VIARA N. POPOVA

Knowledge Discovery and Monotonicity

KNOWLEDGE DISCOVERY AND MONOTONICITY

Opsporen van kennis en monotoniciteit

Thesis

**to obtain the degree of Doctor from the
Erasmus University Rotterdam
by command of the
Rector Magnificus**

Prof.dr. S.W.J. Lamberts

and according to the decision of the Doctorate Board

The public defense shall be held on

Thursday 1 April 2004 at 13:30 hrs

by

**Viara Nikolaeva Popova
born in Bourgas, Bulgaria**

Erasmus Research Institute of Management (ERIM) Rotterdam School of Management / Rotterdam School of Economics Erasmus University Rotterdam

Internet: http://www.erim.eur.nl

ERIM Electronic Series Portal: http://hdl.handle.net/1765/1

ERIM Ph.D. Series Research in Management EPS-2004-37-LIS

The research reported in this thesis has been carried out in cooperation with SIKS, the Dutch Research School for Information and Knowledge Systems.

SIKS Dissertation Series No. 2004-05

ISBN 90 - 5892 - 061 - 5

Acknowledgements

Many people contributed to this research in different ways. They should not be left unmentioned. First of all, my gratitude goes to Cor for his patience and help along the way. I also want to thank Wim, Walter and Jeannette - chapter 5 is based on our joint research on Frequent Patterns and I enjoyed a lot working with you. Eelco and Robbert, the figures in this thesis would not have been good enough without your help. To everybody else, thanks for the moral support and practical help to make these four years easier for me.

Contents

Chapter 1

Introduction

1.1 John Smith and His Grades Prediction System

John is a student and has to prepare for an important exam. He has been through this situation many times before with various success but this time he decides to try something new. Something revolutionary that is bound to succeed. Something scientific!

He has decided to find the answer to the question of how his professor decides which grades to give to the students. And using this, John would be able to predict his own grade even before he went to the exam. Besides, he might also be able to find out what he should do in order to improve his chances.

So, he gets down to work. He starts by diving in all his previous examination experience to select which are the important factors that might influence the professor's decision:

– First of all "smartness" of course – he has seen how easily his very smart colleagues get their high grades.

– Then it should be "appearance". Well if he were a teacher he would definitely give higher grades to all those pretty girls ...

– "Self-confidence" is a very important factor as well – one should be able to present oneself.

– And, well, the amount of material one has studied should also matter somehow, he guesses.

He cannot think of anything else, so, he decides that these are all the relevant factors. He then contacts a number of older students and questions them about the exam and their self-evaluation on the selected criteria. He ends up with a dozen examples which he organizes in a nice table:

smartness	appearance	self-confidence	preparation	grade
high	ugly	low	high	8
average	pretty	high	average	7
average	neutral	low	average	6
...	...	...	...	...

That should do, he concludes, and proceeds with the analysis of the data. Looking at the table he decides that the following rules are true:

$$\text{smartness} = \text{high}, \text{self-confidence} = \text{high} \rightarrow \text{grade} \geq 8$$

$$\text{smartness} = \text{average}, \text{appearance} = \text{pretty} \rightarrow \text{grade} \geq 7$$

Well, he is not too smart, he admits with regret, he is more average. However he is quite good-looking. Therefore the second rule predicts at least 7. And this is definitely a good grade. Hmm, the preparation didn't seem to matter ...

Fortunately not, he smiles in relief, since he didn't have any time for it anyway from all that hard scientific work.

1.2 Knowledge Discovery

The reader should not take the example of John Smith too seriously. However it might help in pointing our attention in the right direction – the area of knowledge discovery. The researchers in this area try to bridge the gap between data and knowledge and the benefits from such research are apparent. However the so-formulated goal is far from trivial.

Looking at the problem from a general point of view, knowledge discovery usually relies on two major resources: data and background information. The background information is all the domain knowledge that we have or the plausible assumptions that we can make about the goal. Having some idea of what we are looking for would guide us through the space of all possible solutions. Data, on the other hand, can help by narrowing the space of all possible solutions and thus make it easier to find the goal.

The two factors obviously complement each other. When enough data is available, the need for background knowledge and assumptions is less. We can simply let data "speak for itself". When, however, data is limited we have to rely on additional information to ensure the quality of the discovered knowledge.

Other issues also play a role. Computational limitations can prevent us from processing amounts of data that are too large. Nowadays computer resources become increasingly more powerful and accessible. At the same time, the data that becomes available and which we would like to use grows as well. The conflict will probably always exist as the more technology advances the more ambitious goals will be considered.

Background information, on the other hand, might be unproven and biased and lead us in the wrong direction. The area of statistics often relies on model assumptions while machine learning methods often try to limit the assumptions and use only the data.

In practice we would prefer to make fewer assumptions but the availability of reliable prior knowledge[1] can be valuable. In the following section we present a brief overview of the types of prior knowledge we might rely on.

1.3 Background Knowledge

We can distinguish the following main types of background knowledge that might be available to us in the process of knowledge discovery:

– knowledge about the describing attributes and the outcome (class): here we include the knowledge we might have about the meaning of the attributes, the way they can be measured, and which is a meaningful way to represent them, etc.

– knowledge about the relationship between the describing attributes and the outcome (class): here we include the available knowledge about how the attributes relate to each other, which attributes are relevant in explaining the class, which of them are the most important ones, how they influence the change in the class, the character/shape of the function, etc.

– knowledge about the desired application and possible use of the system to be built: this is usually normative knowledge which restricts our choice on the methods we can use in order to meet the requirements of the application area. It can include knowledge about misclassification costs, requirements for transparency, interpretability, degree of correctness, some computational issues, the need for explanation of the proposed solution/prediction, etc.

– other knowledge: more general knowledge that can be applied to the specific domain.

In this research we focus on the background knowledge about a specific property – the monotonicity property which will be described in the following section.

[1] In the literature, background knowledge is also referred to as prior knowledge or domain knowledge.

1.4 Monotonicity

Monotonicity is a ubiquitous property in all areas of our life which appears in different ways. Let us start with a number of examples before we proceed with the more formal definition. These examples are from completely different domains but the similarity between them might soon become apparent.

1.4.1 Economics

First we give two examples from the area of economics.

Bankruptcy prediction is the problem of detecting in advance whether a company is going to go bankrupt, so that appropriate measures can be taken on time. A related problem is that of credit rating which has to assign scores to companies applying for credit. The scores are used to help banks decide to which companies they should choose to give the credit. The choice should obviously be made in such a way which maximizes the chance the credit will be paid back and is therefore connected to the company's financial situation.

The data that is available about the companies is the set of financial indicators taken from their annual reports. Among those parameters, a suitable subset is selected and used as the data on which the decision will be based.

For these parameters it holds that the better the company performs on a particular parameter the better its overall situation is. Therefore if we compare two companies such that the first company dominates the second one on all financial indicators then the overall evaluation of the second one cannot be higher than that of the first one. This rule should apply both for the credit rating strategy used by the bank as well as for the bankruptcy prediction strategy.

1.4.2 Natural Sciences

Numerous examples of monotone relationships between factors can be given from all natural sciences. Let us choose a couple of simple cases where monotonicity is present.

In biology it is known that the bigger the animals of a certain group evolve the lower the number of individuals becomes. This is largely due to the increasing amount of food necessary to support a single animal. The availability of food becomes a problem and the same geographical area can support the existence of fewer animals. Such knowledge is used, for example, by paleontologists in studies on dinosaurs.

A different example indicates that bigger containers of the same shape cool down the stored hot substance more slowly than their smaller counterparts. It holds that the bigger the container the slower it cools down. Here the size of the surface that is in contact with the outside world plays a role.

1.4.3 Natural Language

Monotonicity is also reflected in the way we speak. It is a known property in the area of natural language processing. Expressions like “many”, “few”, “some”, “at least two”, “more than five”, “both”, etc. are called quantifiers and some of them can be monotone increasing or monotone decreasing (see, for example, [8]). We demonstrate this with two examples. Consider first the sentence:

Several children sing loudly.

We can make the sentence more general by leaving out “loudly”:

Several children sing.

Notice that if the first statement is true than the second (more general or stronger) statement is also true:

Several children sing loudly. $\Rightarrow$ Several children sing.

By the way, the reverse implication is not true since it is possible that none of the children who sing sings loudly:

Several children sing. $\not\Rightarrow$ Several children sing loudly.

Such quantifiers are called upward monotone. A bit more formal definition will state that upward monotone quantifiers are closed under extension. Other examples of upward quantifiers are: “most”, “all”, “at least three”, etc.

Consider now another sentence:

No children sing.

We make it more specific by adding “loudly”:

No children sing loudly.

If the first statement is true than the second (more specific or weaker) one will also be true:

No children sing. $\Rightarrow$ No children sing loudly.

This is the reverse property of the upward monotonicity – quantifiers that are closed under contraction are called downward monotone. Other downward monotone quantifiers are: “few”, “neither”, “at most two”, etc.

Notice that in the second example the quantifier “no” is not upward monotone since the more specific statement does not imply the more general one:

No children sing loudly. $\not\Rightarrow$ No children sing.

The monotonicity property proves to be important in natural language studies because the monotone quantifiers can be much easier to evaluate. If we ask the question whether a particular quantified statement is true or false, the minimal number of elements from the domain that we have to check to verify or falsify the statement is often significantly lower for the monotone quantifiers than for the non-monotone ones.

1.4.4 Mathematics

The above given examples are fairly different and still there is a red line connecting them. We now look at it from a more formal and precise point of view. Being such an ubiquitous property in our lives, monotonicity is well studied in mathematics. The most straight-forward way of representing it is as a property of a function.

Let f be a function $f : X \to Y$, where X and Y are (partially) ordered sets. f is called non-decreasing if for each pair $x_1, x_2 \in X$ such that $x_1 < x_2$ it is true that $f(x_1) \leq f(x_2)$. Similarly f is called non-increasing if for each pair $x_1, x_2 \in X$ such that $x_1 < x_2$ it is true that $f(x_1) \geq f(x_2)$.

f is called monotone if it is either non-decreasing or non-increasing. An equivalent definition states that a function is monotone if its first derivative does not change sign.[2]

Let us take another point of view. A property of graph is a collection of graphs which is closed under isomorphism. A property P is called monotone if for any graph G satisfying P, every subgraph of G also satisfies P. In other words, P is a family of graphs which is closed under taking subgraphs. An example of such property is the property of being a Hamiltonian graph.

In certain areas the "mirroring" notion of anti-monotonicity is also used. A graph property P is called anti-monotone if for any graph G if G does not satisfy P then no graph $G_1 \supseteq G$ satisfies P.

It is easy to see that monotonicity of functions and of graph properties describe a similar phenomenon in a different setting. For the most part of this thesis we will rely on the functional point of view, however, the other perspective will also play a role in the last chapter (chapter 5).

1.4.5 Operations Research

One example of a monotone function can be given from Game Theory. A co-operative game is a function $v : 2^N \to \mathbb{R}$ such that $v(\emptyset) = 0$, N denotes the set of players and the subsets of N are called coalitions. A simple game is a co-operative game $v : 2^N \to \{0, 1\}$ such that $v(N) = 1$ and v is non-decreasing, therefore monotone. A coalition C is winning if $v(C) = 1$ and losing if $v(C) = 0$. An example of such a game is the weighted majority voting.

[2]Note that we do not restrict the first derivative to be continuous.

Another example comes from reliability theory which considers the probability that a structure will perform the task it was designed for (called the reliability of the structure). With that respect, a structure can be represented as a Boolean function of its components where 0 represents failure of the component/structure and 1 denotes that the component/structure is functioning. This function is called non-degenerate if it has at least one relevant component and semi-coherent if it is 0 when all components fail and 1 when all components function. For non-degenerate functions, the property of semi-coherence is equivalent to monotonicity of the function. It is interesting to realize that reliability theory and game theory are closely connected and a semi-coherent structure in reliability theory corresponds to the notion of a simple game in game theory. The reader is referred to [70] for a detailed discussion on both topics.

1.5 Monotonicity in Knowledge Discovery

We now concentrate on the area of knowledge discovery and the way the monotonicity property appears there. We first briefly discuss a couple of examples within different aspects of knowledge discovery and then concentrate on the sub-area of classification which plays a very important role in this thesis.

The concept of a reduct comes from the Rough Sets theory and is discussed in detail in chapter 2. Informally speaking, a (non-minimal) reduct is such a subset of attributes that does not introduce additional inconsistencies in the data. We can, therefore, use reducts to detect redundant attributes. An important property of reducts is that any superset of a reduct is also a reduct. In other terms, if the subset relation is the ordering relation on the set of all subsets of attributes, then the the concept of a reduct is a monotone Boolean function.

A similar fact can be drawn from Database Theory where one of the most important concepts is that of a key for a data table. A set of attributes X is called a key if no two rows of the data table agree on every attribute in X. Again it is an important property that every key must contain some minimal key and conversely every superset of a key is also a key. In fact, a key is a reduct of a table where no decision attribute is designated.[3]

Association rules generation is another area of knowledge discovery (discussed in chapter 5) where monotonicity plays a role. Here the concept of a frequent pattern is important: a set of items that appear together more often than a predefined threshold. The property of being an infrequent pattern is monotone while the property of being a frequent pattern is anti-monotone. This knowledge can be used in reducing the number of candidate patterns that need to be counted and therefore helps to speed the frequent patterns generation algorithms. A similar argument holds for the generation of association rules.

[3]This variation of the definition of a reduct is not discussed in chapter 2. The reader is referred to [60] for more information.

In this thesis, monotonicity is discussed from the aspects of attribute reduction, function decomposition, missing values handling, frequent patterns generation and classification. We now turn our attention to classification.

1.5.1 Monotone Classification

Classification is a sub-area of knowledge discovery that refers to the problem of predicting the value of a target variable by building a model based on some relevant independent variables where the target and the independent variables are of discrete type.

As background knowledge in classification, monotonicity relates to most of the previously defined groups (see section 1.3). First of all it includes the information that the attributes come from ordered domains. Furthermore it indicates that the target variable is a monotone function of the describing independent attributes. It also poses a requirement to the system to be built that its predictions should also satisfy the same property.

In the following subsections, the effect of monotonicity from the point of view of classification will be defined more precisely. Three points of view can be relevant: the problem, the data set drawn from it and the classifier for prediction of the target variable for future instances.

1.5.2 Monotone Problems

We start by defining a *problem* P as the tuple $P = \langle X, A, f \rangle$ where:

$X = \{x_1, x_2, \ldots, x_m\}$ is the input space, the set of all possible objects that we want to be able to classify;

$A = \{a_1, a_2, \ldots, a_n\}$ is the set of attributes that we use to describe the objects, we will also refer to them as condition attributes;

$f : X \to F$ is a function where F is the set of possible values of the outcome (in the following we will also refer to it as the class or the decision attribute). f is the function that we want to guess or approximate.

We call the problem P a *monotone problem* if:

1. X is partially ordered using the attributes in A;[4]
2. f is a monotone function: $\forall x_i, x_j \in X,\ x_i \leq x_j \Rightarrow f(x_i) \leq f(x_j)$.

Note that, in fact, here we describe a positive function, however, any negative relation can be easily transformed to a positive one by redefining the corresponding attribute. We can change the coding of the values so that the highest one

[4]Although the requirement for partial order is sufficient for our discussion, the definition can be generalized to only requiring quasi-order (see [15]).

gets the lowest label and the lowest one gets the highest label. We can also use a different but "mirroring" attribute – for example an attribute called "degree of lightness" might be replaced by the attribute "degree of darkness".

Since we do not have a complete specification of the function f we cannot check the property by computation. Here we rely on prior knowledge coming from experience or from other research on the underlying mechanisms in the domain.

1.5.3 Monotone Data Sets

A *data set* D_P drawn from the problem P is a tuple $\langle U, A, f_D \rangle$ where:

$U = \{x_1, x_2, \dots, x_k\} \subset X$ is a set of objects;

$A = \{a_1, a_2, \dots, a_n\}$ is the attribute set of P;[5]

$f_D : U \to F_D \subseteq F$ where f_D is completely specified over U so that $f_D(x) \approx f(x)$ where "$\approx$" is an application specific measure of closeness or equality. In the ideal case f_D is equal to f over U, however, in real life applications the available data is sometimes infested with noise and/or imprecise measurements. In the rest of the thesis we assume equality unless noise is explicitly taken into account.[6] Note that, in the case of equality, we may say that f_D is a partially defined function over X which has an extension f.

A data set D (in the following we will skip the subscript P if no confusion arises) is called a *monotone data set* if:

1. U is partially ordered using the attributes in A,
2. the function f_D is monotone: $\forall x_i, x_j \in U,\ x_i \leq x_j \Rightarrow f_D(x_i) \leq f_D(x_j)$.

The monotonicity of a data set is verifiable in a straightforward way by exhaustive computation.

Let us now look at how the monotonicity of the problem relates to the monotonicity of a data set drawn from it.

In a noiseless and precise data collection process where f is an extension of f_D, it is true that the monotonicity of the problem implies the monotonicity of the data set. The reverse, however, is not true. That is easy to see if we consider

[5] A more general formulation might state that $A_D \subseteq A_P$ or $A_P \subseteq A_D$ or even $A_D \cap A_P \neq \emptyset$. However, since the choice of the relevant attributes out of all possible ones comes before the classifier generation process, relies on experts' knowledge and is therefore outside of the scope of this analysis, we choose the simpler formulation which is sufficient for our goals.

[6] In this way we also exclude probabilistic problems where the data points are assigned distributions instead of single labels. This case, however, falls outside of the scope of this thesis and we choose the simplifying assumption of equality.

the extreme case of a data set containing only one object. Such a data set is always monotone because there are no other objects to compare with. But since such a data set can be drawn from any problem, then every problem, monotone or not, can produce many monotone data sets.

When the data set is noisy this might influence the monotonicity as well. If the problem is monotone it might still generate a non-monotone data set as some values will be incorrect or imprecise. In this case we can only rely on domain knowledge to assure us that the problem is monotone and then the non-monotonicity of the data set will be interpreted as a sign for the presence of noise.

More precisely we can represent the situation as follows. Let the data set $D = \langle U, A, f_D \rangle$ be the one generated based on the problem P in the ideal case (with no noise or imprecision). In this case f is an extension of f_D. In practice, however, the data set we record is a noisy version of D denoted by D^* where $D^* = \langle U, A, f_D^* \rangle$ and $f_D^* : U \to F_D$.[7] The monotonicity of D does not imply the monotonicity of D^*.

In the rest of this thesis when we discuss the monotonicity of a data set we assume that we know the underlying problem to be monotone.

1.5.4 Monotone Classifiers

The goal of the classification methods is to find a classifier using the available data set. A *classifier* based on a data set D is defined as a function $\widehat{f} : \widehat{U} \to F_D$ where $U \subset \widehat{U} \subseteq X$ and $\widehat{f}(x) \approx f_D(x)$ for $x \in U$. Here again "$\approx$" is a domain specific measure of closeness. In the ideal case when the data set contains no noise and imprecision we would like the closeness to be equality.[8]

Note that we chose to define $\widehat{f}$ over $\widehat{U}$ such that $U \subset \widehat{U} \subseteq X$. Ideally we would like this $\widehat{U}$ to be equal to X which would mean that we generalize from a small subset to the whole input space. However there exist many algorithms for generating classifiers that do not cover the whole input space – for them the requirement is $\widehat{U}$ to be as close to X as possible.

Let us assume that the available data set is a noisy version D^* of D. In that case we cannot require of the classifier we are about to build to agree with f_D^* as we already know that it is noisy and therefore not equal to f on U (and f is our ultimate goal).

A large number of techniques are available for fighting with noise. Most of them we can represent as methods that (implicitly or explicitly) build an

[7]We make the simplifying assumption that the noisy values do not leave the ranges of the attributes in A and class values in F_D. Such noise is often easy to detect as it results in impossible values that are obviously incorrect.

[8]The function we want to guess or approximate is f and therefore we actually want $\widehat{f}$ to be close or equal to f. But the only data that we have about f is D, therefore we try to approximate that data hoping that it is in turn close enough to f.

approximation of the noisy data set D^{appr} such that $f_D^{appr} \approx f_D^*$ according to some appropriate measure for closeness and $f_D^{appr} \approx f_D$ according to a (different) appropriate measure for closeness. This second measure is often implicit and is based on certain assumptions made by the different methods. The approximation data set is further used for building the classifier.[9]

A classifier $\widehat{f}$ is called a *monotone classifier* if it is a monotone function.

The monotonicity property here is not interpreted as a characteristic that we detect afterwards but more as a requirement coming from the domain area and from the way we are going to apply the classifier. It is possible to generate a monotone classifier for a problem that is known to be non-monotone. That might be taken as a sign that the classifier does not have some of the important characteristics of the problem and is therefore not close enough to the goal. If, however, we know that the problem is monotone then monotonicity might be one of the requirements we pose to the classifier.

A number of methods take explicitly into account the monotonicity property of the problem and generate monotone or nearly monotone classifiers. Some examples can be given from the area of decision trees [9, 55, 18, 64, 13, 30, 65], neural networks [76, 35], rough sets [40, 12, 41], logical analysis of data [33, 25, 23, 24], decision lists [11], instance-based learning [10], etc.

It has to be emphasized that, while in some sub-areas of classification methods have already been developed for taking monotonicity into account, in most algorithms it has not been extensively studied or has not even been addressed at all. This thesis is a contribution in the direction of developing methods that can process monotone data and satisfy the restriction of monotonicity of the resulting classification and prediction model.

1.6 The Contribution of This Thesis

This thesis is divided in four parts each of which is connected to the above discussed monotonicity property however in a different way and from a different point of view. In this section the four problems will be briefly presented with an indication of where the specific contribution of the thesis lies.

The thesis is organized in four independent chapters each devoted to one of the specific problems. The two appendices give additional information on the data used in the experiments for the main chapters and additional experimentation results. In the following subsections we briefly present the four chapters and their contribution.

[9] One of the algorithms that explicitly generates the approximation data set is the Monotone Decision Trees algorithm discussed in chapter 3.2 with the proposed extension for handling noisy data. There the approximation data set is called the updated data set.

1.6.1 Ordinal Attribute Reduction and Rule Induction with Rough Sets

Chapter 2 takes its starting point from the Rough Sets theory. The classical Rough Sets theory does not take into account the monotonicity property of the data and therefore does not guarantee that the class prediction will satisfy the constraint. In our research, an extension of the methodology is proposed for classification with monotonicity constraints. It can be used for attribute reduction of monotone data sets such that preserves the monotonicity property by means of generating monotone reducts. It can further be used to generate rules that compose a monotone classifier.

The main contribution of our research in comparison to the only previous approach in the same direction is the method for rules generation which (unlike the earlier method) composes a consistent monotone classifier even when the data contains monotone inconsistencies. Furthermore, an alternative method for the generation of monotone reducts is proposed which is easier to comprehend and apply than the previous approach.

Most of the research has been previously published in [12].

1.6.2 Monotone Decision Trees

Chapter 3 focuses on monotonicity in the context of decision trees. It extends an existing algorithm that generates monotone decision trees for classification with monotonicity constraints. The original algorithm can only be applied on strictly monotone noiseless data. We propose an extension that allows the generation of monotone trees from noisy inconsistent data, new methods for pruning the trees so that they remain monotone and functions for monotone labelling of the leaves. Empirical comparison is also done between two splitting criteria for decision trees to give more insight into which one performs better in the setting of a monotone problem.

As a result of this research a set of methods become available which provide to the monotone trees generation process tools similar to those available for the general tree generation algorithms: noise handling, simplification, missing values handling, etc. That opens the door to easier application of the monotone decision trees algorithm in real-life setting.

Most of the chapter is based on the publications [17, 16].

1.6.3 Monotone Decomposition

Chapter 4 is in the area of functional decomposition applied to knowledge discovery. Again here the previously available methods do not take into account the monotonicity property. In this thesis, the existing methodology is extended to handle monotone problems in such a way that the decomposed function remains monotone.

We concentrate on the subproblem of determining whether there exists an extension of the positive scheme $f = g(S_0, h(S_1))$ for given subsets S_0 and S_1 of the set of attributes. Furthermore, we propose methods for finding such an extension with a minimal number of distinct values of the intermediate concept in order to minimize the complexity of the decomposed structure.

The main result of this research is in providing methods for generating a monotone classifier through decomposition. It could be used not only for classifying future unseen examples but also in getting better insight into the hierarchical structure of monotone data.

The results of this research have been published in [63].

1.6.4 Frequent Patterns

Finally chapter 5 lies in the area of data mining and, more specifically, association rules generation. It focuses on the first part of the problem – frequent patterns discovery. Two of the best algorithms in the literature are considered, namely, FP-growth and Depth-First implementations of Apriori. They are compared theoretically and empirically in order to give more insight into their differences and how certain features of the data sets can influence the performance of both algorithms.

Very little had previously been done in the area of theoretical comparison of frequent patterns generation algorithms and no previous experimental comparisons had included the Depth-First algorithm. Our research is therefore an important step forward in this direction. We propose two formulas for complexity analysis of the two algorithms in focus measuring two important factors influencing the performance – the number of database queries and the number of nodes in the data structure built by the algorithm. The experimental comparison uncovers some of the data properties that influence the difference in performance.

This research has been published in [36, 51].

Chapter 2

Ordinal Attribute Reduction and Rule Induction with Rough Sets

2.1 Introduction

The theory of Rough Sets dates back to the early 1980's. It was first proposed by Zdzisław Pawlak and was extensively presented in [59, 60]. A large number of researchers contributed to the further development of the field by extending and applying the theory.

Rough Sets theory is a mathematical tool based on sets theory with, as a main goal, the induction of approximations of concepts. The two main applications of the classical Rough Sets theory are in attribute reduction and classification. It was later applied within other areas such as unsupervised learning [53].

An important feature of Rough Sets theory is the fact that it does not make additional model assumptions and does not need special model parameters. It relies solely on the data available. This is a major difference from, for example, the methods of fuzzy sets where the agent is required to assign numerical values to express the imprecision of his knowledge. In Rough Sets theory, instead, the imprecision is expressed by qualitative concepts (approximations) derived from the data [60]. Another important feature is the ability to deal with noise and inconsistent data. In fact inconsistent data is processed in the same way as consistent data and no additional methods are required.

In this chapter an extension of the classical Rough Sets theory is proposed for handling monotone data. It provides methods for generating a classifier, by means of decision rules, which is guaranteed to be monotone over the whole

	age	experience	grades
Anna	21-30	none	good
Bill	21-30	none	good
Cathy	21-30	4-6	average
Dave	31-40	1-3	excellent
Emma	31-40	4-6	good
Frank	31-40	4-6	good

Table 2.1: An information system

input space. The methods can also be applied for monotone attribute reduction where the reduced set of attributes also satisfies the monotonicity constraint. Most of the research has been previously published in [12].

The chapter is organized as follows. Section 2.2 gives a brief introduction to the classical Rough Sets theory. The proposed extension for monotone classification is presented in section 2.3. Further some experiments were performed on the problem of bankruptcy prediction as a monotone classification problem. The results are discussed in section 2.4. The conclusions of the chapter are given in section 2.5.

2.2 Basic Notions in Rough Sets Theory

2.2.1 Decision Tables and Indiscernibility

Traditionally all basic notions in the theory of Rough Sets are defined using sets and equivalence relations. For our purpose, however, it is enough to start directly with a decision table. For a good introduction to the classical Rough Sets Theory the reader is referred to [60].

An *information system* S is a tuple $S = \langle U, A, V \rangle$ where:

$U = \{x_1, x_2, \ldots, x_n\}$ denotes a non-empty, finite set of objects (observations, examples),

$A = \{a_1, a_2, \ldots, a_m\}$ denotes a non-empty, finite set of attributes, and

$V = \{V_1, V_2, \ldots, V_m\}$ is the set of domains of the attributes in A.

An example of an information system is given with table 2.1 where:

$$U = \{Anna, Bill, Cathy, Dave, Emma, Frank\},$$

$$A = \{age, experience, grades\},$$

$$V = \{\{21\text{–}30, 31\text{–}40\}, \{none, 1\text{–}3, 4\text{–}6\}, \{average, good, excellent\}\}.$$

A *decision table* is a special case of an information system where among

	age	experience	grades	hired
Anna	21-30	none	good	yes
Bill	21-30	none	good	no
Cathy	21-30	4-6	average	no
Dave	31-40	1-3	excellent	yes
Emma	31-40	4-6	good	yes
Frank	31-40	4-6	good	yes

Table 2.2: A decision table

the attributes in A we distinguish one or more called a *decision attribute(s)*[1]. The other attributes are called *condition attributes*. In the case of a decision table we use the notation: $A = C \cup \{d\}$, $C = \{a_1, a_2, \ldots, a_m\}$ where a_i – condition attributes, d – decision attribute. As an example, the information system of table 2.1 is extended to a decision table in table 2.2 by adding a decision attribute $d = \mathit{hired}$ with $V_d = \{\mathit{yes}, \mathit{no}\}$.

A closer look at table 2.2 will show that *Anna* and *Bill* have identical values for all condition attributes and the same can be seen about *Emma* and *Frank*. One of the building notions of Rough Sets Theory is the *indiscernibility relation* between objects. It is defined relative to a set of attributes and determines whether, given the set of attributes, two objects are the same (indiscernible) or different. More precisely, the indiscernibility relation generated by a set $B \subseteq A$ in an information system S is defined as:

$$IND_S(B) = \{(x, y) | x, y \in U, \forall a \in B \; a(x) = a(y)\}. \tag{2.1}$$

The indiscernibility relation is an equivalence relation[2] and it partitions the set of objects in equivalence classes of objects that are indiscernible with respect to the given subset of attributes. In the example, for $B = C$ the partition is:

$$IND_S(C) = \{\{\mathit{Anna}, \mathit{Bill}\}, \{\mathit{Cathy}\}, \{\mathit{Dave}\}, \{\mathit{Emma}, \mathit{Frank}\}\}$$

and for $B = \{\mathit{grades}\}$:

$$IND_S(B) = \{\{\mathit{Anna}, \mathit{Bill}, \mathit{Emma}, \mathit{Frank}\}, \{\mathit{Cathy}\}, \{\mathit{Dave}\}\}.$$

The equivalence class of an object x generated by a set of attributes B is denoted by $[x]_B$.

It is clear that for $B_1 \subseteq B_2 \subseteq A$ always $IND_S(B_2) \subseteq IND_S(B_1)$ or, intuitively, the more attributes we add the finer the partition becomes. Therefore

[1]In the following we only consider one decision attribute as this is usually the case in classification, however, all the definitions can easily be generalized for more than one decision attribute.

[2]An equivalence relation R is a binary relation that is reflexive (xRx), symmetric ($xRy \Rightarrow yRx$) and transitive ($xRy, yRz \Rightarrow xRz$).

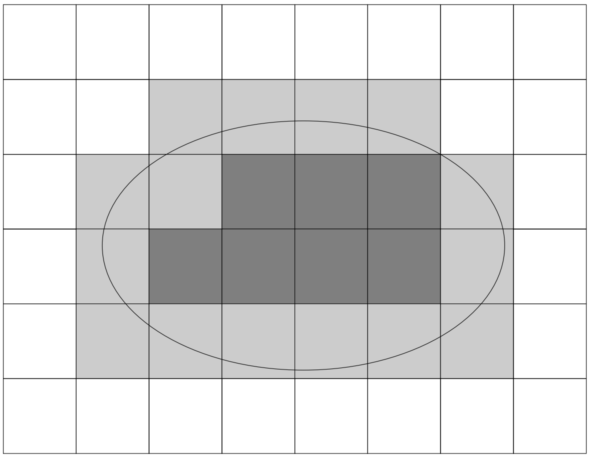

Figure 2.1: The set approximations

by removing attributes from the table in general we reduce the available information for discerning the objects and the important question is which attributes are dispensable and which should not be removed. In Rough Sets Theory those attributes are detected using the notions of lower/upper approximation and positive region.

2.2.2 Set Approximations

Given sets $X \subset U$ and $B \subset A$, the *lower approximation* of X with respect to B is defined as:

$$\underline{B}X = \{x \in U | [x]_B \subset X\} \tag{2.2}$$

or the collection of objects whose equivalence classes are subsets of X.

The *upper approximation* of X with respect to B is defined as:

$$\overline{B}X = \{x \in U | [x]_B \cap X \neq \emptyset\} \tag{2.3}$$

or all objects whose equivalence classes have a nonempty intersection with X.

A *rough set* X is defined as the pair $(\underline{X}, \overline{X})$. X is called *crisp* (or *definable*) if $\underline{X} = \overline{X}$.

Figure 2.1 shows an example of the set approximations where the grid is the partitioning generated by the indiscernibility relation, the oval shape is the set to be approximated, the dark grey area is the lower approximation and the light grey area is the upper approximation.

In the case of a decision table, one is interested in the lower and upper approximation of the equivalence classes of the decision attribute and by varying the set B it is possible to analyze the attributes on whether they are dispensable or not. In the example of table 2.2 the decision attribute generates two equivalence classes: $X_{yes} = \{Anna, Dave, Emma, Frank\}$ and $X_{no} = \{Bill, Cathy\}$. Then the lower approximation of X_{yes} with respect to C is:

$$\underline{C}X_{yes} = \{Dave, Emma, Frank\}.$$

The object *Anna* is not included because its equivalence class is not a subset of X_{yes}. *Anna* is however included in the upper approximation since its equivalence class has a non empty intersection with X_{yes}:

$$\overline{C}X_{yes} = \{Anna, Bill, Dave, Emma, Frank\}$$

Similarly:

$$\underline{C}X_{no} = \{Cathy\},$$

$$\overline{C}X_{no} = \{Anna, Bill, Cathy\}.$$

The *positive region* of $D \subset A$ with respect to $B \subset A$ is defined as:

$$POS_B(D) = \cup\{\underline{B}X : X \in IND(D)\} \tag{2.4}$$

or the collection of the B-lower approximations corresponding to all the equivalence classes of D.

The set D in this particular case will be equal to $\{d\}$ where d is the decision attribute. In the example:

$$POS_C(d) = \{Cathy, Dave, Emma, Frank\} = \underline{C}X_{no} \cup \underline{C}X_{yes}.$$

2.2.3 Reducts

An attribute $a \in B$ is called *dispensable* (or *superfluous*) in $B \subset A$ if $POS_B(d) = POS_{B/\{a\}}(d)$. Otherwise it is called *indispensable*.

In the example the attribute *grades* is dispensable in C because for the attribute set $B = \{age, experience\}$:

$$POS_B(d) = \underline{B}X_{no} \cup \underline{B}X_{yes} = \{Cathy\} \cup \{Dave, Emma, Frank\} =$$

$$= \{Cathy, Dave, Emma, Frank\} = POS_C(d).$$

The set of all indispensable attributes in C with respect to d is called the *core*. Note that the core can be empty if no attributes are indispensable in C. B is called *independent* with respect to d if every attribute in B is indispensable with respect to d.

A *reduct* of C with respect to d is defined as every subset $B \subset C$ such that B is independent with respect to d and $POS_C(d) = POS_B(d)$. A reduct is actually such a subset of conditional attributes that discerns the objects in

	A	B	C	D	E	F
A	∅	∅	{e,g}	∅	∅	∅
B	∅	∅	∅	{a,e,g}	{a,e}	{a,e}
C	{e,g}	∅	∅	{a,e,g}	{a,g}	{a,g}
D	∅	{a,e,g}	{a,e,g}	∅	∅	∅
E	∅	{a,e}	{a,g}	∅	∅	∅
F	∅	{a,e}	{a,g}	∅	∅	∅

Table 2.3: The discernibility matrix

the same way as the full set of conditional attributes while being minimal with respect to inclusion. Therefore it allows all attributes not in the reduct to be ignored and thus to achieve reduction in the number of attributes (or attribute reduction).

One decision table can have more that one (and sometimes very many) reducts. Obviously in some cases the full set of attributes can also be a reduct. Since the attributes in the core are all indispensable for C, then they should appear in all reducts and the core is actually the intersection of all reducts.

The notion of a reduct is fundamental for Rough Sets Theory and a number of equivalent definitions are available in the literature.[3] One that provides an important, for our purpose, point of view on reducts uses the notions of discernibility matrix and discernibility function.

The *discernibility matrix* shows how each two objects differ from each other given the condition attributes. It is formally defined as follows:

$$(c_{ij}) = \begin{cases} \{a \in A : a(x_i) \neq a(x_j)\} & \text{for } i,j : d(x_i) \neq d(x_j) \\ \emptyset & \text{otherwise .} \end{cases} \tag{2.5}$$

Each entry of the matrix contains the (names of the) attributes for which the two objects differ when they are from different classes. The matrix is obviously symmetrical with empty main diagonal. It is therefore enough to only consider the entries on the lower side of the main diagonal.

For the example decision table (table 2.2), the discernibility matrix is shown on table 2.3 where for compactness the objects are denoted by {A,B,C,D,E,F} and the condition attributes by {a,e,g}.

From the discernibility matrix we can construct a Boolean function called a *discernibility function* f of m Boolean variables $\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_m$ where $\hat{a}_i$ corresponds to the attribute a_i for $1 \leq i \leq m$. It is defined as follows:

$$f(\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_m) = \bigvee \left\{ \bigwedge \hat{c}_{ij} | 1 \leq j \leq i \leq n, c_{ij} \neq \emptyset \right\} \tag{2.6}$$

[3]The notion of a reduct as well as many of the other basic notions can also be defined for an information system without a decision attribute. Here, however, we only consider decision tables and those definitions are not taken into account.

where $\hat{c}_{ij} = \{\hat{a}_k | a_k \in c_{ij}\}$.

It is known that the set of all prime implicants[4] of f corresponds to the set of all reducts of the table. They can be computed by dualizing the discernibility function. Here we briefly introduce the notion of dualization (see also [14, 11, 49]).

For a Boolean variable x the complement of x is defined as $\overline{x} = 1 - x$. The complement of a Boolean function is defined as $\overline{f}(x) = \overline{f(x)}$. The dual of a function can then be defined as $f^d(x) = \overline{f}(\overline{x})$. It can be computed by interchanging the conjunction and disjunction signs and simplifying the resulting expression.

Therefore the dual of the discernibility function can be represented as in the following formula:

$$f^d(\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_m) = \bigwedge \left\{ \bigvee \hat{c}_{ij} | 1 \leq j \leq i \leq n, c_{ij} \neq \emptyset \right\} \tag{2.7}$$

When the expression is simplified to disjunctive normal form, each conjunction in it corresponds to a reduct.

Equivalently, the reducts are the minimal transversals[5] of the set of entries of the discernibility matrix. In other words, they are the minimal subsets of condition attributes that have a non-empty intersection with each non-empty entry of the matrix.

In the example, the dual of the discernibility function generated from the matrix of table 2.3 is given below:

$$f^d = (e \vee g)(a \vee e \vee g)(a \vee e \vee g)(a \vee e)(a \vee g)(a \vee e)(a \vee g) \tag{2.8}$$

The function from equation 2.8 can be simplified to:

$$f^d = (a \vee e)(a \vee g)(e \vee g) = ae \vee ag \vee eg$$

from which it can be seen that the prime implicants are {a,e},{a,g} and {e,g}. It is easy to verify from the table that these are indeed the minimal transversals of the entries as well. The core in this case is empty.

2.2.4 Complexity and Heuristics

Generating a reduct of minimum length is an NP-hard problem. Therefore, in practice a number of heuristics are preferred for the generation of only one

[4] An implicant p of a function $f(X)$ is such an elementary conjunction for which $p(X) = 1 \Rightarrow f(X) = 1$ for all Boolean vectors X. p is a prime implicant if no other implicant can be obtained by deleting literals from p.

[5] A transversal of a collection of sets is such a set that has a non-empty intersection with each set from the collection. A minimal transversal is such a transversal that does not properly contain another transversal.

"good" reduct. Two of these heuristics are the "Best Reduct" method [47] and Johnson's algorithm [48].

The complexity of a total time algorithm for the problem of generating all minimal reducts (or dualizing the discernibility function) has been intensively studied in Boolean function theory, see [14, 37, 11]. Unfortunately, this problem is still unsolved, but a quasi-polynomial algorithm is known [39]. However, these results have not yet been mentioned in the rough set literature, see e.g. [49].

