Tallinn University of Technology
Faculty of Information Technology
Department of Informatics

Solving rule set optimisation problem of the MONSA family of algorithms and implementation in Java

Master's thesis

Martin Rebane

Supervisors:
Professor Rein Kuusik
Grete Lind, MSc

Tallinn 2014

Author's declaration

Herewith I declare that this thesis is based on my own work. All ideas, major views and data from different sources by other authors are used only with a reference to the source. The thesis has not been submitted for any degree or examination in any other university.

….....

May 22nd 2014

Martin Rebane

Solving rule set optimisation problem of the MONSA family of algorithms and implementation in Java

Abstract

Martin Rebane

Central research question for this thesis is to solve rule set optimisation problem of MONSA family of algorithms (MONSAMAX, MONSAMIN, and MONSABAN) and analyse the properties of MONSA and cover algorithms both in terms of quality of the solution and resource consumption of the implementation.

Two different cover algorithms for solving rule set optimisation problem are proposed. Greedy algorithm takes a mathematical approach to minimise the number of rules that are selected to the final cover. Approximation algorithm attacks the problem from the other side, selecting the best rule for each data object. They offer different business value to the end user.

Additionally, a unique coverage algorithm is introduced. This algorithm is meant to be used in conjunction with other rule set optimisation algorithms.

All algorithms are implemented in Java programming language. Thesis tests the performance of cover algorithms and of MONSA algorithms on different input datasets.

Finally, efficient handling of zeroes and null-values in the program code for MONSA algorithms is introduced. It allows to develop MONSA algorithms more efficiently than would be possible by using trivial implementation. All algorithms are implemented in Java.

Keywords: *MONSA algorithm, MONSAMAX, MONSABAN, MONSAMIN, rule set optimisation problem, monotone systems*

The thesis is in English and contains 66 pages of text, 9 chapters, 10 illustrations, 11 tables and 7 appendices.

MONSA algoritmiperekonna reeglisüsteemi katteülesande lahendamine. Realisatsioon Javas

Annotatsioon

Martin Rebane

Selle töö keskseks uurimisküsimuseks on MONSA algoritmiperekonna (MONSAMAX, MONSAMIN ja MONSABAN) reeglisüsteemi optimiseerimisülesande lahendamine. Töös analüüsitakse ka nimetatud algoritmide kvalitatiivseid omadusi ning ressursikasutust.

Optimiseerimisülesande sisu on leida väikseim hulk reegleid, mis kirjeldavad ära kogu andmestiku. See ülesanne lahendatakse parima katte leidmise teel, pakkudes välja kaks algoritmi, millest kumbki defineerib *parima katte* veidi erinevalt. Ahne (*greedy*) algoritm läheneb ülesandele pigem matemaatiliselt ja proovib viia lõplikusse valikusse võetavate reeglite hulga miinimumini. Lähendusalgoritm (*approximation algorithm*) valib iga andmestiku rea jaoks kattesse reegli, mis antud objekti seisukohalt tundub kõige paremini erisusi kirjeldav. Kaks alternatiivset lahendust pakuvad erineva suunitlusega väljundit ja on mõeldud erinevateks ärivajadusteks. Lisaks pakub töö välja unikaalse katte algoritmi. See on mõeldud kasutamiseks enne põhialgoritmi ning valib kattesse sellised reeglid, mida täieliku katte korral välistada ei saa.

Kõik käsitletavad algoritmid on töö käigus realiseeritud Java programmeerimiskeeles. Töös testitakse nende jõudlust ja ressursikasutust erinevate sisendite korral.

Realisatsiooni poolelt tutvustatakse võtteid, mis aitasid võrreldes triviaalse lahendusega algoritmide töökiirust tõsta, eelkõige hõlmab see efektiivset nullide ja nullväärtuste käsitlemist.

Võtmesõnad: *MONSA algoritm, MONSAMAX, MONSABAN, MONSAMIN, reeglistiku optimiseerimine, katteülesanne, monotoonsed süsteemid*

Magistritöö on kirjutatud inglise keeles, sisaldab 66 lehekülge, 9 peatükki, 10 graafikut, 11 tabelit ja 7 lisa.

