSM with errors: if the longest increasing subsequence of pattern P in row i is $|P|-\gamma-1$ then any extension $P' = P \cup \alpha$ contain longest increasing subsequence no longer than $|P| - \gamma = |P'| - \gamma - 1$. If no sequence

```
OPSM(n : node, minsupport : integer)
/* Uses DFS traversal of prefix itemset tree
   Uses bit vector compression
   Makes use of Apriori pruning
  */
(1)  t = n.tail
(2)  for each α in t
(3)       Compute s_α = support(n.head ⋃ α) using F2
(4)       if (s_α < minsupport)
(5)            remove α from t
(6)  for each α in t
(7)       n'.head = n.head ⋃ α
(8)       n'.tail = t − α
(9)       Report n'.head as OPSM
(10)      OPSM(n')
```

Figure 4.1: OPSM Algorithm

of length $|P| - \gamma - 1$ exist for a certain pattern, then in $|P'|$ no new valid sequence is created (number of errors is monotone). Therefor $support(P) \geq support(PX)$ and the apriori property holds.

For calculating the longest increasing subsequence we used dynamic programming technique with complexity $O(s \cdot \gamma)$.

We used this for two reasons: (a) It is incremental - if we have calculated for P, then calculation for an extension P' takes γ steps and (b) the algorithm suits vertical bit vectors implementation (With bit vectors extension take γ^2 steps).

In [BS00] similar algorithm for finding longest increasing subsequence in time $O(n \lg \lg n)$ was presented, using efficient data structures. However in practice the algorithm we use is faster for this case because of the low values of γ, and the low constant as we use compressed bit vector ($\frac{1}{w}$).

We transform the problem into a graph, and run the algorithm on the graph. Given a series $D(r_i, \pi_1), D(r_i, \pi_2), \ldots, D(r_i, \pi_s)$ for each row i we build the following directed graph $G_i(V_i, E_i)$:

- $V_i = \{v_j \mid D(r_i, \pi_j)\}$
- $E_i = \{(v_k, v_\ell) \mid D(r_i, \pi_k) < D(r_i, \pi_\ell) \bigwedge k < \ell < k + \gamma + 1\}$.

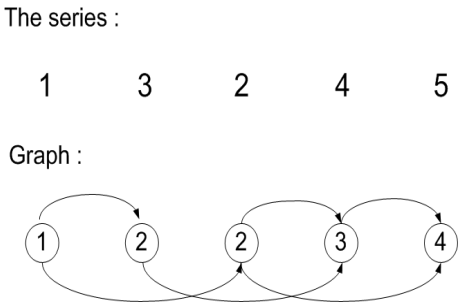


Figure 4.2: Longest Increasing Sequence Graph $s = 5$ $\gamma = 1$

Lemma 4.1.1. *The series $D(r_i, \pi_1), D(r_i, \pi_2), \ldots, D(r_i, \pi_s)$ contains a longest increasing subsequence of length $r \geq s - \gamma$ or more iff the graph $G_i(V_i, E_i)$ contains a direct path of length $r \geq s - \gamma$.*

The proof is in Appendix A.

The longest path algorithm: The algorithm for finding the longest path is based on two phases:

- *Building Phase:* Move from v_1 to v_s and mark each node as follows: If no edge is pointed to the vertex, mark the vertex with '1', otherwise, from all the edges pointing to the vertex, select the edge originating from a vertex with the highest marking, add 1 and set this mark on the current vertex.

- *Search Phase:* Move from $v_{s-\gamma}$ to v_s, check the marking of each vertex and return the highest marking. If this marking is greater or equal then $s-\gamma$ this is the longest increasing subsequence, otherwise no increasing subsequence of length $s - \gamma$ exists.

Lemma 4.1.2. *The longest path algorithm returns the length of the longest directed path in the graph of length $s - \gamma$ or more, and returns no result if no such path exists.*

The proof is in Appendix A.

See figure 4.2 for example of the graph created, the marking in the vertexes after running the algorithm.

The implementation details: Every node in the graph now holds γ vertical bit vectors. The bit vectors holds the support for every possible number of errors (*i.e.* L_0 holds the genes support the pattern with 0 errors, L_1 with one error, and so on until L_γ). The calculation of a new node (Building Phase) is described in Figure 4.3. The calculation of the support (Search Phase) is described in Figure 4.4. The rest of the algorithm remains the same as in OPSM (Figure 4.5).

Errors-CreateNode(n : node, α : item, t : tail)
(1) n'.head = n.head $\bigcup \alpha$
(2) n'.tail = $t - \alpha$
(3) n'.parent = n
(4) Let β be the last item in n.head
(5) $n'.L_0 = n.L_0$BitwiseAND $L^{(\beta,\alpha)}$
(6) for loopMistakes = 1 to γ
(7) noNewErrors = $L^{(\beta,\alpha)}$BitwiseAND $n.L_{\text{loopMistakes}}$
(8) newErrors = noNewErrors
(9) upwardCount = 1
(10) upwardTraversal = n.parent
(11) while loopMistakes − upwardCount ≥ 0
(12) Let δ be the last item in $upwardTraversal$.head
(13) upwardErrors = $L^{(\delta,\alpha)}$ BitwiseAND $upwardTraversal.L_{loopMistakes-upwardCount}$
(14) newError = newError BitwiseOR upwardErrors
(15) upwardTraversal = upwardTraversal.parent
(16) upwardCount = upwardCount + 1
(17) $n'.L_{\text{loopMistakes}}$ = newErrors
(18) return n'

Figure 4.3: Node creation function with errors

4.1.3 Increasing Groups sequence

For the increasing groups sequence algorithm we used the prefix tree DFS traversal, as the order of traversal create different patterns (*e.g.* the pattern $(\{c_1\},\{c_2\},\{c_3\})$ differs from $(\{c_1\},\{c_3\},\{c_2\})$). Given a pattern $(\ldots,\{\ldots,c_i\})$ and a new column c_j, two new patterns are generated - $(\ldots,\{\ldots,c_i,c_j\})$ and $(\ldots,\{\ldots,c_i\},c_j)$ (Considering the restrictions of group length). The support is calculated by performing bitwise AND between the new column c_j and all the tissues in the preceding group, and then preforming AND with the pattern bit vector. This support function pseudo code described in Figure 4.6. The DFS traversal is pruned by the Apriori pruning technique, and groups pruning. See figure 4.7.

4.1.4 Layers sequence

This special case of groups sequence can be mined much faster than the general case of the Increasing Groups sequence. This due the fact that the tissues in every groups (Layers) are known a priori. For that reason when examining each item the group of the item is known, and it dose not matter in what order the items where selected to the sequence. Instead of Prefix

```
Errors-Support(n : node)
(1)  currentSupportVector = n.L_γ
(2)  upwardTraversal = n.parent
(3)  for upward = 1 to γ
(4)       currentSupportVector = currentSupportVector BitwiseOR upwardTraversal.L_{γ−upward}
(5)       upwardTraversal = upwardTraversal.parent
(6)  return currentSupportVector.CountElements()
```

Figure 4.4: Support Calculation function with errors

```
OPSM-with-Errors(n : node, minsupport : integer)
/* Uses DFS traversal of prefix itemset tree
   Uses bit vector compression
   Makes use of Apriori pruning
  */
(1)  t = n.tail
(2)  for each α in t
(3)       n' = Errors-CreateNode(n, α, t)
(4)       Compute s_α = Errors-Support(n')
(5)       if (s_α < minsupport)
(6)            remove α from t
(7)       else
(8)            Report n'.head as OPSM with Errors
(9)            OPSM-with-Errors(n')
```

Figure 4.5: OPSM with Errors Algorithm

Tree traversal the space can be traversed in Lexicographic Order which is much smaller (and can be pruned earlier with dynamic reordering).

4.1.5 Connected DAG

In OPSM a single item from the tail is appended to the head. In connected DAG the algorithm creates $2s$ new nodes by adding the item from the tail, before and after every item in the head. Calculates every time the bit vector of those two items in each order, and perform AND operation with the bit vector of the parent node. There is no need to save the actual ordering of items in the resulting node (only for later printing reasons). Our algorithm does not create DAG that contains circle in the underlying undirected graph. See figure 4.8. In addition to

IncreasingGroups-Support(n : node, α : item, IsNewGroup: boolean)
(1) CurrentGroup = last item in n.groups
(2) if a new group is NOT opened decrease CurrentGroup by 1
(3) foreach β in n.head for which the n.groups value is CurrentGroups
(4) GroupsBitVector = BitwiseAND of all those $L^{(\beta,\alpha)}$
(5) $L' = n.L$ BitwiseAND GroupBitVector
(6) return L'.CountElements()

Figure 4.6: Increasing Groups Sequence support function

the code in the figure, there is a array to prevent each item to have an *in-degree* or *out-degree* greater than 2.