As it was mentioned above, two of the more successful heuristics for generating one reduct are the Johnson's algorithm and the "Best reduct" heuristic. Strictly speaking these methods do not necessarily generate reducts, since the minimality requirement is not guaranteed. Therefore, for this topic we shall make the distinction between (non-minimal) reducts and *minimal* reducts. One possible approach used to solve the problem is to generate the reduct and then check whether any of the subsets are also reducts.

The *Johnson's heuristic* uses a very simple procedure that tends to generate a reduct with minimal length (the minimality is not guaranteed, however). Given the discernibility matrix, for each attribute the number of entries where it appears is counted. The one with the highest number of entries is added to the future reduct. Then all the entries containing that attribute are removed and the procedure repeats until all the entries are covered.

It is logical to start the procedure by simplifying the set of entries: removing the entries that contain strictly or non strictly other elements. In some cases the results with and without simplification might be different.

The *"Best reduct"* heuristic is based on the significance of attributes measure. It is defined based on the measure of *dependency* between a set $R \subset C$ and the decision attribute d:

$$k(R,d) = card(POS_R(d))/card(POS_C(d)).$$

If R is a reduct of C then $k(R,d) = 1$. Further the *significance* of an attribute a added to R is defined as follows:

$$SGF(a,R,d) = k(R + \{a\}, d) - k(R,d).$$

The procedure starts with the core and on each step adds the attribute with the highest significance, if added to the set, until the value reaches one.

In many of the practical cases the two heuristics give the same result, however, they are not the same and a counter example can be given. The data set discussed in our experiments in section 2.4, for example, gives different results when the two heuristics are applied.

2.2.5 Object Reducts

The notions discussed so far give us methods for attribute reduction in a consistent way. Rough Sets theory goes further by generating rules from the reduced table. For that the notions of object-relative discernibility function and object-reduct are introduced.

The *object-relative discernibility function* (or *j-relative discernibility function*) is defined over the j^{th} column of of the discernibility matrix:

$$f_j^d(\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_m) = \bigwedge \left\{ \bigvee \hat{c}_{ij} | j \leq i \leq n, c_{ij} \neq \emptyset \right\} \tag{2.9}$$

It describes how one particular object (denoted here by j) can be discerned from all other objects of different classes.

The prime implicants of the j-relative discernibility function are called *object-reducts* (or *value-reducts*) and they represent the minimal necessary information to discern the object.

From the discernibility matrix of table 2.3, the discernibility function of object E is as follows:

$$f_E^d = (a \vee e)(a \vee g)$$

The prime implicants (and object reducts) are {a} and {e,g}. Similarly for object D the discernibility function is:

$$f_D^d = (a \vee e \vee g)(a \vee e \vee g)$$

and the prime implicants are {a}, {e} and {g}.

2.2.6 Classification Rules

From the object reducts one can generate classification rules in the following way. For each attribute in the reduct the corresponding values are retrieved from the related object. They constitute the left-hand side of the rule. The right-hand side contains the class value (the value of the decision attribute) of the object.

For example the rules corresponding to the object reducts of object E are:

$$a = 31\text{–}40 \Rightarrow h = yes$$

$$e = 4\text{–}6, g = good \Rightarrow h = yes$$

In order to generate a classifier, rules are extracted for each object and each corresponding value-reduct. This usually results in a large number of rules although some of these are repetitious, consequently, pruning the set can avoid this. The classifier is usually applied on an unseen object as follows:

1. The new object is compared to each rule in order to extract the set of rules that fire.

2. If no rule fires a special default rule is applied. This, for example, might assign the majority class from the table to the new object.

3. If at least one rule fires and no conflicting predictions occur the consensus prediction is assigned to the object.

4. If rules with conflicting predictions are detected, a form of voting is performed in order to decide which class to choose. Ranking is usually applied to the rules based on (among other things) the support of the rule in the table.

The Rough Sets methods have been applied in a large number of domain areas including medicine, economics, finance, environmental studies, engineering, information science, social sciences, molecular biology and chemistry. For a list of case-studies and tools the reader is referred to [49].

2.2.7 Extensions of Rough Sets Theory

Many authors argue that the classical Rough Sets theory is too restrictive and the research in that direction results in a number of attempts to extend the theory or relax the constraints in different ways.

The *variable precision rough set model* [80] is a step in that direction. The method is a generalization of the theory of Rough Sets aimed at handling uncertain information by allowing controlled degree of misclassification.

Another approach in that direction is proposed in [57] by relaxing the indiscernibility relation from an equivalence to a tolerance relation.

Other extensions of the methodology provide various ways of refining the results by generating "better" reducts and decision rules. Rough Sets theory gives the possibility of computing all reducts of a set of attributes. This can be both an advantage and a disadvantage. The full set of reducts can provide a better overall picture of the properties of the data and the possibility to choose which reduct to use. However the number of reducts can be overwhelmingly high and the classical theory does not provide assistance in the choice of a reduct. While the reduct of a minimal length might sometimes be preferred, one can argue that it does not need to be the best solution and its calculation is an NP-hard problem.

The idea of *dynamic reducts* was introduced in [7] and strives at generating more "stable" reducts. The underlying idea is that the classical reducts sometimes reflect peculiarities in the data and are sensitive to small changes in the decision table. Therefore, random samples from the data are used and the most frequently appearing reducts (the dynamic reducts) are expected to be more robust and reflect the real structure of the data.

Similar arguments apply to the generation of decision rules. In [56] the notion of *default rules* is introduced as rules that have exceptions and an algorithm is proposed to generate such rules by systematically removing attributes from the data. Such rules are believed to be more general and better for classifying unseen objects.

The extensions mentioned in this section are just some of the existing ones. The large body of research in the area produced many more interesting methods for "customizing" the classical theory for solving different problems.

2.3 Rough Sets on Monotone Problems

In this section we focus on a specific type of problems, namely, the monotone problems. We present the contribution of this chapter in extending Rough Sets theory towards dealing with such problems. In subsection 2.3.3 we also discuss related research and how the approaches differ from our approach.

2.3.1 Monotone Information Systems

Let us consider a decision table $S = (U, C \cup \{d\}, V)$ where for each attribute $a_i \in C \cup \{d\}$ the corresponding set of values V_i is ordered. This induces a partial order over the universe U: for $x, y \in U$, $x \leq y$ if $a_k(x) \leq a_k(y)$ $\forall a_k \in C$. In the rest of this section we only consider decision tables with ordered attribute values.

We call the decision table $S = (U, C \cup \{d\}, V)$ *monotone* when for each couple $x_i, x_j \in U$ the following holds:

$$a_k(x_i) \geq a_k(x_j), \forall a_k \in C \Rightarrow d(x_i) \geq d(x_j) \tag{2.10}$$

where $a_k(x_i)$ is the value of the attribute a_k for the object x_i. The following example will serve as a running example for this section.

Example 1

The following decision table represents a monotone data set (table 2.4):

U	a	b	c	d
x_1	0	1	0	0
x_2	1	0	0	1
x_3	0	2	1	2
x_4	1	1	2	2
x_5	2	2	1	2

Table 2.4: Monotone decision table

Positive Area

We shall now look at the problem from a more general point of view. Let R, S be arbitrary relations on a universe U. Then we define $R \subseteq S \Leftrightarrow (xRy \Rightarrow xSy, \forall x, y \in U)$.

Given a data set, any subset of attributes $A \subseteq C$ induces a relation R_A on U. In the general case, this was an equivalence relation. Here however we consider only attributes with full order over the attribute values. We can then define the relation R_A as follows: for $x, y \in U$, xR_Ay if $\forall a_i \in A \; a_i(x) \leq a_i(y)$. Both in the general and the ordinal case, a data set is called *consistent* if $R_C \subseteq R_{\{d\}}$ where d is the decision attribute.

Furthermore, for each $x \in U$ we define an *upset* and a *downset* with respect to R as follows:

$$x \uparrow R = \{y : xRy\}$$

$$x \downarrow R = \{y : yRx\}.$$

In the next paragraphs we try to redefine the notions of lower and upper approximations for the ordinal case. This would eventually lead us to a new definition for the positive area in the monotone case. We are guided by the following simple considerations.

The intuition behind the definitions is that these should behave in a similar way to those for the general case while taking into account the monotonicity property. In the general case, the lower approximation of a set consists of those points that can be unambiguously assigned to that set. We expect the same from the monotone lower approximation, only that here "unambiguously" takes a new meaning. Intuitively those should be the points that belong to this set which do not participate in inconsistent pairs, therefore no point that dominates them belongs to a lower class and no point that is dominated by them belongs to a higher class.

More formally we can define the upward lower and upper approximation of a set $X \subseteq U$ with respect to the relation R and a relation R_d:

$$\underline{R}\uparrow(X) = \{x \in X : x \uparrow R \subseteq x \uparrow R_d\}$$

$$\overline{R}\uparrow(X) = \{x \in X : x \uparrow R \cap x \uparrow R_d \neq \emptyset\}.$$

Similarly the downward lower and upper approximations can be given with the downset of x:

$$\underline{R}\downarrow(X) = \{x \in X : x \downarrow R \subseteq x \downarrow R_d\}$$

$$\overline{R}\downarrow(X) = \{x \in X : x \downarrow R \cap x \downarrow R_d \neq \emptyset\}.$$

The intersection of the upward and the downward approximations defines the monotone lower and upper approximations:

$$\underline{R}\updownarrow(X) = \underline{R}\uparrow(X) \cap \underline{R}\downarrow(X)$$

$$\overline{R}\updownarrow(X) = \overline{R}\uparrow(X) \cap \overline{R}\downarrow(X).$$

For compactness, in the following $\underline{R}\updownarrow_B(X)$ will be denoted by $\underline{B}\updownarrow(X)$.

A step further is to define the monotone positive area for a set $B \subseteq C$ as:

$$POS_B(d) = \cup\{\underline{B}\updownarrow(X) : X \in [U]_d\}$$

where

$$[U]_d = \{X \subseteq U : \forall x, y \in X, d(x) = d(y)\}.$$

Following the same intuition the positive area contains all points that do not participate in inconsistent pairs. Obviously the data set consisting of only the positive area is monotone.

Now we are ready to redefine the notion of a reduct: $B \subseteq C$ is a monotone reduct if $POS_B(d) = POS_C(d)$ and B is minimal with respect to inclusion.

Before giving a simple example to demonstrate the redefined notions of positive area and reduct, we prove two lemmas to clarify the chosen definitions.

Lemma 1 *Let R_B be the relation on U induced by a set of attributes $B \subseteq C$. Then:*

$$POS_B(d) = \{x \in U : y \in x\uparrow R_B \Rightarrow d(y) \geq d(x) \text{ and } y \in x\downarrow R_B \Rightarrow d(y) \leq d(x)\}.$$

Proof:

$$POS_B(d) = \cup\{\underline{B}\updownarrow(X) : X \in [U]_d\} =$$
$$= \cup\{\underline{B}\uparrow(X) \cap \underline{B}\downarrow(X) : X \in [U]_d\} =$$
$$= \{x \in X : x\uparrow R_B \subseteq x\uparrow R_d, x\downarrow R_B \subseteq x\downarrow R_d, X \in [U]_d\} =$$
$$= \{x \in U : x\uparrow R_B \subseteq x\uparrow R_d, x\downarrow R_B \subseteq x\downarrow R_d\} =$$
$$= \{x \in U : y \in x\uparrow R_B \Rightarrow d(y) \geq d(x) \text{ and } y \in x\downarrow R_B \Rightarrow d(y) \leq d(x)\}.$$

□

Lemma 2 *If the data set is monotone the following statements are equivalent:*

1. *B is a reduct;*
2. *$R_B \subseteq R_d$.*

Proof: It is easy to see that if the data set is monotone then $POS_C(d) = U$. This follows directly from the previous lemma since the data set will contain no inconsistencies that would be excluded from the positive area. Therefore for a reduct $B \subseteq C$ the same holds: $POS_B(d) = U$. Which means that:

$$\forall x \in U : x\uparrow R_B \subseteq x\uparrow R_d, x\downarrow R_B \subseteq x\downarrow R_d \Leftrightarrow$$

U	a	b	c	d
x_1	1	1	1	1
x_2	1	0	0	1
x_3	1	1	0	0
x_4	0	1	1	0

Table 2.5: A small example of a monotone data set with one inconsistent pair of data points

$$\Leftrightarrow \forall x \in U : y \in x{\uparrow}R_B \Rightarrow x{\uparrow}R_d, y \in x{\downarrow}R_B \Rightarrow x{\downarrow}R_d \Leftrightarrow$$

$$\Leftrightarrow \forall x \in U : xR_By \Rightarrow xR_dy, yR_Bx \Rightarrow yR_dx \Leftrightarrow$$

$$\Leftrightarrow R_B \subseteq R_d.$$

□

Let us now consider the following small example given in table 2.5 which contains four objects described by three binary condition attributes and one binary decision attribute.

It is easy to see that objects x_2 and x_3 are conflicting. The positive area here will be $POS_C^U = \{x_1, x_4\}$ which can be calculated using lemma 1. For example $x_2 \notin POS_C^U$ because $x_3 \in x{\uparrow}R_C$ but $d(x_3) < d(x_2)$. Similarly $x_3 \notin POS_C^U$.

We have 6 proper subsets of C which are the candidates for reducts. Let us check for the set $\{a, c\}$. Here again x_2 and x_3 are conflicting and are not included in the positive area which is $POS_{\{a,c\}}^U = \{x_1, x_4\}$.

If we check, however, for the set containing only $\{a\}$ we discover that it is not a reduct. Here again x_2 and x_3 are not in the positive area but also x_1 is excluded because it is in conflict with x_3. Therefore $POS_{\{a\}}^U = \{x_4\} \neq POS_C^U$ which means that $\{a\}$ is not a reduct of C.

An interesting question is what happens if we simply delete the inconsistent pairs of examples from the data and consider the remaining monotone decision table. Do we get the same reducts as from the full data? The answer in general is no. For our example we demonstrate this by deleting x_2 and x_3 so that the resulting set of objects is $W = \{x_1, x_4\}$. Since this set is monotone then the positive area is $POS_C^W = W = \{x_1, x_4\}$. We consider again the candidate reducts $\{a, c\}$ and $\{a\}$. It turns out that as before $\{a, c\}$ is a reduct, however, now also $\{a\}$ is a reduct which was not the case for the full data table.

The intuitive explanation of the difference is that by deleting data points we lose some information that might have otherwise influenced the result. Deleting data is hardly ever a good strategy (even when it is corrupted) which is the motivation for the large amount of research in areas such as handling of noise and missing values in classification data.

It can also be argued that in order to reduce the lost information we can delete only one of the conflicting pair of data points. In this however we are confronted with a different problem – it is not straightforward to choose which of the two points to delete as they can be conflicting with a set of other points. This might result in a complicated graph structure consisting of the points candidates for deleting linked by the inconsistencies between them. Finding the minimal number of points to delete that will make the data monotone is equivalent to finding a minimal cut for the graph.

Monotone Discernibility Matrix

Here we take the other point of view on reducts and try to redefine the discernibility matrix in the monotone case.

Let $S = (U, C \cup \{d\}, V)$ be a decision table. In the classical rough sets theory, the discernibility matrix (DM) is defined as follows:

$$(c_{ij}) = \begin{cases} \{a \in A : a(x_i) \neq a(x_j)\} & \text{for } i,j : d(x_i) \neq d(x_j) \\ \emptyset & \text{otherwise.} \end{cases}$$

The variation of the discernibility matrix proposed here is the *monotone discernibility matrix* $M_d(S)$ defined as follows:

$$(c_{ij}) = \begin{cases} \{a \in A : a(x_i) > a(x_j)\} & \text{for } i,j : d(x_i) > d(x_j) \\ \emptyset & \text{otherwise.} \end{cases} \tag{2.11}$$

Unlike the general discernibility matrix, the monotone one is not symmetric. We assume that the objects are ordered in increasing value of their decision attribute. In this case all entries of the matrix that are above the main diagonal will be empty. A dual definition can be given for the monotone discernibility matrix, $M'_d(S)$, which will be used later for generating dual format rules:

$$(c'_{ij}) = \begin{cases} \{a \in A : a(x_i) < a(x_j)\} & \text{for } i,j : d(x_i) < d(x_j) \\ \emptyset & \text{otherwise.} \end{cases} \tag{2.12}$$

In this case the entries below the main diagonal will be empty.

In the general case the discernibility matrix is closely connected to the positive area. The points which do not belong to the positive area are exactly those which have an empty entry of the discernibility matrix in their row/column corresponding to an object of a different class. We would like to prove that a similar relationship holds for the monotone case.

In the following, by $c_{xy}|_B$ we denote the entry of the monotone discernibility matrix corresponding to the points x, y only restricted to the attributes in B.

Lemma 3 *For $B \subseteq C$ it holds that: $x \notin POS_B(d)$ if and only if at least one of the following conditions is true:*

	x_1	x_2	x_3	x_4
x_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_3	$\{c\}$	$\emptyset$	$\emptyset$	$\emptyset$
x_4	$\{a\}$	$\{a\}$	$\emptyset$	$\emptyset$

Table 2.6: The monotone discernibility matrix for the data set given in table 2.5

1. $\exists y : d(x) > d(y), c_{xy}|_B = \emptyset$

2. $\exists y : d(y) > d(x), c_{yx}|_B = \emptyset$.

Proof:

$$x \notin POS_B(d) \Leftrightarrow x\!\uparrow\! R_B \not\subseteq x\!\uparrow\! R_d \text{ or } x\!\downarrow\! R_B \not\subseteq x\!\downarrow\! R_d \Leftrightarrow$$
$$\Leftrightarrow (\exists y \in x\!\uparrow\! R_B, y \notin x\!\uparrow\! R_d) \text{ or } (\exists y \in x\!\downarrow\! R_B, y \notin x\!\downarrow\! R_d) \Leftrightarrow$$
$$\Leftrightarrow (\exists y \in x\!\uparrow\! R_B, d(y) < d(x)) \text{ or } (\exists y \in x\!\downarrow\! R_B, d(y) > d(x)) \Leftrightarrow$$
$$\Leftrightarrow (\exists y : c_{yx}|_B = \emptyset, d(y) < d(x)) \text{ or } (\exists y : c_{xy}|_B = \emptyset, d(y) > d(x)).$$

□

Let us go back to the small example in table 2.5. The (very simple) monotone discernibility matrix is given in table 2.6. The entry of the matrix that satisfies the conditions of the lemma is c_{23} (considering the full set of attributes C). It corresponds to x_2 and x_3 which, as we showed in the previous subsection, are exactly those two points that do not belong to the positive area for C.

Monotone Discernibility Function

Based on the monotone discernibility matrix, the *monotone discernibility function* can be constructed following the same procedure as in the classical Rough Sets approach. For each non-empty entry of the monotone discernibility matrix M_d, $c_{ij} = \{a_{k_1}, a_{k_2}, \ldots, a_{k_l}\}$, we construct the conjunction $C = a_{k_1} \wedge a_{k_2} \wedge \ldots \wedge a_{k_l}$. The disjunction of all these conjunctions is the monotone discernibility function:

$$f = C_1 \vee C_2 \vee \ldots \vee C_p. \tag{2.13}$$

The *monotone reducts* of the decision table are the minimal transversals of the entries of the monotone discernibility matrix. In other words the monotone reducts are the minimal subsets of condition attributes that have a non-empty intersection with each non-empty entry of the monotone discernibility matrix. That follows directly from lemma 3. In section 2.3.2 we give another equivalent definition for a monotone reduct described from a different point of view.

Looking back to the example which we used for the positive area (table 2.5) we can easily see that the only reduct is $\{a, c\}$ (the only minimal combination

	1	2	3	4	5
1	$\emptyset$	$\{a,b\}$	$\{b,c\}$	$\{a,c\}$	$\{a,b,c\}$
2	$\{a,b\}$	$\emptyset$	$\{a,b,c\}$	$\{b,c\}$	$\{a,b,c\}$
3	$\{b,c\}$	$\{a,b,c\}$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\{a,c\}$	$\{b,c\}$	$\emptyset$	$\emptyset$	$\emptyset$
5	$\{a,b,c\}$	$\{a,b,c\}$	$\emptyset$	$\emptyset$	$\emptyset$

Table 2.7: General discernibility matrix

of attributes covering all non-empty entries). That is indeed the same result as the one we got using the positive area. However this data set is too small to demonstrate the idea and we therefore prefer to use a better one in the following example.

Example 2

Consider the decision table from example 1 (table 2.4). The general and *monotone* discernibility matrix for this table are shown in table 2.7 and table 2.8.

	1	2	3	4	5
1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{a\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3	$\{b,c\}$	$\{b,c\}$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\{a,c\}$	$\{b,c\}$	$\emptyset$	$\emptyset$	$\emptyset$
5	$\{a,b,c\}$	$\{a,b,c\}$	$\emptyset$	$\emptyset$	$\emptyset$

Table 2.8: Monotone discernibility matrix $M_d(S)$

The general discernibility function is:

$$f(a,b,c) = ab \vee ac \vee bc.$$

Therefore, the general reducts of table 2.4 are respectively: $\{a,b\}$, $\{a,c\}$ and $\{b,c\}$ and the core is empty. However, the *monotone* discernibility function looks different:

$$f_{mon}(a,b,c) = a \vee bc.$$

So the *monotone* reducts are: $\{a,b\}$ and$\{a,c\}$, and the *monotone* core is $\{a\}$. It can easily be shown that these monotone reducts preserve the monotonicity property of the data set.

Noise with Respect to Monotonicity

In the beginning of this section we defined a monotone data set. In practice, however, we are not guaranteed that the available data sets will be strictly monotone even when our domain knowledge tells us that they should be. Due to noise, monotone inconsistencies might be present, i.e., pairs of data points $x, y \in U$ such that $x \leq y$ and $d(x) < d(y)$.

One of the attractive features of the general rough sets theory is that it can deal easily with inconsistencies of the type $x = y$, $d(x) \neq d(y)$. In this subsection we would like to emphasize that the monotone extension of the theory proposed here behaves in a similar way towards the monotone inconsistencies.

The definition of a monotone reduct states that it should preserve the positive area of the original set of attributes. For strictly monotone data sets this means that a reduct should preserve the monotonicity property so that the reduced data set remains monotone. However, when the original data set contains monotone inconsistencies they are excluded from the positive area. In this case the reduct is required to introduce no additional inconsistencies and therefore the number of conflicting pairs of points should remain the same in the reduced data set.

When using the monotone discernibility matrix, the conflicting pairs will result in empty entries in the matrix. These are obviously not taken into account in the discernibility function and thus cannot disrupt the generation of reducts. In the data set used for the experiments for this chapter (see section 2.4) contains one inconsistent pair of points. One of these points was removed in order to make the set monotone. That, however, was done only for clarity and in general is not necessary.

The next subsection discusses how rules can be generated from the monotone discernibility matrix. It is easy to see that monotone inconsistencies cannot disrupt that process either, for the same reason – the conflicts result in empty entries in the discernibility matrix which are simply not taken into account in the rules generation.

Rule Generation

The next step in the classical Rough Set approach [60, 49] is, for the chosen reduct, to generate the *value (object) reducts* using a similar procedure as for computing the reducts. A contraction of the discernibility matrix is generated based only on the attributes in the reduct. Further, for each row of the matrix, the object discernibility function is constructed – the discernibility function relative to this particular object. The object reducts are then the minimal transversals of the object discernibility functions.

Using the same procedure but on the monotone discernibility matrix, we can generate the monotone object reducts. Based on them, the classification rules

class $d \geq 2$	class $d \geq 1$
$a \geq 2$	$a \geq 1$
$b \geq 2$	
$a \geq 1 \wedge b \geq 1$	
$c \geq 1$	

Table 2.9: Monotone decision rules

are constructed. For the monotone case we use the following format:

$$\text{if } (a_{i_1} \geq v_1) \wedge (a_{i_2} \geq v_2) \wedge \ldots \wedge (a_{i_l} \geq v_l) \text{ then } d \geq v_{l+1} \ . \tag{2.14}$$

Each such rule should cover at least one example of class higher than the minimal one and no examples of the minimal class value.

It is also possible to construct the classification rules using the dual format:

$$\text{if } (a_{i_1} \leq v_1) \wedge (a_{i_2} \leq v_2) \wedge \ldots \wedge (a_{i_l} \leq v_l) \text{ then } d \leq v_{l+1} \ . \tag{2.15}$$

This type of rules can be obtained by the same procedure by only considering the dual format of the monotone discernibility matrix instead. As a result we get rules that cover at least one example of class smaller than the maximal class value and no examples of the maximal class.

It can be proven that in the monotone case it is not necessary to generate the value reducts for all the objects – the value reducts of the minimal vectors of each class will also cover the other objects from the same class. For the rules with the dual format we consider respectively the maximal vectors of each class. Tables 2.9 and 2.10 show the complete set of rules generated for the whole table.

A set of rules is called a *cover* if all the examples with class $d \geq 1$ are covered, and no example of class 0 is covered. (Assuming, of course, that 0 is the minimal class value which we can ensure by changing the coding of the decision attribute without affecting the order of the values.) A similar definition can be given for the set of dual-format rules.

The minimal covers (computed by solving a set-covering problem) for the full table are shown in tables 2.11 and 2.12. In this case the minimal covers correspond to the unique minimal covers of the reduced tables associated with respectively the monotone reducts {a,b} and {a,c}.

The set of rules with dual format is not an addition but rather an alternative to the set rules of the other format. If used together they may be conflicting in some cases.

It is known that the decision rules induced by object reducts in general do not cover the whole input space. Furthermore, the class assigned by these decision rules to an input vector is not uniquely determined. We therefore briefly discuss the concept of an *extension* of a discrete data set or a decision table in the following section.

class $d \le 0$	class $d \le 1$
$a \le 0 \wedge b \le 1$	$b \le 0$
$a \le 0 \wedge c \le 0$	$c \le 0$

Table 2.10: The dual format rules

class $d \ge 2$	class $d \ge 1$
$a \ge 1 \wedge b \ge 1$	$a \ge 1$
$b \ge 2$	

Table 2.11: The minimal cover for $\{a, b\}$

class $d \ge 2$	class $d \ge 1$
$c \ge 1$	$a \ge 1$

Table 2.12: The minimal cover for $\{a, c\}$

class $d \le 0$	class $d \le 1$
$a \le 0 \wedge b \le 1$	$b \le 0$

Table 2.13: The minimal cover for $\{a, b\}$ (dual format)

class $d \le 0$	class $d \le 1$
$a \le 0 \wedge c \le 0$	$c \le 0$

Table 2.14: The minimal cover for $\{a, c\}$ (dual format)

2.3.2 Monotone Discrete Functions

The theory of monotone discrete functions as a tool for data-analysis has been developed in [11]. Here we only briefly review some concepts that are crucial for our approach. A discrete function of n variables is a function of the form:

$$f : X_1 \times X_2 \times \ldots \times X_n \to Y,$$

where $\mathcal{X} = X_1 \times X_2 \times \ldots \times X_n$ and Y are finite sets. Without loss of generality we may assume: $X_i = \{0, 1, \ldots, n_i\}$ and $Y = \{0, 1, \ldots, m\}$. Let $x, y \in X$ be two discrete vectors. Least upper bounds and greatest lower bounds will be defined as follows:

$$x \vee y = v, \text{ where } v_i = \max\{x_i, y_i\}, \tag{2.16}$$

$$x \wedge y = w, \text{ where } w_i = \min\{x_i, y_i\}. \tag{2.17}$$

Furthermore, if f and g are two discrete functions then we define:

$$(f \vee g)(x) = \max\{f(x), g(x)\}, \tag{2.18}$$

$$(f \wedge g)(x) = \min\{f(x), g(x)\}. \tag{2.19}$$

(Quasi) complementation for X is defined as: $\overline{x} = (\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n})$, where $\overline{x_i} = n_i - x_i$. Similarly, the complement of $j \in Y$ is defined as $\overline{j} = m - j$. The complement of a discrete function f is defined by: $\overline{f}(x) = \overline{f(x)}$. The *dual* of a discrete function f is defined as: $f^d(x) = \overline{f}(\overline{x})$. A discrete function f is called *positive (monotone non-decreasing)* if $x \leq y$ implies $f(x) \leq f(y)$.

Representations

Normal Forms Discrete variables are defined as:

$$x_{ip} = \text{if } x_i \geq p \text{ then } m \text{ else } 0, \text{ where } 1 \leq p \leq n_i,\ i \in (n] = \{1, \ldots, n\}. \tag{2.20}$$

Thus: $\overline{x}_{ip+1} = \text{if } x_i \leq p \text{ then } m \text{ else } 0$. Furthermore, we define $x_{in_i+1} = 0$ and $\overline{x}_{in_i+1} = m$. *Cubic* functions are defined as:

$$c_{v,j} = j.x_{1v_1} x_{2v_2} \cdots x_{nv_n}. \tag{2.21}$$

Notation: $c_{v,j}(x) = \text{ if } x \geq v \text{ then } j \text{ else } 0, \quad j \in (m]$.
Similarly, we define *anti-cubic* functions by:

$$a_{w,i} = i \vee x_{1w_1+1} \vee x_{2w_2+1} \cdots \vee x_{nw_n+1}. \tag{2.22}$$

Notation: $a_{w,i}(x) = \text{ if } x \leq w \text{ then } i \text{ else } m,\ i \in [m) = \{0, \ldots, m-1\}$. Note, that $j.x_{ip}$ denotes the conjunction $j \wedge x_{ip}$, where $j \in Y$ is a constant, and $x_{ip}x_{jq}$

denotes $x_{ip} \wedge x_{iq}$. A cubic function $c_{v,j}$ is called a *prime implicant* of f if $c_{v,j} \leq f$ and $c_{v,j}$ is maximal with respect to this property. The DNF of f:

$$f = \bigvee_{v,j} \{c_{v,j} \mid v \in \ j \in (m]\}, \tag{2.23}$$

is a unique representation of f as a disjunction of all its prime implicants (v is a minimal vector of class $d \geq j$).

If x_{ip} is a discrete variable and $j \in Y$ a constant then $x_{ip}^d = x_{i\overline{p}+1}$ and $j^d = \overline{j}$. The dual of the positive function $f = \bigvee_{v,j} j.c_{v,j}$ equals $f^d = \bigwedge_{v,j} \overline{j} \vee a_{\overline{v},\overline{j}}$.

Example 3

Let f be the function defined by table 2.11 and let e.g. x_{11} denote the variable: if $a \geq 1$ then 2 else 0, etc. Then:

$$f = 2.(x_{11}x_{21} \vee x_{22}) \vee 1.x_{11}, \text{ and}$$

$$f^d = 2.x_{12}x_{21} \vee 1.x_{22}.$$

Decision Lists In [11] it is shown that monotone functions can effectively be represented by decision lists of which the minlist and the maxlist representations are the most important ones. We introduce these lists here only by example. The minlist representation of the functions f and f^d of example 3 are respectively:

$$\begin{aligned} f(x) &= \text{ if } x \geq 11, 02 \text{ then } 2 \\ &\quad \text{ else if } x \geq 10 \text{ then } 1 \text{ else } 0, \end{aligned}$$

$$\begin{aligned} f^d(x) &= \text{ if } x \geq 21 \text{ then } 2 \\ &\quad \text{ else if } x \geq 02 \text{ then } 1 \text{ else } 0. \end{aligned}$$

The meaning of the minlist of f is given by:

$$\text{if } (a \geq 1 \wedge b \geq 1) \vee b = 2 \text{ then } 2$$
$$\text{else if } a \geq 1 \text{ then } 1 \text{ else } 0.$$

The maxlist of f is obtained from the minlist of f^d by complementing the minimal vectors as well as the function values, and by reversing the inequalities. The maxlist representation of f is therefore:

$$\begin{aligned} f(x) &= \text{ if } x \leq 01 \text{ then } 0 \\ &\quad \text{ else if } x \leq 20 \text{ then } 1 \text{ else } 2, \end{aligned}$$

minvectors	maxvectors	class
11, 02		2
10	20	1
	01	0

Table 2.15: Two representations of f

or equivalently:

$$\text{if } a = 0 \wedge b \leq 1 \text{ then } 0$$
$$\text{else if } b = 0 \text{ then } 1 \quad \text{else } 2.$$

The two representations are equivalent to a table as the one represented in table 2.15 that contains respectively the minimal and maximal vectors for each decision class of f. Each representation can be derived from the other by dualization.

Extensions of Monotone Data sets

A *partially defined discrete function* (pdDf) is a function: $f : D \mapsto Y$, where $D \subseteq \mathcal{X}$. We assume that a pdDf f is given by a decision table such as e.g. table 2.4. Although pdDfs are often used in practical applications, the theory of pdDfs is only developed in the case of pdBfs (partially defined Boolean functions). Here we discuss monotone pdDfs, i.e. functions that are monotone on D.

If the function $\hat{f} : \mathcal{X} \mapsto Y$, agrees with f on D: $\hat{f}(x) = f(x),\ x \in D$, then $\hat{f}$ is called an *extension* of the pdDf f. The collection of all extensions forms a lattice: for, if f_1 and f_2 are extensions of the pdDf f, then $f_1 \wedge f_2$ and $f_1 \vee f_2$ are also extensions of f. The same holds for the set of all monotone extensions.

The lattice of all *monotone* extensions of a pdDf f will be denoted here by $\mathcal{E}(f)$. It is easy to see that $\mathcal{E}(f)$ is universally bounded: it has a greatest and a smallest element. The maxlist of the maximal element called the *maximal monotone extension* can be directly obtained from the decision table.

Let us define the following notation:

$$\downarrow x = \{y \in \mathcal{X} : y \leq x\}$$
$$\uparrow x = \{y \in \mathcal{X} : y \geq x\}$$
$$\downarrow D = \bigcup_{x \in D} \downarrow x$$
$$\uparrow D = \bigcup_{x \in D} \uparrow x$$

Now we can define two functions which can be proved to be extensions of their corresponding function.

Definition 1 *Let f be a monotone pdDf. Then the functions $f_{\min}$ and $f_{\max}$ are defined as follows:*

$$f_{\min}(x) = \begin{cases} \max\{f(y) : \; y \in D \cap \downarrow x\} & \textit{if } x \in \uparrow D \\ 0 & \textit{otherwise,} \end{cases} \tag{2.24}$$

$$f_{\max}(x) = \begin{cases} \min\{f(y) : \; y \in D \cap \uparrow x\} & \textit{if } x \in \downarrow D \\ m & \textit{otherwise.} \end{cases} \tag{2.25}$$

Lemma 4 *Let f be a monotone pdDf. Then*
a) $f_{\min}, \; f_{\max} \; \in \mathcal{E}(f)$.
b) $\forall \hat{f} \in \mathcal{E}(f) : \quad f_{\min} \leq \hat{f} \leq f_{\max}$.

Since $\mathcal{E}(f)$ is a distributive lattice, the minimal and maximal monotone extension of f can also be described by the following expressions:

$$f_{\max} = \bigvee \{ \; \hat{f} \mid \hat{f} \; \in \; \mathcal{E}(f)\} \text{ and } f_{\min} = \bigwedge \{ \; \hat{f} \mid \hat{f} \; \in \; \mathcal{E}(f)\} \; . \tag{2.26}$$

We introduce the following notation. Let $T_j(f) = \{x \in D : f(x) = j\}$. A minimal vector v of class j is a vector such that $f(v) = j$ and no vector strictly smaller than v is also in $T_j(f)$. Similarly, a maximal vector w is a vector maximal in $T_j(f)$, where $j = f(w)$. The sets of minimal and maximal vectors of class j are denoted by $\min T_j(f)$ and $\max T_j(f)$ respectively.

According to the previous lemma $f_{\min}$ and $f_{\max}$ are respectively the minimal and maximal monotone extension of f. Decision lists of these extensions can be directly constructed from f as follows. Let $D_j = D \cap T_j(f)$, then $\min T_j(f_{\min}) = \min D_j$ and $\max T_j(f_{\max}) = \max D_j$.

Example 4

Consider the pdDf given by table 2.4, then its maximal extension is:

$$\begin{aligned} f(x) = \; & \text{if } x \leq 010 \text{ then } 0 \\ & \quad \text{else if } x \leq 100 \text{ then } 1 \\ & \quad\quad \text{else } 2. \end{aligned}$$

As described in the last subsection, from this maxlist representation we can deduce directly the minlist representation of the dual of f and finally by dualization we find that f is:

$$f = 2.(x_{12} \vee x_{11}x_{21} \vee x_{22} \vee x_{31}) \vee 1.x_{11}. \tag{2.27}$$

However, f can be viewed as a representation of table 4. This suggests a close relationship between minimal monotone decision rules and the maximal monotone extension $f_{\max}$. This relationship is discussed in the next section. Furthermore, the relationship with the methodology Logical Analysis of Data (LAD) is briefly discussed in subsection 3.5.

The Relationship between Monotone Decision Rules and $f_{\max}$

We first redefine the concept of a monotone reduct in terms of discrete functions. Let $\mathcal{X} = X_1 \times X_2 \times \ldots \times X_n$ be the input space, and let $A = \{1, \ldots, n\}$ denote the set of attributes. Then for $U \subseteq A$, $x \in \mathcal{X}$ we define the set $U.x$ respectively the vector $x.U$ by:

$$U.x = \{i \in U : x_i > 0\} \tag{2.28}$$

and

$$(x.U)_i = \begin{cases} x_i & \text{if } i \in U \\ 0 & \text{if } i \notin U. \end{cases} \tag{2.29}$$

Furthermore, the characteristic set U of x is defined by $U = A.x$.

Now we can give a new definition for a monotone object reduct:

Definition 2 *Suppose $f : D \to Y$ is a monotone pdDf, $w \in D$ and $f(w) = j$. Then $V \subseteq A$ is a monotone w-reduct iff $\forall x \in D : \; (f(x) < j \Rightarrow w.U \not\leq x.U)$.*

Note, that in this definition the condition $w.U \not\leq x.U$ is equivalent to $w.U \not\leq x$. The following lemma is a direct consequence of this definition.

Lemma 5 *Suppose f is a monotone pdDf, $w \in T_j(f)$. Then $V \subseteq A$ is a monotone w-reduct $\Leftrightarrow \forall x(f(x) < j \Rightarrow \exists i \in V$ such that $w_i > x_i)$.*

Corollary 1 *V is a monotone w-reduct iff $V.w$ is a monotone w-reduct. Therefore, without loss of generality we may assume that V is a subset of the characteristic set W of w: $V \subseteq W$.*

Monotone Boolean functions We first consider the case when the data set is Boolean, therefore the objects are described by condition and decision attributes taking one of two possible values $\{0, 1\}$.

The data set represents a *partially defined Boolean function* (pdBf) $f : D \to \{0, 1\}$ where $D \subseteq \{0, 1\}^n$. As we have only two classes, we define the set of true vectors of f by $T(f) := T_1(f)$ and the set of false vectors of f by $F(f) := T_0(f)$.

In the sequel we abuse the notation in the following way: in the Boolean case we will make no distinction between a set V and its characteristic vector v.

Lemma 6 *Let $f : D \to \{0, 1\}$ be a monotone pdBf, $w \in D$, $w \in T(f)$. Suppose $v \leq w$. Then v is a w-reduct $\Leftrightarrow v \in T(f_{\max})$.*

Proof: Since $v \leq w$, we have:

$$v \text{ is a } w\text{-reduct } \Leftrightarrow \forall x(x \in D \cap F(f) \Rightarrow v \not\leq x) \Leftrightarrow v \in T(f_{\max}).$$

□

Theorem 1 *Suppose $f : D \to \{0, 1\}$ is a monotone pdBf, $w \in D$, $w \in T(f)$. Then, for $v \leq w$, $v \in \min T(f_{\max}) \Leftrightarrow v$ is a minimal monotone w-reduct.*

Proof: Let $v \in \min T(f_{\max})$ and $v \leq w$ for some $w \in D$. Then v is a monotone w-reduct. Suppose $\exists u < v$ and u is a monotone w-reduct. Then by definition 2 we have: $u \in T(f_{\max})$, which contradicts the assumption that $v \in \min T(f_{\max})$.

Conversely, let v be a minimal monotone w-reduct. Then by lemma 6 we have: $v \in T(f_{\max})$. Suppose $\exists u < v : u \in T(f_{\max})$. However, $v \leq w \Rightarrow u < w$ therefore u is a monotone w-reduct, which contradicts the assumption that v is a minimal w-reduct. □

The results imply that the irredundant (monotone) decision rules that correspond to the object reducts are just the prime implicants of the maximal extension.