Index of Tables

Table 1: Example dataset with discreet values....13
Table 2: Main frequency table for an example dataset....13
Table 3: Three class frequency tables....14
Table 4: Main frequency table on second level....15
Table 5: Class frequency tables on second level....15
Table 6: Examples of textual and compact representations of the rules....16
Table 7: Benchmark results for coverage algorithms with different datasets....32
Table 8: Running times and results of coverage algorithms depending on input rule set....34
Table 9: Number of rules after each step. Unique + greedy coverage....35
Table 10: Average over-coverage per object after each step. Unique + greedy coverage....35
Table 11: Replacement of values in a dataset....42

Illustration Index

Illustration 1: Complexity growth of MONSAMAX by number of columns....37
Illustration 2: Complexity growth of MONSABAN by number of columns....38
Illustration 3: Complexity growth of MONSAMIN by number of columns....39
Illustration 4: Running time of MONSA algorithms by number of rows in the dataset....40
Illustration 5: Memory usage of MONSA algorithms by number of rows in the dataset....41
Illustration 6: Running time by number of different values....42
Illustration 7: Memory usage by number of different values in range....43
Illustration 8: MONSAMAX memory usage while processing Mushroom 16-column dataset (complete run: MONSAMAX, DSR, Unique + greedy coverage algorithm). Illustration is made using Virtual VM software....45
Illustration 9: MONSABAN memory usage while processing Mushroom 19-column dataset (complete run: MONSABAN, DSR, Unique + greedy coverage algorithm). Illustration is made using Virtual VM software....46
Illustration 10: Class diagram demonstrating the differences in implementation for MONSA algorithms. Private methods are omitted....50

Table of Contents

1 Introduction....9
1.1 Objectives....9
1.2 Overview....9
1.3 Outline of research....10
2 Concepts and theoretical background....11
2.1 Monotone systems algorithms....11
2.2 MONSAMAX algorithm....11
2.2.1 Rule....11
2.2.2 Algorithm....12
2.2.3 An explanation of the MONSAMAX algorithm....13
2.3 MONSABAN algorithm for finding negations....16
2.4 MONSAMIN algorithm....17
2.5 Rule set and determinative set of rules (DSR)....18
2.6 Rule set optimisation problem of MONSA algorithms....18
2.7 Set cover problem....19
3 Rule set optimisation problem....21
3.1 Problem....21
3.2 Analysis of the problem....21
3.2.1 Prefer general patterns....21
3.2.2 Prefer deviance....22
3.2.3 Complexity of a rule....22
3.2.4 Coverage....22
3.3 Solution....25
3.3.1 Integration of MONSAMAX and rule optimisation algorithm....25
3.3.2 Greedy cover algorithm....26
Definitions....26
Algorithm....26
3.3.3 Approximation algorithm....27
Definitions....27
Algorithm....28

3.3.4 Unique coverage algorithm....28
Definitions....29
Algorithm....29
4 Experiments and comparative analyses of algorithms....30
4.1 Overview of experiments....30
4.2 Used datasets....30
4.3 Test environment....31
4.4 Comparison of covering algorithms....31
4.4.1 Speed....32
4.4.2 Coverage....32
4.4.3 Solving rule set optimisation problem on raw rule set....33
4.4.4 Reducing the size of rule set....35
4.4.5 Discussion of test results....35
4.5 Running time and memory usage of MONSA algorithms....36
4.5.1 Number of columns....37
MONSAMAX....37
MONSABAN....38
MONSAMIN....39
4.5.2 Number of rows....39
4.5.3 Number of different values....41
4.6 Analyses of memory usage during the execution....44
4.6.1 MONSAMAX....44
4.6.2 MONSABAN....46
4.6.3 Comparison of memory usage....47
5 Implementation notes of MONSA family of algorithms....48
5.1 Implementation of MONSAMAX using primitive data types....48
5.2 Object oriented implementation of MONSA family of algorithms....48
5.2.1 Efficient handling of zeroes and NULL-values....49
5.2.2 Using abstract data types....49
6 Ideas for future research....52
6.1 Implementation....52
6.2 Integration of MONSAMAX and MONSABAN....52
6.3 Rule set optimisation....52
6.4 Parallel algorithms....53

6.5 Live algorithm....53
6.6 Plug-ins for statistical computing packages....53
7 Conclusion....54
8 Kokkuvõte....56
9 References....58
10 Appendices....60
Appendix 1 – Coverage rule sets for 18-column Mushroom dataset....60
Appendix 2 – Data of MONSAMAX evaluation....62
Appendix 3 – Data of MONSABAN evaluation....63
Appendix 4 – Data of MONSAMIN evaluation....64
Appendix 5 – data of resource consumption depending on number of rows in dataset....65
Appendix 6 – data of resource consumption depending on number of different values in the dataset....65
Appendix 7 – Software and source code in Java (CD)....66

1 Introduction

1.1 Objectives

Main research question for this thesis is to find and implement an efficient solution for solving a rule set optimisation problem of the MONSA[1] family of inductive learning algorithms, three solutions are proposed. This thesis also investigates implementation details of MONSA algorithms, and introduces a new type of MONSA algorithm called MONSABAN. This new algorithm seeks for negations – patterns that do not exist in the dataset. Thesis develops corresponding ideas and methods to improve the rule sets of MONSA algorithms.