4.2 Complexity per output

As shown in the OPSM and variants are NP-Complete. In the following section we analyze the complexity per output result. The summary of the complexity is in the following table (Excluding preprocessing):

Problem	Complexity per output	Details
OPSM	$O(mn)$	
OPSM with Errors	$O(m^{\gamma+1}(\gamma+1)^2 n)$	γ - Number of Errors allowed
Increasing Groups	$O(mqn) \leq O(m^2 n)$	q - Maximal group size
Layered Submatrices	$O(mqn) \leq O(m^2 n)$	q - Maximal layer size
Connected DAG	$O(msn) \leq O(m^2 n)$	s - Longest pattern found

4.2.1 Common preprocessing

The first step, calculation of $OR2$ matrix, is common for all variants of the order relations. In this step, for every two columns combination, for each row we calculate which column is greater, therefor this step takes $O(nm^2)$.

4.2.2 OPSM

After preprocessing, each node in the search tree is traversed by the algorithm if it has a support greater than minsupport, or the parent of that node has such support. Because the degree of the search space is m, for each node with minsupport the algorithm traverse another m nodes.

```
IncreasingGroupsSequence(n : node, minsupport : integer)
/* Uses DFS traversal of prefix itemset tree
   Uses bit vector compression
   Makes use of Apriori pruning
  */
(1)  t = n.tail
(2)  for each α in t
(3)       Loop twice, once appending α to the last group, and once opening a new group.
(4)            s_α = IncreasingGroupsSupport(n,α, is opening new groups)
(5)            if (s_α < minsupport)
(6)                 remove α from t, skip opening new group if current loop is group appending
(7)            else
(8)                 n'.head = n.head ⋃ α
(9)                 n'.tail = t − α
(10)                CurrentGroup = last item in n.groups
(11)                if a new group is opened increase CurrentGroup by 1
(12)                n'.groups = n.groups ⋃ CurrentGroup
(13)                Report n'.head as Increasing Groups Sequence
(14)                IncreasingGroupsSequence(n')
```

Figure 4.7: Increasing Groups Sequence

Traversing a node means building the node itself by adding a new item, and calculating the support, in $n\frac{1}{w}$ time (performing an AND operation between 2 bit-vectors). As the pattern becomes longer the AND operation is performed faster because the spared vertical bit-vectors become shorter. This however has no impact on the complexity as the longer bit-vector in the AND operation is the bit-vector of the new column added to the pattern. Overall, we get $O(mn)$ per output result.

4.2.3 OPSM with Errors

There are two differences between the mining of OPSM and OPSM with Error : (a) Calculating the support is more complex and (b) the search space is larger. Regarding (b), now the number of possible infrequent children that should be traversed is $m^{\gamma+1}$. The calculation of the support, explained in details in previous section, takes $O(n\gamma^2)$ time, and the complexity per output result is $O(m^{\gamma+1}(\gamma+1)^2 n)$

```
ConnectedDAG(n : node, minsupport : integer)
/* Uses DFS traversal of prefix itemset tree
   Uses bit vector compression
   Makes use of Apriori pruning
  */
(1)  t = n.tail
(2)  for each α in t
(3)       for each β in n.head run twice - α < β and α > β
(4)            Compute s_α = support(n.head ⋃ α) using F2,
                                  comparing with β using L^(β,α) or L^(α,β)
(5)            if (s_α ≥ minsupport)
(6)                 n'.head = n.head ⋃ α
(7)                 n'.tail = t − α
(8)                 Report n'.head as Connected DAG
(9)                 ConnectedDAG(n')
(10)      if no pattern using α found remove α from t
```

Figure 4.8: Connected DAG Algorithm

4.2.4 Increasing Groups sequence

The differences between this variant and the original problem are (a) the search space and (b) calculation of the support. As for (a) the degree of the tree is $2m$ because every item added can be added to the last group, or can create a new group. This does not change the complexity, but in practice it increases run time. On the other hand calculating the support increases the complexity as the new item support must be check not only with the last item as in the increasing sequence, but with every item in the last group, and therefor calculating the support take now $O(qn)$ (q if the length of the longest allowed group) time, and the complexity per output result is $O(mqn) \leq O(m^2 n)$

4.2.5 Layered Submatrices

Although the search space is much smaller than the search space of Increasing Groups sequence (Lexicographic tree vs. Prefix tree), the complexity analysis remains the same.

4.2.6 Connected DAG

The difference between this variation and the original problem is the search space, as in the increasing sequence problem the degree of the search tree was m), and here the degree of the

search tree is $2ms$ (s is the length of the longest pattern, and therefor bounded by m) because the new item can be added not only in the end of the sequence but at any given position, before or after every item. Calculating the support remains the same, and the complexity per output result is $O(msn) \leq O(m^2n)$

Chapter 5

Frequent Itemset Mining Experimental Results

The experiments were conducted on an Athlon 1.2Ghz with 256MB DDR RAM running Microsoft Windows XP Professional. All algorithms were implemented in C++. AIM-$\mathcal{F}$ was compiled using VC 7 (2002).

5.1 Data sets

For general comparison of the data representation we used standard datasets from the data mining community: We used five datasets to evaluate the algorithms performance. Those datasets where studied extensively in [Zak00].

1. *connect* — A database of game states in the game connect 4.
2. *chess* — A database of game states in chess.
3. *mushroom* — A database with information about various mushroom species.
4. *pumsb** — This dataset was derived from the *pumsb* dataset and describes census data.
5. *T10I4D100K* - Synthetic dataset.

The first 3 datasets were taken from the UN Irvine ML Database Repository (http://www.ics.uci.edu/ mlearn/MLRepository). The synthetic dataset created by the IBM Almaden synthetic data generator (http://www.almaden.ibm.com/cs/quest/demos.html).

5.2 Comparing Data Representation

We compare the memory requirements of sparse vertical bit vector (with the projection described earlier) versus the standard tid-list. For every itemset length the total memory requirements of all tid-sets is given in figures 5.1, 5.2 and 5.3.

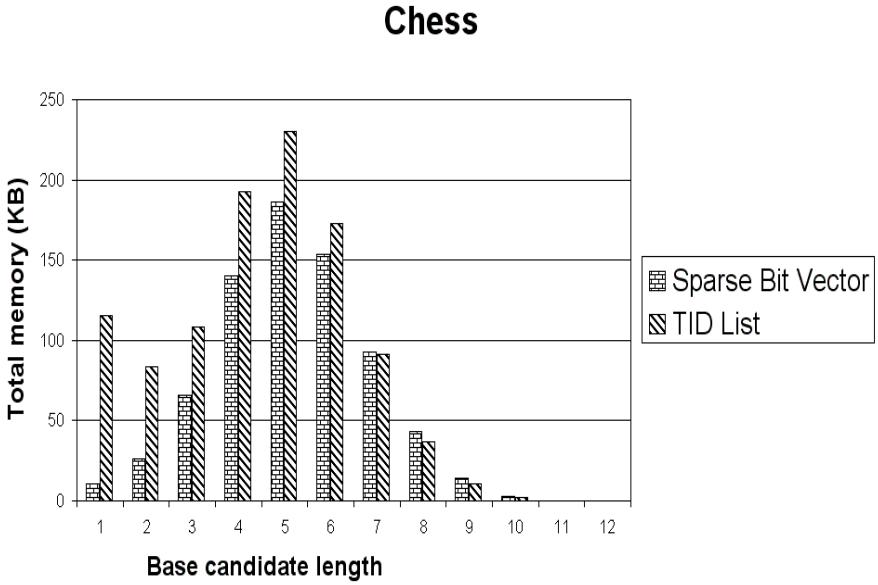


Figure 5.1: Chess - support 2000 (65%)

As follows from the figures, sparse vertical bit vector representation requires less memory than tid-list for the dense datasets (chess, connect). However for the sparse dataset (T10I4D100K) the sparse vertical bit vector representation requires up to twice as much memory as tid-list. Tests to dynamically move from sparse vertical bit vector representation to tid-lists showed no significant improvement in performance, however, this should be carefully verified in further experiments.

Gene expression datasets, after the $F2$ step, are dense (nearly 50of the item are '1's) therefore sparse bit vectors yield better memory usage than tid-lists.