Corollary 2 *The decision rules obtained in rough set theory can be obtained by the following procedure:*

a) find the maximal vectors of class 1 (positive examples)

b) determine the minimal vectors of the dual of the maximal extension, and

c) compute the minimal vectors of this extension by dualization. The complexity of this procedure is the same as for the dualization problem.

Although the above corollary is formulated for monotone Boolean functions, results in [33] indicate that a similar statement also holds for Boolean functions in general.

Monotone Discrete Functions Let us now consider the case when f is a monotone *partially defined discreet function* (pdDf). First we observe that the following lemma is true:

Lemma 7 *Suppose f is a monotone pdDf, $w \in T_j(f)$ and $v \leq w$. If $v \in T_j(f_{\max})$ then the characteristic set V of v is a monotone w-reduct.*

Proof: $f_{\max}(v) = j$ implies $\forall x(f(x) < j \Rightarrow v \not\leq x)$. Since $w \geq v$ we therefore have $\forall x(f(x) < j \Rightarrow \exists i \in V$ such that $w_i \geq v_j > x_i)$. □

Remark: Even if in lemma 7 the vector v is minimal: $v \in \min T_j(f_{\max})$, then still $V = A.v$ is not necessarily a minimal monotone w-reduct.

Theorem 2 *Suppose f is a monotone pdDf and $w \in T_j(f)$. Then $V \subseteq A$ is a monotone w-reduct $\Leftrightarrow f_{\max}(w.V) = j$.*

Proof: If V is a monotone w-reduct, then by definition $\forall x(f(x) < j \Rightarrow w.V \not\leq x)$. Since $w.V \leq w$ and $f(w) = j$ we therefore have $f_{\max}(w.V) = j$.

Conversely, let $f_{\max}(w.V) = j$, $V \subseteq A$. Then, since $w.V \leq w$ and the characteristic set of $w.V$ is equal to V, lemma 7 implies that V is a monotone w-reduct. □

Theorem 3 *Let f be a monotone pdDf and $w \in T_j(f)$. If $V \subseteq A$ is a minimal monotone w-reduct, then $\exists u \in \min T_j(f_{\max})$ such that $V = A.u$.*

Proof: Since V is a monotone w-reduct, theorem 2 implies that $f_{\max}(w.V) = j$. Therefore, $\exists u \in \min T_j(f_{\max})$ such that $u \leq w.V$. Since $A.u \subseteq V$ and $A.u$ is a monotone w-reduct (by lemma 7), the minimality of V implies $A.u = V$. □

Theorem 3 implies that the minimal decision rules obtained by monotone w-reducts are not shorter than the minimal vectors (prime implicants) of $f_{\max}$. This suggests that we can optimize a minimal decision rule by minimizing the attribute values to the attribute values of a minimal vector of $f_{\max}$. For example, if V is a minimal monotone w-reduct and $u \in \min T_j(f_{\max})$ such that $u \leq w.V$ then the rule:

'if $x_i \geq w_i$ then j',

where $i \in V$, can be improved by using the rule:

'if $x_i \geq u_i$ then j',

where $i \in V$. Since $u_i \leq w_i$, $i \in V$, the second rule is applicable to a larger part of the input space X.

The results so far indicate the close relationship between minimal monotone decision rules obtained by the rough sets approach and by the approach using $f_{\max}$. To complete the picture we make the following observations:

Observation 1 *: The minimal vector u (theorem 3) is not unique.*

Observation 2 *: Lemma 7 implies that the length of a decision rule induced by a minimal vector $v \leq w$, $v \in \min T_j(f_{\max})$, is not necessarily smaller than that of a rule induced by a minimal w-reduct. This means that there may exist an $x \in X$ that is covered by the rule induced by v but not by the decision rules induced by the minimal reducts of a vector $w \in D$.*

Observation 3 *: There may be minimal vectors of $f_{\max}$ such that $\forall w \in D$ $v \not\leq w$. In this case if $x \geq v$ then $f_{\max}(x) = m$ but x is not covered by a minimal decision rule induced by a minimal reduct.*

In the next two subsections we briefly compare the rough set approach and the discrete function approach with two other methods.

2.3.3 Related Research

The Approach of Greco et al.

Earlier research presented by Greco et al. in [40, 41] had the same goal of incorporating the monotonicity property of the problem in the Rough Sets analysis. They, however, started from a very different direction - the Multi-Criteria

Decision Aid (MCDA) methodology and hence they position the research in multi-criteria sorting and ranking.

They start by substituting the indiscernibility (equivalence) relation by a dominance relation in the following way. For each criterion q there is an associated outranking relation S_q on the universe U such that xS_qy whenever "x is at least as good as y with respect to q". The intersection of those relations $D_P = \cap_{q \in P} S_q$ for a set of criteria $P \subseteq C$ is a dominance relation.

Furthermore, the following sets are defined (where Cl_s denotes the set of objects of class s):

$$Cl_t^{\geq} = \bigcup_{s \geq t} Cl_s, \;\; Cl_t^{\leq} = \bigcup_{s \leq t} Cl_s,$$

$$D_P^+(x) = \{y \in U : yD_Px\},$$

$$D_P^-(x) = \{y \in U : xD_Py\}.$$

New definitions for the lower and upper approximation are given for the sets $Cl_t^{\geq}$ as follows:

$$\begin{array}{rcl} \underline{P}(Cl_t^{\geq}) & = & \{x \in U : D_P^+(x) \subseteq Cl_t^{\geq}\}, \\ \overline{P}(Cl_t^{\geq}) & = & \bigcup\limits_{x \in Cl_t^{\geq}} D_P^+(x), \end{array}$$

and similarly for the sets $Cl_t^{\leq}$ as follows:

$$\begin{array}{rcl} \underline{P}(Cl_t^{\leq}) & = & \{x \in U : D_P^-(x) \subseteq Cl_t^{\leq}\}, \\ \overline{P}(Cl_t^{\leq}) & = & \bigcup\limits_{x \in Cl_t^{\leq}} D_P^-(x). \end{array}$$

The boundaries of $Cl_t^{\geq}$ and $Cl_t^{\leq}$ are defined as:

$$Bn_P(Cl_t^{\geq}) = \overline{P}(Cl_t^{\geq}) - \underline{P}(Cl_t^{\geq}), \;\; Bn_P(Cl_t^{\leq}) = \overline{P}(Cl_t^{\leq}) - \underline{P}(Cl_t^{\leq}).$$

In order to define a reduct, the quality of approximation measure is used (where Cl is the partition of U generated by the decision attribute):

$$\gamma_P(Cl) = \frac{|U - ((\bigcup_{t \in T} Bn_P(Cl_t^{\geq})) \bigcup (\bigcup_{t \in T} Bn_P(Cl_t^{\leq})))|}{|U|}.$$

A reduct is thus such a minimal subset $p \subseteq C$ for which $\gamma_P(Cl) = \gamma_C(Cl)$.

Three different types of decision rules are defined and used together to form a classifier:

if $q_1(x) \geq v_1$ and $q_2(x) \geq v_2$ and ... and $q_k(x) \geq v_k$ then $x \in Cl_t^{\geq}$, if $q_1(x) \leq v_1$ and $q_2(x) \leq v_2$ and ... and $q_k(x) \leq v_k$ then $x \in Cl_t^{\leq}$,

if $q_1(x) \geq v_1$ and $q_2(x) \geq v_2$ and ... and $q_k(x) \geq v_k$ and $q_{k+1}(x) \leq v_1$ and $q_{k+2}(x) \leq v_2$ and ... and $q_{k+l}(x) \leq v_k$ then $x \in Cl_t \cup Cl_{t+1} \cup \ldots \cup Cl_{t+s}$.

There are a number of differences between our approach and the approach of Greco et al. which are briefly discussed in the following paragraphs.

First of all, the definition of a reduct in our approach is based on the positive area (and the discernibility matrix is used for the computations) while Greco et al. use the quality of approximation measure. As a consequence, different definitions for the lower and upper approximations are appropriate. In fact the approximations proposed by Greco et al. cannot be directly used to re-define the positive area because the union of all lower approximations of $Cl_t^{\geq}$ is always equal to U (and the same holds for the union of all lower approximations of $Cl_t^{\leq}$). That is easy to see starting from the fact that $Cl_1^{\geq} = Cl_n^{\leq} = U$.

It should be emphasized that the resulting definition of a reduct is equivalent to the one of Greco et al. by means of covering the same set of reducts. However the procedure used for their generation is different - Greco et al. use a version of the quality of approximation measure while we use a version of the discernibility matrix and dualization. The notion of monotone discernibility matrix introduced here is an important one and it becomes useful in other settings as well. In chapter 4 we show how it can help in monotone function decomposition.

Another important difference is the type of rules that are generated and how they are used. The three types of rules defined by Greco et al. when used together might lead to monotone inconsistencies and therefore the final classifier is not guaranteed to be monotone. In our approach only the first two types of rules are allowed but they are used as alternative solutions and not together. This results in monotone classifiers. Furthermore, the maximal/minimal extensions allow us to assign single class values and not intervals. They also cover the whole input space which is not necessarily the case for the rule sets of Greco et al.

A fundamental difference is the goal of both approaches. In [40, 41] the goal is to incorporate the additional knowledge of ordering and preference while at the same time taking into account any existing inconsistencies. However, by generating rules covering the conflicting data point the classifier becomes by definition non-monotone. The method presented here, on the other hand, aims at generating a monotone classifier which covers the whole input space.

The above mentioned differences were also demonstrated in our experiments where the same data set was used as in [40] in order to facilitate the parallel.

Rough Sets and Logical Analysis of Data

The Logical Analysis of Data methodology (LAD) was presented in [33] and further developed in [25, 23, 24]. LAD is designed for the discovery of structural information in data sets. Originally it was developed for the analysis of Boolean data sets using partially defined Boolean functions. An extension of LAD for the analysis of numerical data is possible through the process of binarization. The building concepts are the supporting set, the pattern and the theory.

A set of variables (attributes) is called a *supporting set* for a partially defined Boolean function f if f has an extension depending only on these variables. A *pattern* is a conjunction of literals such that it is 0 for every negative example and 1 for at least one positive example. A subset of the set of patterns is used to form a *theory* – a disjunction of patterns that is consistent with all the available data and can predict the outcome of any new example. The theory is therefore an extension of the partially defined Boolean function.

Our research suggests that the LAD and the RS theories are similar in several aspects (for example, the supporting set corresponds to the reduct in the binary case and a pattern with the induced decision rule). The exact connections will be a subject of future research.

2.4 Experiments

2.4.1 The Bankruptcy Prediction Problem

The research in bankruptcy prediction has a long history dating back to the 1930s. A number of statistical methods were applied for developing models able to predict in advance whether a company will go bankrupt or not. The analysis is based on financial information about the company in the form of financial indicators and ratios obtained from the company's annual reports. The main goal is to describe the relationship between these indicators and bankruptcy using available data about companies that have already gone bankrupt and data about "healthy" companies.

A major breakthrough in the research was achieved in the 1960s by applying the method of discriminant analysis for bankruptcy prediction. In 1968 Altman proposed multivariate discriminant analysis for developing the prediction model [4] and the approach has been further improved and tested in a number of studies since then. However, this method makes several assumptions that are not always present in real-life data. This encouraged the researchers to look for alternatives. One of them is the logistic analysis that was proposed in the 1980s. It was applied on bankruptcy data and gave very good results.

The success of the machine learning methods in a number of application domains suggested that they might be useful for predicting bankruptcy as well. Neural networks, decision trees, genetic algorithms, rough sets and other machine learning approaches were applied to bankruptcy data with promising results [6, 52, 62, 74, 75]. In a number of studies these methods are tested and compared to the traditional statistical techniques. Some of the articles suggest that the new approaches outperform the classical methods on data sets from a number of areas including bankruptcy prediction.

In this research the bankruptcy prediction problem is interpreted as a classification problem with monotonicity constraints.

2.4.2 The Bankruptcy Data set

The data set used in the experiments is discussed in [74, 40]. The sample consists of 39 objects denoted by $F1$ to $F39$ – firms that are described by 12 financial parameters. To each company a decision value is assigned – the expert evaluation of its category of risk for the year 1988. The condition attributes denoted by $A1$ to $A12$ take integer values from 0 to 4.

The decision attribute is denoted by d and takes integer values in the range 0 to 2 where: 0 means *unacceptable*, 1 means *uncertainty* and 2 means *acceptable*. This and the other data sets used in the following chapters are described in more details in Appendix A.

The data was first analyzed for monotonicity. The problem is obviously monotone (if one company outperforms another on all condition attributes then it should not have a lower value of the decision attribute). Nevertheless, one noisy example was discovered, namely $F24$. It was removed from the data set and was not considered further.

2.4.3 Reducts and Decision Rules

The minimal reducts have been computed using our program Dualizer. There are 25 minimal general reducts (minimum length 3) and 15 monotone reducts (minimum length 4). They are given in tables 2.16 and 2.17 respectively.

We have also compared the heuristics to approximate a minimum reduct: the "Best reduct" method (for general reducts) and the Johnson strategy (for general and monotone reducts). The results are given in tables 2.18 and 2.19. Note that for the "Best reduct" method the results do not change for the monotone case since we use no additional information. For the Johnson's heuristic, however, the results change because we use the monotone discernibility matrix instead of the general discernibility matrix. The set of generated reducts here is also influenced by whether we use simplification (remove redundancies, etc.) or not both for the general and for the monotone case.

Table 2.20 shows the two sets of decision rules obtained by computing the object (value)-reducts for the monotone reduct $(A1, A3, A7, A9)$. Both sets of rules have minimal covers, of which the ones with minimum length are shown in table 2.21. A minimal cover can be transformed into an extension if the rules

Minimal general reducts:
$A1, A7, A11$
$A3, A6, A7, A12$
$A1, A3, A7, A9$
$A5, A6, A7, A8$
$A1, A3, A7, A12$
$A5, A6, A7, A9$
$A1, A5, A7, A9$
$A5, A6, A7, A10$
$A1, A6, A7, A12$
$A5, A6, A7, A11$
$A1, A7, A8, A9$
$A5, A6, A7, A12$
$A1, A7, A9, A12$
$A6, A7, A8, A10$
$A2, A3, A6, A7$
$A6, A7, A8, A11$
$A2, A5, A6, A7$
$A6, A7, A8, A12$
$A2, A6, A7, A11$
$A6, A7, A9, A11$
$A2, A6, A7, A12$
$A6, A7, A9, A12$
$A3, A6, A7, A8$
$A3, A6, A7, A9$
$A3, A6, A7, A10$

Table 2.16: All minimal general reducts

Minimal monotone reducts:
$A1, A3, A7, A9$
$A1, A5, A7, A9$
$A2, A3, A6, A7$
$A2, A5, A6, A7$
$A3, A6, A7, A9$
$A3, A6, A7, A10$
$A3, A6, A7, A12$
$A5, A6, A7, A8$
$A5, A6, A7, A9$
$A5, A6, A7, A10$
$A5, A6, A7, A11$
$A5, A6, A7, A12$
$A1, A3, A6, A7, A8$
$A1, A3, A6, A7, A11$

Table 2.17: All minimal monotone reducts

The Best Reduct method:	Johnson's algorithm:	
	with simplification	without simplification
$A1, A7, A8, A11$	$A1, A3, A7, A9$	$A1, A7, A8, A11$
$A6, A7, A8, A11$	$A1, A5, A7, A9$	$A6, A7, A8, A11$
	$A3, A6, A7, A9$	$A3, A6, A7, A8$
	$A5, A6, A7, A9$	

Table 2.18: General reducts using the two heuristics

Johnson's algorithm:	
with simplification	without simplification
$A1, A3, A7, A9$	$A2, A5, A6, A7$
$A1, A5, A7, A9$	$A5, A6, A7, A8$
$A3, A6, A7, A9$	$A5, A6, A7, A9$
$A5, A6, A7, A9$	$A5, A6, A7, A10$
	$A5, A6, A7, A11$
	$A5, A6, A7, A12$

Table 2.19: Monotone reducts using the two heuristics

class $d \geq 2$	class $d \geq 1$
$A1 \geq 3$	$A1 \geq 3$
$A7 \geq 4$	$A3 \geq 3$
$A9 \geq 4$	$A7 \geq 3$
$A1 \geq 2 \wedge A7 \geq 3$	$A9 \geq 4$
$A3 \geq 2 \wedge A7 \geq 3$	$A1 \geq 1 \wedge A3 \geq 2$
$A7 \geq 3 \wedge A9 \geq 3$	$A1 \geq 1 \wedge A9 \geq 3$
	$A3 \geq 2 \wedge A7 \geq 2$
	$A3 \geq 2 \wedge A7 \geq 1 \wedge A9 \geq 3$
class $d \leq 0$	class $d \leq 1$
$A7 \leq 0$	$A7 \leq 2$
$A9 \leq 1$	$A9 \leq 2$
$A1 \leq 0 \wedge A3 \leq 0$	
$A1 \leq 0 \wedge A3 \leq 2 \wedge A7 \leq 1$	
$A1 \leq 0 \wedge A3 \leq 1 \wedge A7 \leq 2$	
$A1 \leq 0 \wedge A3 \leq 2 \wedge A9 \leq 2$	
$A3 \leq 0 \wedge A9 \leq 2$	
$A3 \leq 1 \wedge A7 \leq 2 \wedge A9 \leq 2$	
$A3 \leq 2 \wedge A7 \leq 1 \wedge A9 \leq 2$	

Table 2.20: The rules for $(A1, A3, A7, A9)$

are considered as minimal/maximal vectors in a decision list representation. In this sense the minimal cover of the first set of rules can be described by the following function:

$$f = 2.x_{73}x_{93} \vee 1.(x_{33} \vee x_{73} \vee x_{11}x_{93} \vee x_{32}x_{72}). \tag{2.30}$$

The maximal extension corresponding to the monotone reduct $(A1, A3, A7, A9)$ is represented in table 2.22.

Our results show that the function f or equivalently its minlist consists of only 5 decision rules (prime implicants). They cover the whole input space. Moreover, each possible vector is classified as $d = 0, 1$ or 2 and not as $d \geq 1$ or $d \geq 2$ like in [40, 41].

The method presented in [40, 41] uses both formats shown in table 2.20 to describe a minimal cover, resulting in a system of 11 rules. Using both formats at the same time can result in much (possibly exponential) larger sets of rules. As mentioned before, using both formats of rules at the same time may also result in conflicting predictions.

We also computed a monotone decision tree [18, 64] for the bankruptcy data set discussed here. The tree is given in figure 2.2. It appears that monotone decision trees are larger because they contain the information of both an extension and its dual. The topic of generating monotone decision trees is discussed

class $d \geq 2$	class $d \geq 1$
$A7 \geq 3 \wedge A9 \geq 3$	$A3 \geq 3$
	$A7 \geq 3$
	$A1 \geq 1 \wedge A9 \geq 3$
	$A3 \geq 2 \wedge A7 \geq 2$
class $d \leq 0$	**class $d \leq 1$**
$A1 \leq 0 \wedge A3 \leq 2 \wedge A7 \leq 1$	$A7 \leq 2$
$A1 \leq 0 \wedge A3 \leq 1 \wedge A7 \leq 2$	$A9 \leq 2$
$A3 \leq 1 \wedge A7 \leq 2 \wedge A9 \leq 2$	

Table 2.21: The minimal covers for $(A1, A3, A7, A9)$

class $d = 2$	class $d = 1$
$A1 \geq 3$	$A3 \geq 3$
$A3 \geq 4$	$A7 \geq 3$
$A7 \geq 4$	$A1 \geq 1 \wedge A3 \geq 2$
$A9 \geq 4$	$A1 \geq 1 \wedge A9 \geq 3$
$A1 \geq 2 \wedge A7 \geq 3$	$A3 \geq 2 \wedge A7 \geq 2$
$A3 \geq 2 \wedge A7 \geq 3$	$A3 \geq 2 \wedge A7 \geq 1 \wedge A9 \geq 3$
$A7 \geq 3 \wedge A9 \geq 3$	

Table 2.22: The maximal extension for $(A1, A3, A7, A9)$

in detail in the next chapter.

2.5 Conclusions

This chapter discusses an extension of the theory of Rough Sets for generating monotone classifiers from monotone data sets. Our approach uses the concepts of monotone discernibility matrix/function and monotone (object) reduct and the theory of monotone discrete functions. It has a number of advantages over previous research on the problem as it was summarized in section 2.3.3 and in the discussion on the experiment with the bankruptcy data set in section 2.4.

Compared to the previous research, the approach presented here produces smaller sets of rules that are consistent (do not predict conflicting class values), cover the whole input space and form a monotone classifier. When the maximal extension is used, the predictions are of single class values and not sets of values as in the previous research.

Another difference is the use of the discernibility matrix for computing the monotone reducts. This approach provides a comprehensive method for generating all monotone reducts instead of using heuristics for generating one short (but not necessarily of minimum length) reduct.

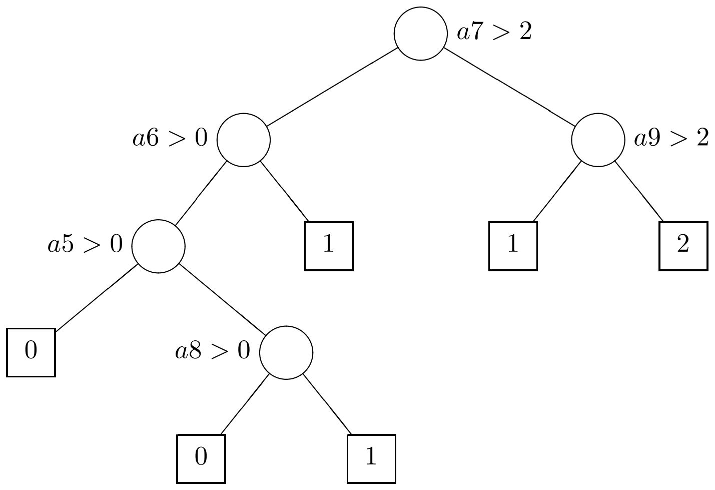

Figure 2.2: Monotone decision tree for the bankruptcy data set

Compared to monotone decision trees, our method produces a more compact classifier since the decision tree contains the information of both the extension and its dual.

Furthermore, it appears that there is a close relationship between the decision rules obtained using the rough set approach and the prime implicants of the maximal extension. Although this has been shown for the monotone case this also holds at least for non-monotone *Boolean* data sets. We have discussed how to compute this extension by using dualization.

The generalization of the discrete function approach to non-monotone data sets and the comparison with the theory of Rough Sets is a topic of further research. Finally, the sometimes striking similarity we have found between Rough Set Theory and Logical Analysis of Data remains an interesting research topic.

Chapter 3

Monotone Decision Trees

3.1 Introduction

In the previous chapter we considered a classifier expressed in decision rules. Another frequently used representation is a decision tree. A *decision tree* is a directed, acyclic, connected graph with a designated starting node (a root) and a designated set of terminal nodes (leaves). At each non-terminal node a test is performed on a certain attribute value(s) and at each leaf a class value is assigned.

Decision trees were first introduced in Machine Learning by Quinlan with the ID3 algorithm [66, 67]. It was applied originally to discrete domains but was extended to C4.5 which is also applicable to continuous domains [69]. C4.5 is now one of the most popular decision tree algorithms. The statistical point of view on the problem was expressed in the other mainstream decision tree algorithm CART (Classification and regression trees) [26].

A number of attempts were made to apply decision trees on the classification for monotone problems, see [9, 18, 55, 64], from which the most successful was the monotone decision trees algorithm introduced in [18, 64].

This chapter addresses the problem of classification with monotonicity constraints in the context of monotone decision trees (MDT). It extends the algorithm presented in [64] in several directions in order to provide a full set of possibilities for solving real-life problems similar to the possibilities available for the classical decision trees.

Data noise is one problem that frequently occurs in real-life classification problems and is extensively studied by a number of authors from different perspectives. In classification problems with monotonicity constraints, noise often causes an additional problem not relevant for the general case – violation of the monotonicity constraint. The MDT algorithm requires a strictly monotone data set. This chapter proposes an extension for dealing with monotonicity noise which allows the generation of a monotone tree from any non-monotone data set.

Decision tree pruning is another area that has attracted a lot of attention (see [27] for a survey). A number of successful methods are available for reducing the tree size and avoiding the overfitting of the particular properties of the data set. However the monotonicity constraint raises new questions, the most important of which is how to label the new leaves so that the tree remains monotone. This chapter tries to answer the question in the setting of pre-pruning as well as post-pruning. Furthermore, we address the more general problem of labelling any tree in a consistent way so that it becomes monotone.

Most of the chapter is based on the publications [17, 16].

The chapter is organized as follows. Section 3.2 presents the original monotone decision tree algorithm. An extension for generating monotone trees from noisy non-monotone data is described in section 3.3. A number of pruning approaches and labelling functions which guarantee the monotonicity of the tree are presented in section 3.4.

Section 3.5 investigates the performance of two different splitting criteria in monotone decision tree generation in order to give more insight into which one better fits the classification problems with monotonicity restrictions. Section 3.6 explores the problem of missing attribute values in the context of monotone classification. A simple preprocessing method is proposed as an extension of a number of general approaches for filling in the unknown values so that the monotonicity property of the resulting data set is guaranteed.

The methods discussed in the paper are tested experimentally and the experimental settings, data sets and results are given in section 3.7. The conclusions of the chapter are given in section 3.8.

3.2 Algorithms for the Induction of Monotone Decision Trees

The classical decision tree algorithm can be characterized by three rules:

– a *stopping* rule defining when to stop growing a branch and turn the current node to a leaf,

– a *splitting* rule which fires when the stopping rule fails to fire – it defines how to split a node by choosing the best attribute and the best split value for the test associated to the node, according to a consistent criterion,

– a *labelling* rule which fires when the stopping rule fires and defines how to label the new leaf – this label will later be assigned to all new examples classified to this leaf.

This algorithm in its general form does not take into account the monotonicity property of the data set. Given a fully monotone data set it is not guaranteed to generate a monotone classifier. The first attempt to confront the problem was presented by Ben-David in [9] where a non-monotonicity index was proposed to measure how monotone a tree is. Used together with the entropy splitting criterion, the index influences the generation process towards building a nearly monotone tree. The monotonicity, however, is not guaranteed. An improvement to the algorithm towards a better measure for the monotonicity of the tree is proposed in [65].

The research of Makino et al. presented in [55] goes a step further by providing a method for generating a monotone tree for the 2-class problem. The algorithm requires a strictly monotone data set. A more general approach applicable to the k-class problem was proposed by Potharst and Bioch in [18, 64]. It also requires a monotone data set. This method is the starting point of the research presented in this chapter.

An overview of the three methods will be presented in the next two subsections. The algorithms of Ben-David [9] and Makino et al. [55] will be briefly described in subsections 3.2.1 and 3.2.2 respectively. More attention will be paid to the algorithm of Potharst and Bioch [18, 64] which is presented in subsection 3.2.3 together with a small example.

3.2.1 The Algorithm of Ben-David

The method of Ben-David modifies the impurity criterion for choosing the best attribute-value pair to split on for the splitting rule. To the traditional measure defined in the ID3 algorithm a measure for the non-monotonicity is added which results in the so-called total-ambiguity-score. This is defined in the following paragraph using the notation of [9].

First, a *non-monotonicity matrix* M is constructed which is a symmetric $k \times k$ matrix for a tree containing k branches. An entry m_{ij} (and m_{ji}) of M is 1 if the corresponding branches i and j are non-monotone with respect to each other and 0 otherwise.

From the non-monotonicity matrix, the sum of the entries is calculated:

$$W = \sum_{i=1}^{k} \sum_{j=1}^{k} m_{ij}.$$

It is used in the so-called *non-monotonicity index* defined as follows for the attribute tests $a_1, a_2, \ldots, a_\nu$:

$$I_{a_1,a_2,\ldots,a_\nu} = \frac{W_{a_1,a_2,\ldots,a_\nu}}{k^2_{a_1,a_2,\ldots,a_\nu} - k_{a_1,a_2,\ldots,a_\nu}}.$$

The non-monotonicity index in turn is used in the definition of the *order-ambiguity-score*:

$$A_{a_1,a_2,\ldots,a_\nu} = \begin{cases} 0 & \text{if } I_{a_1,a_2,\ldots,a_\nu} = 0, \\ -(\log_2 I_{a_1,a_2,\ldots,a_\nu})^{-1} & \text{otherwise.} \end{cases}$$

Finally the *total-ambiguity-score* is defined as the sum of the order-ambiguity-score and the E-score as it was defined in the ID3 algorithm:

$$T_{a_1,a_2,\ldots,a_\nu} = E_{a_1,a_2,\ldots,a_\nu} + A_{a_1,a_2,\ldots,a_\nu}.$$

A variation of the definition employs a parameter R in order to control the trade-off between the entropy measure (the E-score) and the monotonicity measure (the order-ambiguity score A):

$$T_{a_1,a_2,\ldots,a_\nu} = E_{a_1,a_2,\ldots,a_\nu} + RA_{a_1,a_2,\ldots,a_\nu}.$$

It is important to emphasize that the method works both for monotone and non-monotone data, however, the major drawback is that the monotonicity of the generated tree is not guaranteed.

3.2.2 The Algorithm of Makino et al.

This algorithm also uses a modified version of the splitting rule which favours not only splits with equal number of positive and negative points but also those that have the right child-node larger than the left child-node. More precisely, the splitting in the ID3 algorithm is guided by the following functions:

$$gain(x_i : c) = I(|P|, |N|) - E(x_i : c),$$

$$\begin{aligned} E(x_i : c) &= \frac{|B_0(P)| + |B_0(N)|}{|P| + |N|} I(|B_0(P)|, |B_0(N)|) \\ &+ \frac{|B_1(P)| + |B_1(N)|}{|P| + |N|} I(|B_1(P)|, |B_1(N)|), \end{aligned}$$

$$I(p, n) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n},$$

where $x_i : c$ is the candidate split on attribute x_i and value c, P is the set of positive vectors, N is the set of negative vectors and for the candidate split:

$$B_0(P) = \{x \in P | x_i < c\},$$

$$B_0(N) = \{x \in N | x_i < c\},$$

$$B_1(P) = \{x \in P | x_i \geq c\},$$

$$B_1(N) = \{x \in N | x_i \geq c\}.$$

The above introduced functions are modified as follows:

$$\begin{aligned} E^+(x_i : c) &= \frac{|B_0(P)| + |B_0(N)|}{|P| + |N|} I^+(|B_0(P)|, |B_0(N)|) \\ &+ \frac{|B_1(P)| + |B_1(N)|}{|P| + |N|} I^+(|B_1(P)|, |B_1(N)|), \end{aligned}$$

$$I^+(p, n) = \begin{cases} 1 & \text{if } p < n, \\ I(p, n) & \text{otherwise.} \end{cases}$$

The main contribution of the algorithm, however, is the procedure of adding points to the nodes in such a way that the resulting tree will be monotone[1]. This procedure works as follows:

– Whenever a leaf t is labelled with 0, the maximal vector $\alpha^{(t)}$ in the leaf is added also with label 0. Furthermore, for each node which has yet to be split or labelled (active nodes), the maximal vector smaller or equal to $\alpha^{(t)}$ is added with label 0 to the data in the node.

– Alternatively if the leaf t is labelled with 1, the minimal vector $\beta^{(t)}$ is added with label 1. For each active node the minimal vector larger than $\beta^{(t)}$ is also added to the node with label 1.

We emphasize again that this algorithm can only be applied on strictly monotone Boolean data sets.

3.2.3 The Algorithm of Potharst and Bioch

The algorithm of Potharst and Bioch presented in [18, 64] extends in a non-trivial way the algorithm of Makino et al. towards the k-class problem. It also uses a procedure for adding new points.

Let T be a node of the tree $\mathcal{T}$ generated thus far on the data set $D \subseteq \mathcal{X}$ where $\mathcal{X}$ is the input space. T can be represented as:

$$T = \{x \in \mathcal{X} : a(T) \leq x \leq b(T)\}$$

where $a(T)$ is called the *lower corner* and $b(T)$ is called the *upper corner* of T. In the following, the corners will be denoted simply by a and b when no ambiguity occurs.

The original MDT algorithm is given in figure 3.1. In order to guarantee the monotonicity of the tree, an update procedure is performed on the data set by

[1]In the algorithm there is also a procedure for adding points to ensure the quasi-monotonicity of the generated tree but we do not discuss this aspect here as we are only interested in the monotonicity property.

```
split(node $T$):
    **update($T$)**;
    if $T$ is homogeneous
        label($T$);
    else
        split $T$ into disjoint $T_L$ and $T_R$;
        split($T_L$);
        split($T_R$);

**update(node $T$)**:
    if $a \notin D$
        $\lambda(a) = \lambda_{\max}(a)$;
        add $a$ to $D$;
    if $b \notin D$
        $\lambda(b) = \lambda_{\min}(b)$;
        add $b$ to $D$;
```

Figure 3.1: The monotone decision tree algorithm

adding at most 2 new data points with consistent labels. The update procedure is performed every time a node is considered for splitting and the added points are the lower and the upper corners of the node (if they are not already present in the data set). The labels are chosen to be $\lambda_{\max}(a)$ and $\lambda_{\min}(b)$ respectively, which are defined in the following, where $c_{\min}/c_{\max}$ are the lowest/highest class in the data set:

$$\downarrow x = \{y \in \mathcal{X} : y \leq x\}$$

$$\uparrow x = \{y \in \mathcal{X} : y \geq x\}$$

$$\downarrow D = \bigcup_{x \in D} \downarrow x$$

$$\uparrow D = \bigcup_{x \in D} \uparrow x$$

$$\lambda_{\min}(x) = \begin{cases} \max\{\lambda(y) : \ y \in D \cap \downarrow x\} & \text{if } x \in \uparrow D \\ c_{\min} & \text{otherwise.} \end{cases}$$

$$\lambda_{\max}(x) = \begin{cases} \min\{\lambda(y) : \ y \in D \cap \uparrow x\} & \text{if } x \in \downarrow D \\ c_{\max} & \text{otherwise.} \end{cases}$$

Using this way of labelling we guarantee that the lower corner gets the minimal label possible for the node and the upper corner gets the maximal possible label. At the same time the new points remain consistent with the rest of the data, therefore the monotonicity constraint is not violated.

Figure 3.2 is meant to give some intuition in how this works. The figure shows the two corners (a and b) of a leaf T. For corner b, the area of points x in the data set such that $x \leq b$ is given. Similarly, for the corner a, the area of points y such that $y \geq a$ is indicated. The intersection of those two

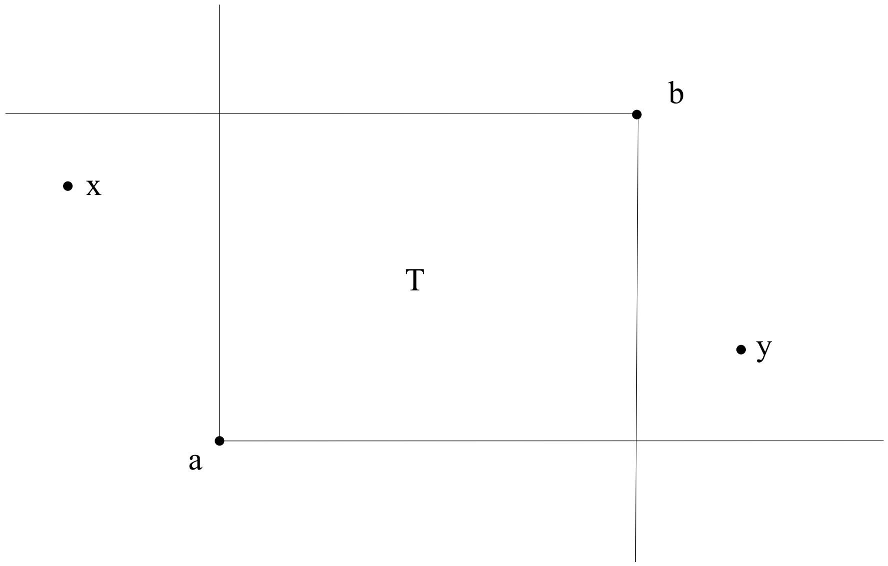

Figure 3.2: A leaf T with corners a and b

areas is obviously the leaf T. Let there exist a point $y \geq a$. Then the labelling function $\lambda_{\max}$ will assign a label to a that is not greater than the label of y because otherwise we introduce inconsistency in the data set. In a similar way, the labelling function $\lambda_{\min}$ will assign to b a label that is not lower than the label of $x \leq b$.

In order to illustrate how the algorithm works we use the example data set from table 3.1. In the first step, the root of the tree (containing the whole data set) is considered for splitting. The update rule fires and the corners $a = (0,0,0,0,0,0)$ and $b = (4,4,4,4,3,3)$ are added with labels $\lambda(a) = 0$ and $\lambda(b) = 3$. Further $a_1 > 2$ is chosen for a test of the node and the data set is divided between the two children. Following the left-depth-first strategy

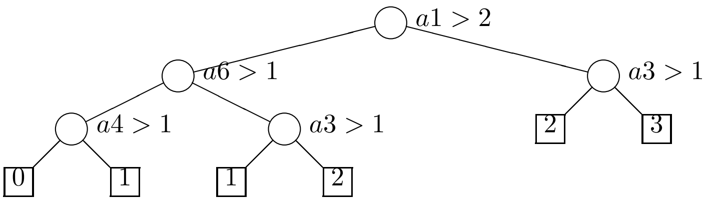

Figure 3.3: The MDT generated for the example data set

x	a_1	a_2	a_3	a_4	a_5	a_6	λ
1	0	0	1	1	0	1	0
2	1	1	2	1	3	1	0
3	0	1	1	0	0	1	0
4	2	3	1	3	3	1	1
5	1	0	2	2	3	1	1
6	0	0	0	3	2	2	1
7	2	2	1	1	1	2	1
8	2	4	2	2	2	3	2
9	1	1	2	1	3	2	2
10	3	2	1	0	0	1	2
11	3	2	2	1	2	2	3
12	3	3	4	1	2	2	3
13	4	2	3	3	3	3	3
14	3	3	3	4	1	3	3
15	4	4	2	3	0	1	3

Table 3.1: The example data set

we consider the left child for splitting. The corners are $a = (0,0,0,0,0,0)$ and $b = (2,4,4,4,3,3)$ where a is already present and b is added with a label $\lambda(b) = 2$. The algorithm continues with finding the best split, etc. The full monotone tree generated from this data set is given in figure 3.3.

Let $\mathcal{L}$ be the set of leaves of a tree $\mathcal{T}$ and $\mathcal{N}$ be the set of nodes of $\mathcal{T}$. We define a relation on $\mathcal{N}$: for $T, T' \in \mathcal{N}$, $T \leq T' \Leftrightarrow a(T) \leq b(T')$.

Note that a simple criterion for checking the monotonicity of a tree ([64]) can be defined as follows. Let $T, T' \in \mathcal{L}$ such that $T = \{x \in \mathcal{X} : a(T) \leq x \leq b(T)\}$ and $T' = \{x \in \mathcal{X} : a(T') \leq x \leq b(T')\}$. Then the tree is monotone if for any choice of T and T': $T \leq T' \Rightarrow \lambda(T) \leq \lambda(T')$.

3.3 Monotone Decision Trees from Noisy Data

Data noise is a problem that often occurs in practical applications of the classification algorithms and is extensively studied by many authors. The traditional definition of noise considers data points which do not agree with the underlying function because of wrong classification, incorrect/imprecise measurements, typing mistakes, etc. Such points can mislead the classification algorithm and cause the generation of an overly complicated and/or inaccurate classifier.

In monotone problems, however, noise will often manifest itself in a specific way that is not relevant for the general methods. The restriction of monotonicity of the data might be violated and data points can be inconsistent with each

```
update $D$ for $T$:
    $l_1 = \lambda_{\max}(a)$; $l_2 = \lambda_{\min}(b)$;
    if $a \in D$
      relabel $a$: $\lambda(a) = l_1$;
    else
      label $a$: $\lambda(a) = l_1$;
      add $a$ to $D$;
    if $b \in D$
      relabel $b$: $\lambda(b) = l_2$;
    else
      label $b$: $\lambda(b) = l_2$;
      add $b$ to $D$;
```

Figure 3.4: The new update rule

other, i.e., one point might dominate another on all attribute values but be classified in a lower class. We refer to this type of noise as *monotonicity noise*. The MDT algorithm requires strictly monotone data. In this section we propose an extension which will allow the generation of a monotone tree from any noisy data set.