1.2 Overview

MONSA family of algorithms provides an efficient way to describe given dataset by finding regularities in them (Roosmann et al. 2008). These patterns are called rules and all the rules for given dataset form a rule set. MONSA algorithms offer excellent descriptive patterns of the dataset, making it easier to understand and interpret data. Two implementations of MONSAMAX will be introduced in this thesis. One of them has been developed along writing this paper and can be found on accompanying CD. This implementation will be used to test and analyse the performance of algorithms throughout the work.

While MONSAMAX is efficient and easy to use, main deficit of MONSAMAX from user's perspective is that the algorithm finds too many overlapping rules that describe same data examples. According to Kuusik and Lind, authors of MONSAMAX algorithm, some overlapping is desirable and even strength of the method (Kuusik and Lind 2012), but in practice it finds too many overlapping rules which makes it difficult to interpret the findings efficiently. Rule set optimisation algorithms developed in this thesis target that problem.

For solving a rule set optimisation problem, this thesis investigates different methods and sorting criteria of MONSA rule set. Intention is to find a best suitable criterion to evaluate the

1 "MONSA" is a shorthand of Monotone System Algorithm. This work considers three of them: MONSAMAX, MONSAMIN and MONSABAN. For this reason, author uses term "MONSA" to refer to these three algorithms.

quality of a rule and to remove weaker rules. Ideal solution finds a minimum set that covers all objects in dataset. This is both a technical task and a decision problem. Author will propose two different algorithms to solve the task. One of them is a greedy algorithm that reduces rule set optimisation problem to a classical set covering problem. Other method is an approximation algorithm that assigns weights to each rule and approximates a solution based on the calculated weights[2]. Additionally unique coverage algorithm is introduced, which speeds up greedy algorithm on some cases.

1.3 Outline of research

As will be explained in Chapter 2, a rule selection problem reduces to a classical cover set problem, hence a sub-chapter is also dedicated to analyse different approaches to solving a general cover set problem. New cover algorithms are developed based on the theoretical assumptions covered there.

After introducing new cover algorithms to solve rule set optimisation problem, thesis continues to investigate them empirically. Proposed cover algorithms are tested against different datasets and different types of compressed and uncompressed rule sets. Their running time, memory usage and quality of the result are compared. Empirical evaluation was chosen because it is the first time these algorithms are implemented. Empirical tests quickly help to determine main characteristics of algorithms while using less time than theoretical analyses of algorithms might take.

Thesis continues with the empirical evaluation of implementations of MONSAMAX, MONSAMIN, and MONSABAN algorithms. Aim is to determine what affects the resource consumption most and therefore algorithms are tested against different types of datasets and against different characteristics of datasets, like the number of columns and number of rows.

Finally, this work analyses the implementation of MONSA algorithms in Java, highlighting the differences between current and trivial implementations. Thesis concludes by suggesting new research ideas for investigation of MONSA algorithms and rule set optimisation problem.

2 In a mathematical sense, both algorithms could be called greedy approximation algorithms. This work calls one of them „greedy” as it uses classical greedy approach and aims at finding minimal solution. Other is called approximation algorithm as it solves a problem that is close to the original.

2 Concepts and theoretical background

2.1 Monotone systems algorithms

Monotone Systems algorithms are a type of inductive learning[3] algorithms that take datasets with an unknown structure as input, process those datasets using frequency tables and output rules that describe the dataset (Roosmann et al. 2008). These algorithms use frequency tables to process datasets instead of using the data directly, hence they achieve higher efficiency while not losing information (Roosmann et al. 2008). This work implements and tests three such algorithms: MONSAMAX, MONSAMIN and MONSABAN. As all three are very similar, this chapter gives full explanation and details only for MONSAMAX. For MONSAMIN and MONSABAN, only the differences from MONSAMAX are highlighted.

2.2 MONSAMAX algorithm

MONSAMAX algorithm finds regularities and patterns in a dataset. MONSAMAX works on datasets that are using or can be converted to use discreet values. A dataset is viewed as a table where each row represents a data *object*, and columns represent variables that are called *attributes*. Attributes can have different discreet *values* within the dataset, but one value for any given object. One of the attributes for each run of the algorithm will be called a *class* – this is the attribute that is being characterized by the algorithm. MONSAMAX computes a frequency tables for the dataset and uses them to find patterns that determine a certain class value. Such pattern is called a *rule*. Finally, a collection of all the rules that are found for given class is called a *rule set*.

2.2.1 Rule

A found pattern in a data set is called a *rule*. Given that A, B and C are attributes (variables) in a dataset then an example of a MONSAMAX[4] algorithm rule might be:

if A=2 and B=3 then C=6;

3 In essence, it means "learning by example"

4 Rule of MONSAMAX is identical to rules found by other MONSA algorithms, MONSABAN and MONSAMIN

C is a class variable in this example.

Each rule is composed of one or more attribute-value pairs and of exactly one class value that is determined by those attribute-value pairs. A rule also has metadata about number of objects it covers. *Rule set* is a collection of rules.