5.2.1 Comparing The Various Optimizations

We analyze the influence of the various optimization techniques on the performance of the algorithm. First run is the final algorithm on a given dataset, then returning on the task, with a single change in the algorithm. Thus trying to isolate the influence of every optimization

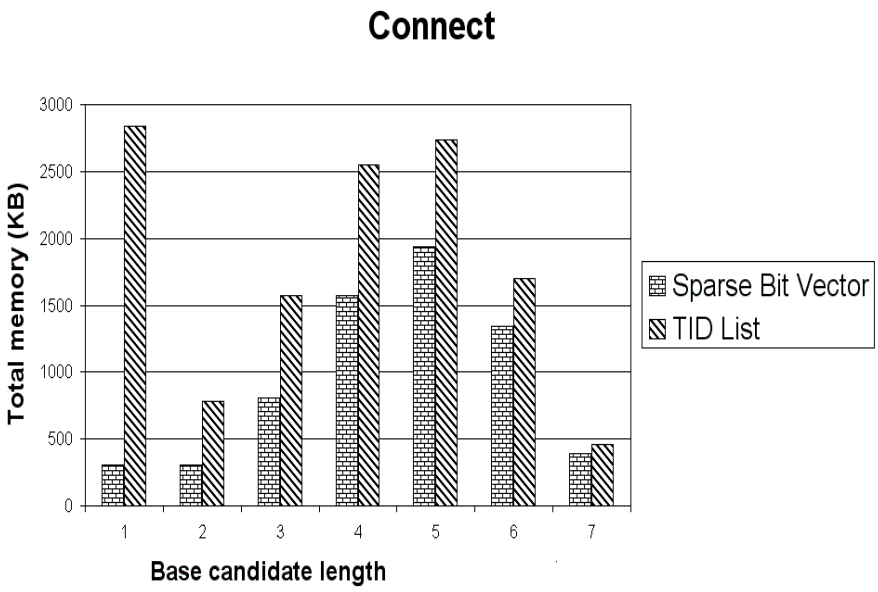


Figure 5.2: Connect - support 50000 (75%)

technique, as shown in figures 5.4 and 5.5.

As follows from the graphs, there is much difference in the behavior between the datasets. In the dense dataset, Connect, the various techniques had tremendous effect on the performance. PEP, dynamic reordering and diffsets behaved in a similar manner, and the performance improvement factor gained by of them increased as the support dropped. From the other hand the sparse bit vector gives a constant improvement factor over the tid-list for all the tested support values, and projection gives only a minor improvement.

In the second figure, for the sparse dataset T10I4D100K, the behavior is different. PEP gives no improvement, as can expected in sparse dataset, as every single item has a low support, and does not contain existing itemsets. There is drop in the support from k-itemset to k+1-itemset due to the low support therefore diffset also gives no impact, and the same goes for projection. A large gain in performance is made by optimized initialization, however the performance gain is constant, and not by a factor. Last is the dynamic reordering which contributes to early pruning much like in the dense dataset.

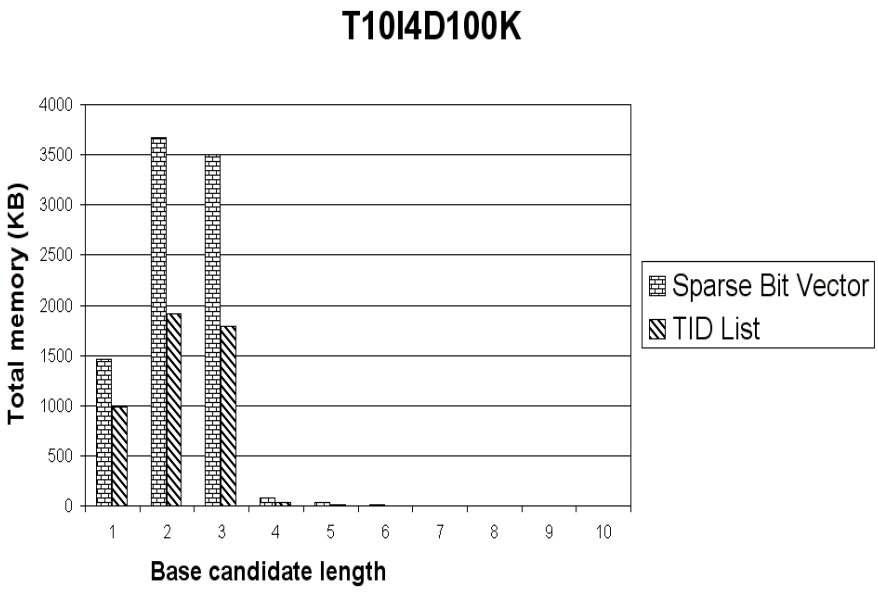


Figure 5.3: T10I4D100K - support 100 (0.1%)

5.3 Comparing Frequent Itemset Mining algorithms

We first compare the base algorithm on which we build (AIM-$\mathcal{F}$) with other frequent itemset mining algorithms.

For comparison, we used implementations of

1. Apriori [AS94] - horizontal database, BFS traversal of the candidates tree.

2. FPgrowth [HPY00] - tree projected database, searching for frequent itemsets directly without candidate generation, and

3. dEclat [Zak00] - vertical database, DFS traversal using diffsets.

All of the above algorithm implementations were provided by Bart Goethals (http://www.cs.helsinki/u/goethals/) and used for comparison with the AIM-$\mathcal{F}$ implementation.

Figures 5.6 to 5.10 gives experimental results on the various algorithms and datasets. Not surprising, Apriori [AS94] generally has the lowest performance amongst the algorithms compared, and in some cases the running time could not be computed as it did not finish even at the highest level of support checked. For these datasets and compared with the specific algorithms

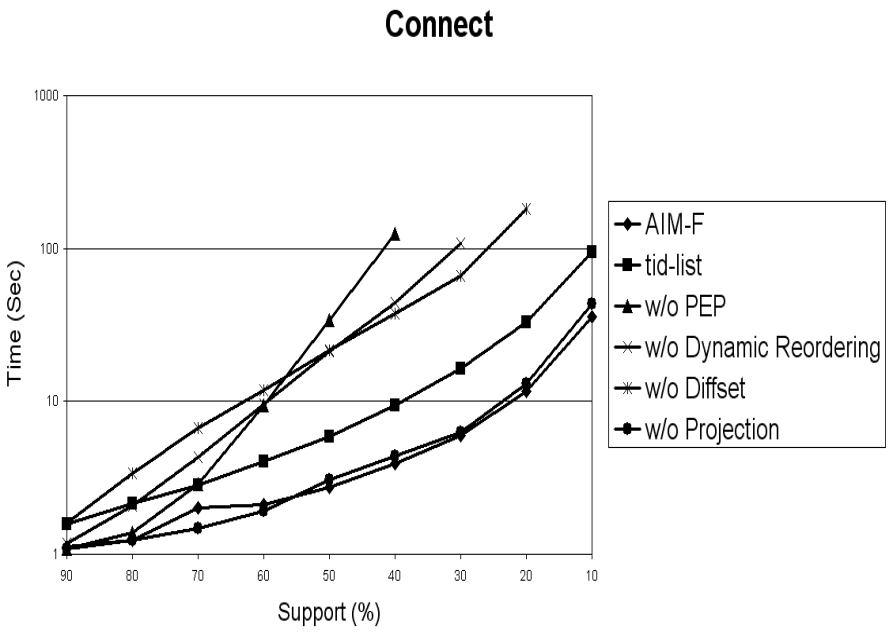


Figure 5.4: Influence of the various optimization on the Connect dataset mining

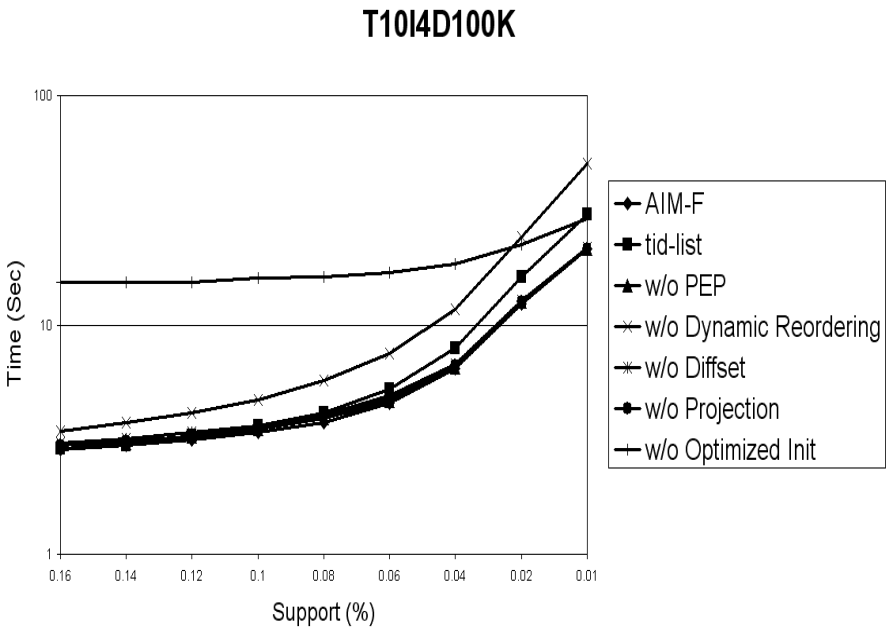


Figure 5.5: Influence of the various optimization on the T10I4D100K dataset mining