When monotonicity noise occurs in the data, it appears as pairs of data points that are inconsistent with respect to monotonicity. For the MDT algorithm that results in tree nodes for which the lower left corner is assigned a higher label than the upper right corner. More precisely, let $T = \{x \in \mathcal{X} : a(T) \leq x \leq b(T)\}$ be the set (node) considered for splitting and let $\lambda(a)$ and $\lambda(b)$ be the labels of $a(T)$ and $b(T)$ respectively. Then it might occur that $\lambda(a) > \lambda(b)$.

In order to solve the problem we propose a simple extension of the update rule that not only grows the data set but also tries to repair the inconsistencies. The new update rule is given in figure 3.4. The procedure always relabels the corners with the consistent labels that are calculated from the rest of the data. This algorithm always generates a monotone tree.

Theorem 4 *The MDT algorithm of figure 3.1 with the update rule of figure 3.4 always generates a monotone tree.*

Proof: Let us assume that the generated tree is not monotone:

$$\exists T, T' \in \mathcal{L} : T \leq T' \text{ and } \lambda(T) > \lambda(T')$$

By assumption T and T' are homogeneous. Therefore $\lambda(T) = \lambda(a(T)) = \lambda(b(T))$ and $\lambda(T') = \lambda(a(T')) = \lambda(b(T'))$. This implies

$$\lambda(a(T)) > \lambda(b(T')). \tag{3.1}$$

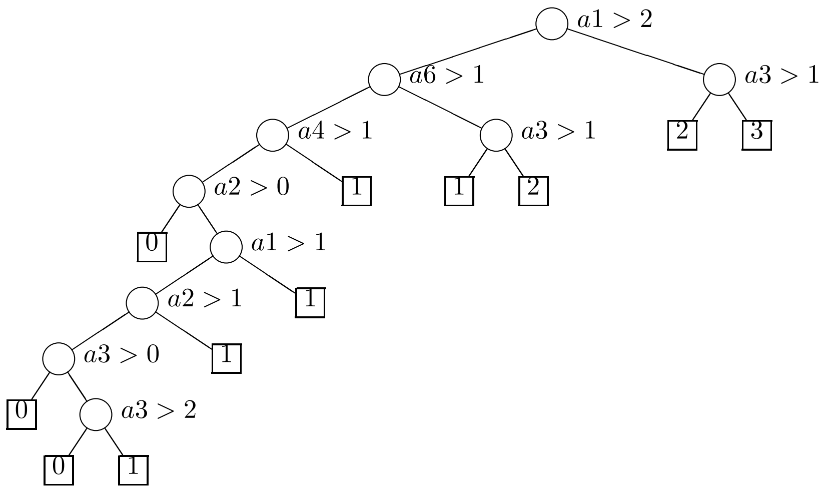

Figure 3.5: MDT on the non-monotone data set

The labels of the leaves are assigned as follows:

$$\text{at a moment } t \text{ we assign } \lambda(a(T)) = \lambda_{\max}(a),$$

$$\text{at a moment } t' \text{ we assign } \lambda(b(T')) = \lambda_{\min}(b).$$

Let $t < t'$. Since $T \leq T'$ then $a(T) \in \downarrow b(T') \cap D \neq \emptyset$

$$\Rightarrow \lambda_{\min}(b) \geq \lambda(a(T))$$

$$\Rightarrow \lambda(a(T)) \leq \lambda(b(T'))$$

which is a contradiction with condition 3.1. The case of $t' < t$ is analogous. $\square$

An interesting observation is that, after a leaf is created, all the points belonging to the leaf except the corners can be deleted from the data set since they will not be used further in the tree generation.

To illustrate the algorithm we introduce monotone inconsistency in the example data set of table 3.1 – we change the label of data point $x3$ from 0 to 1. Thus we introduce an inconsistent pair of data points ($x2$,$x3$). The output of the algorithm on the new data set is given in figure 3.5.

3.4 Pruning and Labelling Rules that Guarantee the Monotonicity

Pruning is a general tree simplification method that has proven to give good results both for reduction of the size of the tree and for reducing the overfitting of the particular data set. Methods for pruning will be a valuable addition to the monotone decision tree algorithms; however, the general approaches cannot be applied directly. The main problem is how to guarantee that, after pruning a node, the new leaf is given a label which will be consistent with the rest of the labels and that the resulting tree will still be monotone.

Moreover, this raises the more general question of labelling monotone decision trees. Given a tree generated by any classical algorithm, how do we (re)label the leaves in a consistent way so that the tree becomes monotone?

Section 3.4.1 is devoted to the monotone labelling problem. Section 3.4.2 investigates how the labelling approaches can be applied when simplification of the tree is necessary.

3.4.1 Labelling Rules

Two approaches to the problem of labelling are considered – dynamic and static labelling. *Dynamic labelling* is applied while generating the tree, as soon as a node is turned to a leaf and its label has to be chosen. Therefore it is assumed that in most of the cases a part of the tree is not yet generated, thus information about it is not available.

The *static labelling* approach on the other hand assumes that the whole tree is already generated but the leaves are not assigned labels (or their labels have to be ignored and re-assigned in order to generate a monotone tree). Therefore the assumption is that usually more than one leaf will not yet be labelled and for that part of the leaves only information about the corner labels is available.

The dynamic and the static approaches are presented in sub-sections 3.4.1 and 3.4.1 respectively.

Dynamic Labelling

One important difference between the static and the dynamic way of labelling is to what extent the information in the tree is already available. While in the static case the tree is grown and all the information about the shape of the tree, the corners of the leaves and the labels of these corners is available, in the dynamic case we only have a part of the tree built and the new label should be based on partial information.

Therefore, for dynamic labelling, an important factor is the search strategy used for building the tree, i.e., which part of the tree is expected to have been built already and what kind of information will influence the new labels. In the following we assume the *depth-first strategy* for growing the tree. First we note an observation that holds for this strategy.

Lemma 8 *Let $T, T' \in \mathcal{L}(\mathcal{T})$ in the monotone tree $\mathcal{T}$ generated with depth-first strategy. Let $T \leq T'$. Then leaf T is generated before leaf T'.*

```
label leaf $T$:
    $\lambda(T) = L(T)$;
    $\lambda(a(T)) = \lambda(T)$;
    $\lambda(b(T)) = \lambda(T)$;
```

Figure 3.6: The dynamic labelling rule

Proof: Let N be a node in $\mathcal{T}$ such that N is the least common ancestor of T and T'. Therefore $\exists i$ such that exactly one of the following is true:

$$\forall x \in T, \forall y \in T' : x(i) \leq y(i) \tag{3.2}$$

$$\forall x \in T, \forall y \in T' : x(i) > y(i) \tag{3.3}$$

Condition 3.3 contradicts the requirement $T \leq T'$. Therefore condition 3.2 is true and T belongs to the left branch of N while T' belongs to the right branch of N. Therefore, using the depth-first strategy, T will be generated before T'.$\square$

The general form of the dynamic labelling rule for a non-homogeneous leaf T is given in figure 3.6 where L is the labelling function. Two possible forms are proposed for the labelling function as follows:

$$L \in \{L_{\min}, L_{\max}\},$$

$$L_{\min}(T) = \max\{\lambda(a(T'))|T' \leq T\},$$
$$L_{\max}(T) = \min\{\lambda(b(T'))|T \leq T'\}.$$

Note that for $T' \not\equiv T$ in the above formulas, $a(T') = b(T') = \lambda(T')$.

Theorem 5 *Let $\mathcal{T}$ be a tree generated using the extended MDT update rule for a threshold of at least m points in a leaf, $m \geq 1$. Let the leaves be labelled using the dynamic labelling rule of figure 3.6 where one of the following strategies is applied:*

1. $L(T) = L_{\min}(T), \forall T \in \mathcal{L}$,
2. $L(T) = L_{\max}(T), \forall T \in \mathcal{L}$.

Then $\mathcal{T}$ is monotone.

Proof: Let the tree be generated using the left-depth-first strategy. The newly generated leaf to be labelled is denoted by T. If T is the first leaf then the current set of labelled leaves is monotone.

Let T be not the first leaf and let the current set of labelled leaves be monotone. We label $\lambda(T) = L(T)$.

Let $L = L_{\min}(T)$. Then

$$\lambda(T) = \max\{\lambda(a(T'))|T' \leq T\}.$$

Therefore, for each $T' \leq T$, $\lambda(T') \leq \lambda(T)$. Therefore the tree remains monotone. The proof is analogous for $L = L_{\max}$.

In the same way it can be proved for right-depth-first strategy. □

The following observations can help speed up the computation:

Observation 4 *For the left-depth-first strategy the following holds:*

$$L_{\max}(T) = \lambda(b(T)).$$

Observation 5 *For the right-depth-first strategy the following holds:*

$$L_{\min}(T) = \lambda(a(T)).$$

The experiments suggest that $L_{\max}(a)$ tends to favor the lower classes while $L_{\min}(b)$ tends to favor the higher classes. Therefore they provide a choice to the decision-maker for a more pessimistic against a more optimistic prediction.

Static Labelling

As mentioned before, static labelling is performed on the fully generated tree, where the information about all the leaves (except their labels) is already available.

The earlier defined relation over the nodes/leaves of a tree is not transitive. We define now a *transitive closure* relation, denoted by $\trianglelefteq$, in the following way. For $T', T'' \in \mathcal{N}$:

$$T' \trianglelefteq T'' \Leftrightarrow \exists T_1, T_2, \ldots, T_m \in \mathcal{N} \text{ such that } T' \leq T_1 \leq T_2 \leq \ldots \leq T_m \leq T''.$$

In our implementations we used the algorithm of Warshall [77] for computing the transitive closure.

We define $\Lambda_{\min}$ and $\Lambda_{\max}$ over the set of leaves $\mathcal{L}$ as follows:

$$\Lambda_{\min}(T) = \max\{\lambda(a(T_1))|T_1 \in \mathcal{L}, T_1 \trianglelefteq T\}$$

$$\Lambda_{\max}(T) = \min\{\lambda(b(T_2))|T_2 \in \mathcal{L}, T \trianglelefteq T_2\}$$

Then Λ is defined as: $\Lambda \in \{\Lambda_{\min}, \Lambda_{\max}\}$.

It can be shown that $\Lambda_{\min} \leq \Lambda_{\max}$ and they are *monotone labellings*, i.e., for leaves $T_1 \leq T_2$, $\Lambda_{\min}(T_1) \leq \Lambda_{\min}(T_2)$ and $\Lambda_{\max}(T_1) \leq \Lambda_{\max}(T_2)$.

Theorem 6 *Let $\mathcal{T}$ be a tree generated without labelling. Let the leaves be visited following either left-depth-first or right-depth-first order. The leaves are labelled using one of the following strategies:*

1. $\lambda(T) = \Lambda_{\min}(T), \forall T \in \mathcal{L}$,

2. $\lambda(T) = \Lambda_{\max}(T), \forall T \in \mathcal{L}$.

Then the resulting tree is monotone.

Proof: Let $\Lambda = \Lambda_{\min}$ and the order of visiting the leaves is left-depth-first. Let the last leaf labelled be T. If T is the first labelled leaf then the current set of labelled leaves is monotone.

Let T be not the first labelled leaf and let the current set of labelled leaves be monotone. The label of T is:

$$\lambda(T) = \Lambda_{\min}(T) = \max\{\lambda(a(T'))|T' \trianglelefteq T\}.$$

Let us assume that the resulting set of labelled leaves is no longer monotone. Using left-depth-first means that there are no labelled leaves T' such that $T \trianglelefteq T'$. Therefore $\exists T' \trianglelefteq T, \lambda(T') > \lambda(T)$. Since $\lambda(T) \geq \lambda(a(T'))$, then $\lambda(T') > \lambda(a(T'))$

$$\Rightarrow \exists T'' : T'' \trianglelefteq T', \lambda(T') = \lambda(a(T'))$$

$$\Rightarrow T'' \trianglelefteq T \Rightarrow \lambda(T) \geq \lambda(a(T'')) = \lambda(T')$$

which is a contradiction with the assumption.

The proof is analogous for $\Lambda_{\max}$ and for right-depth-first order. □

Comparison of the Two Labelling Approaches

In order to get more insight into the performance of the two presented approaches for labelling, experiments were conducted. The main goal was to compare the results from the four possible combinations of settings: dynamic labelling with $L_{\min}$, dynamic labelling with $L_{\max}$, static labelling with $\Lambda_{\min}$ and static labelling with $\Lambda_{\max}$.

The trees were generated using left-depth-first search strategy with pre-pruning at predefined thresholds for the minimal number of points in a node (see section 3.4.2). Therefore the size of the trees was almost always the same.

The results showed that:

- In general, the dynamic labelling produces trees with lower misclassification rate on unseen data. The reason for this lies most probably in the fact that the tree changes dynamically and it is possible to generate trees that have a different shape, while in the static case the shape of the tree is already fixed and the only feature that changes is the labels of the leaves.
- The difference between the two labelling functions for both approaches was not so clear-cut and no conclusion can be made yet on which one is preferable.

Details on the experimental setting and the results are given in section 3.7.

3.4.2 Pruning

When noise is present in the data set this creates difficulties for most of the classification algorithms, e.g. by causing areas in the data to become difficult to describe, and thus causes (sometimes substantial) increase in the complexity of the generated classifier.

For the MDT algorithm the same effect is present when monotonicity noise is introduced – the tree sometimes grows much larger and some isolated deep branches might be generated in order to describe the inconsistent areas in the data. This can be illustrated by the following example. We introduce inconsistency in the data set from table 3.1 by changing the label of $x8$ from 2 to 0 which results in one pair of inconsistent points ($x7$,$x8$). The monotone tree generated by the algorithm has 148 leaves and 288 data points in the updated data set.

Therefore we need methods for pruning the monotone tree in order to reduce the size of the tree and avoid the overfitting of the noisy areas in the data in such a way that we keep the monotonicity property of the tree.

This paper proposes methods for pruning within two main approaches – pre-pruning and post-pruning. *Pre-pruning* is a general approach for pruning while generating the tree by not growing branches which fail to satisfy a predefined criterion and turning them to leaves. Therefore pre-pruning modifies the stopping and the labelling rule of the algorithm. *Post-pruning* on the other hand first grows the full tree and then tries to cut branches from it while a predefined criterion is satisfied. It is therefore a post-processing step which requires two separate rules – for choosing a branch to cut and for labelling the new leaves.

Pre-pruning

One criterion often used in traditional pruning techniques for prematurely stopping the generation of a branch is a predefined threshold for the minimal number of points in a leaf. Splitting is not allowed if the number of points in any of the new leaves drops below the threshold, the current node is then turned to a leaf and assigned an appropriate label.

Pre-pruning is a typical application of dynamic labelling. Once a non-homogeneous node is turned to a leaf, the labelling rule of figure 3.6 is applied and the leaf (and its corners) is given a consistent label.

Static labelling can also be used with pre-pruning if the labelling is delayed until the whole tree is generated. The leaves are then visited in left-depth-first or right-depth-first order and assigned labels as in theorem 6.

To illustrate the algorithm we use the example from table 3.1 with the change described in section 3.4.2. Figure 3.7 shows the tree generated using pre-pruning with threshold of at least 4 points in a leaf, using left-depth-first and dynamic labelling with $L = L_{\max}$. The tree misclassifies 2 points from the original data set.

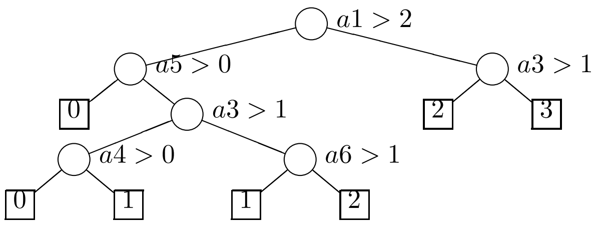

Figure 3.7: MDT generated with pre-pruning

In some cases both children-leaves of a node might be assigned the same label. In this case the node can be pruned without further increase in the misclassification rate.

Pre-pruning can be used together with the extension of the MDT algorithm for noisy data as well as with the original algorithm on monotone data.

Post-pruning

The general approach of post-pruning the already generated tree defines two additional rules – for choosing a branch to prune and for choosing a label for the new leaf. First we address the second problem taking into account the monotonicity property of the tree.

The approach we use for labelling is different from the ones already discussed in section 3.4.1. In the particular application, we need a local labelling strategy which assumes that the whole tree is fully generated and labelled except for the current leaf, which needs to be assigned a label.

Let $\mathcal{T}$ be a monotone decision tree. For a node $T = \{x \in \mathcal{X} : a(T) \leq x \leq b(T)\}$ we define a *consistency interval* $CI(T)$ where:

$$CI(T) = \begin{cases} [l_{\min}(T), l_{\max}(T)] & \text{if } l_{\min}(T) \leq l_{\max}(T) \\ \emptyset & \text{otherwise,} \end{cases}$$

$$l_{\min}(T) = \begin{cases} \max\{\lambda(T') : T' \in \mathcal{L}, T' \leq T\} & \text{if } \exists T' : T' \leq T \\ c_{\min} & \text{otherwise,} \end{cases}$$

$$l_{\max}(T) = \begin{cases} \min\{\lambda(T') : T' \in \mathcal{L}, T \leq T'\} & \text{if } \exists T' : T \leq T' \\ c_{\max} & \text{otherwise.} \end{cases}$$

If $CI(T) \neq \emptyset$ then any value in $CI(T)$ is a possible consistent label for T preserving the monotonicity property of the tree.

Theorem 7 *For a given monotone tree $\mathcal{T}$ and an arbitrary node T in $\mathcal{T}$, suppose that the children of T are pruned and that T is turned to a leaf. Let $CI(T) \neq \emptyset$. Then for any $l \in CI(T)$, l can be assigned as a label of T and the resulting tree remains monotone.*

Proof: Let us assume that the new tree is not monotone. Therefore there exists $T' \in \mathcal{L}$ such that one of the following occurs:

$$T' \leq T \text{ and } \lambda(T') > \lambda(T) \text{ or} \tag{3.4}$$

$$T \leq T' \text{ and } \lambda(T) > \lambda(T'). \tag{3.5}$$

Let condition 3.4 be the case. Since $T' \leq T$, we have $\lambda(T') \leq l_{\min}(T)$. But $\lambda(T) = l \geq l_{\min}(T)$, therefore:

$$\lambda(T') \leq l_{\min} \leq \lambda(T) < \lambda(T')$$

which is a contradiction.

The case of condition 3.5 is analogous. □

When the consistency interval contains only one point $l = l_{\min} = l_{\max}$, there is only one possibility for a consistent label of the pruned node. However, if $l_{\min} < l_{\max}$ then a choice has to be made about which point from the interval to assign. This choice is often domain dependent and reflects, for example, how optimistic or pessimistic the prediction is required to be.

The second open question with monotone pruning is the choice of a node to prune. It includes the order of visiting the nodes and the criterion for approval or rejection of the current candidate for pruning. We consider two search strategies for visiting the nodes. The first follows the depth-first order of visiting the nodes and tries to prune the current node if both its children are leaves. The second strategy iteratively tries to prune the frontier of the tree in depth-first order. On each iteration it tries to prune all nodes whose (both) children are leaves none of which has just be pruned. The loop terminates when the tree is traversed without pruning any node. Our experiments point out that the second strategy produces more balanced trees while the size of the trees is comparable to the size of the trees produced by the first strategy.

Once a candidate for pruning is reached the decision needs to be made whether to prune it or not. One logical criterion is the misclassification rate. The algorithm computes the new label and then checks whether the misclassification rate of the tree with the new leaf is below a predefined threshold for the percentage of misclassified data points. It is a general approach to use a separate pruning set for checking the accuracy of the tree.

To illustrate the post-pruning algorithm we use the same example. As noted before, the full tree contains 148 leaves. Figure 3.8 shows the pruned tree at misclassification threshold 25% and assigning label $l_{\max}$. For simplicity, no

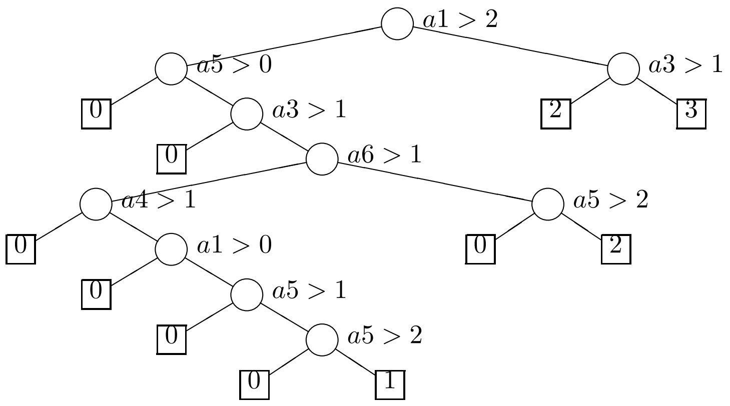

Figure 3.8: MDT generated with post-pruning

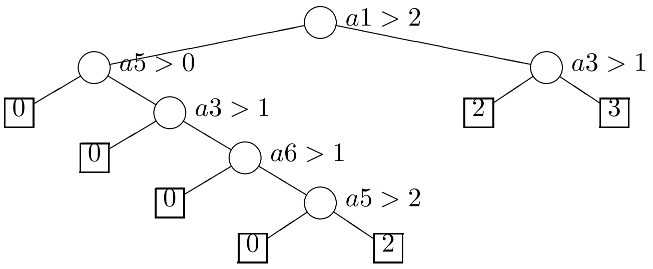

Figure 3.9: MDT generated with post-pruning

separate pruning set was used but the misclassification rate was checked on the original data set instead. The pruned tree misclassifies 3 points from the original data set. Figure 3.9 shows the tree pruned at threshold 30% and 4 misclassified points.

Again as with pre-pruning it might happen that both children of a node are assigned the same label – then again we can prune the node without increasing the misclassification rate.

Figure 3.10 illustrates the same algorithm with choosing $l_{\min}$ as the label of the new leaf. The tree is pruned at misclassification threshold 25% and 3 misclassified points. Figure 3.11 shows the tree at threshold 30% and 4 misclassified points.

The post-pruning algorithm can be applied as a post-processing step on any monotone tree generated with another algorithm as soon as the information about the leaf corners is available. It can also be used on a monotone tree

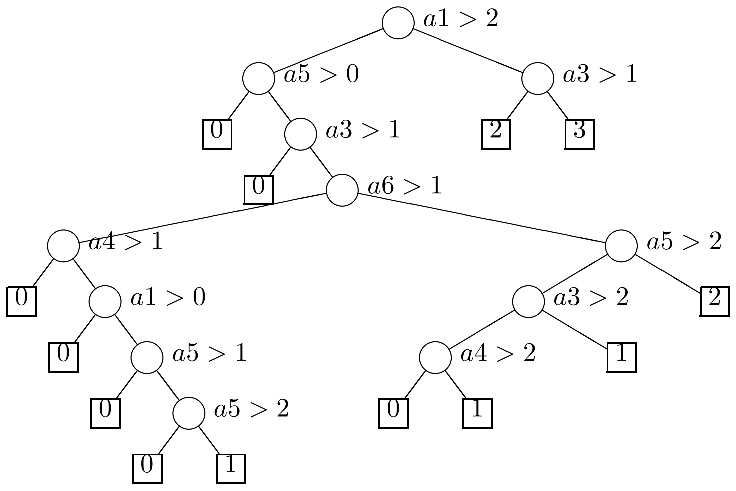

Figure 3.10: MDT generated with post-pruning

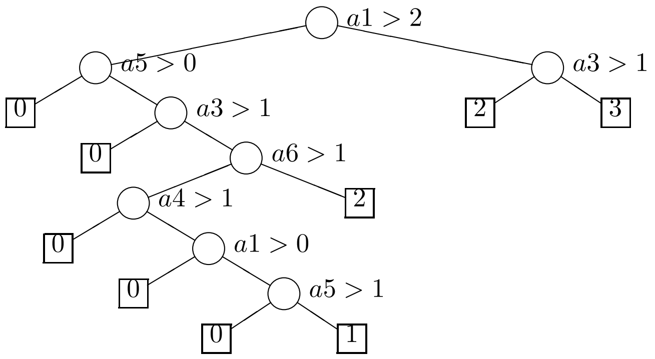

Figure 3.11: MDT generated with post-pruning

generated with the pre-pruning algorithm for further simplification of the tree.

Comparison of the Two Pruning Approaches

Experiments were conducted in order to compare and study the specifics of the two pruning approaches. Here the main observations from these experiments are presented.

It is theoretically clear that pre-pruning will generate smaller data sets and therefore consumes less resources than growing the whole tree and pruning it afterwards. This was confirmed to be a significant advantage of pre-pruning against post-pruning, especially when the noise affects more severely the generation process.

The relation between the number of inconsistent pairs of data points and the size of the tree is not straightforward – more important is the type of inconsistency that can confuse the tree generation. When the noise severely disturbs the tree generation it is possible for the data set and the tree size to grow exponentially. This is a known result for the original MDT algorithm for monotone data. The problem occurs more often with noisy data since it is originally inconsistent and can more easily confuse the generation process. However, using pre-pruning the problem can be easily overcome and manageable trees can be generated from any noisy data set.

On the other hand, for some of the data sets post-pruning seems to produce better results by pruning a large part of the tree with no change in the misclassification rate.

As was expected, for several data sets using smaller thresholds for pre- and post-pruning improves the accuracy of the tree by giving lower misclassification rate than the full tree. This is a result of the reduction in overfitting the particular samples due to the pruning and it is a known effect of tree pruning in general.

Details about the data used, the performed experiments and the experimental results are given in section 3.7.

3.5 Splitting Criteria for Monotone Decision Trees

One of the important rules in building a decision tree is the splitting rule which defines how to split the current node into two branches. For a binary tree, a split is a tuple of the type $\langle a_i, v_j \rangle$ where a_i is the attribute to split on and v_j is the cut-off value. A lot of research has been carried out on finding appropriate criteria for choosing good splits for the classical DT algorithms. One of the most successful approaches is to choose the split which produces the highest decrease in the entropy or the highest information gain. These two notions are usually defined as follows. The *entropy* of a node T is:

$$Ent(T) = -\sum_{i=1}^{n} p_i \, log_2 p_i$$

where p_i is the proportion of data points with class i in T for an n-class problem. Then the *information gain* of an attribute a_i and cut-off value v_j is:

$$Gain(T, a_i, v_j) = Ent(T) - \sum_{k \in \{L,R\}} \frac{|T_k|}{|T|} Ent(T_k)$$

where T_L/T_R are respectively the left and the right child of T when split on $\langle a_i, v_j \rangle$.

However, for the specific case of MDT it is not clear which splitting criteria are appropriate. Intuitively they should not only attempt to produce smaller trees but also provide fast decrease in the inconsistencies in the tree. One such criterion within the Multicriteria Decision Aid methodology was suggested in [30, 29] although no experimental results were presented on its performance compared to other criteria. The criterion aims at reducing the number of non-monotone pairs of points in the resulting branches. It chooses the split with the least number of inconsistencies/conflicts.

Let the current node T be split into the following non-overlapping subsets:

$$T' = \{a(T') \leq x \leq b(T')\}$$

and

$$T'' = \{a(T'') \leq x \leq b(T'')\}.$$

Let T' be the left branch. Therefore $T' \leq T''$. If for all points $x \in T', y \in T''$ it is true that $\lambda(x) \leq \lambda(y)$ then the split is monotone. There might be, however, points such that $\lambda(x) > \lambda(y)$. The number of those inconsistent pairs is counted for all possible splits of the current node and the one with the lowest count is chosen.

The experiments that were conducted for this section aim at giving more insight into the performance of the two mentioned criteria in the context of MDT. The two main aspects for comparison are the size and the accuracy of the generated trees. The experimental results point at the following observations:

- On monotone data sets no criterion is systematically better than the other.
- With the increase of monotonicity noise in the data, the entropy criterion tends to generate smaller trees.
- With the increase of monotonicity noise in the data the conflicts criterion generates more accurate trees with lower misclassification rate on unseen data.

Intuitively the reason for the difference is probably in the orientation of the two criteria. The entropy criterion strives at generating smaller trees by reducing the diversity of classes in the leaves as quickly as possible. The conflicts criterion, on the other hand, only cares about the consistency/monotonicity of the tree and therefore produces trees that are larger but better fit the "character" of the data. In this way it handles monotonicity noise more successfully but at the cost of generating bigger trees.

Details about the data used, the performed experiments and the experimental results are given in section 3.7.

3.6 Missing Values in Monotone Data Sets

Missing values is a known problem in knowledge discovery. It comes as a result of human mistakes and omissions but also when data for certain attributes is difficult, expensive or even impossible to get. A special case is when for a certain object this attribute is not relevant, for example, in the records of a hospital the attribute "has given birth" is only relevant for female patients. This specific case requires different methods, such as reorganization of the data or specific classification algorithms, that can handle such data. In this section we only consider the former type of missing values in the context of a monotone classification problem.

We assume that the missing values are only found among the condition attributes. If a value of the decision attribute is missing then this object cannot contribute much information for the classification. If however a value of a condition attribute is missing, the rest of the values, together with the decision attribute, can still give important information. For this reason, the obvious solution to simply ignore all objects with missing values is usually not the best solution. The results are especially bad if the percentage of missing values is high.

A number of approaches for handling unknown values are available in the literature, see [68, 54, 43] for surveys and experimental comparison of the methods. However the only general approach that can be applied on monotone problems without introducing inconsistencies is to discard the objects with missing values. We propose here a simple extension to a number of preprocessing methods which guarantees that the resulting data set will still be monotone.

Let x be a data point with a missing value in attribute $a_i \in C$. By $x|A$ we denote the vector consisting of all values in x for the attributes in $A \subset C$. We define the relation $[\leq]$ on the set of examples as follows. Let x have known values for attributes in A and missing values for all the rest in $C\backslash A$ and let y have values for the attributes in B and missing values for $C\backslash B$. Then we define the relation as:

$$x\,[\leq]\,y \text{ if } x|A \cap B \leq y|A \cap B.$$

Notation: In the following, $\lfloor D \rfloor$ will denote the subset of objects in D that are fully defined, i.e. contain no missing values.

We define, for an object x and a missing value in x for attribute a, the following interval $[v_{\min}(x|a), v_{\max}(x|a)]$ (which we call *possible values interval*) where:

$$v_{\max}(x|a) = \begin{cases} \min\{(y|a-1): \ y \in \lfloor D \rfloor, \ y \,[\leq]\, x, \ d(y) > d(x)\} & \text{if such } y \text{ exists} \\ a_{\max} & \text{otherwise.} \end{cases}$$

$$v_{\min}(x|a) = \begin{cases} \max\{(y|a+1): \ y \in \lfloor D \rfloor, \ x \,[\leq]\, y, \ d(x) > d(y)\} & \text{if such } y \text{ exists} \\ a_{\min} & \text{otherwise.} \end{cases}$$

Here $a_{\min}$ and $a_{\max}$ refer to the minimal and the maximal possible value respectively for attribute a.

Note that the possible values interval might contain no values in the case when $v_{\min}(x|a) > v_{\max}(x|a)$. This is a clear sign of noise in the data. The following theorem explains why.

Theorem 8 *Let D be a monotone data set described by condition attributes C without missing values. Then for each object $x \in D$ and each attribute $a \in C$ the possible values interval $[v_{\min}(x|a), v_{\max}(x|a)]$ is non-empty.*

Proof: Let the right side of the interval be extracted from the object y: $v_{\max} = y|a - 1$ where $y|C\backslash\{a\} \leq x|C\backslash\{a\}$ and $d(y) > d(x)$.

Let $x|a > v_{\max}$. Then $x \geq y$ but $d(x) < d(y)$ therefore the D is not monotone, which is a contradiction to the conditions.

Similarly let $v_{\min} = z|a + 1$. If $x|a < v_{\min}$ then if follows that $x \leq z$ but $d(x) > d(z)$, which is a contradiction to the monotonicity of D.

Therefore $x|a \in [v_{\min}(x|a), v_{\max}(x|a)] \neq \emptyset$. $\square$

Taking this result into account, a possible method to deal with cases when the interval is empty is to ignore the object. If however the interval is not empty, then any value from it is a valid assignment for x which preserves the monotonicity. This is expressed in the following theorem.

Theorem 9 *Let D be a data set with missing values such that $\lfloor D \rfloor$ is monotone. Let $x \in D$ contain one missing value in attribute a. Let us assume that $\exists v \in [v_{\min}(x|a), v_{\max}(x|a)]$. Then $\lfloor D \rfloor \cup x$ is monotone.*

Proof: Let us assume that $\lfloor D \rfloor \cup x$ is not monotone. Let the inconsistent pair be $y \leq x$ such that $d(y) > d(x)$. Therefore $x|a \geq y|a > v_{\max} \geq x|a$, which is a contradiction.

Similarly the assumption that $x \leq y$ while $d(x) > d(y)$ will lead us to a contradiction. $\square$

Note that in the theorem, x is allowed to have only one missing value. The case of more missing values is more complicated since they depend on each other and cannot be considered separately. In the following we assume that no object has more than one unknown value.

The general algorithm proposed here goes through the following steps for each object with a missing value:

1. Compute the possible values interval taking into account the fully defined objects in the data set.
2. If $[v_{\min}, v_{\max}] = \emptyset$ then discard the object x.
3. If $v_{\min} = v_{\max}$ then the interval contains only one value. Assign that value to $x|a$.
4. If $v_{\min} < v_{\max}$ then apply procedure **fill-value**($v_{\min}, v_{\max}$).

The procedure **fill-value** which appears in the algorithm depends on the chosen general method for filling in the missing values. Not all available approaches are applicable since we want to restrict the possible values to the interval we have calculated and not all methods can accommodate this restriction. We consider here three candidates from the literature: most frequent value, k-nearest neighbour and vector multiplication.

The most frequent value method is very simple – choose the most frequent value for this attribute. The CN2 algorithm uses that idea (see [31]). A refinement of the approach is the maximum relative frequency method proposed in [50] which assigns the most common value within the decision class. For our purpose we only consider the set of values included in the possible values interval and choose the most common of them within the decision class.

The second method applies a version of the k-nearest neighbour algorithm [32] to choose one value for the attribute. It applies some relevant distance measure to extract from the data set the k objects that are closest to the object x. Then it chooses the best (for example the most frequent) value for the missing attribute. In the monotone case we are only interested in neighbours with values within the computed interval. This restriction can easily be incorporated in the original algorithm.

The third method adds new objects to the data set – x is multiplied by assigning each of the possible values of the missing attribute [42]. By only considering the values from the computed interval, this approach can also be used in the monotone case. A refinement of the approach is to restrict the set to only those values that appear within the decision class.

Applying the above presented algorithm with one of the three methods discussed for the **fill-value** procedure, results in a monotone data set, as long as the initial fully defined data set $\lfloor D \rfloor$ is monotone.

Obviously the end result will depend on the order in which we visit the objects and the attributes (because we use the already repaired objects for the calculation of the new intervals). One possible approach to finding a suitable order is to choose at each step the case with the smallest interval of possible values. Therefore we have to update all intervals after each change. This, however, does not add a lot of overhead since only one or a very limited number of objects change at each step. On the other hand, the intervals cannot get larger after a change because no objects in $\lfloor D \rfloor$ are removed. They might, however, get smaller, which might in some cases speed up the algorithm a little (if for example the interval is reduced to only one value).

3.7 Experiments

In the following sub-sections we describe the experiments on the different methods presented in this chapter. Three data sets were used which will be discussed first.

The Nursery data set was obtained from UCI Machine Learning Repository [19]. It is a real-world monotone data set which represents applications for a nursery school, contains 12960 instances described by 8 attributes and covers the whole input space.

The Cars data set was also obtained from UCI Machine Learning Repository [19] and is an artificial set describing cars by their properties and classifying them according to their acceptability for a buyer. The original data was not strictly monotone and for that reason one of the values of one attribute was removed. The resulting set was monotone. It contains 1153 instances described by 6 attributes.

Both data sets are described in more details in Appendix A.

For the experiments random samples of size 200 points were drawn from both data sets. Monotone inconsistencies were introduced in the data in the following way: a pair of comparable data points (such that either $x \leq y$ or $x \geq y$) from different classes was chosen at random and the labels were switched. This results in one or more inconsistent pairs. The procedure can be repeated to introduce more noise. Since both data sets cover the whole input space, they were also used in the experiments as test sets for the misclassification.

The third data set used in the experiments is discussed in [40, 74]. The sample consists of 39 objects representing firms that are described by 12 financial parameters. To each company a decision value is assigned – the expert evaluation of its category of risk. Since this data set is very small it was only used in one of the experiments.

3.7.1 Experiments on the Dynamic and Static Labelling Approaches

For the comparison of the two labelling approaches, four samples were used – one from the Cars data set and three from the Nursery data. For each sample, the algorithms were applied for the thresholds of minimum 5, 10, 15 and 20 points in a node. For the static case the leaves were left unlabelled and the labelling was performed at the end. Eight different combinations of settings were tested: entropy versus conflicts splitting criteria, $L_{\min}$ versus $L_{\max}$ and $\Lambda_{\min}$ versus $\Lambda_{\max}$. The results about the splitting criteria are discussed in section 3.7.3 and for the purpose of the current discussion the results will the averaged over the two criteria.

As test sets, the full data sets were used as they cover the whole input space (1153 points for the Cars data set and 12960 points for the Nursery data set). The number of misclassified points per sample is given in table 3.2. The first column contains the value of the threshold. Columns 2 and 3 give the results for the tree labelled with static labelling for $\Lambda_{\min}$ and $\Lambda_{\max}$ respectively. Columns 4 and 5 give the analogous results for dynamic labelling and $L_{\min}$ and $L_{\max}$ respectively.

As mentioned above, the size of the trees was practically the same for the static and the dynamic case (per sample and threshold). Bearing this in mind, it can be seen that the number of points misclassified by the trees which were labelled dynamically is systematically lower than (often more than twice as low as) that of the trees labelled statically.

On the other hand, no definite answer can be given to the question of which labelling function to use: $L_{\min}$ versus $L_{\max}$ and $\Lambda_{\min}$ versus $\Lambda_{\max}$. In the static case the choice of labelling function made hardly any difference in the results, while in the dynamic case for some samples $L_{\min}$ is better and for the other $L_{\max}$ gives better results.

3.7.2 Experiments on the Pre-pruning and Post-pruning Approaches

In order to compare and study the specifics of the two pruning approaches presented in the chapter, experiments were conducted using a sample from the Nursery data set and the bankruptcy data set. For each sample, 3 noisy sets are generated by switching the labels of 1, 2 or 3 pairs of comparable points. The new data sets are used to build the full MDT, the pre-pruned MDTs with varied threshold of 2 to 5 points in a node and the post-pruned trees with varied misclassification rate threshold of 5% to 20%. Each tree is represented by the following indicators: number of points in the updated data set(u), number of nodes (n), number of leaves (l), average depth (d), number of misclassified points on the original data (o), and on the separate test set (t).

	static		dynamic	
thr	$\Lambda_{\min}$	$\Lambda_{\max}$	$L_{\min}$	$L_{\max}$
5	205	205	93	285
10	350	350	156	407
15	408	408	217	425
20	408	408	219	434
5	2495	2495	1708	1915
10	4012	4012	2172	2154
15	4530	3524	2395	2292
20	4950	4950	2512	2424
5	2436	2436	1821	1694
10	3380	3380	2486	1892
15	4653	4653	2587	2404
20	5055	5055	2967	2404
5	4152	4152	2097	3351
10	5402	5402	2489	3509
15	6892	6892	2983	4219
20	7398	7403	3130	4555

Table 3.2: Experimental data on the labelling approaches

The results for the Bankruptcy data set are given in table 3.3, where the results from the monotone data set are presented in column 2. The three generated noisy data sets are given in the rest of table 3.3. Since the original set is very small, no separate test data set is used. The last data set took too long to generate the full tree due to exponential growth of the updated data set. However, the pre-pruning algorithm generates manageable trees (even for a threshold of 2 points) which are represented in the table.