2.2.2 Algorithm

MONSAMAX was developed by Rein Kuusik and Grete Lind (2012), their algorithm code from the same source follows[5]:

```
S0. t:=0; U_t:=∅
S1. Find frequencies in whole dataset and each class
    If t>0 then Bring zeroes down
S2. For each factor A such that its frequency in some
class C is equal to its frequency in whole set
    output rule {U_i}&A→C, i=0,…,t
    A←0
S3. If not enough free factors for making an extract
then
    If t=0 then Goto End
    Else t:=t-1; Goto S3
S4. Choose a new (free) factor U_t with largest frequency
     U_t ←0; t:=t+1;
     extract subtable of objects containing U_t;
     Goto S1
End. Rules are found
```

5 Kuusik and Lind offered this algorithm together with explaining notes. Author of this work has integrated some of the notes into algorithm description for clarity.

2.2.3 An explanation of the MONSAMAX algorithm

To get a better sense of how MONSAMAX works, a detailed description of steps of algorithm follows. Readers familiar with MONSAMAX can safely skip this section.

An example dataset is given in Table 1 where third attribute is chosen as a class. Class is an attribute that algorithm describes using other attributes in the data set.

	Attribute A	Attribute B	Attribute C (Class)
Object 1	1	1	1
Object 2	2	3	2
Object 3	2	2	3
Object 4	3	3	3

Table 1: Example dataset with discreet values

Step 1. MONSAMAX computes a frequency table for the dataset and for each class. Columns of a frequency table represent attributes, and rows represent values of the attributes. Each entry in the frequency table shows how many times one value occurs for given attribute. Main frequency table for example dataset is given in Table 2. As Attribute C was chosen as a class, there is no need to compute frequencies for that.

	Attribute A	Attribute B
Value “1”	1	1
Value “2”	2	1
Value “3”	1	2

Table 2: Main frequency table for an example dataset

Similar frequency tables are computed for each class value. In this example dataset we chose Attribute C as a class value. It has three possible values: 1, 2 and 3. Frequency table for class value 1 would be built using only the objects where Attribute C has value 1, and frequency table for class value 2 would be built using only the objects where Attribute C has value 2. Similarly frequency table for class value 3 would be built using only the objects where Attribute C has value 3. If we add up the values for any given cell (attribute-value combination) from all class-based frequency tables, it adds up to the number in the main

frequency table. Class frequency tables are shown in Table 3.

	Table for Class = 1		Table for Class = 2		Table for Class = 3	
	Attr A	Attr B	Attr A	Attr B	Attr A	Attr B
Value “1”	1	1	0	0	0	0
Value “2”	0	0	1	0	1	1
Value “3”	0	0	0	1	1	1

Table 3: Three class frequency tables

Step 2. Rules are found by comparing frequencies. If frequencies for some attribute-value combination are the same in any of the class frequency tables and in the main frequency table, then this match is called a *rule*. Given the example dataset above, we can easily determine a rule for class value 1:

- if the value of Attribute A is 1, then class value in this dataset is always 1 (meaning that there are no occurrences where Attribute A is 1, but class value is not 1). This can be shortened as IF A=1 THEN C=1;
- there is also an alternative rule based on Attribute B: IF B=1 THEN C=1. This rule is found as frequency of value 1 for Attribute B matches the frequency in main frequency table.

Algorithm continues by comparing all the frequencies and finds all the rules. In this example, no rules were found from a second class frequency table. Rules IF A=3 THEN C=3 and IF B=2 THEN C=3 are found after comparing the frequencies in third class frequency table against the main frequency table.

If at the end of process there are any frequencies over 0 in main frequency table that have not been used, a largest of them is selected. Should several frequencies be equal, first one is selected. In our case, such value is where Attribute A has frequency 2 for value 2. Hence we build a new frequency table, based on the original dataset, and include only rows, where A=2. Column for Attribute A is omitted from the table, as we used this as a basis for selection.

	Attribute B
Value “1”	0
Value “2”	ǂ 0
Value “3”	1

Table 4: Main frequency table on second level

Frequency for value “2” will be replaced by 0 as we already found a rule for B=2 on previous level. Next, we build class frequency tables based on data where A=2.

	Table for Class = 1	Table for Class = 2	Table for Class = 3
	Attribute B	Attribute B	Attribute B
Value “1”	0	0	0
Value “2”	0	0	ǂ 0
Value “3”	0	1	0

Table 5: Class frequency tables on second level

First class frequency table is empty, nothing to do there. From the second class frequency table we will find a rule for class value 2 as frequency matches with the main table. This rule would be a made from 2 components:

- as this table contains only rows where A=2, we will add this as first part of a rule: if value of Attribute A is 2 and value of Attribute B is 3 then the class value will be always 2 for given dataset.