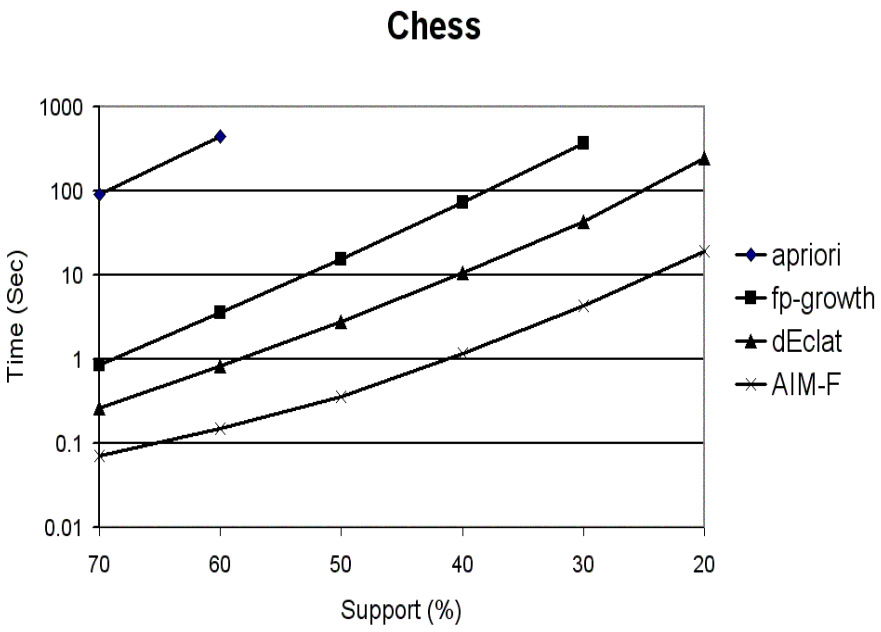


Figure 5.6: Chess dataset

and implementations described above, our algorithm/implementation, AIM-$\mathcal{F}$, seemingly outperforms all others.

In general, for the dense datasets (Chess, Connect, Pumsb* and Mushroom, figures 5.6,5.7,5.8 and 5.9 respectively), the sparse bit vector gives AIM-$\mathcal{F}$ an order of magnitude improvement over dEclat. The diffsets gives dEclat and AIM-$\mathcal{F}$ another order of magnitude improvement over the rest of the algorithms.

For the sparse dataset T10I4D100K (Figure 5.10), the optimized initialization gives AIM-$\mathcal{F}$ head start, which is combined in the lower supports with the advantage of the sparse vertical bit vector (See details in figure 5.5)