The results for the Nursery data set sample are presented in a similar way in table 3.4.

Further experiments were performed on two of the series from table 3.6. These data sets are samples from the Nursery data set with added noise and they are described in section 3.7.3. The goal was to see how the size and the accuracy of the trees change on the full range of pre-pruning and post-pruning thresholds. It is also possible to see how the chart changes with the addition of more noise.

The trees were built using left-depth-first strategy and the entropy splitting criterion. The labelling function for pre-pruning was $L_{\min}$ and for post-pruning $\Lambda_{\min}$. Similar results, however, were observed for $L_{\max}$ and $\Lambda_{\max}$.

In this section only the charts for the monotone sample are included. The rest of the results are presented in appendix B. Figure 3.12 shows how the size of the tree (the number of nodes) changes with the increase of the pre-pruning

	mon	full	pre2	pre3	pre4	pre5	po5	po7	po10	po15	po20
u)	50	91	91	90	75	73	91	91	91	91	91
n)	11	53	53	51	35	33	11	9	7	7	7
l)	6	27	27	26	18	17	6	5	4	4	4
d)	2	13	13	12	8	8	2	2	2	2	2
o)	0	1	1	1	1	2	1	2	3	3	3
u)		123	123	121	104	83	123	123	123	123	123
n)		87	87	83	65	43	87	45	21	17	17
l)		44	44	42	33	22	44	23	11	9	9
d)		12	12	12	9	7	12	9	4	4	3
o)		2	2	2	2	3	2	2	3	5	7
u)		*	195	180	121	100	*	*	*	*	*
n)		*	163	145	83	61	*	*	*	*	*
l)		*	82	73	42	31	*	*	*	*	*
d)		*	14	14	9	8	*	*	*	*	*
o)		*	4	5	5	6	*	*	*	*	*

Table 3.3: Experimental data for the bankruptcy data set

	mon	full	pre2	pre3	pre4	pre5	po5	po7	po10	po15	po20
u)	482	598	459	353	311	283	598	598	598	598	598
n)	321	471	297	161	111	83	143	75	73	25	17
l)	161	236	149	81	56	42	72	38	37	13	9
d)	10	10	10	8	7	7	9	7	7	4	4
o)	0	1	35	38	39	40	9	13	19	29	36
t)	17	16	41	44	45	44	24	27	31	36	47
u)		592	410	341	295	275	592	592	592	592	592
n)		493	239	149	95	75	35	31	29	37	21
l)		247	120	75	48	38	18	16	15	19	11
d)		11	9	8	7	6	5	5	5	5	5
o)		1	27	34	36	39	9	13	19	29	37
t)		26	38	43	43	44	24	34	38	46	48
u)		2795	576	440	377	328	2795	2795	2795	2795	2795
n)		3449	419	255	179	129	3449	3449	3449	45	13
l)		1725	210	128	90	65	1725	1725	1725	23	7
d)		14	10	9	8	7	14	14	14	6	3
o)		24	16	28	41	47	24	24	24	28	37
t)		40	29	37	43	52	40	40	40	39	37

Table 3.4: Experimental data for the Nursery data set

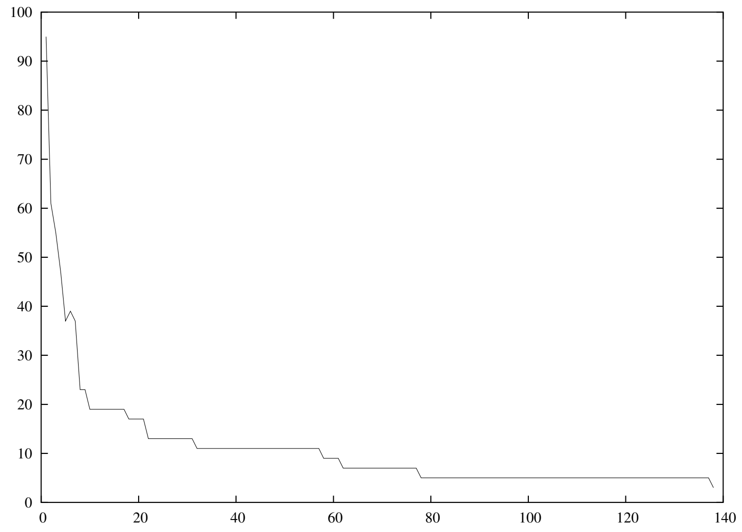

Figure 3.12: The number of nodes for the full range of pre-pruning thresholds

threshold. Figure 3.13 shows the change in accuracy (number of misclassified points on the full data set) with the increase of the pre-pruning threshold. Figure 3.14 shows the size of the tree for the whole range of post-pruning thresholds. Figure 3.15 shows the accuracy for all post-pruning thresholds.

From these charts and from the ones included in appendix B it can be seen that with the increase of noise the curve for the number of nodes becomes steeper both for pre-pruning and for post-pruning.

The flat area which appears at the beginning of the charts for post-pruning are due to the initial number of misclassified points of the full tree. The lowest thresholds for those points are lower than that number and the pruning does not start, therefore the observed values are for the full tree. This flat area gets larger when the noise in the data increases since then the tree becomes generally more inaccurate.

An interesting observation is that with the increase of noise, pruning at low thresholds improves the accuracy of the tree. This becomes more apparent as the noise increases and is better seen in pre-pruning.

3.7.3 Experiments on the Comparison of the Entropy and the Conflicts Splitting Criteria

These experiments were designed to answer two main questions in the comparison of the two splitting criteria:

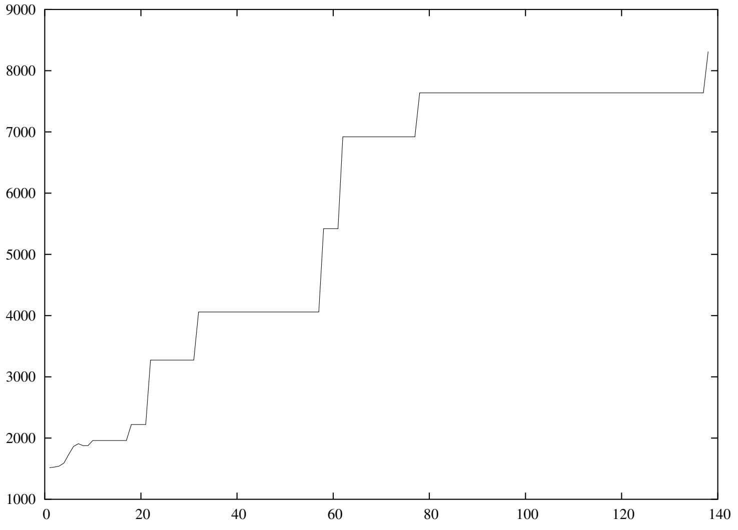

Figure 3.13: The number of misclassified points for the full range of pre-pruning thresholds

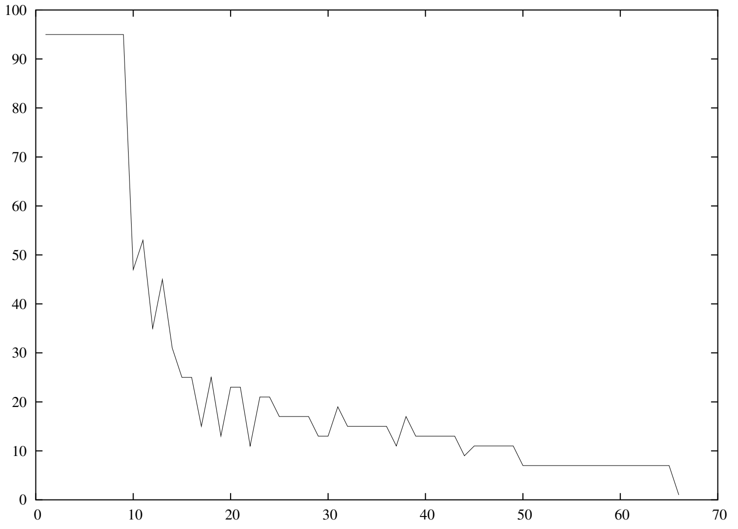

Figure 3.14: The number of nodes for the full range of post-pruning thresholds

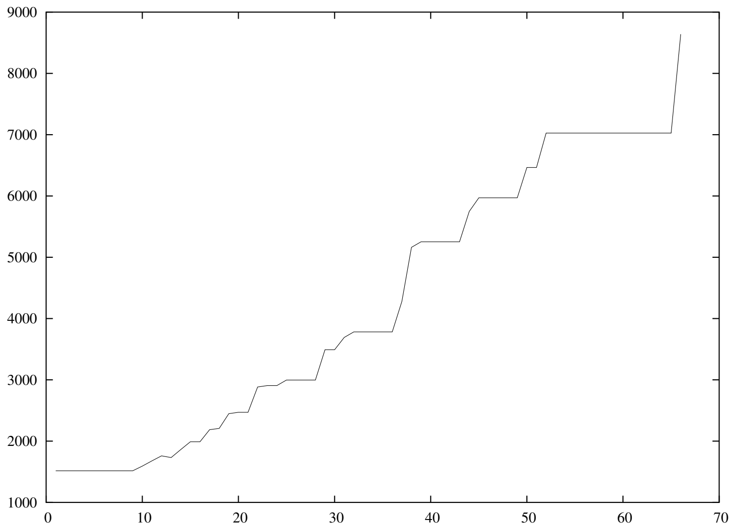

Figure 3.15: The number of misclassified points for the full range of post-pruning thresholds

- Which criterion performs better on monotone data by means of generating smaller and/or more accurate monotone trees?
- Which criterion handles better monotonicity noise (monotone inconsistencies) by means of generating smaller and/or more accurate monotone trees?

In order to give some insight into these questions the experiments were conducted in two different settings. In the first part 10 monotone samples were drawn from the Nursery data set. Monotone decision trees were generated from each of them using the two criteria. The results – the number of misclassified points over the full data sets and the number of tree nodes – are given in table 3.5. It can be seen that the performance of the two criteria is different on the different samples and no definite conclusion can be made on which one fits monotone problems better.

For the second part of the experiments, 4 samples were chosen – 1 from the Cars data set and 3 from the Nursery data set (rows 2, 3 and 7 from table 3.5). For each data set, 6 noisy versions (5 for the Cars data) were generated by switching the labels of 1 to 6 pairs of points. For sample 3 (row 3 from table 3.5) the procedure was repeated 4 times generating 4 different series of noisy data sets. For each of the generated 7 series, MDTs were built using the entropy and the conflicts criteria.

entropy		num conflicts	
miscl	nodes	miscl	nodes
1612	321	1512	233
1516	95	1522	211
1520	87	1752	131
1645	85	1458	247
1478	143	1330	159
1354	151	1429	783
1277	163	1718	195
1266	107	1232	149
1087	401	1672	241
1504	343	1622	327

Table 3.5: Experimental data on the splitting criteria for monotone samples

The results per series are presented in table 3.6. The first column contains the number of non-monotone pairs of points in the set. Columns 2 and 3 give the number of misclassified points on the full data set and the number of nodes for the tree generated with the entropy criterion, while columns 4 and 5 give the respective information for the number of conflicts criterion.

It can be seen that with the increase of inconsistencies the number of conflicts criterion gives systematically better misclassification rates while producing bigger trees than the entropy criterion.

3.8 Conclusions

This chapter is a contribution to the area of decision trees building in the context of classification for monotone problems. The starting point is the Monotone Decision Trees algorithm already available in the literature.

The original algorithm was extended in order to handle noisy data containing monotone inconsistencies. This allows monotone decision trees to be generated from any data set.

Furthermore, a number of additional methods were provided for pruning both in the direction of pre-pruning and post-pruning. The problem of pruning raised the question of finding consistent labelling strategies that guarantee the monotonicity of the resulting tree. Several labelling functions were proposed for this within the two approaches of dynamic labelling and static labelling. These methods can easily be applied in the context of pruning but are nevertheless separate methods that can be used in different contexts as well.

The pruning and labelling methods were tested experimentally in order to give more insight into their performance and how they compare with each other.

Another question considered in this chapter is the choice of a splitting crite-

incons	entropy		num conflicts	
pairs	miscl	nodes	miscl	nodes
0	52	53	47	83
19	90	293	74	171
55	196	311	103	373
86	218	421	172	501
111	225	469	176	521
118	224	473	186	609
0	1516	95	1522	211
2	1500	115	1520	303
21	2326	561	1789	397
28	2496	593	1773	583
31	2366	605	1737	589
40	2432	963	1763	639
42	2586	1787	1831	1069
0	1516	95	1522	211
31	2338	1071	2148	3303
71	2498	1581	2217	4447
91	2965	1887	2558	2969
109	3866	2987	2655	3137
128	4238	3299	2857	3975
129	4226	3309	2831	3873
0	1516	95	1522	211
11	1610	107	1741	227
22	1928	679	1817	1077
33	2084	699	1847	1069
42	2193	875	2006	1077
50	2678	1375	2273	2141
59	2575	1733	2273	2162
0	1516	95	1522	211
24	1528	77	1538	181
34	2054	361	1746	1113
36	2041	369	1758	1131
92	4035	3335	3287	8751
101	4250	4653	4306	8141
124	4467	5603	4740	9771
0	1520	87	1752	131
2	1512	157	1812	125
7	1673	747	1850	317
39	3052	3655	2692	3497
64	3607	3719	3003	3829
66	3685	3353	3327	4339
69	3798	3751	3322	4437
0	1277	163	1718	195
39	3035	2093	3607	3945
59	3518	3755	4283	5645
74	4577	2969	4369	5737
78	4603	3137	4553	5863
81	4547	3125	4504	5887
83	4547	3135	4466	5861

Table 3.6: Experimental data on the splitting criteria on non-monotone samples

rion in building monotone decision trees. Two criteria from the literature were discussed – the entropy criterion as the most popular general approach used in decision trees and the conflicts criterion designed specifically for the monotone context. Those were compared experimentally to determine which one gives better results. The figures showed that the entropy criterion produces smaller trees while the conflicts criterion generates more accurate trees.

The chapter also looks at the more general problem of missing values in monotone data sets. The monotonicity constraint poses additional restrictions to the process of filling in these values. A general scheme was proposed which can be applied as a preprocessing step for filling in unknown values in monotone data sets in such a way that the resulting data set is guaranteed to be monotone.

A number of directions for future research are still open. For example, it might be possible to construct more "moderate" labelling functions which are not necessarily either maximal or minimal.

The pruning methods can be refined as well, for example to accommodate a better pruning criterion than just the misclassification rate. One possibility is to incorporate a way to take into account the degree of how wrong the prediction is. Intuitively a prediction that differs slightly from the real value is better than such that differs a lot from it and the latter should be penalized more severely.

A different approach for generating monotone decision trees was recently proposed in [65] and proper comparison between the two algorithms will be beneficial in order to give insight into their performance and how they differ in practical applications. Another new approach for pruning monotone decision trees was presented in [38] and an experimental comparison might give interesting results.

The presented method for filling in missing values can be further refined and experiments should be conducted in order to compare how well the three different methods predict the missing values. Other aspects of data preprocessing in the contexts of monotone data sets might be interesting as well, such as, for example, noise reduction and inconsistencies removal.

Chapter 4

Monotone Decomposition

4.1 Introduction

One of the most effective approaches for solving a complex problem is by splitting it into smaller subproblems that can be solved (relatively) separately. The gains from such an approach are many, both for the human and the computational aspect. Complex problems are often impossible for humans to comprehend and solve at once, while smaller subproblems would be easier to grasp. Big problems usually require a lot of computational power to solve at once and it might be the case that such an amount is not available. Then by splitting the problem we make it possible to solve within the available computational resources. On the other hand if the runtime for finding the solution is the critical resource, then we can process the subproblems in parallel and achieve better performance.

Problem decomposition approaches are used in many areas of science. Examples of applications can be given from switching theory, game theory, reliability theory, machine learning. One of the applications in machine learning is in structured induction which aims at splitting a concept to be learnt in a hierarchy of sub-concepts which can be used separately to generate classification rules. The methods vary in their way of building the concept hierarchy but the majority of them involve a human expert who provides domain knowledge of the structure of the problem. This process can take a long time and a lot of energy while the availability of the expert is not necessarily guaranteed. Therefore the development of tools to assist the process or completely automate it might be highly beneficial.

The contribution of this chapter is within the research in automating the decomposition process. We concentrate on the problem of classification for monotone data sets and aim at building a decomposition hierarchy that preserves the monotonicity property with the ultimate goal of generating a monotone classifier. The research was previously published in [63].

The chapter is organized as follows. Section 4.2 reviews related research in the area. Section 4.3 presents the main results of our research. Section 4.4 gives the conclusions and some directions for future research.

4.2 Function Decomposition

The research presented in this chapter is closely related to the methods presented by Zupan et al. in [81, 82] and implemented in the system HINT. We first give a short overview of those methods and then discuss other related research.

4.2.1 The Function Decomposition Methods Introduced by Zupan et al.

The methods developed in [81, 82] aim at building a hierarchical structure based on the attribute set describing the data points. A distinct feature of the approach is that no predefined intermediate functions or operators are used. Instead the sub-concepts are induced from the data.

The goal at each step is to decompose a function $y = f(X)$ into $y = g(A, h(B))$ where X is a set of input attributes $X = \{x_1, x_2, \ldots, x_n\}$ and y is the class label variable. f, g and h are partially defined discrete functions defined by examples while A and B are disjoint subsets of the input attributes such that $A \cup B = X$. To find such a decomposition corresponds to finding a new intermediate concept $c = h(B)$.

As mentioned above, the functions g and h are not predefined in the application of the method and are induced during the decomposition process. The requirement for them is to have joint complexity which is lower than the complexity of f and that is determined using some complexity measure. By applying the method recursively on the two new functions h and g we can generate a hierarchy of concepts. Figure 4.1 (taken from [82]) shows one step of the building of such hierarchy.

The structure shown in figure 4.2 gives an example of a concept hierarchy. The names of the intermediate concepts would not be given by the decomposition method – those are just interpretations of the results given in order to make the structure clearer.

The decomposition method is organized in the following steps: basic function decomposition step, attribute partition selection and overall function decomposition. They are described as follows:

– The attribute partition selection step determines which is the best partition of the attribute set X into A and B. The criterion for selecting the best one out of the set of all possible partitions is to minimize some predefined complexity measure.

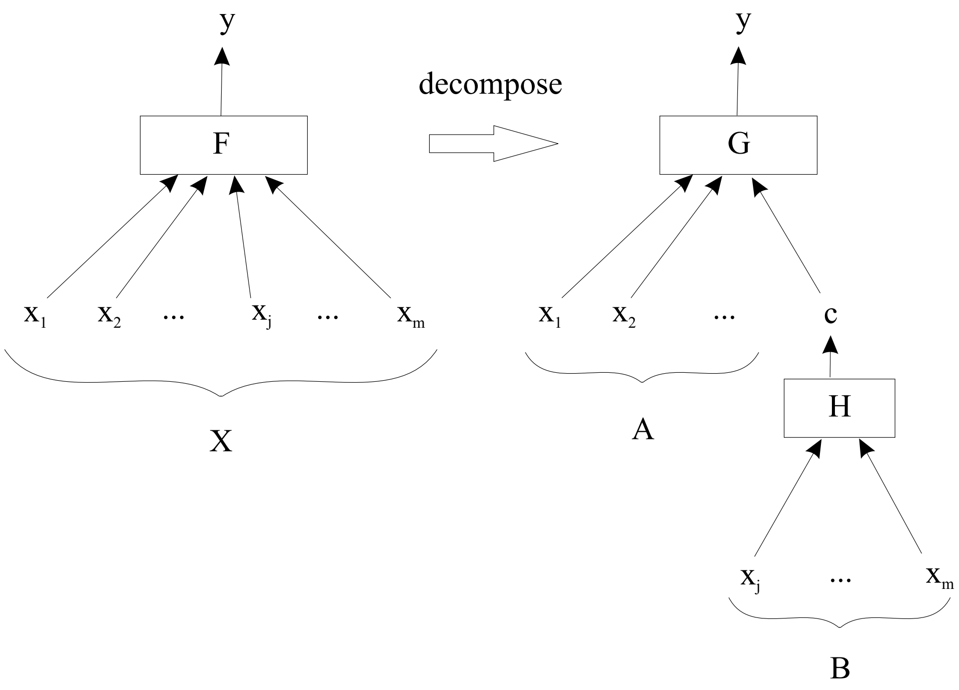

Figure 4.1: The decomposition step

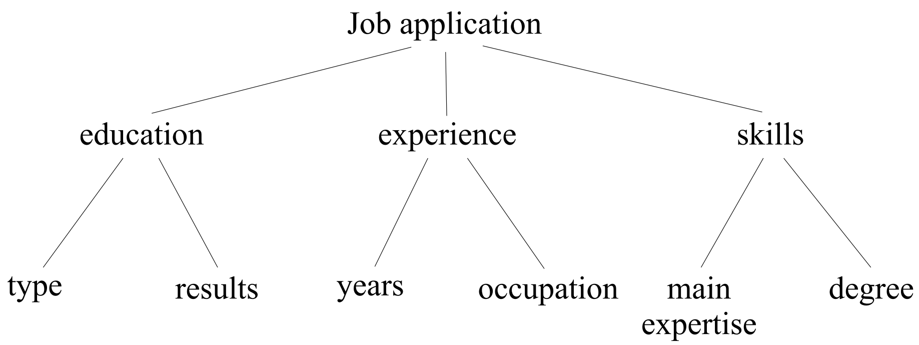

Figure 4.2: A concept hierarchy

– The basic decomposition step takes as an input the partition of the attributes in sets A and B and finds the corresponding functions g and h such that $y = g(A, c)$ and $c = h(B)$.

– The overall function decomposition step is the recursive application of the above two steps for the current g and h functions. The recursion stops if no decomposition can be found such that the resulting functions are less complex than the function being decomposed.

In its general form the algorithm will suffer from its time complexity. A number of heuristics can be applied in order to reduce the problem, for example by limiting the size of B.

The method considers only disjoint partitions $A \cap B = \emptyset$ which limits the shape of the discovered hierarchies to hierarchical trees. Note that those trees are different from the decision trees generated by methods such as C4.5 and CART. The nodes of the hierarchical trees correspond to sets of attributes forming a new concept while the nodes of the decision trees correspond to sets of data points grouped by the attribute-value tests present in the branch leading to the node.

The decomposition algorithm achieves higher generalization than the original data set due to the basic decomposition step. The construction of the sets defining the new functions g and h might lead to adding new points previously not present in the data set. That however does not necessarily extend to a cover of the whole input space. In the experiments reported by the authors a default rule was used in order to be able to classify the uncovered points from the input space. This rule assigns the value of the most frequently used class in the example set that defines the intermediate concept.

Let us now concentrate on the basic decomposition step of the algorithm. It starts by constructing the so called *partition matrix.* The rows of this matrix correspond to the distinct combinations of values of the attributes in A and similarly the columns correspond to the distinct values of the attributes in B. The entries of the matrix contain the class label for the specific combination of A and B values from the row and the column. Obviously some of those will be empty and considered as “don’t care”.

As an example of a partition matrix we consider the data set from table 4.2. Let us choose $A = \{a_1, a_2, a_3, a_6\}$ and $B = \{a_4, a_5\}$. The corresponding partition matrix is given in table 4.1.

Two columns of the matrix are called *compatible* if they don’t contradict each other or, more precisely, if they don’t contain entries for the same row that are non-empty and are labelled with a different class label. The number of such pairs of entries is called the *degree of compatibility* of the two columns. If two columns are not compatible, they are called *incompatible.* In our example we have two incompatible columns: 11 and 22. For them the values of the attributes in A are the same but the class label is different.

	13	23	11	12	22
3221	3	*	*	*	*
2221	*	3	*	*	*
3132	3	*	*	*	*
2112	*	*	2	*	*
2231	*	*	*	2	*
1121	*	*	1	*	2
1211	*	*	1	*	*
h	1	1	1	1	2

Table 4.1: An example of a partition matrix

In order to find a new intermediate concept for B we need a labelling for the columns of the partition table (in other words, values for the intermediate concept c) such that g and h are consistent with f. That is exactly the case when incompatible columns are never assigned the same label. The problem is equivalent to graph colouring and the corresponding graph is the so called *incompatibility graph*. Its vertices correspond to the columns of the partition matrix and there is an edge between two vertices if and only if they are incompatible.

In the example, the graph contains five vertices and only one edge between 11 and 22 which indicates that they should be assigned different colours. That restriction becomes clearer when we remember that for those two columns there is a pair of data points from different classes that differ only in the values in B. If we assign the same value for B then we introduce an inconsistency - two points that have the same attribute values but are classified in different classes. Therefore the last row of the matrix in table 4.1 is a valid assignment for the intermediate concept.

The increase in the generalization achieved by the algorithm is due to the empty entries in the partition matrix which now might be assigned a value. This happens exactly when a non-empty entry exists in the same row which corresponds to the same value of the new concept c (i.e. the same colour).

The overall decomposition algorithm starts by trying to decompose X by considering all possible partitions such that B is not larger than a predefined threshold for the number of attributes. For each candidate, a partition selection measure is evaluated and the best partition is chosen. Then the complexity of the two new example sets is determined and if it is lower than that of the original set the decomposition is accepted. This is applied recursively until all leaves in the concept structure are found to be non-decomposable.

A number of different partition selection measures were investigated, however, the most simple one, column multiplicity, proved to be the best. It chooses the partition which leads to the least number of values of the new concept c.

Two information-based measures are proposed for determining the complexity of the functions.

4.2.2 Other Related Research

The earliest approaches to decomposition were developed in the area of switching circuits design back in 1940's and 1950's (see [5, 34]). Those methods decompose the truth table of a Boolean function realized through binary gates and consider both disjoint and non-disjoint decomposition.

Within machine learning the first attempts in decomposition were presented in [71] where manually defined concept structures were used with two layers of intermediate concepts.

A related field to function decomposition is feature extraction (also called constructive induction) which aims at constructing new better attributes from the existing ones through different methods. These methods in general need to be given in advance the operators they can use for generating the new attributes.

Other approaches use an expert to decompose the problem and construct the concept hierarchy, see for example [73] and [20]. Further, the hierarchical structure is used to generate decision rules.

A lot of research has been done in the specific case of Boolean functions decomposition. Important results relevant for our approach were presented in [22] which investigates the problem of decomposability of partially defined Boolean functions. It concentrates on the complexity of determining whether a Boolean function is decomposable using a specific scheme. A *scheme* determines the number of intermediate concepts to which the function should be decomposed and the most general scheme considered is:

$$f = g(S_0, h_1(S_1), h_2(S_2), \dots, h_k(S_k))$$

where S_i are (not necessarily disjoint) subsets of the set of all attributes A such that $\bigcup_{i=0}^{k} S_i = A$.

The results presented in the paper show that deciding whether a partially defined Boolean function is decomposable is an NP-complete problem for $k \geq 2$. For $k = 1$ the time complexity is proven to be $O(mn)$ where m is the number of data points defining the function and n is the number of attributes. The article also examines the problem of positive schemes in the frames of Boolean functions, which will be discussed in the following section.

4.3 Function Decomposition with Monotonicity Constraints

Let us take a monotone data set D with an attribute set A and a monotone labelling function which we view as a partially defined discrete function λ, λ :

X	a_1	a_2	a_3	a_4	a_5	a_6	λ
x_1	3	2	2	1	3	1	3
x_2	2	2	2	2	3	1	3
x_3	3	1	3	1	3	2	3
x_4	2	1	1	1	1	2	2
x_5	2	2	3	1	2	1	2
x_6	1	1	2	2	2	1	2
x_7	1	1	2	1	1	1	1
x_8	1	2	1	1	1	1	1

Table 4.2: An example of a monotone data set

$D \to \{0, m\}$. The attribute set A is split into two disjoint subsets such that $S_0, S_1 \subseteq A$, $S_0 \cup S_1 = A$, $S_0 \cap S_1 = \emptyset$. A scheme of the type $f = g(S_0, h(S_1))$ is called *positive* if the functions g, h are required to be positive.

Intuitively the monotone decomposition problem tries to answer the question whether for the given S_0 and S_1 there exists an extension of the positive scheme $f = g(S_0, h(S_1))$ and if 'yes' to find such an extension. A more general formulation of the problem will consider schemes of the type $f = g(S_0, h_1(S_1), \ldots, h_k(S_k))$ where $\bigcup_{i=0}^{k} S_i = A$ and the sets S_i might or might not be disjoint. However, we only look at the case where the attribute set is split into two non-intersecting parts with one intermediate concept.

Let us now try to give a more precise picture of the problem.

The requirement that $h(S_1)$ should be positive implies that the data set generated by S_1 and the corresponding values given by h should satisfy the monotonicity constraint. We denote this data set by $S_1|h$. Similarly the requirement that g should be positive implies that the resulting set after applying h on S_1 in D should also satisfy the monotonicity constraint. That data set is denoted by $S_0h|\lambda$.

This is demonstrated using the example in table 4.2. The attributes are split in subsets $S_0 = \{a_1, a_2, a_3\}$ and $S_1 = \{a_4, a_5, a_6\}$. Let us assume that g and h are known. Then the data set generated by applying h on S_1 is given in table 4.3. That data set is required to be monotone. Using the values of h in the original table we construct the set from table 4.4. It is also required to be monotone.

The problem has so far been investigated in the context of Boolean functions in [22]. There a criterion is given for the existence of an extension of positive schemes of a number of different types. Here we are only interested in schemes of the type $f = g(S_0, h(S_1))$. The corresponding proposition in [22] states that a partially defined Boolean function has an extension of positive scheme $f = g(S_0, h(S_1))$ if and only if there is no pair of vectors $x \in T^*$ and $y \in F^*$ such that $x[S_1] \le y[S_1]$. Here $x[S_1]$ denotes the vector containing only the values of

X	a_4	a_5	a_6	h
x_1	1	3	1	3
x_2	2	3	1	3
x_3	1	3	2	3
x_4	1	1	2	1
x_5	1	2	1	2
x_6	2	2	1	2
x_7/x_8	1	1	1	1

Table 4.3: The new data set generated using h, $S_1|h$

X	a_1	a_2	a_3	h	λ
x_1	3	2	2	3	3
x_2	2	2	2	3	3
x_3	3	1	3	3	3
x_4	2	1	1	1	2
x_5	2	2	3	2	2
x_6	1	1	2	2	2
x_7	1	1	2	1	1
x_8	1	2	1	1	1

Table 4.4: The new data set generated using g, $S_0h|\lambda$

x for the attributes in S_1 and T^* and F^* are defined as follows:

$$T^* = \{x \in T | \exists y \in F : y[S_0] \geq x[S_0]\},$$

$$F^* = \{y \in F | \exists x \in T : y[S_0] \geq x[S_0]\}.$$

Therefore deciding if a partially defined Boolean function has an extension of a positive scheme $f = g(S_0, h(S_1))$ can be done in polynomial time with complexity $O(m^2n)$.

In the rest of this chapter we investigate the corresponding problem in the context of discrete functions.

4.3.1 Monotone Decomposition of Discrete Functions

We choose to take a slightly different point of view and go back to the monotonicity restrictions over the two new data sets. For a representation of those restrictions we use the monotone discernibility matrix we proposed in the context of rough sets theory in chapter 2.

First, we prove the following lemma:

Lemma 9 *There exists a positive extension for the scheme $f = g(S_0, h(S_1))$ if and only if there exists an assignment of values $h_D : D \rightarrow \{h_i\}_{i=1}^k$ such that the two new data sets $S_1|h$ and $S_0h|\lambda$ are monotone.*

Proof: Let there exist an assignment h_D such that the two data sets are monotone. Then we define the following functions h and g over the input space where we denote the minimal possible value of h_D with $h_{\min}$ and the minimal possible value of λ with $l_{\min}$:

$$h(x) = \begin{cases} h_i & \text{if } \exists y : h_D(y) = h_i, y[S_1] \leq x[S_1] \text{ and} \\ & \nexists z \neq y : z[S_1] \leq x[S_1] \text{ such that } h_D(z) \geq h_D(x) \\ h_{\min} & \text{otherwise.} \end{cases}$$

$$g(x) = \begin{cases} \lambda_i & \text{if } \exists y : \lambda(y) = \lambda_i, y[S_0h] \leq x[S_0h] \text{ and} \\ & \nexists z \neq y : z[S_0h] \leq x[S_0h] \text{ such that } \lambda(z) \geq \lambda(y) \\ 0 & \text{otherwise.} \end{cases}$$

Let us check whether applying the two functions results in a positive extension. For this we need to prove that the two functions are positive or, in other words, that applying them to new points always preserves the monotonicity property over the currently labelled data.

Let $S_1|h$ be monotone and x be a new point on which we apply h: $h(x) = h_i$. Let $\exists y, z \in D$ such that $z \geq x \geq y$. Therefore $z[S_1] \geq x[S_1] \geq y[S_1]$.

According to the definition of h, $h(x) \geq h_D(y)$. Let $h(x) = h_i = h(v)$ for some $v \in D$ such that $x[S_1] \geq v[S_1]$. Then $z[S_1] \geq v[S_1]$, therefore $h_D(z) \geq h_D(v) = h_i = h(x)$. $h_D(z) \geq h(x)$ also holds if $\nexists y \in D$ such that $x \geq y$. Then $h(x) = h_{\min}$.

This proves that, after adding the new point x, $S_1|h$ remains monotone. We only used the property that $S_1|h$ is monotone, therefore, we can apply the same arguments to the new data set including x.

In a similar way we can prove that the data set $S_0h|\lambda$ remains monotone when adding the new point x (where $x[S_1]$ has already been labelled by h) using the function g to label it.

If for every assignment h_D at least one of the two data sets is not monotone then it is not possible to find a positive extension of the scheme by means of positive h and g functions. $\square$

In the following we sometimes abuse the notation by using h instead of h_D when no confusion arises.

Let us consider the data set generated by h, $S_1|h$. The monotonicity constraint here will restrict the new class values in such a way that if $x[S_1] \leq y[S_1]$ then $h(x[S_1]) \leq h(y[S_1])$. That implies that if $x[S_1] = y[S_1]$ then $h(x[S_1]) = h(y[S_1])$. Therefore we can simply join the identical vectors $x[S_1]$ and consider them together. In the new data set we assign some unknown values to the class

$h_2 \geq h_1,$	$h_3 \geq h_8$
$h_2 \geq h_5,$	$h_4 \geq h_7$
$h_2 \geq h_6,$	$h_4 \geq h_8$
$h_2 \geq h_7,$	$h_5 \geq h_7$
$h_2 \geq h_8,$	$h_5 \geq h_8$
$h_3 \geq h_1,$	$h_6 \geq h_7$
$h_3 \geq h_4,$	$h_6 \geq h_8$
$h_3 \geq h_5,$	$h_7 \geq h_8$
$h_3 \geq h_7,$	$h_8 \geq h_7$

Figure 4.3: The set of constraints from the data table $S_1|h$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
x_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_4	$a_1a_2a_3c_{14}$	$a_2a_3c_{24}$	$a_1a_3c_{34}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_5	a_1c_{15}	c_{25}	a_1c_{35}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_6	$a_1a_2c_{16}$	$a_1a_2c_{26}$	$a_1a_3c_{36}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
x_7	$a_1a_2c_{17}$	$a_1a_2c_{27}$	$a_1a_3c_{37}$	a_1c_{47}	$a_1a_2a_3c_{57}$	c_{67}	$\emptyset$	$\emptyset$
x_8	$a_1a_3c_{18}$	$a_1a_3c_{28}$	$a_1a_3c_{38}$	a_1c_{48}	$a_1a_3c_{58}$	a_3c_{68}	$\emptyset$	$\emptyset$

Table 4.5: The monotone discernibility matrix using the variables c_{ij}

attribute $\{h_i\}_{i=0}^k$. Further we generate constraints of the type $h_i \leq h_j$ for the class values for each couple of data points such that $x_i[S_1] \leq x_j[S_1]$.

Looking back to the example of table 4.2, the constraints generated in this way will be as shown in figure 4.3 where we denote $h_i = h(x_i[S_1])$.

As a next step, we replace the vectors $x[S_1]$ with the corresponding (for the moment still unknown) values h_i and that results in the data set $S_0h|\lambda)$. This new data set should also be monotone which here means that if $x \leq y$ then x cannot belongs to a lower class than y. We build the monotone discernibility matrix as in chapter 2 with the only difference that here we have unknown values for one attribute[1]. Therefore for that attribute we use special Boolean variables c_{ij} which are true if $h(x_i[S_1]) > h(x_j[S_1])$ and false otherwise. The monotone discernibility matrix for our example will look as shown on table 4.5.

If we look at the matrix closer, we realize that the constraints/variables that are necessary in order to keep the data set monotone are the ones that appear in the core and therefore will be present in every monotone reduct generated

[1]The monotone discernibility matrix was defined in chapter 2. For the purpose of the current discussion we only recapitulate that this is an $n \times n$ matrix with entries consisting of the subset of attributes for which the object of the higher class dominates the object of the lower class.

$$\boxed{\begin{array}{l} h_2 > h_5 \\ h_6 > h_7 \end{array}}$$

Figure 4.4: The set of constraints generated from the data table $S_0 h|\lambda$

from the matrix. The notion of the core in rough sets theory is explained in more details in chapter 2. Here we only recall that the core consists of all attributes that appear as single-attribute entries in the discernibility matrix. In our particular case, when a c_{ij} variable appears in the core that indicates that for the corresponding couple of objects x_i and x_j it holds that $x_i[S_0] \leq x_j[S_0]$ while $\lambda(x_i) > \lambda(x_j)$. Therefore to avoid the inconsistency we need to compensate by ensuring that $h(x_i[S_1]) > h(x_j[S_1])$.

In this way we discover a second set of constraints for the new attribute h. For the example the generated constraints are those shown in figure 4.4 where again we denote $h_i = h(x_i[S_1])$.

Note that in the case where the core of the monotone discernibility matrix is empty, we have no constraints of the second type. In this specific case the remaining constraints of only the first type can be satisfied by assigning the same value to all h-variables and that would be a valid solution to the problem.

A natural way of representing those two sets of constraints is by using a directed graph. The set of distinct values $\{h_i\}_{i=1}^{k}$ will be represented by the vertices of the graph. For the representation of the constraints, however, we need two different types of directed edges which are described in the following.

The first type of constraints (larger or equal) will be denoted by a dashed arrow in the figures or by a double arrow in the text $' \Rightarrow '$ where $x \Rightarrow y$ will mean that $x \geq y$. The special case when $x \leq y$ and $y \leq x$ (corresponding to $x = y$) can be denoted with a dashed undirected edge which can be traversed in both directions in the figures or an equal sign $' = '$ in the text. For the particular example this, however, does not occur.

The second type of constraints (larger) will be represented in the text and in the figures by single (solid) arrows $' \rightarrow '$ where $x \rightarrow y$ will mean that $x > y$. Here it cannot occur that both constraints $x > y$ and $y > x$ are present.

In our example both sets of constraints result in the graph given in figure 4.5.

4.3.2 Existence of a Positive Extension of the Scheme $f = g(S_0, h(S_1))$

We are now faced with the problem of finding out whether there exists an assignment for the values $\{h_i\}_{i=1}^{k}$ such that it is consistent with all constraints and if such assignments exist then to find one of them.