MONSAMAX would now return to previous level and make another extract until all unused frequencies above 0 are used and all rules are found.

We can now formalize the representation of the rules as follows

IF [ATTRIBUTE] = [SOME VALUE] THEN [CLASS] = [SOME VALUE]

Kuusik and Lind (2011b) also write rules more formally as follows:

- [optionally add capital “T” for “Attribute”] and give the number of attribute. For example, for first attribute, use “1”;
- denote attribute value with a dot followed by the value;
- show class value after the equals sign.

Examples of both representations are given in Table 6.

Textual representation	Compact representations
if the value of Attribute A (first in dataset) is 1, then class value in this dataset is always 1	T1.1 = 1 or 1.1 = 1 IF A=1 THEN C=1
if the value of Attribute A is 2 and Attribute B (second in dataset) is 3 then the class value will be always 2 for given dataset	T1.2 & T2.3 = 2 or 1.2 & 2.3 = 2 IF A=2 AND B=3 THEN C=2

Table 6: Examples of textual and compact representations of the rules

For another example and longer explanation of MONSAMAX, please see (Kuusik and Lind 2012).

2.3 MONSABAN algorithm for finding negations

MONSABAN is an algorithm that utilizes most of the MONSAMAX code, but instead of finding regularities, it seeks negations – combinations of attributes and values that do not exist in a dataset. This idea was suggested to the author by Professor Leo Võhandu during master's seminar in spring 2013. Algorithm is based on MONSAMAX and has only a small modifications in Step 2:

```
S0. t:=0; Ut:=∅
S1.  Find frequencies in whole dataset and each class
    If t>0 then Bring zeroes down
S2. For each factor A such that its frequency
 in some class C is 0
    output rule {Ui}&A != C, i=0,…,t
S3. If not enough free factors for making an extract then
    If t=0 then Goto End
    Else t:=t-1; Goto S3
S4. Choose a new (free) factor Ut with largest frequency
     Ut ←0; t:=t+1;
     extract subtable of objects containing Ut;
     Goto S1
End. Rules are found
```

Instead of comparing frequencies with main frequency table, MONSABAN compares frequencies in sub-tables to 0. If a frequency is 0 for some combination, then we have found a negation. Implementation in program code differs more from MONSAMAX than a written algorithm as it uses somewhat different comparators and sorting. Efficiency and results of MONSABAN will be introduced in a later chapter.

2.4 MONSAMIN algorithm

MONSAMIN is similar to MONSAMAX, it even shares the same algorithm logic. The difference with MONSAMAX is that it processes frequency tables starting from the lowest frequency instead of highest. MONSAMIN is also developed by Rein Kuusik and Grete Lind. Algorithm code is almost identical with MONSAMAX (see Kuusik and Lind 2012), except for sorting in Step 4:

```
S0. t:=0; U_t:=∅
S1. Find frequencies in whole dataset and each class
    If t>0 then Bring zeroes down
S2. For each factor A such that its frequency in some
class C is equal to its frequency in whole set
    output rule {U_i}&A→C, i=0,…,t
    A←0
S3. If not enough free factors for making an extract
then
    If t=0 then Goto End
    Else t:=t-1; Goto S3
S4. Choose a new (free) factor U_t with smallest frequency
     U_t ←0; t:=t+1;
     extract subtable of objects containing U_t;
     Goto S1
End. Rules are found
```

Final rule set for MONSAMAX and MONAMIN should be the same, only the order in which

rules are found, is different.

2.5 Rule set and determinative set of rules (DSR)

MONSA algorithms find many rules that contain same set of attribute-value pairs and determine the same class value. This problem is solvable by finding *determinative set of rules* (DSR), method which was introduced in (Kuusik and Lind 2011a). This method eliminates rules that are contained in other rule, e.g. if one rule is a subset of other rule then this subset is removed.

In the following example Rule 2 will be eliminated as A = 1 (attribute A has value 1) is enough information to determine Class 1 and B = 4 is unnecessary addition:

Rule 1: IF A = 1 THEN Class = 1
Rule 2: IF A = 1 & B = 4 THEN Class = 1

As MONSAMAX finds more overlapping rules that could be helpful to the user, there is also a need to discard rules that are different in terms of attributes, but which cover same objects. This article contributes to finding such solution.

2.6 Rule set optimisation problem of MONSA algorithms

MONSA algorithms may find and usually finds many rules that each describe a set of objects in a dataset. Such sets tend to overlap and hence describe many data objects several times. The problem usually arises when different attributes (variables) can be used to describe a data object.

Rule set optimisation problem of MONSA algorithms is combinatorial task of selecting best rules from the MONSA DSR rule set. Such selection ideally covers all objects that are covered by DSR set, but with fewer rules. This problem is a third step after (1) generating rules from the dataset and (2) compressing them into DSR set, and is one of the main topics of this thesis. As we will demonstrate, should we wish to solve rule set optimisation problem by minimising data object coverage then this optimisation problem is reducible to a set covering problem.