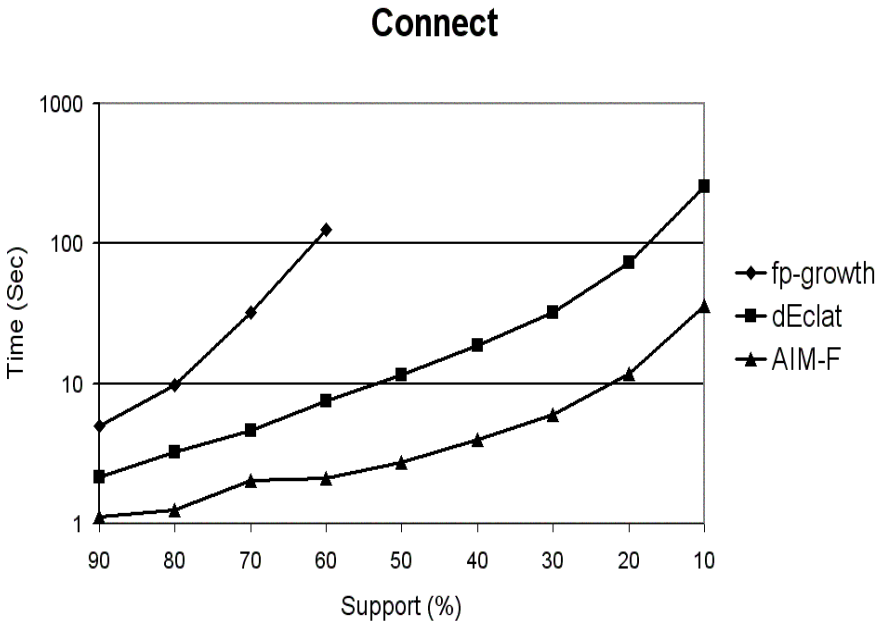


Figure 5.7: Connect dataset

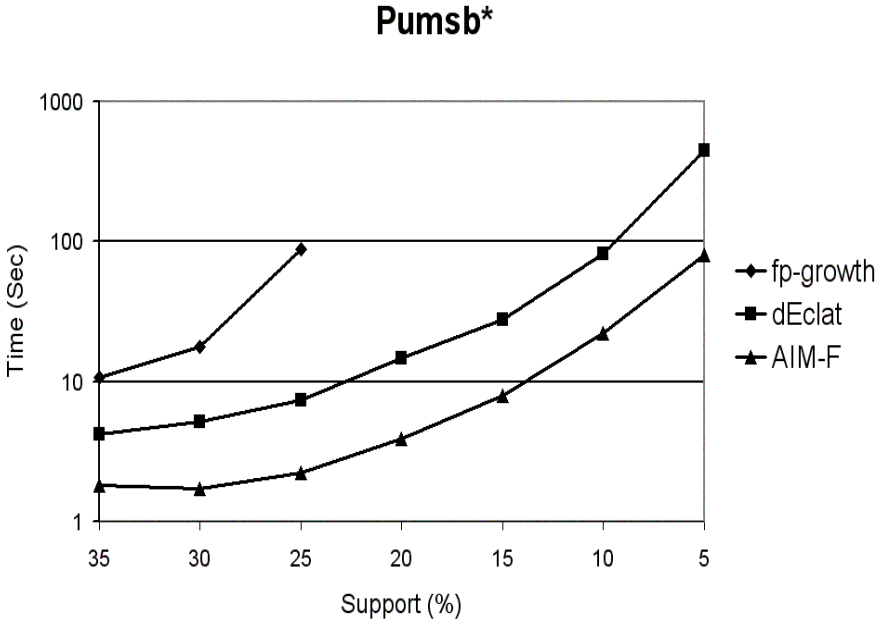


Figure 5.8: Pumsb* dataset

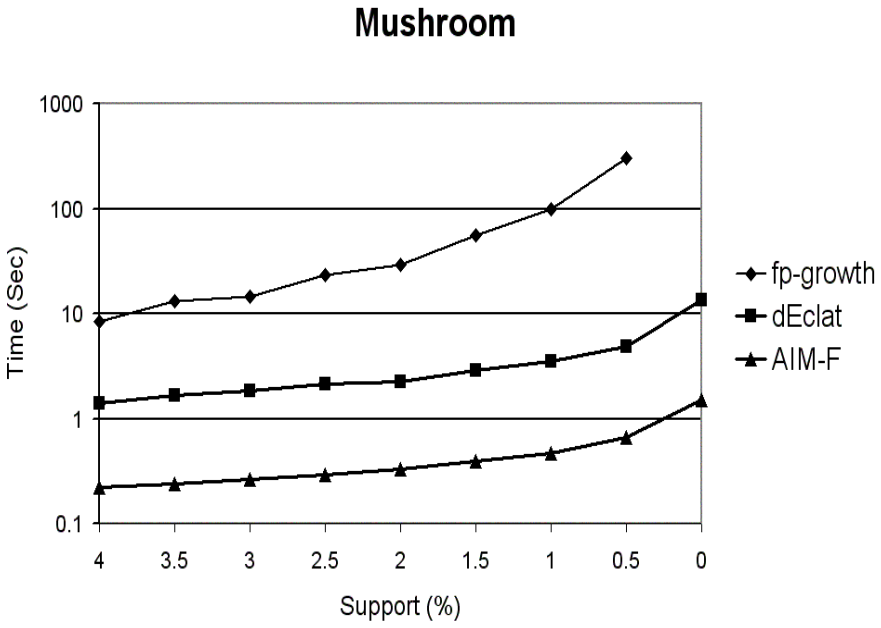


Figure 5.9: Mushroom dataset

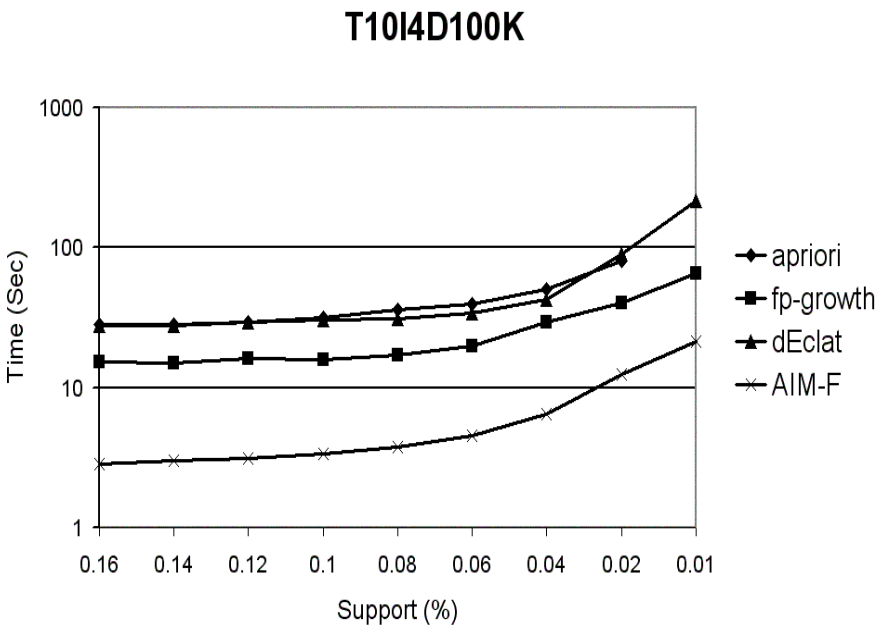


Figure 5.10: T10I4D100K dataset

Chapter 6

Order Relations Experimental Results

The experiments were conducted on an Athlon 1.2Ghz with 256MB DDR RAM running Microsoft Windows XP Professional. All algorithms were implemented in C#. Experiments were conducted with .NET Framework 1.0.

6.1 Data sets

The gene expression datasets used are real-life gene expression databases used, some where normalized by [LL02]:

- Breast cancer tumor dataset from [HDC+01], $n = 3226$ genes, $m = 22$ tissues : 8 with *brca1* mutations, 8 with *brca2* mutations, and 6 sporadic breast tumors.
- Colon cancer tumor dataset from [ABN+99], $n = 2000$ genes, $m = 62$ tissues : 40 tumor biopsies, 22 normal biopsies. The 2000 genes where selected from the original paper by [LL02] according to the confidence in the results.
- MLL Leukemia samples from [ASS+02], $n = 12582$ genes, $m = 57$ tissues (train group) : 20 ALL, 17 MLL and 20 AML.
- AML-ALL Leukemia dataset from [GSPT+99], $n = 7129$ genes, $m = 38$ tissues (train group) : 27 ALL, 11 AML.
- Central Nervous System Embryonal Tumor data set from [SPM+02], $n = 7129$ genes, $m = 60$ tissues. The dataset is dataset C mentioned in the paper, of the tissues, 21 belong

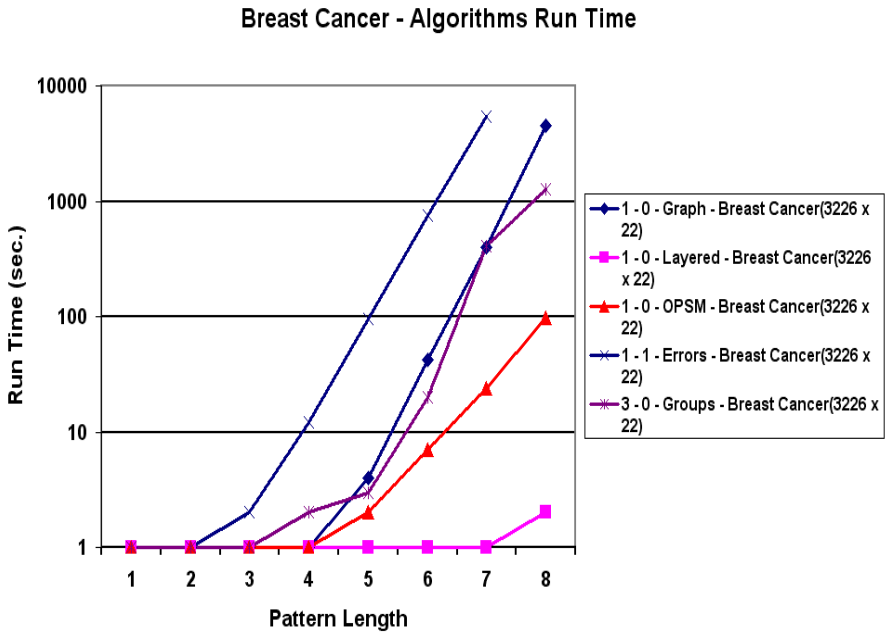


Figure 6.1: Running various order relation algorithms on the Breast Cancer dataset - Run time

to survivors, and 39 belong to patients who succumbed to their disease.

6.2 Experiments

The experiments were conducted as follows - for every pattern length, the pattern with the highest support was found. Later a mining task for that specific pattern was run to calculate the running time.

In figure 6.1 there is a comparison of the running times for all algorithms (Figure 6.2 shows the pattern size found at each of those runs). The dataset is the breast cancer dataset. The algorithms parameters selected were as follows:

- OPSM with Errors - $\gamma = 1$.
- Increasing Groups Sequence - All groups are equal in size, and set to 2.
- Layered Submatrices - 3 Layers (Normal, BRCA1, BRCA2).

From this graph it is clear that the problem of finding layered submatrices is by far the "easiest". This was expected as the layered submatrices is the only pattern, of those presented here, that could be mined using the lexicographic tree. All other pattern's algorithms are more than order of magnitude slower.

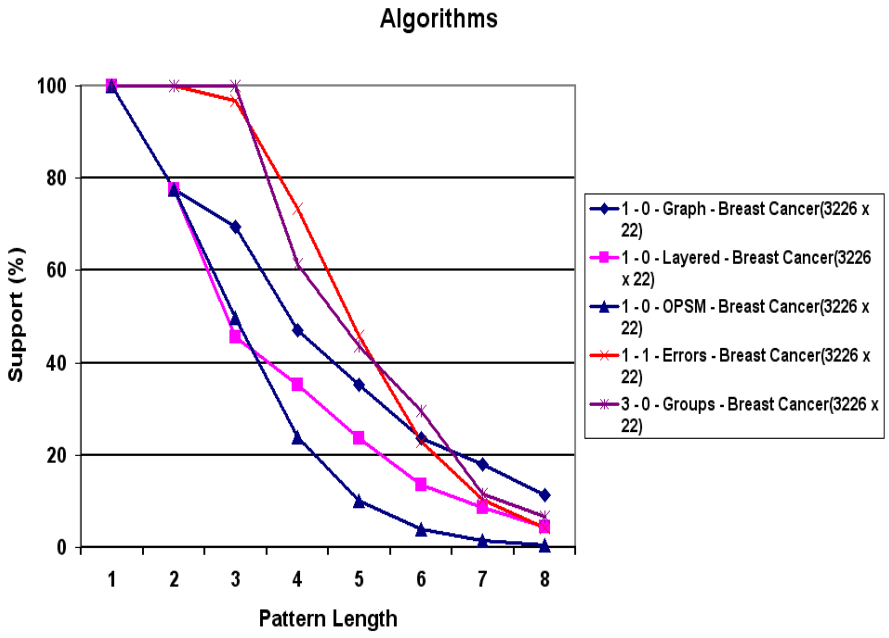


Figure 6.2: Running various order relation algorithms on the Breast Cancer dataset - Support

All further graphs in this section are support graph - the X-Axis is the length of the pattern and the Y-Axis is the support, as percentage of the dataset size (number of genes). The graphs show for each pattern length, the pattern with the highest support found.

In appendix B we show patterns found during mining of the breast cancer dataset.

6.2.1 OPSM

In figure 6.3 we compare the maximal support level of each pattern size found. Except the Colon Tumor dataset, all other dataset behaved in similar manner. In [BDCKY02] the reported run time was about 30 seconds for OPSM of length 4 for the breast cancer dataset. The running time of our algorithms for the breast cancer dataset is less than a second for OPSM of length 5 or less. The quality of the results in our algorithm (which promise to return the exact result) is also superior. The results (The support of pattern with the highest support):

Algorithm \ Pattern Length	4	6	8
[BDCKY02]	347	42	7
Our	772	127	18

As the algorithm in [BDCKY02] is heuristical the differences between the best pattern found may vary from dataset to dataset. Also, changing the size of the window (the number