First, we introduce the following notation for $i \in [0, m-1]$ where we assume

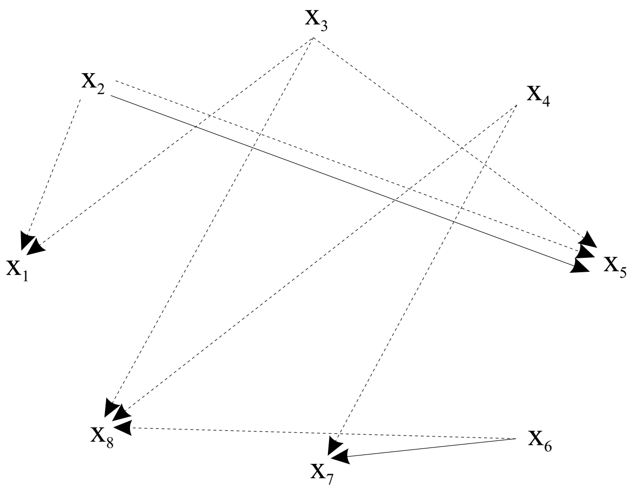

Figure 4.5: The constraints graph for the example

that 0 is the minimal value of the function λ and m is the maximal one:

$$T_i = \{x \in D : \lambda(x) = i\},$$

$$T_{>i} = \{x \in D : \lambda(x) > i\} \text{ for } i \in [0, m-1],$$

$$T_{<i} = \{x \in D : \lambda(x) < i\} \text{ for } i \in [1, m].$$

Using that, we define the following two sets:

$$T^*_{>i} = \{x \in T_{>i} \mid \exists y \in T_i, x[S_0] \leq y[S_0]\} \text{ for } i \in [0, m-1],$$

$$T^*_{<i} = \{x \in T_{<i} \mid \exists y \in T_i, y[S_0] \leq x[S_0]\} \text{ for } i \in [1, m].$$

We can now formulate the following criterion for existence:

Theorem 10 *There exists a positive extension of the scheme $f = g(S_0, h(S_1))$ if and only if there are no data points $x_i, x'_i, x_{i+1}, x'_{i+1}, \ldots, x_{i+j}, x'_{i+j}, x_{i+j+1}$ such that $x_{i+j+1} = x_i$ and such that for all x_k, x'_k the following conditions hold:*

1. *if $x_k \in T_l$ then $x'_k \in T^*_{>l}$,*
2. *$x'_k[S_1] \leq x_{k+1}[S_1]$.*

Proof: First note that in the conditions of the theorem x'_k is not required to be different from x_{k+1}. However x_k needs to be different from x'_k because otherwise the first condition cannot be true.

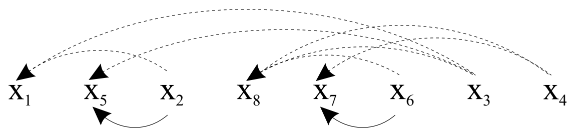

Figure 4.6: The constraint graph from figure 4.5 after applying the topological ordering procedure

Also note that, because of the transitivity of the relation $\leq$ for vectors, if the following two constraints are present: $x[S_1] \leq y[S_1]$, $y[S_1] \leq z[S_1]$ then the constraint $x[S_1] \leq z[S_1]$ is also present.

Let us assume that there exists such a sequence of data points that satisfies the conditions. This means that there exists a cycle in the constraints graph such that

$$h(x_i) < h(x'_i) \leq h(x_{i+1}) < h(x'_{i+1}) \leq \ldots < h(x'_{i+j}) \leq h(x_i).$$

It is clear that no such assignment for the values of h exists.

Let us now assume that no such sequence of data points exists. This means that the constraints graph will be acyclic. It is known that in every acyclic directed graph there exists at least one vertex with zero in-degree and at least one vertex with zero out-degree. Therefore we can use topological sorting to find an ordering of the vertices such that all edges point in the same direction in this ordering. A simple algorithm exists to find such ordering which proceeds as follows.

Since the graph is acyclic then we can find a vertex with a zero in-degree. We choose it to be the first in the ordering and do not consider it or its edges further in the sorting. The new graph is obviously also acyclic therefore we can again find a vertex with zero out-degree which will be the second in the ordering. We proceed like thais until all vertices are included in the ordering. We can now represent the graph in such a way that all vertices are in one line and all edges point to the left in the structure. As an example we use the constraint graph of figure 4.5. After applying the topological sorting procedure we can represent the graph as in figure 4.6.

The complexity of the topological sorting in $O(|V||E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges.

Now we need to find a labelling for the vertices consistent with all the constraints. The most straightforward way is to use the topological ordering $\{x_1, x_2, \ldots, x_{|V|}\}$ and assign the corresponding values $\{1, 2, \ldots, |V|\}$ in the same order. Such a labelling is obviously consistent with the constraints because due to the topological sorting if a constraint $h(x_i) \leq h(x_j)$ or $h(x_i) < h(x_j)$ exists then $i < j$ which are exactly the labels of x_i and x_j. $\square$

4.3.3 Assignments with a Minimal Number of Values

In the previous subsection we proved that if the condition of the theorem is satisfied then there exists a labelling consistent with the constraints. However, the simple labelling that we used in the proof is probably not the best one we can find as it assigns different value to each data point, which is in most cases not necessary. In general we would be more interested in assignments with a (close to the) minimal number of different values.

In order to address this problem we first define a *path* in the constraint graph as a sequence of vertices $x_1, x_2, \ldots, x_j$ such that for each pair x_i, x_{i+1} there exists either a single edge from x_i to x_{i+1}(constraint $h(x_i) < h(x_{i+1})$) or a double edge (constraint $h(x_i) \leq h(x_{i+1})$). The *length* of a path P is the number of single edges participating in it, denoted by $|P|$. For an acyclic constraint graph, we denote the maximal length of a path in the graph by χ.

Theorem 11 *If there exists a positive extension of the scheme $f = g(S_0, h(S_1))$, then the minimal number of values necessary for an assignment consistent with the constraints equals the number $\chi + 1$ of the constraint graph.*

Proof: Let the path of a maximal length be $x_1, x_2, \ldots, x_i$ with length χ. Then $\chi + 1$ is obviously a lower bound for the number of values necessary for a consistent assignment as for each end point of a single edge we need a higher value. We need to prove that there exists a consistent assignment with exactly $\chi + 1$ values.

Consider the following assignment:

$$h_D(x) = \begin{cases} h_{\min} + \max\{|P| : P - \text{path starting from } x\} & \text{if such path exists,} \\ h_{\min} & \text{otherwise.} \end{cases}$$

where $h_{\min}$ denotes the minimal possible value of h.

Is this assignment consistent with all the constraints? Let us assume that a constraint $x \rightarrow y$ exists in the graph. Then $h_D(x) \geq h_D(y) + 1$ since the longest path starting from x will contain at least one single edge more than the longest path starting from y. Let a constraint $x \Rightarrow y$ exist. Then $h_D(x) \geq h_D(y)$ because the length of the path from x will be at least as much as the path from y.

Let y be such a vertex that $h_D(y) > \chi + h_{\min}$. Therefore $h_D(y) = |P| + h_{\min}$ for some path P which is a path starting from y with length more than χ. This is a contradiction with the definition of χ. Therefore the proposed assignment needs exactly $\chi + 1$ values. □

The assignment used in the proof of the theorem can be found by using the topological ordering in the following way. We start by assigning $h_{\min}$ to the first vertex where $h_{\min}$ denotes the minimal possible value for the h-variables. Following the ordering we label each other vertex using the following procedure:

X	a_1	a_2	a_3	h	λ
x_1	3	2	2	1	3
x_2	2	2	2	2	3
x_3	3	1	3	1	3
x_4	2	1	1	1	2
x_5	2	2	3	1	2
x_6	1	1	2	2	2
x_7	1	1	2	1	1
x_8	1	2	1	1	1

Table 4.6: The new assignment for the h-variables using only two values

1. If no edges start from the vertex, assign $h_{\min}$;
2. Otherwise for each such edge:
   (a) extract the label of the end vertex;
   (b) for single edges add 1 to the corresponding number;
3. Find the maximal among the numbers for all edges ending in x;
4. Assign this maximal number to the current vertex;

The complexity of the procedure is $O(|V||E|)$ where V is the set of all vertices of the constraint graph and E is the set of all edges.

For our example we proposed an assignment in table 4.4. However, applying the above described procedure we discover a better solution (by means of fewer values) which is given in table 4.6.

An alternative assignment can be the following:

$$h'_D(x) = \begin{cases} h_{\min} + \chi - \max\{|P| : P - \text{path leading to } x\} & \text{if such path exists,} \\ h_{\min} + \chi & \text{otherwise.} \end{cases}$$

In the same way as in theorem 11 we can prove that h'_D is a consistent assignment and uses $\chi + 1$ number of values.

Lemma 10 *For each vertex x of an acyclic constraints graph, it holds that $h_D(x) \leq h'_D(x)$.*

Proof: Let P be the longest path leading to x and Q be the longest path starting in x. Therefore:

$$|P| + |Q| \leq \chi \Rightarrow$$
$$\Rightarrow |Q| \leq \chi - |P|,$$

which is equivalent to $h_D(x) \leq h'_D(x)$.

In case no such Q exists for x then $h_D(x) = 0 \leq h'_D$. Similarly if no such P exists for x then $h'_D = h_{\min} + \chi \geq h_D$. □

4.3.4 Default Rule for Covering the Whole Input Space

As was mentioned before, the decomposition method achieves higher generalization than the original data set. However there is no guarantee that the whole input space will be covered. In lemma 9 we proposed two specific functions for h and g that will cover the whole input space. Here we would like to take a more practical point of view and reformulate the solution using simpler default rules that classify examples not covered by the extended data set generated by the full concept hierarchy. Obviously for the case when monotonicity constraints exist this default rule should also satisfy the restrictions so that the resulting classifier is guaranteed to be monotone over the whole input space.

The default rules we propose here will be discussed in the case of one intermediate concept as in the previous sections. It can easily be generalized for the whole concept structure.

As the new example x will have to be compared to two data sets we need two labelling functions. At the first step we compare $x[S_1]$ with the data set $S_1|h$. If none of the data points in it is equal to $x[S_1]$ then we need to find a new label consistent with $S_1|h$. Once such a label is found we replace $x[S_1]$ with it and compare the new example with the data set $S_0h|\lambda$. Here again if none of the already existing examples is equal to the new one we have to find a new label that is consistent with the rest of the data.

In general it might not be necessary to apply a labelling function on both steps. For example after applying the function at the first step the new example might turn out to be already present at the second step.

Since those two steps look very similar, we can naturally use the same labelling function where the only difference is the data set we consider. We therefore denote the data set by D which in the first step should be replaced by $S_1|h$ and in the second step by $S_0h|\lambda$. The labelling function of the particular data set will be denoted by $\lambda(x)$ which in the first step should be replaced by h and in the second step by g. We propose two different alternative versions of the labelling function denoted by $\lambda_{\min}$ and $\lambda_{\max}$ as follows:

$$\lambda_{\min}(x) = \begin{cases} \max\{\lambda(y) :\ y \in D \cap \downarrow x\} & \text{if } x \in \uparrow D \\ c_{\min} & \text{otherwise;} \end{cases}$$

$$\lambda_{\max}(x) = \begin{cases} \min\{\lambda(y) :\ y \in D \cap \uparrow x\} & \text{if } x \in \downarrow D \\ c_{\max} & \text{otherwise.} \end{cases}$$

where $\mathcal{X}$ is the input space, $c_{\min}/c_{\max}$ are the minimal and the maximal possible label respectively and we use the following other notation:

$$\downarrow x = \{y \in \mathcal{X} : y \leq x\},$$

$$\uparrow x = \{y \in \mathcal{X} : y \geq x\},$$

$$\downarrow D = \bigcup_{x \in D} \downarrow x,$$

$$\uparrow D = \bigcup_{x \in D} \uparrow x.$$

In the proof of lemma 9 we used $\lambda_{\min}$ both for h and g. However, in general this is not necessary. Different functions can be used on the different steps, e.g. $\lambda_{\min}$ for labelling at the first step and $\lambda_{\max}$ at the second step, as long as the same function is applied every time we are at the same step. If we apply both functions at the same step the monotonicity is no longer guaranteed.

Note that those functions are the same labelling functions used for adding points to the updated data set in the Monotone Decision Trees algorithm (chapter 3). They have been proven to give consistent labels when the data set is monotone. It is also known that $\lambda_{\min}$ tends to give higher labels and more optimistic predictions than $\lambda_{\max}$.

As an example to demonstrate how the default rule works we consider again the decomposed data set of tables 4.2, 4.3 and 4.4. Let us try to classify the new data point $x = (2,2,2,2,2,2)$ which is not present in the original set. First we consider $x[S_1] = (2,2,2)$ which does not appear in $S_1|h$ (table 4.3). We apply the labelling functions and both $\lambda_{\min}$ and $\lambda_{\max}$ give the same label 2.

At the second step we replace that label in x and compare the new data point $(2,2,2,2)$ with $S_0h|\lambda$ (table 4.4). It is not present there, therefore we need to again apply a labelling function. We try both $\lambda_{\min}$ and $\lambda_{\max}$ and discover that $\lambda_{\min}$ predicts a label 2 and $\lambda_{\max}$ predicts 3. Let us assume that we prefer more optimistic predictions and we choose $\lambda_{\max}$. Therefore the final label assigned to $x = (2,2,2,2,2,2)$ will be 3.

4.4 Conclusions

In this chapter we propose a decomposition method for discrete functions which can be applied to monotone problems in order to generate a monotone classifier based on the extracted concept hierarchy. We formulated and proved a criterion for the existence of a positive extension of the scheme $f = g(S_0, h(S_1))$ in the context of discrete functions.

We also propose a method for finding an assignment for the intermediate concept with a minimal number of values based on topological sorting. The complexity of the procedure is $O(|V||E|)$ where V is the set of all vertices of the constraint graph and E is the set of all edges. Two alternative assignment functions were defined for giving values to the intermediate concept.

Furthermore we propose two alternative default rules for the classification of points not covered by the extended data set of the concept structure. These default rules applied together with the concept structure produce a monotone classifier in the case of a monotone problem.

A number of directions for further research can be mentioned. Our research was focused on decompositions using the scheme $f = g(S_0, h(S_1))$. It would be interesting to investigate whether some of the results could be extended to the case of more complicated schemes using more intermediate concepts. The research in the Boolean case points out a trend of fast increase in the complexity when the scheme becomes more complicated and it is expected that in the discrete case similar development will be observed.

It would also be interesting to consider decompositions with non-disjoint attribute sets. In practical applications, the requirement for disjointness of the attribute sets might be too restricting. However, this would probably increase the complexity of the problem.

Another research direction would be to consider more than one intermediate concepts based on the same subset of attributes. That would result in a scheme of the type $f = g(S_0, h_1(S_1), h_2(S_1))$ in the case of two intermediate concepts over S_1. For such a small number the complexity might still be lower than that of the original set of attributes.

Chapter 5

Frequent Patterns

5.1 Introduction

Data Mining emerged back in the late 1980s and received a significant boost during the 1990s. This large multidisciplinary field connects to a number of areas, such as machine learning, statistics, information retrieval, pattern recognition, database technology, knowledge-based systems, etc. For a good overview of the area the reader is referred to [44]. In this chapter the focus is on association rules mining and its major subproblem of frequent patterns generation.

The problem of association rules mining was first introduced in [1]. A popular way of explaining the problem is to give an example from the so-called basket data, which constitutes one of the most important applications of the research in this area. Due to the progress in bar-code technology, massive amounts of data were collected and attracted the attention of analysts. This data would usually consist of rows of records where each row is with various length and represents the set of products bought by one customer at one visit to the shop/supermarket/etc. Such data has huge potential benefits. Typical questions that might be asked for example are: Which products are frequently sold together? If the customer buys product A which other products is he/she likely to buy? The answers to those questions can be used to analyze customer buying patterns and sales trends and can be applied in marketing campaigns such as promotions and special offers as well as for developing better ways of arranging the products on the shop shelves. Association rules can help in giving non-trivial answers to those questions.

An *association rule* reflects relations between the products in customer carts. An example of such a rule can be: 74% of the customers who buy cereal also buy milk. Of course this rule is obvious and we do not need to perform complicated data analysis to retrieve it. Nevertheless, the data very often contains previously unknown rules which can give food for thought to marketeers and analysts.

Imagine, however, how many customers visit a relatively big supermarket per day, week, and month. This points us to one of the main problems of the field – the huge amounts of data require very efficient algorithms for processing. Various proposals have been presented in the literature (among which [1, 28, 45, 61, 72, 78]) and some experimental comparisons have been performed (see for example [46, 79]). Nonetheless, very little is done on the theoretical comparison between the algorithms and even the experiments include only some of the algorithms and do not provide “easy answers” to the question which ones to choose for what kind of data.

This chapter is a contribution in that direction. It concentrates on the main subproblem of association rules mining – frequent patterns generation. Two of the best existing algorithms Depth-First [61] and FP-Growth [45] are considered. They have not been previously compared although FP-Growth has given very good results in experimental comparisons with other algorithms. The goal of this research is to provide both theoretical and experimental insight into the performance of the two algorithms, where they differ and which features of the data sets influence the differences in their performance. The research was previously published in [36] and extended in [51].

The chapter is organized as follows. Section 5.2 will give a formal definition of the problem of association rules mining. Section 5.3 presents one of the first and most popular algorithms in the literature – Apriori [1, 2] which became the basis for many of the later algorithms. The FP-Growth algorithm is described in section 5.4 and the Depth-First algorithm – in section 5.5. Their connection with the Apriori algorithm is investigated in sections 5.6 and 5.7 respectively. Section 5.8 gives the theoretical comparison of FP-growth and Depth-First and they are experimentally compared in section 5.9. And finally section 5.10 gives the conclusions.

5.2 Association Rules

In this chapter the problem of association rules mining will be formally described using the notation and terminology of [1, 2].

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals called *items.* A set $X \subseteq \mathcal{I}$ with $|X| = k$ is called a k-itemset. Let $\mathcal{D}$ be a multiset[1] of *transactions* where each transaction T consists of a set of items, $T \subseteq \mathcal{I}$. Each transaction has a unique identifier associated to it called *TID.*

An *association rule* is an implication of the type: $X \Rightarrow Y$ with $X, Y \subseteq \mathcal{I}$ and $X \cap Y = \emptyset$, where X is called the antecedent and Y is called the consequent of the rule. Such a rule expresses that if a transaction T contains X then it will, with some probability, also contain Y. Formulated like this, it is obvious

[1]A multiset is such a collection of objects where the order is ignored but identical objects might be present. For example $\{a, b\}$ is the same as $\{b, a\}$ but is different from $\{a, a, b\}$.

that a rule can be generated for each two sets of items that appear together in a transaction. Among these we are only interested in the "significant" ones – those which appear often enough.

The *support* of an itemset $X \subseteq \mathcal{I}$ is defined as the fraction of transactions in the database which contain all items in X, or:

$$support(X) = \frac{|\{T \in \mathcal{D} \,|\, X \subseteq T\}\,|}{|\mathcal{D}|}.$$

The support of a rule $X \Rightarrow Y$ is defined as the fraction of transactions that contain both X and Y or the probability $p\,(X \cup Y)$. The *confidence* of a rule is the conditional probability $p\,(Y \subseteq T \,|\, X \subseteq T)$ or, in other words, the ratio:

$$confidence(X \Rightarrow Y) = \frac{support(X \cup Y)}{support(X)}$$

if $support(X) \neq 0$.

Now we can formulate the problem of mining association rules as: generate all rules with support and confidence at least as high as some predefined thresholds for minimal support and minimal confidence. An itemset with support at least as high as the minimal support is called a *frequent itemset* (or a *frequent pattern*). The problem can now be split into the following two subproblems that can be solved separately:

1. Find all frequent itemsets with their supports.

2. For each frequent itemset X, generate rules by splitting the itemset into two non-intersecting parts $X \backslash Y \Rightarrow Y$ as long as the confidence of the resulting rule is at least as high as the minimal confidence.

The second step is the easier of the two and the overall performance of the algorithm is determined by the first step, on which will be the focus of the remainder of this chapter.

Note that as soon as step 1 is completed we need not read the database anymore since all necessary information is already available for step 2. This is a consequence of an important property of the set of frequent itemsets which will be elaborated on in the following paragraphs.

First we give a quick review of the necessary terminology from lattice theory. A binary relation R over a set U which is reflexive, antisymmetric and transitive is called a *partial order* and (U, R) is called a *partially ordered set.* A partially ordered set (U, R) in which for each pair $x, y \in U$ there exists an element z that is the least upper bound (also called join) and an element w that is the greatest lower bound (also called meet) is a *lattice.* (U, R) is a *complete lattice* if join and meet exist for each subset. Every finite lattice is complete.

A partially ordered set is called a *join semi-lattice* (*meet semi-lattice*) if only the join (meet) exists. The powerset of a set together with the inclusion relation is a complete lattice.

Let $\mathcal{L} = 2^{\mathcal{I}}$ be the powerset of $\mathcal{I}$. Then $\mathcal{L}$ is a lattice. The maximal element of $\mathcal{L}$ is $\mathcal{I}$ and the minimal element is the empty set $\{\}$. It is easy to see that the set of all frequent itemsets is a meet semi-lattice. Similarly the set of all infrequent itemsets forms a join semi-lattice. The following lemma is a consequence:

Lemma 11 *All subsets of a frequent itemset are also frequent.*

Analogically we can see that all supersets of an infrequent itemset are also infrequent. Formulated in a different way, the property of being an infrequent pattern is monotone while the property of being a frequent pattern is anti-monotone. This knowledge is important as it can be used in reducing the number of candidate patterns. Bearing in mind the huge amount of data that usually needs to be analyzed, such a property not only helps speed up the algorithms but also in some cases makes the computation feasible.

Let us consider the example in figure 5.1. It represents the set of all possible itemsets in $\mathcal{I} = \{a, b, c, d, e\}$. The lines connecting the itemsets are the inclusion relation. The frequent itemsets are indicated by oval frames. It can be seen that for every frequent itemset all its subsets are also frequent. However not all itemsets that are supersets of frequent ones turn out to be frequent as is the case with $\{a, c\}$ and $\{b, c, d\}$.

5.3 The Apriori Algorithm

In this section we will introduce the Apriori algorithm as it was originally presented by Agrawal and Srikant in 1994. The algorithm relies heavily on the property discussed in the previous section that all subsets of a frequent itemset are also frequent. That allows to limit the number of candidates for frequent itemsets that need to be counted. A k-itemset is considered as a candidate if all of its subsets of length k-1 have been found to be frequent.

The algorithm performs a breadth first search starting from the 1-itemsets. It requires as many scans of the whole database as the maximal length of a candidate itemset.

Figure 5.2 gives the Apriori algorithm. It uses the following notation: L_k is the set of frequent k-itemsets, C_k is the set of candidates of length k. The algorithm starts with a database pass to count the supports of all 1-itemsets. The frequent ones among them are used to generate the candidates of length 2 by the **apriori-gen** function. Next the database is scanned again to count the support of those candidates. For each transaction t the candidates that are subsets of it are found (those are the subsets C_k^t of t for each C_k) and their counts incremented. The candidates that turn out to be frequent are used to

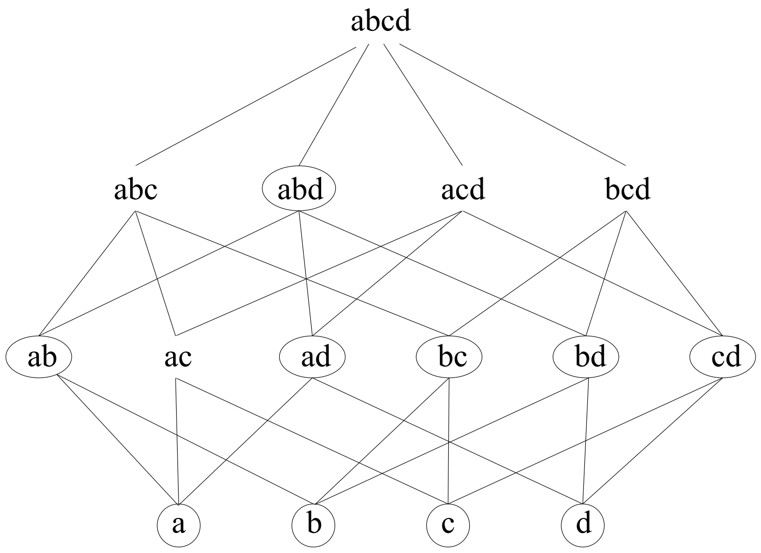

Figure 5.1: The lattice of itemsets

generate the candidates of the next level. The procedure goes on until there are no frequent itemsets at the current level.

The **apriori-gen** function takes as parameter L_{k-1} and returns the new set of candidates C_k. It works as follows. For each pair $p = \{i_1, i_2, \ldots, i_{k-1}\} \in L_{k-1}$ and $q = \{j_1, j_2, \ldots, j_{k-1}\} \in L_{k-1}$ such that $i_1 = j_1, \ldots, i_{k-2} = j_{k-2}$, $i_{k-1} < j_{k-1}$ a new candidate $c = \{i_1, i_2, \ldots, i_{k-1}, j_{k-1}\}$ is generated and added to C_k. Further the set C_k is pruned by removing all candidates that contain a subset of length (k-1) not present in L_{k-1}.

5.4 The FP-Growth Algorithm

One of the most efficient algorithms which appeared after Apriori was FP-growth [45]. The most distinct feature of the algorithm is the use of a very complicated but compact data structure for storing the necessary part of the database. This structure, called *FP-tree*, allows an efficient procedure of extracting the frequent patterns.

Let us first focus on the FP-tree. It is a tree-like structure which can be defined as follows:

1. The FP-tree consists of: a root labelled by *null*, nodes connected in a tree structure starting from the root, a header table.

2. Each node contains the following information: item name, count and node

```
Apriori($D$):
    $L_1$=all frequent 1-itemsets;
    for($k$=2; $L_{k-1} \neq \emptyset$; $k$++)
        $C_k$=apriori-gen($L_{k-1}$);
        forall transactions $t \in D$
            $C_k^t$=subset($C_k$,$t$);
            forall candidates $c \in C_k^t$
                $c.count$++;
        $L_k = \{c \in C_k | c.count \geq minsup\}$;
    result=$\bigcup_k L_k$;
```

Figure 5.2: The Apriori Algorithm

link. The item name is the item represented by the node, the count represents the number of transactions contained in the path from the root to the node. The node link is a link to the next node with the same item in the tree or *null* if such a node does not exist.

3. Each of the rows of the header table contains an item name and a node link which points to the first node in the tree labelled by the same item.

The structure is created using the **FP-tree** procedure which implements the following algorithm in two steps:

1. The frequent 1-itemsets are discovered by a scan of the database. The set is sorted in descending value of their support – it is denoted by L.

2. The root of the tree, R, is created. The database is scanned again and for each transaction the items in the transaction are sorted following the order of L and inserted in the tree using the procedure **insert-tree**($t|T$,R) on the root R and the transaction $t|T$ where t is the first item in the sorted transaction and T is the rest.

The **insert-tree** procedure considers the first item in the transaction t and checks if R has a child N labelled by the same item. If so, the count of that child is incremented, otherwise a new child N is created and assigned count 1. Further, the header table and the node links are updated if necessary. If the rest of the transaction T is not empty then the procedure is called recursively on T and N: **insert-tree**(T,N).

Clearly two database scans are necessary for the generation of the tree. Afterwards the information contained in this tree is enough for the process of frequent patterns mining.

An example will give a clearer picture of the structure of the tree and the building procedure. Table 5.1 shows a small transaction database which we

TID	transaction					
1	a	b	d			
2	b	c	d	f	g	h
3	a	b	d	g		
4	d	e	f			
5	b	d	f			
6	a	f				

Table 5.1: The example transaction database

use for constructing an FP-tree. The full tree is given in figure 5.3 where the parent-child links are denoted by solid lines and the node links by dashed lines.

The first step is to scan the database and find the frequent 1-itemsets. Let us have a minimal support threshold of 2 which means that we are only interested in itemsets that appear in at least two transactions. The frequent 1-itemsets[2] (sorted in decreasing value of the support) are: $(d : 5), (b : 4), (f : 4), (a : 3), (g : 2)$, where the numbers after the colon signs are the support values. The portion of the database that contains only the frequent 1-itemsets is shown in table 5.2 where the transactions are also sorted in decreasing support.

We start with the first transaction $\{d, b, a\}$. The tree contains only the root and the transaction will be inserted as a branch with counts equal to 1 for each node. The second transaction is $\{d, b, f, g\}$. Starting from the root, we try to insert d. But the root already has a child labelled with d, thus we only increase the count of the node to 2 and proceed by trying to insert b in the subtree originating at the d-node. However, there is again a child with the label b and we increase the count of the b-node to 2 and proceed with f. The b-node this time has no child carrying the label f and we create a new f-node as a child with count 1. From that new node we try to insert g which results in creating a child with item g and count 1.

The insertion of the third transaction $\{d, b, a, g\}$ results in increasing the counts in the branch $d \to b \to a$ and creating a new child of the a-node labelled with g. Proceeding in the same way we generate the full FP-tree given in figure 5.3.

The FP-tree is a highly condensed way of representing the relevant part of the database. This is due to the "collapsing" of transactions having the same prefix – the prefixes of such transactions are stored together and further processed together and only the count indicates that more than one transaction is present in this branch. In the extreme case of having a database of identical transactions, the tree will contain only one path corresponding to all the transactions.

[2]In the following we reserve the round brackets notation for itemsets that are found to be frequent and appear in the output of the algorithm. The curly brackets will be used for transactions in a (conditional) data base.

TID	transaction			
1	d	b	a	
2	d	b	f	g
3	d	b	a	g
4	d	f		
5	d	b	f	
6	f	a		

Table 5.2: The sorted database containing only items originating from frequent 1-itemsets

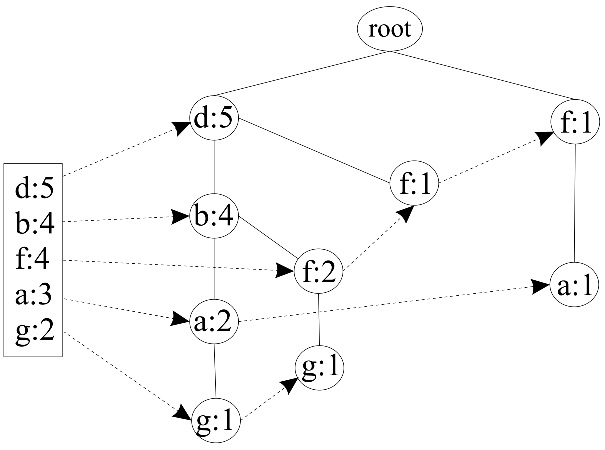

Figure 5.3: The full FP-tree

```
FP-growth($Tree$,$\alpha$):
    if $Tree$ consists of a single path $P$
        for each combination $\beta$ of nodes in $P$
            generate $\beta \cup \alpha$ with support= min support of a node in $\beta$;
    else for each $a_i$ in the header of $Tree$
        generate $\beta = a_i \cup \alpha$ with support= $a_i.support$;
        construct $\beta$'s conditional pattern base;
        construct $\beta$'s conditional FP-tree $Tree_\beta$;
        if $Tree_\beta \neq \emptyset$
            FP-growth($Tree_\beta$,$\beta$);
```

Figure 5.4: The FP-growth procedure for mining the frequent itemsets

By sorting the items in decreasing support we influence the structure of the tree. The goal is to minimize the number of nodes by starting with the most frequent ones and to avoid generating new nodes by "collapsing" the identical ones.

The special structure of the FP-tree will obviously need specific procedures for extracting the frequent itemsets. This procedure is called **FP-growth** and is given in figure 5.4. It takes as an input an FP-tree and a (possibly empty) set of items that are called the prefix on the current step (denoted by α). At the first call of the procedure the parameters are the full FP-tree generated from the data set and an empty set for α.

The algorithm starts by checking if the tree consists of a single path. If this is the case each subset of it is a frequent itemset which reveals another efficiency feature of FP-growth – it is not necessary to mine the tree further. In the extreme case of a database containing only identical transactions the tree will contain a single path and the algorithm will simply terminate at this step.

If the tree contains more than one path, then the current frequent itemsets are generated and the algorithm goes into recursion with each of them and their corresponding conditional FP-trees. The conditional FP-trees are generated using the **FP-tree** procedure discussed earlier based on the conditional pattern base. The conditional pattern base of an item a_i is the collection of all prefix paths of the a_i-nodes. They are extracted by starting from the head of the node-link for a_i in the header table. From the first node reached we extract the path connecting it to the root and we attach to it the support of the a_i-node. We go on following the node-links to the next a_i-node to extract the next path.

Let us look again at the example from figure 5.3. The mining process starts from the header table and the items are considered from the bottom (in other words we move from the leaves of the tree upwards). However, since the processing step of an item is independent from the processing of the other items in the header table, the order does not influence the result.

First we output the pattern $(g : 2)$ and then, starting from the header entry for g and following the node links, we extract the conditional pattern base for g. It contains two paths (considered here as transactions although there is no one-to-one correspondence with the original transaction database): $\{d, b, a\}$ and $\{d, b, f\}$. They both have support 1 which is the support of the corresponding g-node connected to them. We count the support of the items and discover that a and f are not frequent (together with g). The remaining parts of the transactions are used to generate a conditional FP-tree. It contains a single path $d \to b$ with support 2, thus we output all combinations of items from the path postfixed with g. The patterns are: $(b, g : 2), (d, g : 2), (d, b, g : 2)$.

Note that the four patterns generated in this first step are the only ones in which the item g participates. Therefore there is no need to consider this item later in the mining process.

We go one step back in the recursion to the original tree and go on with the item a in the header table. First we output the pattern $(a : 3)$. The conditional pattern base consists of two paths/transactions: $\{d, b : 2\}$ and $\{f : 1\}$ which shows that f is not frequent and the conditional FP-tree again contains a single path $d \to b$. As a result we output the following patterns: $(d, a : 2), (b, a : 2), (d, b, a : 2)$.

The algorithm goes on in a similar way with the items f, b and d.

5.5 The Depth-First Algorithm

The Depth-First algorithm was introduced in [61]. As the name suggests, a depth-first strategy is used for the generation of candidates. Another important feature is that the structure used to represent the itemsets is a trie which can be described as follows.

A *trie* consists of nodes and links. Each node contains a number of cells which are also called *buckets*. Each bucket is labelled by an item. The links connect a bucket (as a parent) to a node (as a child). Each path in the trie starting from the root is an itemset and therefore each bucket corresponds to a unique itemset. An example trie is given in figure 5.5. The root node of that trie contains five cells/buckets labelled with the items a, b, c, d and e. Each path starting from one of those cells represents an itemset. For example the path $e \to c \to b \to a$ represents the itemset $\{e, c, b, a\}$ but that means this also $\{e, c, b\}$ is an itemset as well as $\{e, c\}$ and $\{e\}$.

In the following, the terms "cell", "path" and "itemset" will be used interchangeably in the context of an itemset trie. That is because each cell determines exactly one path that leads to it starting from the root. This path corresponds to an itemset containing the items present in the path.

The trie is an appropriate data structure for storing frequent patterns be-

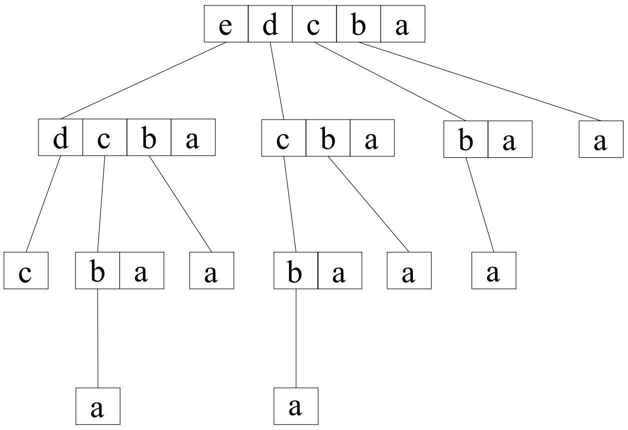

Figure 5.5: An example of a trie

cause of the property that the set of all frequent itemsets is a meet semi-lattice and every subset of a frequent itemset is also frequent. In this case each bucket also contains a number which gives the support of the itemset represented by the path from the root to that bucket.

Note that the trie is very different from the FP-tree as in the trie every cell determines uniquely an itemset. This is not the case in the FP-tree where the same itemset may be present in different branches of the tree and extracting only one of those branches does not necessarily give us the full support of the itemset.

The Depth-First algorithm is given in figure 5.6. As a preprocessing step, the frequent 1-itemsets are extracted by one scan of the database. The infrequent ones are not further considered in the algorithm. The frequent 1-itemsets, denoted by $i_1, i_2, \ldots, i_n$ form the root of the trie. For each item in the root starting from i_{n-1} and moving towards i_1, the subtrie that has been built to the right of it is copied under the bucket. This new subtrie contains all the current candidates. Their support is counted by a database scan and the infrequent ones are pruned (removed from the trie). The algorithm proceeds with the next root-item to the left.

The **count** procedure is performed by extracting the transactions one by one and "pushing" them through the trie. If the current transaction does not contain the root item then we ignore it and proceed with the next transaction. If it does it is checked for the first child-cell item. If that is present then we go deeper recursively, if not we continue with the next child-cell.

A refinement of the algorithm is to sort the items in the root in increasing order so that the most frequent one is in the rightmost cell of the root. This would mean that by moving to the left in the structure fewer transactions need

```
**Depth-First():**
    $T$ = the trie including only bucket $i_n$;
    **for** $m = n - 1$ **downto** 1
        $T' = T$;
        $T = T'$ with $i_m$ added to the left and a copy of $T'$ appended to $i_m$;
        $C = T \setminus T'$ (the subtrie rooted in $i_m$);
        **count**($C$);
        delete the infrequent itemsets from $T$;
```

Figure 5.6: The Depth-First algorithm

to be inspected in order to count the support of candidates. This improvement will be assumed in the following.

The algorithm is illustrated with the example in figures 5.7 to 5.10 which uses the database from table 5.1 and a support threshold of 2. For simplicity, the supports of the cells in the trie are not shown in the pictures. We start with the frequent 1-itemsets which here are a, b, c, d, e. They form the root node. First we consider a and we create a root-cell for it. We then consider item b and add it to the root. The subtrie to the right of it contains only a which is copied under b. The resulting trie is shown on figure 5.7 where the newly copied subtrie is drawn with dashed lines. The database is scanned for the support of the candidate itemset (b, a) which turns out to be frequent.

The algorithm proceeds with the root-item c. The subtrie to the right of it is copied under it, see figure 5.8. Again the database is scanned for the support of the new part. The candidates that are actually being counted are (c, b), (c, b, a) and (c, a). They turn out to be frequent and are kept in the trie. We then proceed with the item d and the situation from figure 5.9. The database is scanned again for the support of the following candidates: (d, c), (d, c, b), (d, c, b, a), (d, c, a), (d, b), (d, b, a) and (d, a). They all prove to be frequent and are therefore kept in the trie.

We finally consider the item e. The updated trie is shown in figure 5.10. The database is scanned for the new candidates. This time however some of them turn out to be not frequent: (e, d, c, b), (e, d, b) and (e, d, a). Therefore their supersets (e, d, c, b, a), (e, d, c, a) and (e, d, b, a) cannot be frequent either. The corresponding parts of the trie are pruned and the resulting structure is the one from figure 5.5.

5.6 FP-growth and Apriori

The FP-growth algorithm was presented with the claim that it avoids one of Apriori's important drawbacks: the candidate generation process. Indeed it

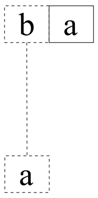

Figure 5.7: An example of the Depth-First algorithm – item *b*

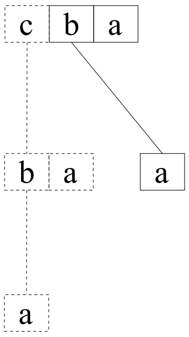

Figure 5.8: An example of the Depth-First algorithm – item *c*

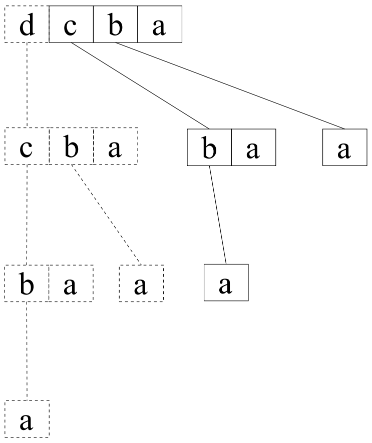

Figure 5.9: An example of the Depth-First algorithm – item *d*

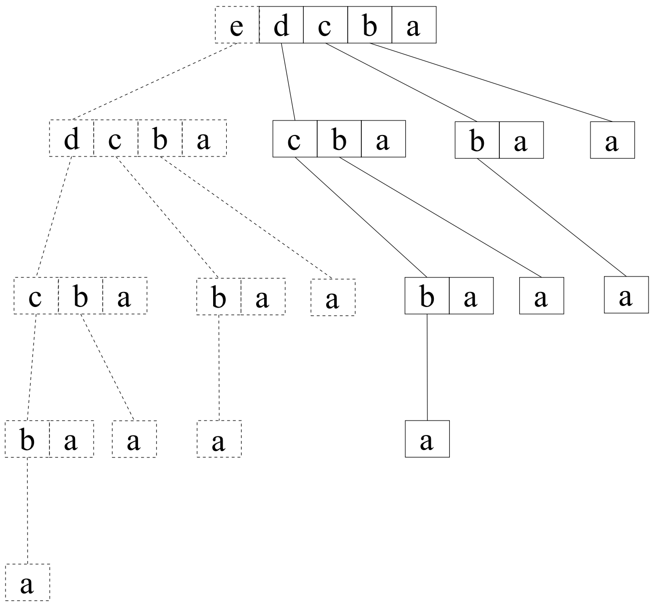

Figure 5.10: An example of the Depth-First algorithm – item e

has been pointed out that in some circumstances Apriori needs to generate a huge number of candidates and subsequently scans the database repeatedly to count their support. For example, if a frequent pattern of length 100 is present in the data then the number of candidates rises up to 2^{100}. However it is an interesting question whether indeed FP-growth generates no candidates. In order to investigate the problem we take a different view on the algorithm.