Following is an example of rule set optimisation problem. Given dataset with variables A, B, C, D, E and class variable F, MONSA algorithm might come up with three different rules to describe F:

IF A=1 & B=2 THEN F=1
IF C=2 & E=4 THEN F=1
IF D=4 THEN F=1

They all describe objects where the value of class variable F is 1. Each rule uses different variables and covers a different set of objects, but with some overlapping. Assume that first and third rule will cover the same amount of objects that all three combined. Successful solution for rule selection problem considers this overlapping and may select only first and third rule. This is equivalent to solving a set cover problem. There are also other possibilities that are described later in this thesis.

2.7 Set cover problem

Set cover problem can be defined as follows: given a finite set U which contains n objects, we have subsets of U named $S_1...S_k$ where each contains one or more objects from U. Intention is to select as few subsets S_i as possible, but cover all n objects from U with this selection. We can also define vector x of size k to denote whether a subset S_i is selected or not. For each S_i there is decision variable x_i to denote selection – if given subset is selected then $x_i=1$ and in case it is not then $x_i=0$. Given these conditions, the goal is to minimise $\sum_{i=0}^{k} x_i$.

Set cover problem was proved to be NP-complete in 1972 by Richard Karp (Lund and Yannakakis 1994:960) after Stephen A. Cook had proved a year earlier that boolean satisfiability problem is NP-complete (Praust 1996:86). Karp demonstrated that satisfiability is reducible to different problems that belong to class NP, including a set covering problem (Karp 1972:98). There are no known exact algorithms for solving NP-complete problems efficiently, best exact algorithms still take exponential time in the worst case (Sedgewick and Wayne 2011:917,919). Hence set covering problem can not be solved exactly for larger problems.

Greedy algorithm is one of the best algorithms to solve set covering problem (Slavík 1996:435) and one that is most widely studied for it. Basic concept of greedy algorithm is that it iterates through the sets and on each iteration selects such set to the cover that offers best value at that time[6]. Running time of greedy set covering algorithms is polynomial, but nearness to optimality is not always outstanding. To evaluate algorithm's nearness to optimality, a concept of *ratio* is used (Feige 1998:634; Lund and Yannakakis 1994:961). Let k be number of sets that are selected to final cover. Ratio between k of greedy algorithm and k of optimal solution is used to evaluate the results of algorithm (ibidem). Lund and Yannakakis proved ratio of approximately $0.72 \ln n$ as lower bound for greedy algorithm (n stands for the number of objects in finite set U) in their work under stricter complexity assumptions while Johnson (for greedy algorithm) and Lovász (for linear approximation algorithm) had shown a ratio of $\ln n$ earlier (Feige 1998:635). They showed that below these ratios, a solution to set cover problem can not be approximated efficiently.

6 Best value can vary by implementation and objective. Algorithm in this work prefers sets that have most uncovered objects in it.

3 Rule set optimisation problem

3.1 Problem

Rule set optimisation problem of MONSA algorithms aims to reduce over-coverage of data objects. After MONSA algorithm finds rules from the dataset, a rule set compression process removes rules that are fragments of other rules. Such cleaned rule set is called determinative set of rules (DSR) (Kuusik and Lind 2011a). Even after DSR process there are too many rules to be efficiently used by the analyst. This is caused by the fact that MONSA algorithm finds many possible combinations of rules and hence covers one data object many times. Full description of the problem is given in previous chapter. The task of this thesis is to develop and build an algorithm that reduces the amount of rules by reducing, perhaps minimising over-coverage of data objects[7].

3.2 Analysis of the problem

MONSA algorithm finds rules that are represented by one or more rule parts, each containing a variable name and variable value, and class variable name and class variable value. An example:

IF A=1 AND B=1 THEN C=3, where A and B are variables and C is class variable.

We also know information about frequency of the rule (how many data objects a rule covers)[8]. Given this information we can consider different approaches to solving rule set optimisation problem. These approaches could be used independently, but most of them can also be combined with each other.

3.2.1 Prefer general patterns

Should a user of the algorithm be interested in describing general patterns, rule set

7 In a strictly mathematical task, one would prefer minimal coverage to reduce the size of a rule set. This work also considers business requirements where minimal coverage might be neither best nor preferred solution; or minimal coverage might be only part of the solution. Details will follow later in this chapter.

8 MONSA algorithms are based on frequency tables. Therefore saving how many objects each rule covers (frequency) is only an implementation detail and does not change the algorithm.

optimisation problem can be solved exactly by sorting rules by frequency and selecting all rules above some frequency threshold. This problem is easy to solve and requires no additional analysis.

3.2.2 Prefer deviance

Should a user be interested in deviating cases, the solution is identical to *general patterns* approach, except that rules below some frequency threshold are selected.