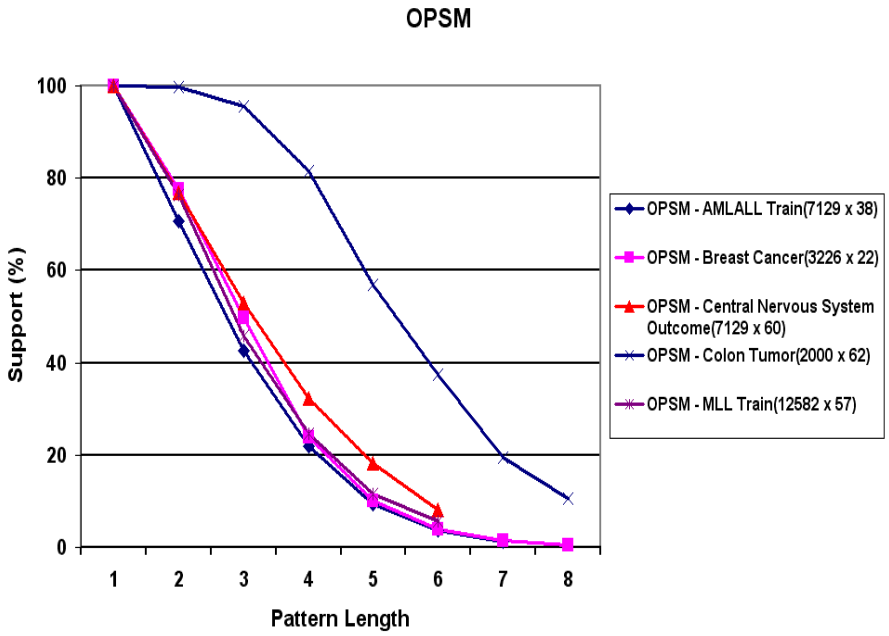


Figure 6.3: Order Preserving Submatrices for various gene expression datasets

of patterns held in each iteration) has a major effect on the results found by this algorithm. Here we compare with results reported by [BDCKY02]. Increasing the window size, will result in better results (and longer running time).

6.2.2 OPSM with Errors

OPSM with Errors algorithm was run on two datasets: Breast Cancer dataset (Figure 6.4) and the Colon Tumor dataset (Figure 6.5). For each dataset we ran two series of experiments, once with 1 error allowed, and once with 2 errors. In each graph we also present the OPSM result to compare with. As seen in both figure, allowing more errors in the pattern results in longer patterns with higher support.

6.2.3 Increasing Groups Sequence

Increasing Groups Sequence algorithm was run on two datasets: Breast Cancer dataset (Figure 6.6) and the Colon Tumor dataset (Figure 6.7). For each dataset we ran two series of experiments, once with groups of size 2, and once with groups of size 3. In each graph we also present the OPSM result to compare with. Again, as in the case of OPSM with errors, longer groups result in longer patterns with higher support. The reason is that unlike OPSM in which every

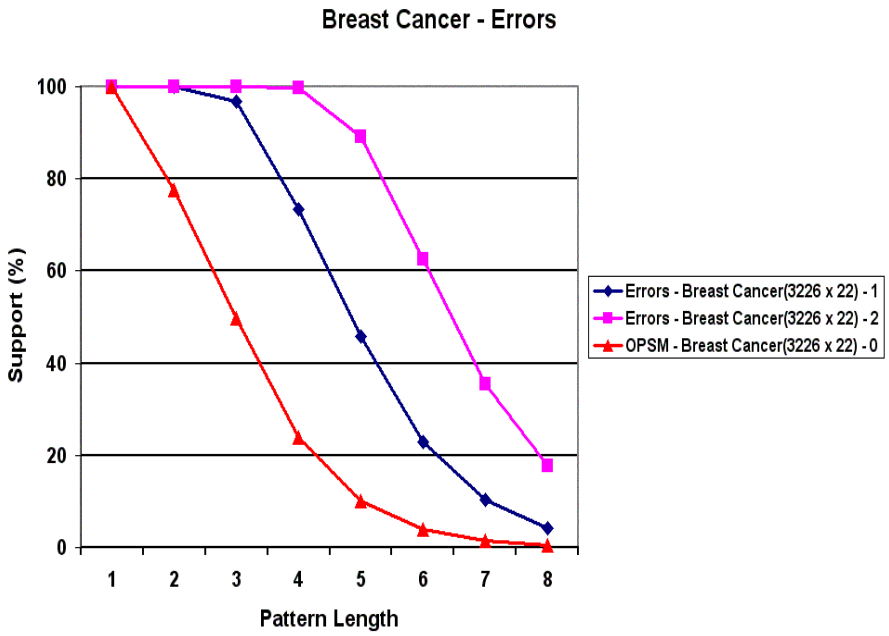


Figure 6.4: Breast Cancer Order Preserving Submatrices with Errors Allowed

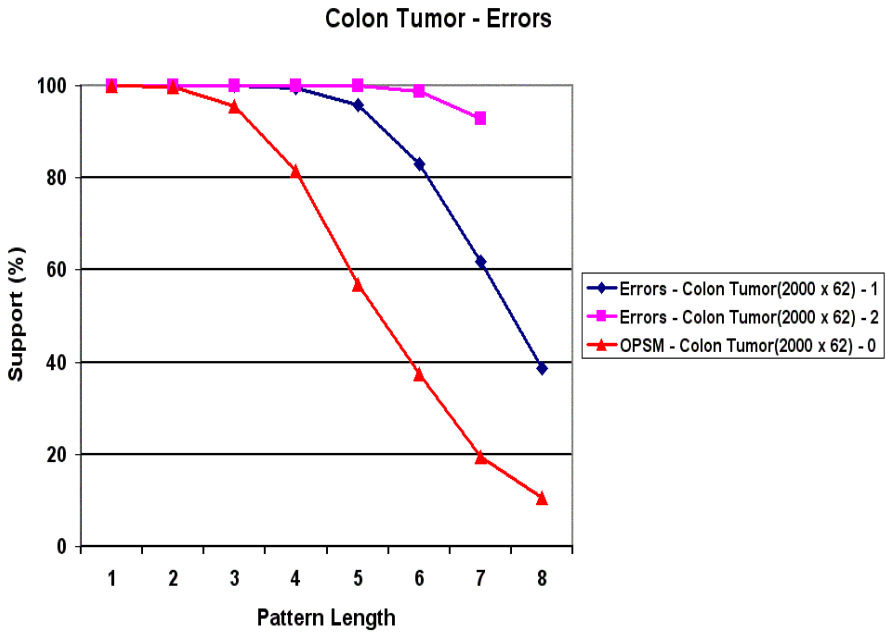


Figure 6.5: Colon Tumor Order Preserving Submatrices with Errors Allowed

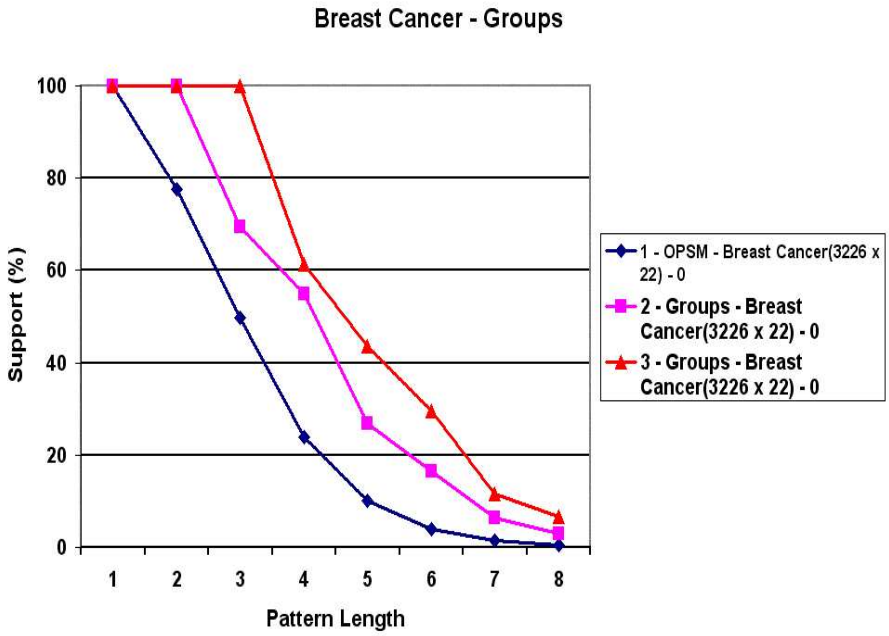


Figure 6.6: Breast Cancer Order Preserving Submatrices with groups of tissues

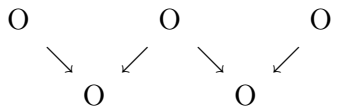
column is in order with all others (transitive), in increasing groups sequence the order relation is weaker. This result in more rows supporting every patterns.

6.2.4 Layered Submatrices

For the layered submatrices the breast cancer dataset was divided to 3 groups: Normal, BRCA1 and BRCA2. We have limited the patterns to balanced layers - the layers must be in the same size (± 1). The results are shown in the prefix of this section (Figure 6.2).