Let us consider another representation of the mining process – the *recursion tree*. It is in fact a trie with all frequent 1-itemsets in the root. Every path in the final trie starting from the root is a frequent itemset. Furthermore the items appearing in the same node have been generated in the same step of the recursion and a child node is always a step deeper in the recursion from its parent.

In order to explain clearer how the recursion tree is built and how it compares to the FP-tree let us go back to the example of table 5.2 and figure 5.3.

In the first step of the algorithm we count the supports of the 1-itemsets. Then we build the tree and start the mining procedure. Note that all items that appear in the FP-tree are already proven to be frequent. The recursion starts with the item g. By extracting the conditional pattern base we actually extract all the transactions from the original database (although in a condensed form) that contain the item g. We count the supports and as a result know those

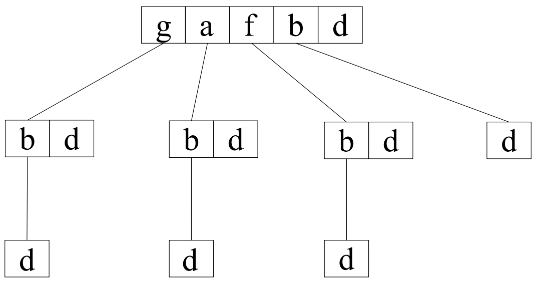

Figure 5.11: The underlying recursion tree for the example of table 5.2 and figure 5.3

items that are frequent together with g, i.e., d and b. Therefore we have just showed the itemsets (d, g) and (b, g) to be frequent and thus they appear in the recursion tree – d and b are added as a child node for g.

Here we build the conditional FP-tree which contains a single path. However for the sake of clarity we ignore the rule that we can extract directly all combinations of items in the path and proceed as before (the final results don't change). Now we go one level deeper in the recursion and investigate the item b. From the conditional database it turns out that d is frequent and it is added as a child of b. We now go back one level up in the recursion and proceed with d, which does not bring any new results, so we return one more level up and focus on the next item in the root node, i.e., a.

The final trie (or the recursion tree) is given in figure 5.11 without the support counts of the cells.

Now let us go back to the discussion concerning candidates generation. Each node in the recursion tree corresponds to a (conditional) database scan and counting of the support of all candidate items to be included in the node. For all those items (that appear in such a database) we know that they have been frequent on the previous step with the corresponding prefix and they appear in the node together with the parent. If we look closer at what actually happens at each step in the recursion we discover that at every node a number of candidates are considered and their support counted. For each candidate of length k, $k > 1$, we already know that two of its subsets of length $k - 1$ are frequent, otherwise it will not appear as a candidate. Let us illustrate it with an example.

Going back to the recursion tree of figure 5.11 we focus on the child node of the root-item g. The item candidates at this step are a, f, b and d because they are present in the FP-tree. We already know that they are frequent and they are listed as neighbours to the right of g in the root-node. By extracting the conditional pattern database of g we discover which ones actually appear together with g in a transaction. By counting their support we select the frequent ones (together with g). These are b and d.

We consider b – its candidate is the only item that appears in the conditional tree, d (which means the itemset (d, b, g)). What do we know about this candidate? One of its subsets of length $k-1$ is frequent, i.e., the path from the root to the parent, namely (b, g). We also know that (d, g) is frequent, which is another $k-1$ subset. In fact if it was not frequent we would not have considered (d, b, g) as a candidate as that would mean that d does not appear in the conditional FP-tree of b.

Proceeding in the same way we realize that at each step, for each candidate we know that *at least two subsets of length $k-1$ and identical prefix of length $k-2$ are frequent.* But that is very similar to how Apriori works before the pruning step is applied.

What is then the improvement in FP-growth?

One of the important improvements is the use of the FP-tree in order to reduce the amount of data which needs to be scanned at each counting step. Instead of going through the whole database we scan only the path in which the prefix appears. Furthermore the transactions are listed in an efficient way by joining those that are identical and processing them together.

5.7 Depth-First and Apriori

In this section we try to answer the similar question for Depth-First: how does it connect to Apriori. In this case the answer can easily be seen in the examples given earlier.

Looking at the example of figures 5.7 to 5.10 we see that the candidates at a particular step are all itemsets confirmed as frequent at that moment, appended to the new item. Therefore for each candidate of length k we know that at least one subset of length $k-1$ is frequent – the one that excludes the new item. However we know nothing more for the rest of the $k-1$ subsets as at that moment no connection has been investigated with the new item. In fact the only thing that we know about this item is that it is a frequent 1-itemset which is guaranteed by the preprocessing step of the algorithm.

Therefore Depth-First is not a full implementation of Apriori since for each candidate we only know that one subset of length $k-1$ and the remaining part of length 1 are frequent.

The difference compared to Apriori is that at each step we are not limited to counting only itemsets of length k but also longer patterns. The number of database scans is equal to the number of frequent 1-itemsets.

5.8 Theoretical Comparison of FP-Growth and Depth-First

For the theoretical analysis of the two algorithms we use the following notation (as in [36] and [51]). Let the set of items be $\mathcal{I} = \{1, 2, \ldots, n\}$, where we denote each item by an integer number for simplicity. The database contains m transactions. Usually m is much larger than the number of items n. The threshold of minimal support of a frequent itemset is denoted by $minsup$.

We assume that each item is known to be frequent. That is ensured by the first step of both algorithms to count the support of the 1-itemsets, which we consider as a preprocessing step.

Let $A \subseteq \mathcal{I}$ be a non-empty itemset. Then we define:

- $supp(A)$ is the support of A;
- $sm(A)$ is the smallest number/item in A;
- $la(A)$ is the largest number/item in A.

Further we define for the special case of the empty itemset, denoted by $\emptyset$:

- $supp(\emptyset) = m$;
- $sm(\emptyset) = n + 1$.
- $la(\emptyset) = 0$;

In the complexity analysis we consider two important factors for comparison: the number of database queries and the number of nodes in the data structure(s). A database query is defined as a question of the form “Does customer C buy product P?” or in other words “Does transaction T contains item I?”.

Intuitively both factors influence the runtime of the algorithms and the used memory. A better explanation will be given in the following subsections for the case of each algorithm.

5.8.1 The Complexity of FP-Growth

In order to make the analysis clearer we choose for a different representation of the FP-growth algorithm. Following our new notation we assume without loss of generality that the item numbers are ordered according to the support of the corresponding 1-itemsets. In other words, in $\mathcal{I} = \{1, 2, \ldots, n\}$ item 1 is less frequent than item 2 which is less frequent than item 3, etc. Therefore the same order will be followed when sorting the transactions for inserting in the FP-tree and the branches of the FP-tree will have the items with larger numbers closer to the root.

```
**FP-growth**(itemset $B$, database $D$):
    forall $i > la(B)$ with $B \cup \{i\}$ frequent
        $D'$ = the subset of transactions in $D$ that support $i$;
        count support for all items $k$ from $D'$ with $k > i$;
        remove infrequent items and items $k$ with $k \leq i$ from $D'$;
        build new FP-tree for $D'$;
        if there are items left
            call FP-growth($B \cup \{i\}$,$D'$);
```

Figure 5.12: A different representation of the FP-growth algorithm

Bearing this in mind we can represent the FP-growth procedure as given in figure 5.12 where we have skipped the printing of the results.

FP-growth is a very complex algorithm which is difficult to analyze. In order to make the picture clearer the following assumptions are made:

– In the original algorithm, the transactions that are identical are joined and processed together. In the experiments reported in section 5.9 this did not happen too often. Therefore in our analysis we assume that this does not occur (the transactions are never joined).

– When the database is detected to contain only a single path, the recursion terminates and outputs all combinations of items in the path. This also did not often occur in our experiments. Thus we assume that instead of terminating the recursion goes on as normal.

– At every step of the recursion the items in the new sub-database are again sorted according to their new supports and not according to the order found at the first step. In the experiments this refinement did not influence the results significantly and therefore we safely assume that only the first order is used at every step.

Let us now go back to the notion of database query. Here the question "Does customer C buy product P?" is posed to the local database used at the "count support" step of the algorithm from figure 5.12. That local database corresponds to the current conditional pattern base.

Note that the preprocessing step when the support of the 1-itemsets is counted makes exactly mn queries to the original database. This number is not included in the calculations for the total number of queries.

In this case the following theorem gives an insight into the number of database queries that the algorithm performs.

Theorem 12 *The number of database queries for the FP-growth algorithm except for the preprocessing step equals:*

$$\sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{\substack{j = la(A)+1 \\ \{j\} \cup A \setminus \{la(A)\} \text{ frequent}}}^{n} supp(A).$$

Proof: In order to prove the correctness of the formula we again consider the underlying recursion tree. Every step of the recursion corresponds to a cell/bucket in this tree. Every cell represents an itemset A given by the path from the root to the cell. For this itemset we already know that it is frequent. We also know that, due to the initial sorting of the transactions, the current cell contains the "largest" item in A, or $la(A)$.

Referring to the discussion in section 5.6, we know that the candidates for new frequent itemsets are the right neighbours of the current cell for all of which we already know that they (the corresponding paths) are frequent. Taking this into account it is easy to see that the formula is correct.

The first sum sums over each step of the recursion or each cell in the recursion tree. The second sum goes over each right neighbour of the current cell. For both sums we add the support of A as that determines the number of all transactions containing A and those are exactly the transactions that have to be checked for the new items. □

The second important factor we would like to examine is the number of nodes created during the run of the algorithm. By nodes here we mean not the nodes of the recursion tree but the real nodes of the original FP-tree and all conditional FP-trees necessary for the mining process.

Again FP-growth proves to be difficult to analyze. One reason for this is the large number of nodes created and destroyed on the fly. Another reason is the procedure of partly joining transactions having the same prefix and thus reducing the number of nodes in the tree.

Assuming that no prefixes are joined in the tree, we establish an upper bound for the real number of nodes that will be created by the algorithm.

Theorem 13 *Taking into account the previously mentioned assumptions, the number of FP-tree nodes created is at most equal to:*

$$\sum_{A \text{ frequent}} \sum_{\substack{j = la(A)+1 \\ A \cup \{j\} \text{ frequent}}}^{n} supp(A \cup \{j\}).$$

Proof: It is easy to see that the contribution of the original FP-tree equals to:

$$\sum_{i=1}^{n} supp(\{i\}).$$

Note that we consider only frequent items in our set $\mathcal{I}$.

The nodes of the conditional FP-trees amount to the number of items in the conditional pattern databases constructed in the mining process. That number is equal:

$$\sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \quad \sum_{\substack{j = la(A)+1 \\ A \cup \{j\} \text{ frequent}}}^{n} supp(A \cup \{j\})$$

which can easily be justified. For each frequent itemset A we construct a conditional pattern database containing only items that are larger than the largest item in A. Those items are inserted in the tree only if they are frequent in this database or in other words if the corresponding pattern $A \cup \{j\}$ is frequent. Each of the inserted items contributes its support to the sum, or $supp(A \cup \{j\})$.

Therefore the total number is the sum of the two:

$$\sum_{i=1}^{n} supp(\{i\}) + \sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \quad \sum_{\substack{j = la(A)+1 \\ A \cup \{j\} \text{ frequent}}}^{n} supp(A \cup \{j\}) =$$

$$= \sum_{A \text{ frequent}} \quad \sum_{\substack{j = la(A)+1 \\ A \cup \{j\} \text{ frequent}}}^{n} supp(A \cup \{j\}).$$

□

Since this is only an upper bound for the number of nodes, in the experiments in section 5.9 we count both the number of nodes according to the formula and the real number of nodes.

5.8.2 The Complexity of Depth-First

The Depth-First algorithm searches for frequent patterns by first extending the trie with the new candidates and then counting their support. The infrequent patterns are removed from the trie and the algorithm proceeds in the same way with the next root item. The counting of the support of the candidates is an important step which determines to a large degree the performance of the algorithm as it involves scanning the database.

The counting is performed by "pushing" the transactions one by one through the sub-trie and incrementing the counters of those cells (or patterns) that are present in the transaction. Only after the whole database is scanned do we have enough information on which patterns are infrequent and should be removed. This leads to the following specific feature of the algorithm. If a large number of frequent itemsets of length k are present in the database but only a few (or none) of the itemsets of length $k+1$ are frequent we still have to count a huge number of candidates of length $k+1$ at each step of the algorithm. In Apriori those candidates will be counted in a single scan of the database.

The difference in the performance in this case will depend on how the number of items, n, compares to k. When n is much larger than k then Depth-First will be forced to count a large number of "false" candidates and perform n scans of the database that will not produce longer patterns, while Apriori will only have to perform about k scans. If, however, a high enough number of longer patterns are present then Depth-First might prove to be more efficient by producing a number of patterns of various length at each scan. We come back to this point in the experimental comparison of FP-Growth and Depth-First in section 5.9.

Going back to the notion of database query, in this case we define it as a question of the type "Does customer C buy product P?" (or "Does transaction T contain item I?") posed to the original database. Here again the preprocessing step of counting the support of the 1-itemsets makes mn queries in the database. This number is not included in the total calculations.

The number of database queries as defined above for the Depth-First algorithm is given in the following theorem:

Theorem 14 *The number of database queries for the Depth-First algorithm is equal to:*

$$m(n-1) + \sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{j=1}^{sm(A)-1} supp(\{j\} \cup A \backslash \{la(A)\}).$$

Proof: First we notice that the component $m(n-1)$ is the contribution of the 1-itemsets. For each of them except for the last one we try to locate them in every transaction in order to determine whether the transaction has to be "pushed" down in the sub-trie or not.

The rest of the formula is contributed by the candidate itemsets. The first sum runs through all frequent itemsets A – those will determine the candidates that have to be counted in the next steps. They will appear in each sub-trie copied under a root cell to the left of them. In other words, if $(4,5,6)$ proves to be a frequent itemset at the step of item 4, then we always have to check $(3,4,5,6)$ as well as $(2,4,5,6)$ and $(1,4,5,6)$ at the later steps.

The number of these appearances is counted by the internal summation which runs through all root items to the left of the current one. The contribution of each one of them equals $supp(\{j\} \cup A \backslash \{la(A)\})$. This number determines the support of the parent cell of A which shows how many times a transaction will be "pushed" as deep as A – this portion of the transaction will have to be checked for the last item in A as well. □

It is easy to see that the contribution of the 1-itemsets in the number of database queries for both algorithms is the same and is equal to:

$$\sum_{i=1}^{n}(n-i)supp(\{i\}).$$

We now focus on the notion of number of nodes. In the context of Depth-First a node is any cell in a node of the trie that needs to be created during the run of the algorithm (including the candidates that will later be deleted). Defined in this way, the total number of nodes is given in the following theorem.

Theorem 15 *The number of nodes for the Depth-First algorithm equals:*

$$\sum_{A \text{ frequent}} [sm(A)-1].$$

Proof: The root node contains n cells. Further we notice that the contribution of all children cells in the trie is given by:

$$\sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{j=1}^{sm(A)-1} 1,$$

which counts, for each cell, how many times it will be copied under a root item to the left in the trie. Therefore the total number is:

$$\begin{aligned} & n + \sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{j=1}^{sm(A)-1} 1 = \\ & = n + \sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} [sm(A)-1] = \\ & = \sum_{A \text{ frequent}} [sm(A)-1]. \end{aligned}$$

□

5.9 Experimental Comparison of FP-Growth and Depth-First

In the experiments, four data sets were used. Three of them are artificial, generated by the well-known Almaden software (see [3]). The fourth data set is a sample of real-life data collected from a big retail chain. Details about the data can be found in appendix A.

The goal of the experiments is to give an insight into the "real" complexity of the algorithms measured by the execution time, by comparing it to the theoretically determined complexity discussed in the previous section. We try to find out where the algorithms' performance differs and whether the theoretical predictions go inline with the practical results.

The experiments were conducted at a Pentium-III machine with 256 MB memory at 733 MHz, running Windows NT. The programs were developed under the Borland C++ 5.02 environment, but are also usable with the GNU C++ compiler.

The results are shown in tables 5.3 to 5.6 where the following indicators are compared for the minsup thresholds of 0.5%, 1.0%, 1,5%, 2.0% for the synthetic data and 0.25%, 0.5%, 0.75%, 1.0% for the retail data set:

- the number of frequent 1-itemsets,
- the total number of frequent itemsets (including the frequent 1-itemsets),
- the execution time for Depth-First,
- the execution time for FP-growth,
- the number of queries for Depth-First according to the formula from theorem 14,
- the number of queries for FP-growth according to the formula from theorem 12,
- the number of nodes for Depth-First according to the formula in theorem 15,
- the real number of FP-tree nodes created during the run of the program,
- the number of nodes for FP-growth as given by the formula from theorem 13.

The experimental results given in tables 5.3 to 5.6 show some interesting phenomena. One of them is the surprising difference between the performance of the two algorithms for the retail data set at support 0.25%. This is a result of the different character of this real-life data set which turns out to be a worst-case scenario for the Depth-First algorithm. Compared to the artificial data sets, the retail data contains a lot more frequent 1-itemsets but few longer frequent itemsets. For Depth-First this results in a huge number of candidate 2-itemsets that have to be counted but turn out to be infrequent. This is also reflected in the difference between the small number of tree nodes for FP-growth and the large number of trie nodes for Depth-First for this particular case.

In fact the number of candidate 2-itemsets (influenced by n) affects the performance of the Depth-First algorithm in one more way through the procedure

D100T20I6				
minsup	0.5%	1.0%	1.5%	2.0%
# fr. 1-itemsets	730	561	435	358
# fr. itemsets	25,040	1,090	493	361
$\mathcal{DF}$ exec. time	29	11	8	6
$\mathcal{FP}$ exec. time	22	14	11	9
# queries $\mathcal{DF}$	804,898,976	427,097,723	301,111,212	234,051,516
# queries $\mathcal{FP}$	442,562,012	329,761,608	252,044,086	198,043,844
# nodes $\mathcal{DF}$	10,558,723	401,703	116,816	65,303
# nodes-real $\mathcal{FP}$	1,961,267	1,479,935	1,322,362	1,188,134
# nodes-formula $\mathcal{FP}$	17,071,465	2,442,218	1,740,313	1,514,332

Table 5.3: Experimental results for the artificial data set D100T20I6

D100T20I4				
minsup	0.5%	1.0%	1.5%	2.0%
# fr. 1-itemsets	688	559	439	369
# fr. itemsets	9,438	2,046	548	382
$\mathcal{DF}$ exec. time	23	12	8	6
$\mathcal{FP}$ exec. time	19	14	12	10
# queries $\mathcal{DF}$	731,815,487	446,136,987	315,448,527	251,949,116
# queries $\mathcal{FP}$	417,996,012	338,850,828	262,917,783	213,788,927
# nodes $\mathcal{DF}$	3,912,166	676,159	134,778	72,546
# nodes-real $\mathcal{FP}$	1,908,370	1,544,726	1,389,513	1,267,268
# nodes-formula $\mathcal{FP}$	8,629,486	3,663,093	1,876,177	1,596,385

Table 5.4: Experimental results for the artificial data set D100T20I4

D100T20I2				
minsup	0.5%	1.0%	1.5%	2.0%
# fr. 1-itemsets	635	527	445	359
# fr. itemsets	5,435	1,415	708	441
$\mathcal{DF}$ exec. time	20	11	9	6
$\mathcal{FP}$ exec. time	20	15	14	12
# queries $\mathcal{DF}$	757,631,634	432,277,091	331,223,803	249,973,785
# queries $\mathcal{FP}$	396,861,892	321,545,022	271,150,238	210,231,042
# nodes $\mathcal{DF}$	2,512,289	456,566	174,620	83,544
# nodes-real $\mathcal{FP}$	2,265,841	1,584,755	1,476,595	1,324,171
# nodes-formula $\mathcal{FP}$	5,827,026	3,137,017	2,284,728	1,828,513

Table 5.5: Experimental results for the artificial data set D100T20I2

Retail data				
minsup	0.25%	0.50%	0.75%	1.00%
# fr. 1-itemsets	2,101	920	493	320
# fr. itemsets	3,058	1,086	536	330
$\mathcal{DF}$ exec. time	66	12	4	2
$\mathcal{FP}$ exec. time	7	4	3	2
# queries $\mathcal{DF}$	778,171,180	281,647,840	114,469,871	66,731,996
# queries $\mathcal{FP}$	605,209,096	213,361,480	82,454,945	46,351,860
# nodes $\mathcal{DF}$	4,148,742	571,763	142,475	54,486
# nodes-real $\mathcal{FP}$	828,336	545,504	383,085	290,001
# nodes-formula $\mathcal{FP}$	1,175,688	751,809	545,116	434,039

Table 5.6: Experimental results for the real-life retail data set

of copying. In this procedure, the current frequent itemsets are copied under the new root item to be counted. Even though the infrequent items are not copied, they are still traversed in order to find the frequent ones. Further experiments showed that this results in a high number of node-visits, which explains why for D100T20I6, *minsup* 0.5%, Depth-First creates more trie nodes than for the retail data at 0.25% and still for the retail data performs much worse.

A different picture can be seen in the figures for the three artificial data sets. There, due to the higher percentage of real frequent itemsets from the candidate itemsets, Depth-First performs better than FP-growth in most of the cases. Only for the lowest support in D100T20I6 and D100T20I4, does Depth-First show slower runtime due to the rapid increase in the number of trie nodes that are generated and visited.

The overall results of the experiments show the following picture of the comparison of the two algorithms. The recorded execution time is proportional to both the number of queries and the number of nodes of the data structures which confirms the relevance of the two factors chosen in the theoretical analysis.

Furthermore from the experiments one might conclude that Depth-First outperforms FP-growth for data sets with larger values of the ratio m/n. But if there are fewer transactions compared to the number of items, FP-growth seems better. The importance of the ratio m/n is that it partly determines the percentage of real frequent itemsets out of the generated candidates. This influences the algorithms in a different way, as was discussed in the previous section, and hence results in differences in the performance.

5.10 Conclusions

This chapter focused on the problem of frequent patterns generation for associate rules mining. More specifically, two of the best known algorithms from the literature on this subject, Depth-First and FP-growth, are compared both theoretically and practically with the goal to give more insight into their performance as well as into which factors influence their differences.

The theoretical analysis examines two important factors for the algorithms' performance: the number of database queries performed by the algorithm and the number of nodes in the data structures created during the run of the algorithm. Formulas for the calculation of each of those factors were established and proved in theorems 12, 13, 14 and 15.

Further, the algorithms were compared experimentally by applying them to four data sets, three artificial and one real-life data set. The experiments show that the execution time is proportional to both the number of queries and the number of nodes of the data structures. From the experiments one might conclude that Depth-First outperforms FP-growth for data sets with larger values of the ratio m/n. But if there are fewer transactions compared to the number of items, FP-growth seems to perform better.

The experiments also uncovered some interesting phenomena in the performance of the algorithms discussed in section 5.9.

As this is the first attempt at achieving a theoretical grip on complexity, a number of directions for future research are available. In the case of FP-growth, for example, three simplifying assumptions were made. It would be therefore interesting to investigate how those assumptions influence the result and perhaps to incorporate them in new extended formulas.

Appendix A

Data Sets Used for the Experiments

A.1 Bankruptcy Data Set

The bankruptcy data set used in some of the experiments is discussed in [74, 40]. The sample consists of 39 objects denoted by $F1$ to $F39$ – firms that are described by 12 financial parameters. To each company a decision value is assigned – the experts' evaluation of its category of risk for the year 1988.

The condition attributes denoted by $A1$ to $A12$ take integer values from 0 to 4 and can be described as follows:

$A1$=earnings before interests and taxes / total assets

$A2$=net income / net worth

$A3$=total liabilities / total assets

$A4$=total liabilities / cash flow

$A5$=interest expenses / sales

$A6$=general and administrative expense / sales

$A7$=manager work experience

$A8$=firm's market niche position

$A9$=technical structure-facilities

$A10$=organization-personnel

$A11$=special competitive advantage of the firm

$A12$=market flexibility

The first six attributes are quantitative (financial ratios) and the rest are qualitative. They are coded on the ordinal scale where 4 is better than 3, 3 is better that 2, etc.

The decision attribute is denoted by d and takes integer values in the range 0 to 2 where: 0 means *unacceptable*, 1 means *uncertainty* and 2 means *acceptable.*

Because of the small number of objects in the data set it can be printed here in its full size, see table A.1.

The data was first analyzed for monotonicity. The problem is obviously monotone (if one company outperforms another on all condition attributes then it should not have a lower value of the decision attribute). Nevertheless, one conflicting pair of data points was discovered, namely $F24$ and $F31$. It was treated as noise and closer observations suggested that more probably example $F24$ was noisy. For those experiments in which a monotone data set was required, this example was removed from the data set and was not considered further.

A.2 Nursery Data Set

The Nursery data set was obtained from the UCI repository of machine learning databases [19] and has been first presented in [58]. It is based on a real-life problem and classifies applications for nursery schools. It was generated by a hierarchical model developed with the help of experts using the expert system shell for decision making DEX [20]. The model was applied in practice and was subsequently used for several years in the 1980's in Ljubljana, Slovenia. Because of the large number of applications for nurseries at that time, an objective explanation for those rejected was necessary which was the reason for developing the model.

The experts divided the problem in three subproblems (also called intermediate concepts):

– occupation of parents and child's nursery (denoted by EMPLOY),

– family structure and financial standing (denoted by STRUCT_ FINAN),

– social and health picture of the family (denoted by SOC_ HEALTH).

The EMPLOY intermediate concept groups two input attributes:

– *parents* – parents' occupation,

– *has_ nurs* – child's nursery.

U	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	d
F1	1	1	1	1	0	2	4	2	4	3	1	3	2
F2	3	4	1	2	2	2	4	3	4	4	3	4	2
F3	2	4	0	0	1	1	4	2	4	4	2	4	2
F4	1	2	1	0	1	3	4	1	4	3	2	3	2
F5	2	3	2	1	1	1	4	2	4	4	2	4	2
F6	2	4	2	2	2	1	4	2	3	3	2	3	2
F7	2	4	1	2	3	3	4	3	3	4	2	4	2
F8	0	0	3	0	1	2	4	1	3	3	0	3	2
F9	2	3	2	2	1	3	3	1	3	2	0	2	2
F10	2	3	1	0	1	1	3	1	3	3	0	3	2
F11	1	4	0	0	2	3	3	2	3	3	2	3	2
F12	2	2	3	3	2	3	3	1	3	3	0	2	2
F13	0	0	1	0	0	2	3	1	3	3	0	3	2
F14	1	0	0	0	3	2	3	1	3	3	2	2	2
F15	1	2	1	0	0	1	3	3	3	3	1	4	2
F16	1	2	3	2	0	4	3	1	3	2	1	2	2
F17	1	1	1	0	0	3	3	3	3	3	1	3	2
F18	1	0	2	0	0	2	4	1	3	1	0	2	2
F19	1	0	1	0	0	2	3	1	3	3	1	3	2
F20	1	0	1	0	0	4	3	1	3	3	1	3	2
F21	1	0	0	0	0	2	1	1	3	3	1	2	1
F22	0	0	2	0	1	0	2	3	3	3	2	3	1
F23	1	0	1	0	0	1	3	2	2	1	0	1	1
F24	0	0	0	0	0	0	2	1	3	3	1	2	1
F25	1	1	1	0	0	2	2	1	3	3	1	2	1
F26	1	1	0	0	0	2	1	1	3	3	1	2	1
F27	1	0	1	0	0	2	1	1	3	3	1	3	1
F28	0	0	3	0	2	0	1	1	2	2	0	1	1
F29	2	3	3	2	1	2	2	3	3	3	2	3	1
F30	2	0	2	2	0	1	1	2	3	3	1	2	1
F31	0	0	1	0	0	0	2	2	3	3	1	2	0
F32	2	4	1	0	0	0	2	1	2	3	0	2	0
F33	1	1	0	0	0	0	2	2	2	3	2	3	0
F34	1	0	0	0	0	0	1	1	2	3	2	3	0
F35	0	0	1	0	0	0	2	0	3	2	0	1	0
F36	0	0	2	0	1	0	1	0	2	2	1	2	0
F37	0	0	0	0	0	0	1	1	3	3	1	2	0
F38	0	0	2	0	0	0	0	0	3	2	0	2	0
F39	1	0	0	0	0	0	0	0	1	0	0	1	0

Table A.1: The bankruptcy data set

The STRUCT_ FINAN subproblem contains one intermediate concept and two input attributes:

– STRUCTURE which is an intermediate concept defining the family structure and containing the following input attributes:

  - the form of the family, denoted by *form*,
  - the number of children, denoted by *children*

– *housing* – an input attribute describing the housing conditions of the family,

– *finance* – an input variable for the financial standing of the family.

The SOC_ HEALTH intermediate concept contains the following two input variables:

– *social* – an input attribute which describes the social conditions of the family,

– *health* – the health conditions of the family.

The Nursery data set used in our experiments has been derived from this model and the structural information has been removed so that the resulting data set contains eight input attributes. The attributes take the following values:

parents – usual, pretentious, great_pret

has_ nurs – proper, less_proper, improper, critical, very_crit;

form – complete, completed, incomplete, foster;

children – 1, 2, 3, more;

housing – convenient, less_conv, critical;

finance – convenient, invonv;

social – non-prob, slightly_prob, problematic;

health – recommended, priority, not_recom.

The decision attribute classifies the applicants in five groups according to how strongly it is recommended to accept their application: not_recom, recommend, very_recom, priority, spec_prior. They were coded in our representation of the data set as 0,1,2,3,4 respectively. The classes turned out to be not evenly distributed and class 1 and 2 were rare in the data set. Class 1 appeared in only two examples and had therefore hardly any influence on the induction process. The examples belonging to class 2 were about 2% of the data. Each of the other three class values (0,3,4) appeared in roughly 30% of the objects.

It is easy to see that the values of all attributes are ordered according to how inconvenient the situation is (e.g. for the *housing* attribute convenient < less_conv < critical). They were coded with numerical labels with lowest value 0 and following the natural order. For example the *housing* attribute was coded as convenient=0, less_conv=1, critical=2.

Furthermore the problem from which the data set was drawn is obviously monotone since the worse the situation of the family is the more recommended it is to accept the child's application. Nevertheless the data was checked for monotonicity and was confirmed to have no monotone inconsistencies.

The number of instances is 12960, which completely covers the input space. For our experiments samples were drawn randomly from the data set. Class 2 was drawn in most of the samples but was often too rare. That, for example, in the experiments on monotone decision trees sometimes resulted in trees that had only one leaf for that class. That leaf had few belonging points and was subsequently pruned from the tree even for low pruning thresholds.

A.3 Cars Data Set

The Cars data was also obtained from the UCI repository [19]. Similarly to the Nursery data set, the Cars data set was drawn from a hierarchical model with the structural information removed from the data. The model was developed for the demonstration of the DEX system [20] and was first presented in [21].

The model assigns evaluation to cars based on their characteristics from the point of view of the buyer. On the first level there are two subproblems (intermediate concepts):

- PRICE – the overall price of the car
- TECH – the technical characteristics of the car

The PRICE subproblem contains two input attributes:

- *buying* – the buying price of the car
- *maint* – the price of the maintenance

The TECH subproblem contains one intermediate concept and one input attribute:

- COMFORT – an intermediate concept which includes three input attributes:
    - *doors* – the number of doors,
    - *persons* – capacity in terms of persons to carry,
    - *lug_boot* – the size of luggage boot;

– *safety* – an input attribute describing the estimated safety of the car.

The six input variables take the following values:

buying – v-high, high, med, low

maint – v-high, high, med, low

doors – 2, 3, 4, 5-more

persons – 2, 4, more

lug_boot – small, med, big

safety – low, med, high

The examples are classified in four groups labelled as either unacc, acc, good or v-good. These values are clearly ordered and were coded in our representation as 0, 1, 2, 3. Our monotonicity analysis showed that all but one attribute behave monotonically (the classification function is monotone over those attributes). They were coded with integers with lowest value of 0 following the natural order. For example the attribute *maint* has values v-high $<$ high $<$ med $<$ low which were coded with 0, 1, 2, 3 respectively (note that the higher the maintenance costs of the car are the less attractive it is).

The remaining attribute, *doors* has values such that $2 < 3 < 4$ but $4 \not< $ 5-more. For our experiments we require monotone problems and therefore we decided to slightly modify the input space by ignoring the single value 5-more that behaves non-monotonically.

The original data set contains 1728 examples which covers the whole input space. After removing the 5-more value of the attribute *doors* with the corresponding data points, the data set was reduced to 1153 and was found to be monotone. The sample covers the whole modified input space. For the experiments random samples were drawn with the size of 200 examples.

A.4 Almaden Data Sets

The Almaden data sets (see [3]) are synthetic data samples generated using the software developed at IBM Almaden Research Center. The data mimics the transactions in the retailing environment. It contains a set of transactions represented using 0's and 1's where 0/1 codes absence/presence of the corresponding item in the transaction.

The generating program allows a large range of data characteristics. The parameters that can be used are as follows:

$|D|$ – the number of transactions,

$|T|$ – the average size of the transactions,

$|I|$ – the average size of the maximal potentially large itemsets,

$|L|$ – the number of maximal potentially large itemsets,

N – the number of items.

We do not give here a description of the algorithm used by the program. For more details the reader is referred to [3].

For our experiments on the comparison of the FP-Growth and the Depth-First algorithms for generating frequent patterns we used three synthetic data sets produced by the program. They were labelled as *D100.T20.I2*, *D100.T20.I4* and *D100.T20.I6* following the parameter values used to generate them.

The samples contain 100000 transactions with the number of different items equal to 1000. The average size was 20 items in a transaction, the number of maximal frequent itemsets was set to 2000 and 2, 4 or 6 respectively were chosen as the average size of the maximal potentially large itemsets.

A.5 Retail Data Set

The Retail data set is a sample drawn from a real life transaction data collected by a big retail chain and it was used in our exaperiments on frequent patterns generation in chapter 5. From the whole set only those customers were considered who bought not more than 100 items. The motivation for this was based on the observation that when more than 100 items were bought it is quite often not individual customers but small businesses such as restaurants, offices, etc. The sample that was drawn contains 59498 transactions and the number of different items is 4644.

The experiments showed that the Retail data set differs in some characteristics from the synthetic sets generated using the Almaden software (discussed in the previous section). The main difference is in the fact that the Retail data contains a much smaller percentage of frequent itemsets of length more than 1 out of the total number of frequent itemsets. In fact, for *minsup* equal to 1% only 10 frequent itemsets had length more than 1 and none had length more than 2 while 320 frequent 1-itemsets were found. For the same *minsup* the artificial data sets produce much larger percentage of longer frequent itemsets (529 out of 1090, 1487 out of 2046 and 888 out of 1415 respectively).

Appendix B

Experimental Results

In chapter 3 we discussed experiments meant to give insight into the behavior of the pre-pruning and post-pruning strategies for monotone decision trees. There only the results for the monotone case were given. The rest of the charts are presented in this appendix.

The data sets used are based on one of the samples from the Nursery data set (the second row from table 3.5). Since the Nursery data is monotone, the sample is also monotone. Further we generated six versions of the sample by adding increasing number of inconsistencies. That resulted in 2, 21, 28, 31, 40 and 42 conflicting pairs of data points. For these experiments we chose the first four sets in the series.

For each set we generated monotone decision trees using the full range of pre-pruning and post-pruning thresholds. For pre-pruning thais meant from a threshold of at least 1 point in a leaf up to the threshold for which the tree contains only one node. For post-pruning we start from a threshold of 1% misclassification rate up to the point where the tree contains only one node.

The indicators that we measured were the number of misclassified points over the whole data set and the number of nodes in the trees. The charts per set, indicator and pruning method are given in the following figures.