3.2.3 Complexity of a rule

MONSA algorithms find rules of different length. Minimal length for any rule is 1 and it uses only one variable to describe a class variable (example: IF A=1 THEN C=4). Maximum length of the rule is k where k is the number of variables in the data set (excluding class variable). Length of the rule could also be viewed as a complexity of a rule. More variables it contains, more complex it is. For some scenarios a user might prefer shorter rules, while for others longer rules are better suited. Reducing the rule set by complexity is also a trivial challenge and easy to solve. One needs to sort the rule set by length and apply desired length limits.

3.2.4 Coverage

Mathematically straightforward approach to reduce the size of rule set would be to minimise the coverage of data objects. This approach does not depend on business requirements but could also be used in combination with any business-driven approach. This method ensures that each data object that is covered by DSR is also covered by optimised solution, but as few times as possible. It does not make any assumptions on rule quality other that preferring rules that can contribute to minimal coverage.

This method can also utilize information about the number of objects that each rule covers. One approach to minimising a rule coverage of dataset is to convert rules to Boolean expressions and try to simplify these expressions. Praust (1996:82–84), also Lensen and Kruus (2012) offer good examples on solving such expressions. Other feasible option would be constraint programming. Working principle for constraint programming is that a user

specifies constraints for the solution and solver needs to satisfy them (Barták 1999).

The efficiency of such approaches would be questionable as too much information about data objects is lost during the generation of rules. For example, let us take three rules from a fictional dataset:

IF A=1 AND B=1 THEN C=3 (frequency 3)
IF A=2 AND E=1 THEN C=3 (frequency 3)
IF D=4 THEN C=3 (frequency 6)

They all describe class C on value 3, but they do not contain any further information about objects they cover. From such set we can only infer that first and the second rule must be describing different set of objects as they contain different value for variable A, but we can not tell anything about the third rule as it may:

1. cover distinct set of objects than first and second rule,
2. partly cover same set of objects that first and second set are covering,
3. fully cover same set of objects that first and second set are covering.

Any combination might offer minimal coverage, but we have no means to verify. Continuing with this path only makes sense if we are looking for exact coverage i.e. no object can be covered more than once. Additionally we know that MONSA algorithm always describes all non-contradictory data objects in the dataset.

Considering these limitations, in some cases it would be possible to solve the problem by using constraint programming models. As we know the frequencies of rules and of class, it would be possible to set constraints based on frequencies. An example: let us assume total frequency of class variable-value combination C=3 in the dataset is 8. Now we can compose a following expression for minimal coverage without overlapping rules:

$$x_1 * f_1 + x_2 * f_2 + x_3 * f_3 \leq 8$$

where x_n is a Boolean decision variable which tells if given rule is selected for final set of rules (value is either 0 or 1) and f_n stands for the frequency of *n*th rule.

Computing all the possibilities based on such expression would take too much time in a larger dataset. We set additional constraint for each pair of arbitrary rules <n, m> from the rule set so that no pair will be able to exceed total frequency:

$$x_n * f_n + x_m * f_m \leq 8 \quad .$$

An advanced solver can now solve the problem efficiently by eliminating impossible combinations quickly. Alternatively, we can derive constraints from rule parts like we did earlier. As 1st and 2nd rule can not cover same objects, we know that optimal solution covers at least 6 objects.

$$x_1 * f_1 + x_2 * f_2 + x_3 * f_3 \geq 6 \quad .$$

This constraint sets a lower bound to the solution and hence prunes the search space by removing all combinations below this threshold[9]. Together with upper bound set by the total frequency of C=3 it leaves two options: either 1st and 2nd together or 3rd rule. Any other combination would violate the constraints. Now we can see that third rule must offer minimal coverage by itself. Selecting it gives us maximum possible frequency with fewest rules i.e. minimal cover.

Unfortunately this approach only works if we are interested in minimal coverage without overlapping rules and even then it has its limitations. For example, let total frequency of C=3 in the dataset be 9. Now we do not have a single best solution. Recall that MONSA algorithm always covers all objects. Therefore perfect exact minimum cover is possible in this case. Selecting 3rd rule and one of the others gives us this minimum cover. Alas without additional information there is no way to determine which of the two we should select.

Additionally, removing the requirement of exact cover poses even greater challenges. All the

9 This formula is necessary in more complex datasets. For example, given a total frequency of 14 and 4 rules with frequencies 7, 5, 10, 3 where first two are known to cover different objects and there is no information about last two, we must use this constraint. Lower bound would be 12 and this would also remove some combinations of rules, not only single rules. Single rules can also be removed by primitive threshold: selecting a maximum frequency among all rules for given class variable-value combination and setting it as a lower bound.

combinations that exceed given class variable-value frequency are then possible. But we are not interested in any kind of over-coverage. We only care about such over-coverage which at the same time increases total coverage of objects[10]. As we do not have any criteria for selecting rules for over-coverage, we need more information to solve the problem.