6.2.5 Connected DAG

First few runs of the algorithm produced a zigzag graph:



The reason is that such graph implies the least number of constraints between the tissues. For example OPSM pattern of length s, $\pi = (\pi_1, \ldots, \pi_s)$ implies the following constrains:

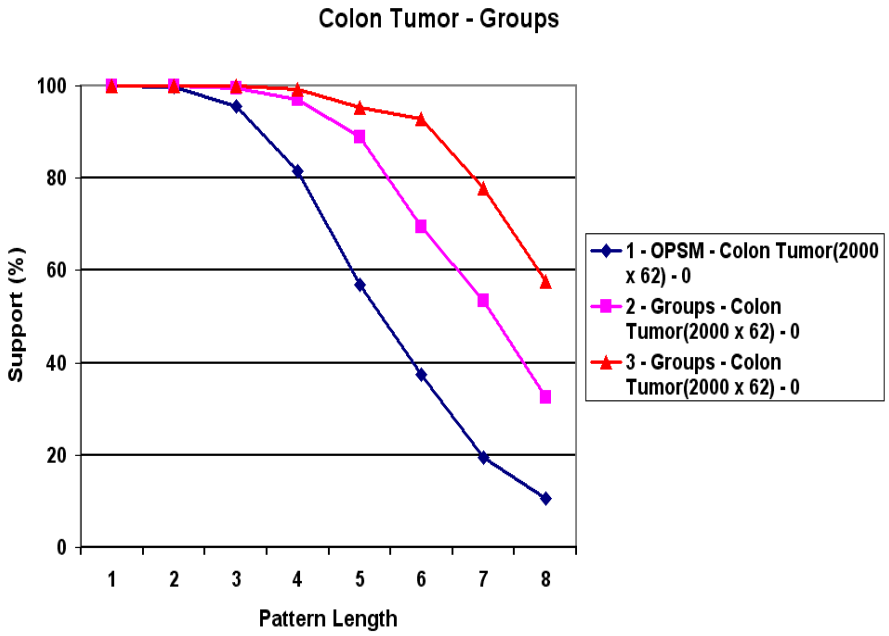


Figure 6.7: Colon Tumor Order Preserving Submatrices with groups of tissues

$$
\begin{aligned}
&\pi_2: \quad \pi_1 < \pi_2 \\
&\pi_3: \quad \pi_1 < \pi_3 \quad \pi_2 < \pi_3 \\
&\pi_4: \quad \pi_1 < \pi_4 \quad \pi_2 < \pi_4 \quad \pi_3 < \pi_4 \\
&\ldots
\end{aligned}
$$

Total of $\frac{s^2}{2}$ constraints. On the other hand, a zigzag graph implies only $s-1$ constraints (which is the minimal number for a connected DAG, trivial to prove). For that reason further tests were done under a more limited schema - binary tree pattern.

Under the tree pattern limit the resulting patterns were balanced trees. The reason for balanced trees is that balanced tree contains the least constraint of all trees. The number of constraints for a balanced tree (assuming a full tree) is

$$\sum_{i=1}^{\log s - 1} i2^i = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \ldots$$

Claim. *Balanced binary tree pattern implies the least constraints of all binary tree patterns.*

Proof. (Sketch) In a tree, each level implies a certain number of constraints (1st level zero, 2nd level one, 3rd level two and so on). If we compare a balanced to any non-balanced tree, and create a difference set of nodes. The nodes can be put in pairs such that every tree node in

difference set from the non-balanced tree have a pair from the balanced tree which implies less constrains (less deeper in the tree). □

Chapter 7

Future Work

Optimizations : We have used some current state-of-the-art techniques for frequent itemset mining to mine the patterns described in the paper. Any other technique that would adapted could bring an order of magnitude improvement. One other problem with the algorithms described in this paper is they assume the dataset can reside in the main memory. Development of out-of-core algorithms will create an opportunity for mining those patterns on much larger datasets.

New Pattern : So far, in all the patterns we have shown, we ignored the values of the gene expression (after found the 2-items order). For example a pattern where we allow errors of size ϵ.

Bibliography

[ABN+99] U. Alon, N. Barkai, D.A. Notterman, K. Gish, S. Ybarra, D. Mack, and A.J. Levine. Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide array. In *Proc. Natl. Acad. Sci. USA*, pages 6745–6750, 1999.

[AGY+02] Jay Ayres, J. E. Gehrke, Tomi Yiu, , and Jason Flannick. Sequential pattern mining using bitmaps. In *SIGKDD*, 2002.

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *ICDE*, pages 3–14, 1995.

[ASS+02] Scott A. Armstrong, Jane E. Staunton, Lewis B. Silverman, Rob Pieters, Monique L. den Boer, Mark D. Minden, Stephen E. Sallan, Eric S. Lander, Todd R. Golub, and Stanley J. Korsmeyer. Specify a distinct gene expression profile that distinguishes a unique leukemia. *Nature Genetics*, 30:41–47, 2002.

[BCG01] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke. Mafia: a maximal frequent itemset algorithm for transactional databases. In *ICDE*, 2001.

[BDCKY02] Amir Ben-Dor, Benny Chor, Richard M. Karp, and Zohar Yakhini. Discovering local structure in gene expression data: the order-preserving submatrix problem. In *RECOMB*, pages 49–57, 2002.

[BEB99] D. Bassett, M. Eisen, and M. Boguski. Gene expression it's all in your mine. *Nature Genetics supplement*, 21:51–55, 1999.

[BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD*, pages 255–264, 1997.

[BS00] S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Inform. Process. Lett.*, 76(1-2):7–11, 2000.

[CC00] Yizong Cheng and George M. Church. Biclustering of expression data. In *ISMB 2000*, volume 8, pages 93–103, 2000.

[CYBD+01] Y. Chen, Z. Yakhini, A. Ben-Don, E. Dougherty, J. Trent, and M. Bittner. Analysis of expression patterns: The scope of the problem, the problem of scope. In *Disease Markers*, 2001.

[ESBB98] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *PNAS*, 95:14863–14868, 1998.

[FS03] Amos Fiat and Sagi Shporer. Aim: Another itemset miner. In *Proceedings of the Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, 2003.

[GB00] T. Gaasterland and S. Bekiranov. Making the most of microarray data. *Natute Genetetics*, 24:204–206, 2000.

[GJ79] M.R. Garey and D.S. Johnson. *COMPUTERS AND INTERACTABILITY A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[GLD00] G. Getz, E. Levine, and E. Domany. Coupled two-way clustering analysis of gene microarray data. *PNAS*, 94:12079–12084, 2000.

[GSPT+99] T. R. Golub, D. K. Slonim, C. Huard P. Tamayo, J.P. Mesirov M. Gaasenbeek, H. Coller, M. L. Loh, J. R. Downing, M. A. Caligiuri, C. D. Bloomfield, and E. S. Lander. Molecular. Classification of cancer: class discovery and class prediction by gene expression monitoring. *Science*, 286:531–537, 1999.

[HDC+01] Ingrid Hedenfalk, David Duggan, Yidong Chen, Michael Radmacher, Michael Bittner, Richard Simon, Paul Meltzer, Barry Gusterson, Manel Esteller, Mark Raffeld,

Zohar Yakhini, Amir Ben-Dor, Edward Dougherty, Juha Kononen, Lukas Bubendorf, Wilfrid Fehrle, Stefania Pittaluga, Sofia Gruvberger, Niklas Loman, Oskar Johannsson, Hakan Olsson, Benjamin Wilfond, Guido Sauter, Olli-P. Kallioniemi, Ake Borg, and Jeffrey Trent. Gene-expression profiles in hereditary breast cancer. *NEJM*, 334:539–548, 2001.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, 2000.

[Joh87] D.S. Johnson. The np-completeness column: an ongoing guide. In *J. Algorithms*, volume 8, pages 438–448, 1987.

[Jr98] Roberto J. Bayardo Jr. Efficiently mining long patterns from databases. In *SIGMOD*, pages 85–93, 1998.

[LK98] Dao-I Lin and Zvi M. Kedem. Pincer search: A new algorithm for discovering the maximum frequent set. In *EDBT'98*, volume 1377 of *Lecture Notes in Computer Science*, pages 105–119, 1998.

[LL02] Jinyan Li and Huiqing Liu. Bio-medical data analysis. In *http://sdmc.lit.org.sg:8080/GEDatasets/index.html*, 2002.

[LO02] L. Lazzeroni and A. Owen. Plaid models for gene expression data. *Statistica Sinica*, 12:61–86, 2002.

[SES02] R. Sharan, N. Elkon, and Ron Shamir. Cluster analysis and its application to gene expresion data. In *Ernst Schering workshop on Bioinformatics and Genome Analysis*, 2002.

[SHS+00] Pradeep Shenoy, Jayant R. Haritsa, S. Sundarshan, Gaurav Bhalotia, Mayank Bawa, and Devavrat Shah. Turbo-charging vertical mining of large databases. In *SIGMOD*, 2000.

[SK02] M. Seno and G. Karypis. Slpminer: An algorithm for finding frequent sequential patterns using length decreasing support constraint. In *ICDE*, 2002.