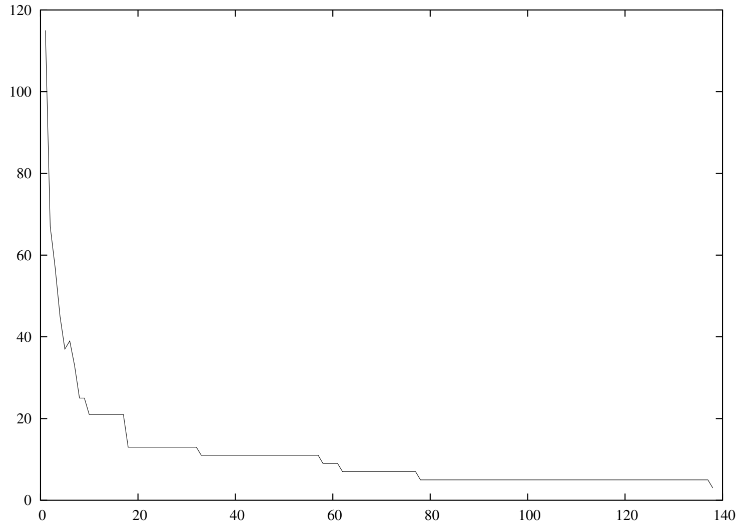

Figure B.1: 2 non-monotone pairs, number of nodes for all pre-pruning thresholds

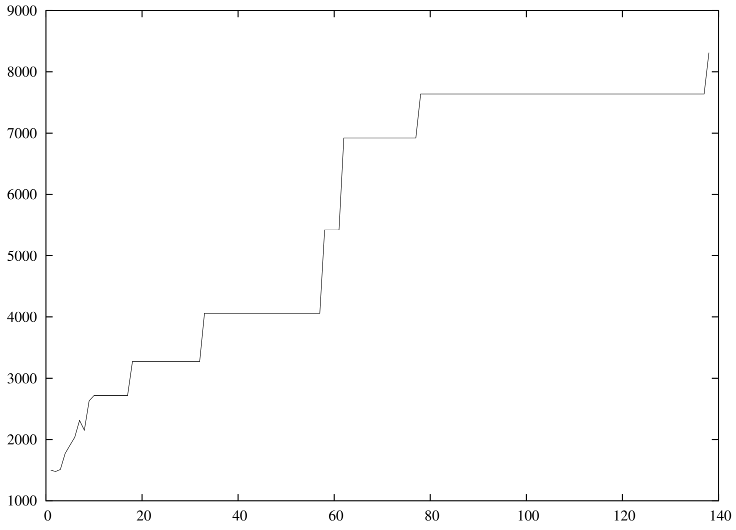

Figure B.2: 2 non-monotone pairs, number of misclassified points for all pre-pruning thresholds

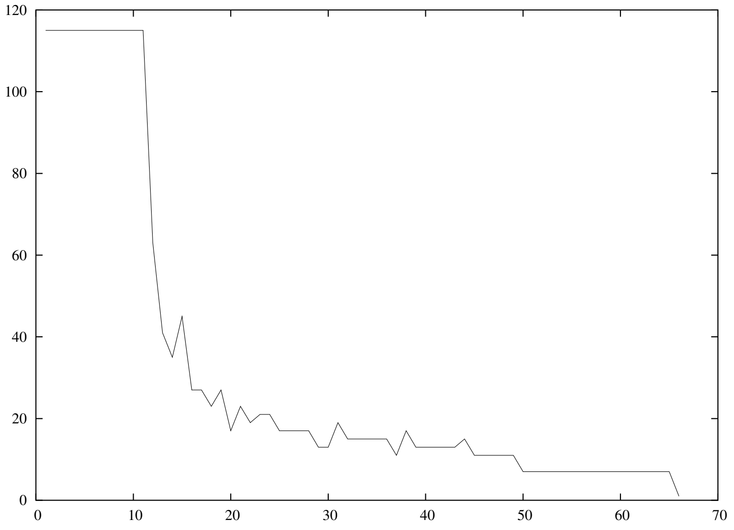

Figure B.3: 2 non-monotone pairs, number of nodes for all post-pruning thresholds

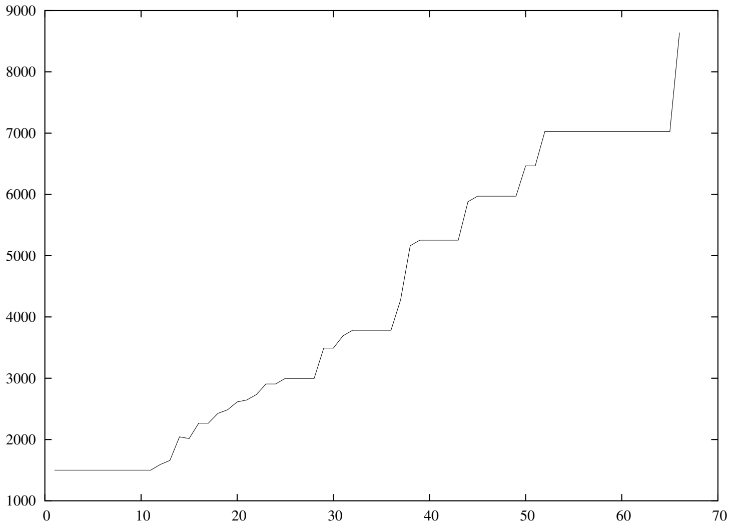

Figure B.4: 2 non-monotone pairs, number of misclassified points for all post-pruning thresholds

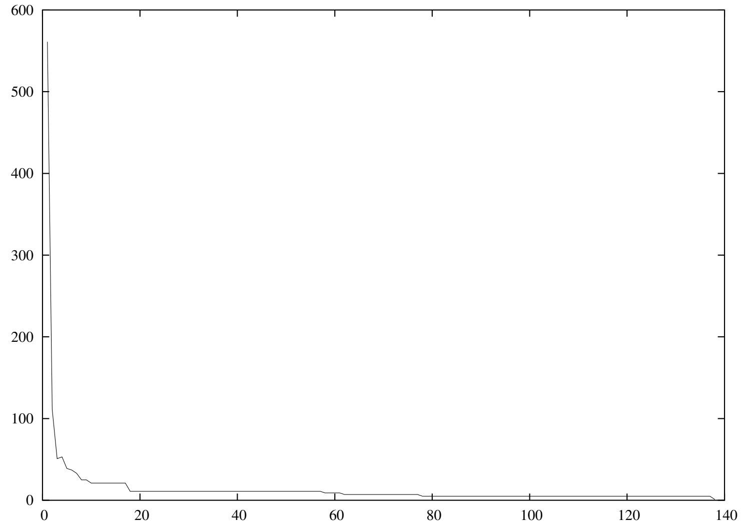

Figure B.5: 21 non-monotone pairs, number of nodes for all pre-pruning thresholds

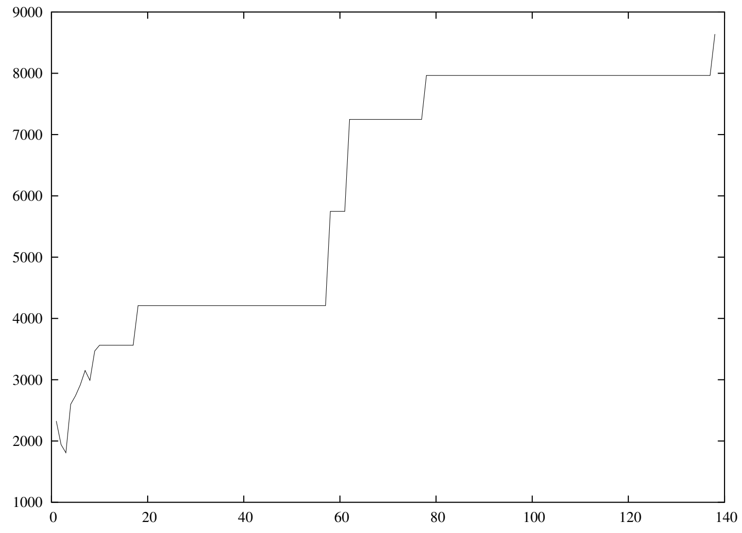

Figure B.6: 21 non-monotone pairs, number of misclassified points for all pre-pruning thresholds

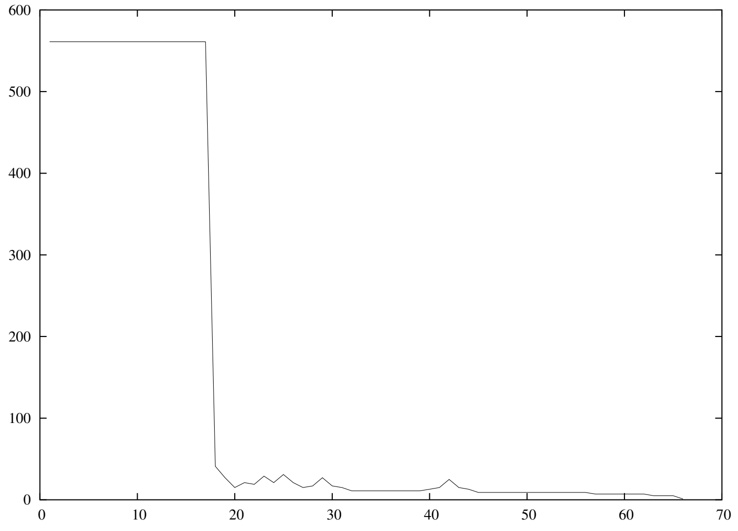

Figure B.7: 21 non-monotone pairs, number of nodes for all post-pruning thresholds

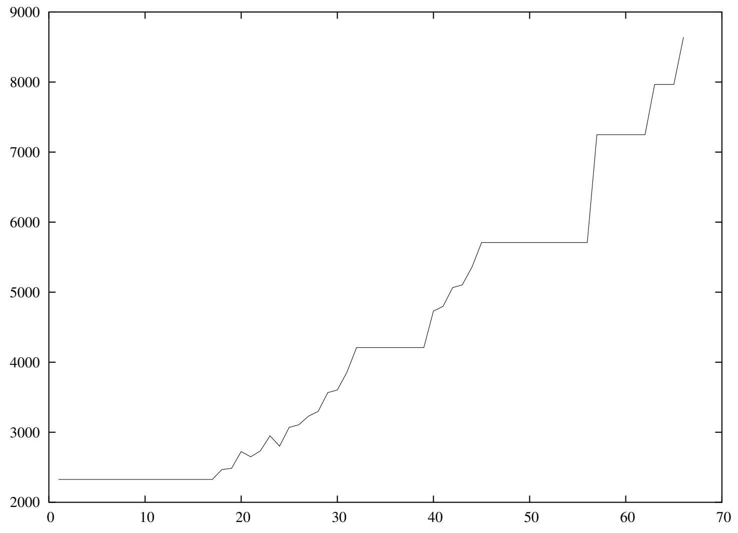

Figure B.8: 21 non-monotone pairs, number of misclassified points for all post-pruning thresholds

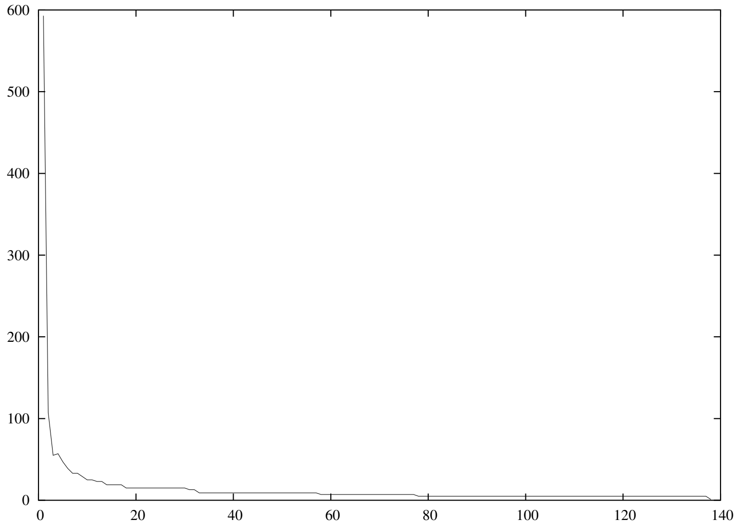

Figure B.9: 28 non-monotone pairs, number of nodes for all pre-pruning thresholds

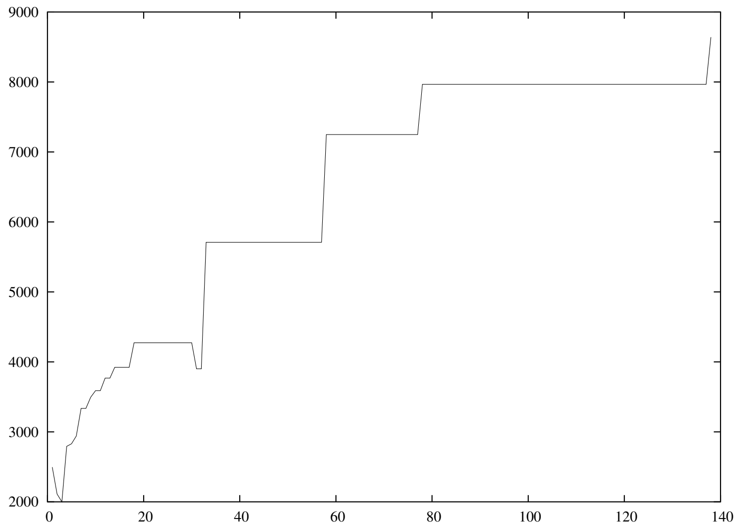

Figure B.10: 28 non-monotone pairs, number of misclassified points for all pre-pruning thresholds

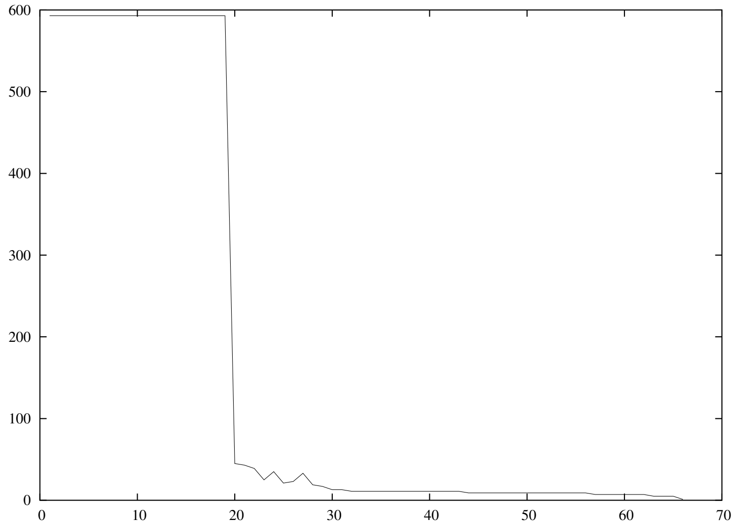

Figure B.11: 28 non-monotone pairs, number of nodes for all post-pruning thresholds

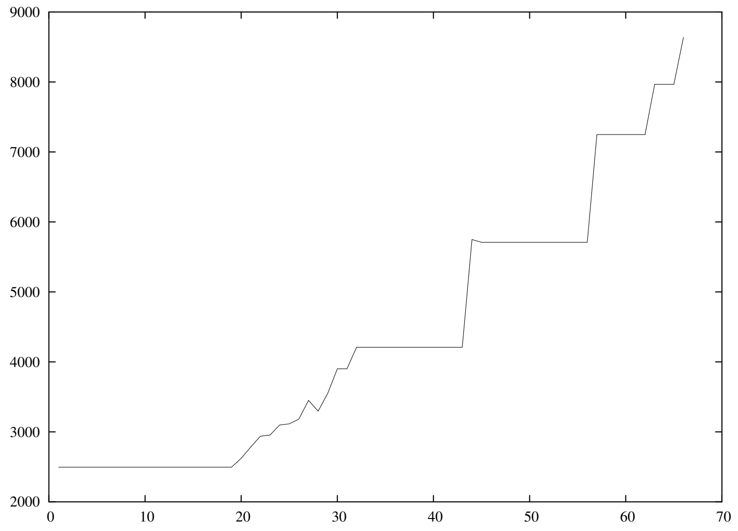

Figure B.12: 28 non-monotone pairs, number of misclassified points for all post-pruning thresholds

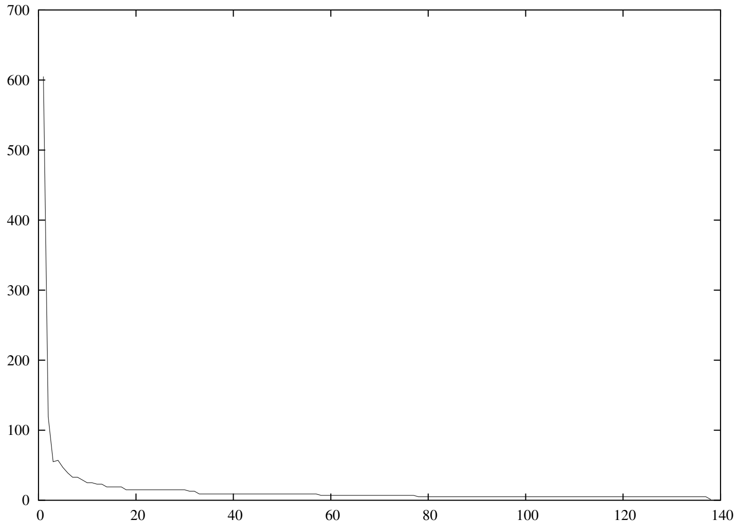

Figure B.13: 31 non-monotone pairs, number of nodes for all pre-pruning thresholds

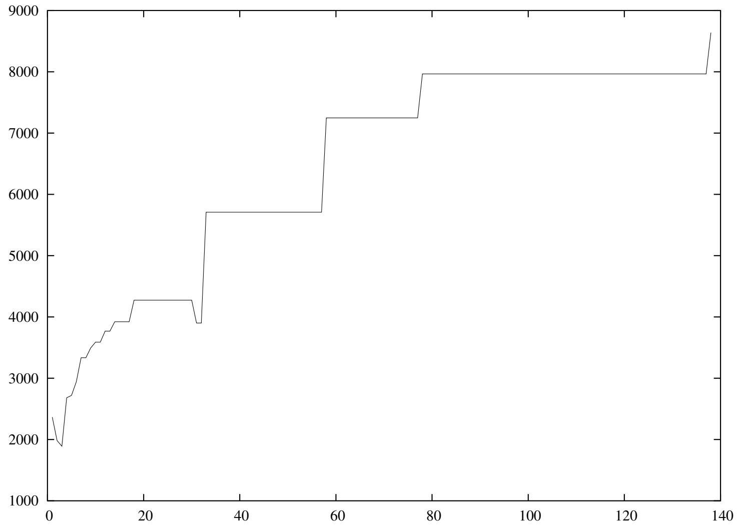

Figure B.14: 31 non-monotone pairs, number of misclassified points for all pre-pruning thresholds

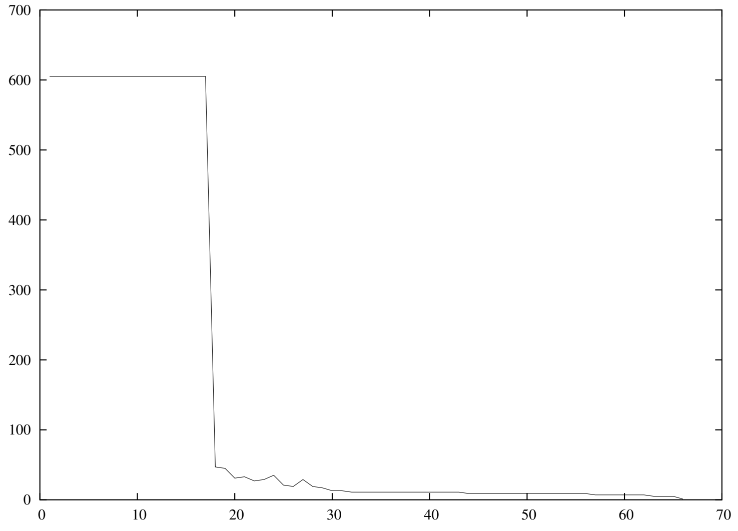

Figure B.15: 31 non-monotone pairs, number of nodes for all post-pruning thresholds

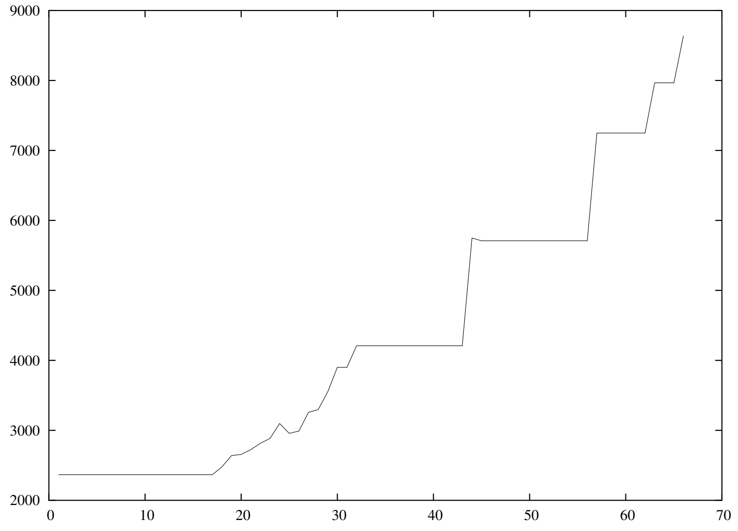

Figure B.16: 31 non-monotone pairs, number of misclassified points for all post-pruning thresholds

Summary

This thesis is positioned in the area of knowledge discovery with special attention to problems where the property of monotonicity plays an important role. Monotonicity is an ubiquitous property in all areas of life and has therefore been widely studied in mathematics. Monotonicity in knowledge discovery can be treated as available background information that can facilitate and guide the knowledge extraction process. While in some sub-areas methods have already been developed for taking this additional information into account, in most methodologies it has not been extensively studied or even has not been addressed at all. This thesis is a contribution to a change in that direction.

In the thesis, four specific problems have been examined from different sub-areas of knowledge discovery. The research results are organized in four separate chapters devoted to the four problems. The first three chapters are positioned in the area of classification. The starting point for classification is a training data set described by a number of attributes and a class attribute which represents the target concept we want to predict in new examples. The training data is used to build a classifier (a prediction model) which covers the whole input space and is able to predict the values of the target concept for new, previously unseen data.

The first chapter is focused on the rough sets methodology. An extension of the methodology is proposed for classification with monotone constraints. The approach can be used for attribute reduction of monotone data sets in such a way that the reduced data set remains monotone – this is achieved by means of generating monotone reducts. The proposed approach can further be used to generate rules that compose a monotone classifier.

The extension utilizes a variation of the discernibility matrix – the so called monotone discernibility matrix. It can provide a straightforward way for generating all monotone reducts or for using a heuristics for generating an approximation of a 'good' monotone reduct. For theoretical completeness a connection was also made to the notion of a positive area for which a new definition was given in the monotone case. The developed methods have a number of advantages over the only other existing approach for incorporating the monotonicity property in rough sets theory. In order to underline the differences, the same

data set was used in our experiments and the results were compared.

The second chapter focuses on monotone decision trees. It extends an existing algorithm that generates monotone decision trees within the area of classification. The original algorithm can only be applied on monotone data with no inconsistencies. In practice, however, noiseless data is hardly ever available. The proposed new methods allow the generation of monotone trees from any noisy and inconsistent data.

Furthermore methods for pre-pruning and post-pruning of the tree are developed so that it remains monotone. For this, special labelling functions are used to guarantee the monotonicity property of the pruned trees. The problem of monotone leaf labelling, however, is also treated as a separate topic in a more general setting and not only in connection with pruning. The proposed labelling functions can, for example, be used on any tree already generated by any other method such that the labels of the corners are known or can be computed and the monotonicity of the tree is required.

The chapter also includes empirical comparison between two available in the literature splitting criteria for decision trees in order to give more insight into which one performs better in the setting of a monotone problem. One of them is a traditional information gain splitting for general decision trees and the other one is oriented at reducing the monotone inconsistencies in the tree. The results show that the information gain splitting tends to produce smaller trees while the monotonicity-oriented one tends to generate more accurate trees.

The third chapter is in the area of functional decomposition applied to knowledge discovery. The goal is to decompose a complex function to a concept hierarchy that reflects the structure of the domain. In knowledge discovery the function is represented by a data set and the hierarchical structure built from it is further used for classification. In our research, the methodology is extended to handle monotone problems so that the decomposed function remains monotone.

We concentrate on the subproblem of determining whether there exists an extension of positive scheme of the type $f = g(S_0, h(S_1))$ for given subsets S_0 and S_1 of the set of attributes. The function h here is the intermediate concept we need to introduce so that the decomposed function agrees with the original one. In our research we also propose methods for finding an extension of this type (when it exists) with minimal number of distinct values of the intermediate concept in order to reduce the complexity of the decomposed function.

The last chapter lies in the area of data mining and association rules generation. It focuses on the first part of the problem – frequent patterns discovery. Here again the monotonicity property plays a role as a property of the set of infrequent patterns while the set of frequent patterns is anti-monotone. This knowledge can be used in reducing the number of candidate patterns that need to be counted and therefore helps speed up the frequent patterns generation algorithms.

In our research, two of the best algorithms in the literature are considered, namely FP-growth and Depth-First implementations of Apriori. They are compared theoretically and empirically in order to give more insight into their differences and how certain features of the data sets can influence the performance of both algorithms. In the theoretical analysis complexity formulas were derived for counting two of the most important factors for the performance of the algorithms: the number of nodes created in the data structures and the number of queries to the database during the run of the algorithm on the current data.

In the experimental research those formulas were tested against the actual counts as well as the runtime performance of the algorithms for data sets with different characteristics. It was shown that the proposed formulas are adequate for counting the two factors. The results were also analyzed in order to explain some interesting phenomena in the performance of the algorithms on the four data sets used for the experiments.

Bibliography

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 207–216, 1993.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U.M. Fayaad et al., editor, *Advances in Knowledge Discovery and Data Mining*, chapter 12, pages 307–328. AAAI/MIT Press, 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J.B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings 20th International Conference on Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 1994.

[4] E.I. Altman. Financial ratios, discriminant analysis and the prediction of corporate bankruptcy. *The Journal of Finance*, 4:589–609, 1968.

[5] R.L. Ashenhurst. The decomposition of switching functions. In *Proceedings on an International Symposium on the Theory of Switching*, pages 74–116, 1959.

[6] B. Back, T. Laitinen, K. Sere, and M. Van Wezel. Choosing bankruptcy predictors using discriminant analysis, logit analysis, and genetic algorithms. Technical Report 40, Turku Centre for Computer Science, 1996.

[7] J. Bazan, A. Skowron, and P. Synak. Dynamic reducts as a tool for extracting laws from decision tables. In *Proceedings of the Symposium on Methodologies for Intelligent Systems, Charlotte, NC*, Lecture Notes in Artificial Intelligence, pages 346–355. Springer-Verlag, 1994.

[8] Gilad Ben-Avi and Yoad Winter. Monotonicity and collective quantification. *Journal of Logic, Language and Information*, 12(2):127–151, 2003.

[9] A. Ben-David. Monotonicity maintenance in information-theoretic machine learning algorithms. *Machine Learning*, 19:29–43, 1995.

[10] A. Ben-David, L. Sterling, and Y. Pao. Learning and classification of monotonic ordinal concepts. *Computational Intelligence*, 5:45–49, 1989.

[11] J. C. Bioch. Dualization, decision lists and identification of monotone discrete functions. *Annals of Mathematics and Artificial Intelligence*, 24:69–91, 1998.

[12] J. C. Bioch and V. Popova. Rough sets and ordinal classification. In *Proceedings of the 11th International Conference on Algorithmic Learning Theory (ALT'2000)*, Lecture Notes in Artificial Intelligence, pages 291–305. Springer-Verlag, 2000.

[13] J. C. Bioch and V. Popova. Monotone classification and noisy data. Technical Report ERS-2002-53-LIS, Erasmus Research Institute of Management, http://www.erim.nl, 2002.

[14] J.C. Bioch and T. Ibaraki. Complexity of identification and dualization of positive boolean functions. *Information and Computation*, 123:50–63, 1995.

[15] J.C. Bioch and T. Ibaraki. Version spaces and generalized monotone boolean functions. Technical Report ERS-2002-34-LIS, Erasmus Research Institute of Management, http://www.erim.nl, 2002.

[16] J.C. Bioch and V. Popova. Labeling and splitting criteria for monotone decision trees. In M. Wiering, editor, *Proceedings of the 12th Belgian-Dutch Conference on Machine Learning (BENELEARN'2002)*, pages 3–10, Utrecht, 2002.

[17] J.C. Bioch and V. Popova. Monotone decision trees and noisy data. In H. Blockeel and M. Denecker, editors, *Proceedings of the 14th Belgium-Dutch Conference on Artificial Intelligence (BNAIC'2002)*, pages 19–26, Leuven, 2002.

[18] J.C. Bioch and R. Potharst. Decision trees for monotone classification. In K. van Marcke and W. Daelmans, editors, *Proceedings of the Dutch Artificial Conference on Artificial Intelligence (NAIC'97)*, pages 361–369, 1997.

[19] C. L. Blake and C. J. Mertz. UCI repository of machine learning databases. Irvine, CA: University of California, Department of Information and Computer Science [http://www.ics.uci.edu/~mlearn/ MLRepository.html], 1998.

[20] M. Bohanec and V. Rajkovič. DEX: An expert system shell for decision support. *Sistemica*, 1(1):145–157, 1990.

[21] M. Bohanec and V. Rajkovic. Knowledge acquisition and explanation for multi-attribute decision making. In *8th International Workshop on Expert Systems and Their Applications*, pages 59–78, Avignon, France, 1988.

[22] E. Boros, V. Gurvich, P.L.Hammer, T. Ibaraki, and A. Kogan. Decomposability of partially defined Boolean functions. *Discrete Applied Mathematics*, 62:51–75, 1995.

[23] E. Boros, P.L. Hammer, T. Ibaraki, and A. Kogan. Logical analysis of numerical data. RUTCOR Research Report RRR 04-97, Rutgers University, 1997.

[24] E. Boros, P.L. Hammer, T. Ibaraki, and A. Kogan. Logical analysis of numerical data. *Mathematical Programming*, 79:165–190, 1997.

[25] E. Boros, P.L. Hammer, T. Ibaraki, A. Kogan, E. Mayoraz, and I. Muchnik. An implementation of logical analysis of data. RUTCOR Research Report RRR 22-96, Rutgers University, 1996.

[26] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Wadsworth & Brooks, Monterey, 1962.

[27] L. Breslow and D. W. Aha. Simplifying decision trees: A survey. *Knowledge Engineering Review*, 12:1–40, 1997.

[28] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 255–264. ACM Press, 1997.

[29] K. Cao-Van. *Supervised ranking, from semantics to algorithms (to be published).* PhD thesis, University of Gent, Belgium, 2003.

[30] K. Cao-Van and B. De Baets. Growing decision trees in an ordinal setting. *submitted to International Journal of Intelligent Systems*, 2002.

[31] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[32] T.M. Cover and P.E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13:21–27, 1967.

[33] Y. Crama, P. L. Hammer, and T. Ibaraki. Cause-effect relationships and partially defined boolean functions. *Annals of Operations Research*, 16:299–326, 1988.

[34] H.A. Curtis. *A new approach to the design of switching functions.* Van Nostrand, Princeton, N.J., 1962.

[35] H. Daniels and B. Kamp. Application of MLP networks to bond rating and house pricing. *Neural Computation and Applications*, 8:226–234, 1999.

[36] J.M. de Graaf, W.A. Kosters, W. Pijls, and V. Popova. A theoretical and practical comparison of depth first and FP-growth implementations of Apriori. In H. Blockeel and M. Denecker, editors, *Proceedings of the Fourteenth Belgium-Netherlands Artificial Intelligence Conference (BNAIC 2002)*, pages 115–122, 2002.

[37] T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24:1278–1304, 1995.

[38] A.J. Feelders and M. Pardoel. Pruning for monotone classification trees. In M.R. Berthold et al., editor, *Advances in intelligent data analysis V*, volume 2810 of *Lecture Notes in Computer Science*, pages 1–12, Berlin, 2003. Springer-Verlag.

[39] M. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21:618–628, 1996.

[40] S. Greco, B. Matarazzo, and R. Slowinski. A new rough set approach to evaluation of bankruptcy risk. *C. Zopounidis (ed.), Operational Tools in the Management of Financial Risks, Kluwer, Dordrecht*, pages 121–136, 1998.

[41] S. Greco, B. Matarazzo, and R. Slowinski. Rough sets theory for multicriteria decision analysis. *European Journal of Operational Research*, 129:1–47, 2001.

[42] J.W. Grzimala-Busse. On the unknown attribute values in learning from examples. In *Proceedings of the Sixth International Symposium on Methodologies for Intelligent Systems (ISMIS'91)*, volume 542 of *Lecture Notes in Artificial Intelligence*, pages 368–377. Springer-Verlag, 1991.

[43] J.W. Grzymala-Busse and M. Hu. A comparison of several approaches to missing attribute values in data mining. In *Proceedings of the Second International Conference on Rough Sets and Current Trends in Computing (RSCTC'2000)*, pages 340–347, Banff, Canada, 2000.

[44] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufman Publishers, 2001.

[45] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, 2000.

[46] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for association rule mining - a general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, 2000.

[47] X. Hu and N. Cercone. Learning in relational databases: a rough set approach. *Computational Intelligence*, 11:323–338, 1995.

[48] D.S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[49] J. Komorowski, L. Polkowski, and A. Skowron. Rough sets: a tutorial. In S.K. Pal and A. Skowron, editors, *Rough-Fuzzy Hybridization: A New Method for Decision Making*, pages 3–98. Springer-Verlag, 1998.

[50] I. Kononenko, I. Bratko, and E. Roskar. Experiments in automatic learning of medical diagnostic rules. Technical report, Jozef Stefan Institute, Ljubljana, Yugoslavia, 1984.

[51] W. Kosters, W. Pijls, and V. Popova. Complexity analysis of depth-first and fp-growth implementations of apriori. In *Machine Learning and Data Mining in Pattern Recognition, Proceedings of the 3rd International Conference on Machine Learning and Data Mining (MLDM'2003)*, volume 2734 of *Lecture Notes in Artificial Intelligence*, pages 284–292, Leipzig, Germany, 2003.

[52] M. Leshno and Y. Spector. Neural network prediction analysis: The bankruptcy case. *Neurocomputing*, 10:125–147, 1996.

[53] P. Lingras. Unsupervised rough set classification using GAs. *Journal of Intelligent Information Systems*, 16(3):215–228, 2001.

[54] W.Z. Liu, A.P. White, S.G Thompson, and M.A. Bramer. Techniques for dealing with missing values in classification. In *Proceedings of Advances in Intelligent Data Analysis (IDA'97)*, volume 1280 of *Lecture Notes in Computer Science*, pages 527–536. Springer, 1997.

[55] K. Makino, T. Suda, K. Yano, and T. Ibaraki. Data analysis by positive decision trees. In *Proceedings International symposium on cooperative database systems for advanced applications (CODAS)*, pages 282–289, 1996.

[56] T. Mollestad. *A Rough Set Approach to Data Mining: Extracting a Logic of Default Rules from Data.* PhD thesis, Norwegian University of Science and Technology, 1997.

[57] S.H. Nguyen and A. Skowron. Searching for relational patterns in data. In *Principles of Data Mining and Knowledge Discovery*, pages 265–276, 1997.

[58] M. Olave, V. Rajkovic, and M. Bohanec. An application for admission in public school systems. In I.Th.M. Snellen, W.B.H.J. van de Donk, and J.-P. Baquiast, editors, *Expert Systems in Public Administration*, pages 145–160. Elsevier Science Publishers (North Holland), 1989.

[59] Z. Pawlak. Rough sets. *International Journal of Computer and Information Sciences*, 11:341–356, 1982.

[60] Z. Pawlak. *Rough Sets: Theoretical Aspects of Reasoning about Data.* Kluwer Academic Publisher, 1991.

[61] W. Pijls and J.C. Bioch. Mining frequent itemsets in memory-resident databases. In E. Postma and M. Gyssens, editors, *Proceedings of the Eleventh Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 1999)*, pages 75–82, 1999.

[62] P.P.M. Pompe and A.J. Feelders. Using machine learning, neural networks, and statistics to predict corporate bankruptcy. *Microcomputers in Civil Engineering*, 12:267–276, 1997.

[63] V. Popova and J.C. Bioch. Monotone function decomposition. Technical report, ERIM, http://www.erim.nl, 2004.

[64] R. Potharst and J. C. Bioch. Decision trees for ordinal classification. *Intelligent Data Analysis*, 4:1–15, 2000.

[65] R. Potharst and A.J. Feelders. Classification trees for problems with monotonicity constraints. *SIGKDD Explorations*, 4:1–10, 2002.

[66] J.R. Quinlan. Discovering rules by induction from large collections of examples. In D. Michie, editor, *Expert systems in the micro-electronic age.* Edinburgh University Press, 1979.

[67] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[68] J.R. Quinlan. Unknown attribute values in induction. In *Proceedings of the sixth international Machine Learning workshop*, pages 164–168. Morgan Kaufmann, 1989.

[69] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[70] K.G. Ramamurthy. *Coherent Structures and Simple Games.* Kluwer Academic Publishers, 1990.

[71] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:221–229, 1959.

[72] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *The VLDB Journal*, pages 432–444, 1995.

[73] A.D. Shapiro. *Structured induction in expert systems.* Turing Institute Press in association with Addison-Wesley, Wokingham, UK, 1987.

[74] R. Slowinski and C. Zopounidis. Application of the rough set approach to evaluation of bankruptcy prediction. *Intelligent Systems in Accounting, Finance and Management*, 4:27–41, 1995.

[75] T.K. Sung, N. Chank, and G. Lee. Dynamics of modelling in data mining: Interpretive approach to bankruptcy prediction. *Journal of Management Information Systems*, 16:63–85, 1999.

[76] S. Wang. A neural network method of density estimation for univariate unimodal data. *Neural Computation and Applications*, 2:160–167, 1994.

[77] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

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

[79] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In F. Provost and R. Srikant, editors, *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2001)*, pages 401–406, 2001.

[80] W. Ziarko. Variable precision rough set model. *Journal of Computer and System Sciences*, 46:39–59, 1993.

[81] B. Zupan. *Machine learning by function decomposition.* PhD thesis, University of Ljubljana, 1997.

[82] B. Zupan, M. Bohanec, J. Demsar, and I. Bratko. Learning by discovering concept hierarchies. *Artificial Intelligence*, 109(1-2):211–242, 1999.

Curriculum Vitae

Viara Popova was born in Bourgas, Bulgaria in 1972. She followed her secondary education at Mathematics High School "Nikola Obreshkov" in Bourgas. In 1996 she finished her higher education at Sofia University, Faculty of Mathematics and Informatics where she graduated with major in Informatics and specialization in Information Technologies in Education. She then joined the Department of Information Technologies, first as an associated member and from 1997 as an assistant professor.

In 1999 she became a PhD student at Erasmus University Rotterdam, Faculty of Economics, Department of Computer Science. In 2004 she joined the Artificial Intelligence Group within the Department of Computer Science, Faculty of Sciences at Vrije Universiteit Amsterdam as a PostDoc researcher.

Ganzaroli , A., Creating Trust between Local and Global Systems, (co-) Promotor(es): Prof.dr. K. Kumar & Prof.dr. R.M. Lee, EPS-2002-018-LIS, ISBN: 90-5892-031-3, http://hdl.handle.net/1765/361

Gilsing, V.A., Exploration, Exploitation and Co-evolution in Innovation Networks. (co) Promotor(es): Prof.dr. B. Nooteboom & Prof.dr. J.P.M. Groenewegen, EPS-2003-032-ORG, ISBN 90-5892-05-42, http://hdl.handle.net/1040

Graaf, G. de, Tractable Morality. Customer discourses of bankers, veterinarians and charity workers. (co-) Promotor(es): Prof.dr. F. Leijnse & Prof.dr. T. van Willigenburg. EPS-2003-031-ORG, ISBN 90-5892-051-8, http://hdl.handle.net/1038

Heugens, P.M.A.R., Strategic Issues Management: Implications for Corporate Performance, (co-) Promotor(es): Prof.dr.ing. F.A.J. van den Bosch & Prof.dr. C.B.M. van Riel, EPS-2001-007-STR, ISBN: 90-5892-009-7, http://hdl.handle.net/1765/358

Hooghiemstra, R., The Construction of Reality, (co-) Promotor(es): Prof.dr. L.G. van der Tas RA & Prof.dr. A.Th.H. Pruyn, EPS-2003-025-F&A, ISBN: 90-5892-047-X, http://hdl.handle.net/1765/871

Jong, C. de, Dealing with Derivatives: Studies on the role, informational content and pricing of financial derivatives, (co-) Promotor(es): Prof.dr. C.G. Koedijk, ISBN: 90-5892-043-7, http://hdl.handle.net/1043

Koppius, O.R., Information Architecture and Electronic Market Performance, (co-) Promotor(es): Prof.dr. P.H.M. Vervest & Prof.dr.ir. H.W.G.M. van Heck, EPS-2002-013-LIS, ISBN: 90-5892-023-2, http://hdl.handle.net/1765/921

Loef, J., Incongruity between Ads and Consumer Expectations of Advertising, (co-) Promotor(es): Prof.dr. W.F. van Raaij & prof. dr. G. Antonides, EPS-2002-017-MKT, ISBN: 90-5892-028-3, http://hdl.handle.net/1765/869

Meer, J.R. van der, Operational Control of Internal Transport, (co-) Promotor(es): Prof.dr. M.B.M. de Koster & Prof.dr.ir. R. Dekker, EPS-2000-001-LIS, ISBN:90-5892-004-6, http://hdl.handle.net/1765/859

Miltenburg, P.R., Effects of modular sourcing on manufacturing flexibility in the automotive industry. A study among German OEMs. (co) Promotor(es): Prof.dr. J. Paauwe & Prof.dr. H.R. Commandeur, EPS-2003-030-ORG, ISBN 90-5892-052-6, http://hdl.handle.net/1039

Mol, M.M., Outsourcing, Supplier-relations and Internationalisation: Global Source Strategy as a chinese puzzle, (co-) Promotor(es): Prof.dr. R.J.M. van Tulder, EPS-2001-010-ORG, ISBN: 90-5892- 014-3, http://hdl.handle.net/1765/355

Knowledge Discovery and Monotonicity

The monotonicity property is ubiquitous in our lives and it appears in different roles: as domain knowledge, as a requirement, as a property that reduces the complexity of the problem, and so on. It is present in various domains: economics, mathematics, languages, operations research and many others. This thesis is focused on the monotonicity property in knowledge discovery and more specifically in classification, attribute reduction, function decomposition, frequent patterns generation and missing values handling. Four specific problems are addressed within four different methodologies, namely, rough sets theory, monotone decision trees, function decomposition and frequent patterns generation. In the first three parts, the monotonicity is domain knowledge and a requirement for the outcome of the classification process. The three methodologies are extended for dealing with monotone data in order to be able to guarantee that the outcome will also satisfy the monotonicity requirement. In the last part, monotonicity is a property that helps reduce the computation of the process of frequent patterns generation. Here the focus is on two of the best algorithms and their comparison both theoretically and experimentally.

ERIM

ERIM ERASMUS RESEARCH INSTITUTE OF MANAGEMENT

www.erim.eur.nl

ISBN 90-5892-058-5

37

VIARA N. POPOVA Knowledge Discovery and Monotonicity

ERIM

VIARA N. POPOVA