[SPM+02] Pomeroy SL, Tamayo P, Gaasenbeek M, Sturla LM, Angelo M, McLaughlin ME, Kim JY, Goumnerova LC, Black PM, Lau C, Allen JC, Zagzag D, Olson JM, Curran T, Wetmore C, Biegel JA, Poggio T, Mukherjee S, Rifkin R, Califano A, Stolovitzky G, Louis DN, Mesirov JP, Lander ES, and Golub TR. Prediction

of central nervous system embryonal tumour outcome based on gene expression. *Nature*, 415:436–442, 2002.

[Toi96] Hannu Toivonen. Sampling large databases for association rules. In *VLDB*, pages 134–145, 1996.

[TSS02] Amos Tanay, Roded Sharan, and Ron Shamir. Discovering statistically significant biclusters in gene expression data. *Bioinformatics*, 18:S136–S144, 2002.

[WYY01] Wei Wang, Jiong Yang, and Philip S. Yu. Meta-patterns: Revealing hidden periodic patterns. In *ICDM*, pages 550–557, 2001.

[YC96] S. Yen and A. Chen. An efficient approach to discovering knowledge from large databases. In *4th International Conference on Parallel and Distributed Information Systems*, 1996.

[YWY00] Jiong Yang, Wei Wang, and Philip S. Yu. Mining asynchronous periodic patterns in time series data. In *Knowledge Discovery and Data Mining*, pages 275–279, 2000.

[Zak00] Mohammed J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000.

[Zak01] Mohammed Javeed Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1/2):31–60, 2001.

[ZG03] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *SIGKDD*, 2003.

Appendix A

Longest Increasing Subsequence

We transform the problem sequence into a graph, and run the algorithm on the graph. Given a series $D(r_i, \pi_1), D(r_i, \pi_2), \ldots, D(r_i, \pi_s)$ for each row i we build the following directed graph $G_i(V_i, E_i)$:

- $V_i = \{v_j \mid D(r_i, \pi_j)\}$
- $E_i = \{(v_k, v_\ell) \mid D(r_i, \pi_k) < D(r_i, \pi_\ell) \bigwedge k < \ell < k + \gamma + 1\}$.

Lemma. *The series $D(r_i, \pi_1), D(r_i, \pi_2), \ldots, D(r_i, \pi_s)$ contains a longest increasing subsequence of length $r \geq s - \gamma$ or more iff the graph $G_i(V_i, E_i)$ contains a direct path of length $r \geq s - \gamma$.*

Proof. $\Rightarrow$ Assume there is an increasing sequence in $D(r_i, \pi_1), D(r_i, \pi_2), \ldots, D(r_i, \pi_s)$ of length $r \geq s - \gamma$. Let the series be $D(r_i, \pi_{p_1}), D(r_i, \pi_{p_2}), \ldots, D(r_i, \pi_{p_r})$. There are r vertexes in the graph that corresponds to the items in the series. There is an edge between each two neighbors in the series (the gap between each two items is less than $\gamma + 1$ because the series length is $s - \gamma$). Those edges creates a path of length r.

$\Leftarrow$ Assume there is a path of length $r \geq s - \gamma$ in the graph. From the construction, the items in the series with the same indices of the vertexes create an increasing sequence of length $r \geq s - \gamma$. □

The algorithm: The algorithm for finding the longest path is based on two phases:

- *Building Phase:* Move from v_1 to v_s and mark each node as follows: If no edge is pointed to the vertex, mark the vertex with '1', otherwise, from all the edges pointing to the vertex, select the edge originating from a vertex with the highest marking, add 1 and set this mark on the current vertex.

- *Search Phase:* Move from $v_{s-\gamma}$ to v_s, check the marking of each vertex and return the highest marking. If this marking is greater or equal then $s-\gamma$ this is the longest increasing subsequence, otherwise no increasing subsequence of length $s-\gamma$ exists.

See figure 4.2 for example of the graph created, the marking in the vertexes after running the algorithm.

Lemma. *The algorithm returns the length of the longest directed path in the graph of length $s-\gamma$ or more, and returns no result otherwise.*

Proof. We show by negation that any result but the correct one is impossible. There are four possible wrong answers:

- A valid longest path exists, and the algorithm returns no result - Let $\lambda \geq s-\gamma$ be the length of the longest path. Let $V'' = \{v_{\ell_1}, \ldots, v_{\ell_\lambda}\}$ be the vertexes on the path. By the first step of the algorithm, v_{ℓ_1} is marked '1' (no other vertex has a smaller value which points to v_{ℓ_1} - contradiction to the maximality of V''), v_{ℓ_2} is marked 2 ($v_{\ell_1}+1$), again there aren't two vertexes smaller than v_{ℓ_2} as it contradicts the maximality of V''', and so on until v_{ℓ_λ} is marked λ. Because the length of V'' is no less than $s-\gamma$ then the last vertex in V'', v_{ℓ_λ}, is one of the last γ vertexes, therefor it is selected (the marking is λ, which is equal or greater than $s-\gamma$), in contradiction that the algorithm returns no result.

- A valid longest path exists, and the algorithm returns lower value - Let $\lambda \geq s-\gamma$ be the length of the longest path. Let $V'' = \{v_{\ell_1}, \ldots, v_{\ell_\lambda}\}$ be the vertexes on the path. As shown in the previous item, the last vertex, v_{ℓ_λ}, which lies in the last $s-\gamma$) vertexes, is marked λ and the second step returns the maximal value, this contradicts that a value lower than λ was returned.

- A valid longest path exists, and the algorithm returns higher value - from the value λ returned a path of length λ can be build which is longer than the longest path, in contradiction that the longest path is the longest path.

- A valid longest path does not exists, and the algorithm returns a value - from the value returned a path of length $s-\gamma$ or greater can be build, in contradiction that no valid longest path exists.

□

Appendix B

Patterns Found in the Breast Cancer Dataset

In this appendix we show some of the patterns found in the breast cancer dataset.

NEJM-PatientID	Mutation	Title	Column
1	BRCA1	s1996	0
5	BRCA1	s1822	1
3	BRCA1	s1714	2
7	BRCA1	s1224	3
2	BRCA1	s1252	4
4	BRCA1	s1510	5
10	BRCA2	s1900	6
9	BRCA2	s1787	7
8	BRCA2	1721	8
10	BRCA2	s1486	9
16	Sporadic	s1572	10
17	Sporadic	s1324	11
15	Sporadic	s1649	12
18	Sporadic	s1320	13
19	Sporadic	s1542	14
21	Sporadic	s1281	15
20	Sporadic/Meth.BRCA1	s1321	16
6	BRCA1	s1905	17
13	BRCA2	s1816	18
14	BRCA2	s1616	19
11	BRCA2	s1063	20
12	BRCA2	s1936	21

Figure B.1: List of all the columns in the breast cancer dataset, with the meaning of every column

Length	Pattern	Abs. Support	Support (%)
2	19 4	2506	77.6%
3	19 21 4	1602	49.6%
4	19 20 21 4	772	23.9%
5	19 20 21 6 4	326	10.1%
6	19 20 21 3 4 1	127	3.9%
7	19 20 21 9 6 4 1	48	1.4%
8	19 20 11 5 6 3 4 1	18	0.5%

Figure B.2: OPSM patterns found (Maximal support for every length)

γ	Length	Pattern	Abs. Support	Support (%)
1	3	15 19 4	3123	96.8%
1	4	19 20 21 4	2367	73.3%
1	5	19 20 21 18 4	1474	45.6%
1	6	19 20 21 6 4 1	739	22.9%
1	7	19 20 21 9 6 4 1	332	10.2%
2	7	19 20 11 21 6 1 4	1141	35.3%
2	8	19 20 11 21 6 13 4 1	576	17.8%

Figure B.3: OPSM with Errors patterns found

Group Len	Length	Pattern	Abs. Support	Support (%)
2	3	{19} {1 4}	2243	69.5%
2	4	{19 20} {1 4}	1772	54.9%
2	5	{19 20} {21} {1 4}	870	26.9%
2	6	{19 20} {6 21} {1 4}	530	16.4%
2	7	{19 20} {21} {6 9} {1 4}	205	6.3%
2	8	{19 20} {14 21} {6 9} {1 4}	99	3.0%
3	4	{19} {1 4 21}	1977	61.2%
3	5	{14 19 20} {1 4}	1409	43.6%
3	6	{14 19 20} {1 4 13}	956	29.6%
3	7	{19} {18 20 21} {1 4 7}	377	11.6%
3	8	{19 20} {11 14 21} {1 4 13}	214	6.6%

Figure B.4: Increasing Groups Sequence patterns found

Length	Pattern	Abs. Support	Support (%)
6	19⇒20, 19⇒21, 20⇒1, 20⇒7, 21⇒4	766	23.7%
8	19⇒14, 19⇒20, 14⇒1, 14⇒13, 20⇒7, 20⇒21, 21⇒4	366	11.3%

Figure B.5: Connected DAG - Binary tree patterns found