We saw that using MONSA DSR rules as an input for optimisation problem is somewhat efficient only if we are using exact minimal coverage and if we are able to set bounds. This clearly is not satisfactory for most use cases where we can not meet these conditions. Therefore this thesis proposes a solution that integrates with original MONSA algorithm and saves concrete reference to the data objects that each rule covers. This allows to approach the problem differently and to use wider set of methods. It also makes it feasible to solve rule set optimisation problem by finding minimal coverage without above-mentioned constraints.

3.3 Solution

3.3.1 Integration of MONSAMAX and rule optimisation algorithm

This thesis proposes that the solution for rule optimisation problem would be closer to optimality if the algorithm can access information about data objects that each rule covers. One approach would be to take both the DSR rule set and original dataset as input and match rules to data. This approach would cause a lot of extra work that could be avoided by integration with MONSA algorithm.

When MONSA algorithm finds a rule in Step 2, the code has access to a frequency table which was used to find a rule. Data that was used to compute this frequency table is kept available as it might be necessary to make another extract based on this table. Software can easily process this data and save references to data objects as rule metadata. This does not make MONSA algorithms more complex and adds very little overhead. Software that is developed in this thesis implements this functionality. This chapter proposes three such approaches for solving rule set optimisation problem.

10 If rule covers objects that are not yet covered, it increases total coverage. At the same time it may cover some objects that are already covered, therefore also increasing over-coverage.

3.3.2 Greedy cover algorithm

This greedy algorithm for rule set optimisation is similar to classical approach of solving cover set problem (a good example is Chvatal 1979). It works by selecting the best rule to the final rule set, then ordering the rule set to determine a new best rule, selecting it, sorting again etc. until all the objects are covered. Here we define best rule by unique coverage – the rule which at the moment of decision covers most uncovered objects, is the best. The goal is to minimise the number of rules that are going to be selected.[11]

Definitions

n – number of objects (= rows in dataset)

R – rule set after DSR, containing rules. Each rule covers >= 1 object. Rule set covers <= *n* objects

R2 – rule set for final selection of rules, initially empty set.

¬R2 – negation of R2. Objects that are covered by rule set R, but not by R2.

N – vector of size *n* to store the status of object, all values initially 0. If object (data row) has been covered by rule set R2, object's value in N is set to 1.

current unique coverage of rule – count of objects that are covered by rule *r* and rule set ¬R2 (=are covered by R and not covered by R2)

Algorithm

Step 1. Sort rule set R by current unique object coverage, in descending order.

Step 2. Take and remove first rule from set R, name it r.

Step 3. Add r to set R2

Step 3. Mark all objects that r covers in N.

Step 4. Remove rule r from R.

Step 5. Sort R by current unique objects coverage (same as step 1).

Step 6. Take and remove first rule from set R, name it r. (same as step 2)

Step 6. If size(R) > 0 AND at least 1 object that r covers is not covered by R2 GOTO STEP 3
ELSE END;

Final rule set is R2.

11 There are also other possibilities. For example, if one might need to minimise over-coverage, it is better to calculate some sort of ratio between uncovered and already covered objects and sort by that ratio.

3.3.3 Approximation algorithm

While greedy algorithm works by selecting best rules from the rule set, this approximation algorithm works by selecting best rule for every data object. At the first step, algorithm computes a weight to each rule. We start by computing variable *k* for each data object. *k* denotes how many times a data object is covered by original rule set. Then we sum all *k*'s of objects that a rule covers and divide the sum by the number of objects. For better separation, *k* is squared. Finally, algorithm selects a rule with smallest *k* for each data object. This selection is a final rule set.

Definitions

n – number of objects (= rows in dataset)

R – rule set after DSR, containing rules. Each rule covers >= 1 object. Rule set covers <= *n* objects

R2 – rule set for final selection of rules, initially empty set.

¬R2 – negation of R2. Objects that are covered by rule set R, but not by R2.

N – vector of size *n* to store the status of object, all values initially 0. If object (data row) has been covered by rule set R2, object's value in N is set to 1.

current unique coverage of rule – count of objects that are covered by rule *r* and rule set ¬R2 (=are covered by R and not covered by R2)

F – data structure to save references between data objects and rules. For each data object there is as set of rules that it covers.

M – vector of size *n* to store the rule count for each object, all values initially 0. For each rule from the initial set R that covers given object, this value is increased by 1

Algorithm

R2 = Ø (final set is initially empty)

N = int[n] (integer array of size n, all values initially 0)

M = int[n] (integer array of size n, all values initially 0)

Step 1. Build frequency table to mark how many times each data row is covered by rules.

E.g if data row is covered by 3 rules then its frequency is 3

For each data row, save references to rules that they cover.

F[i] = List<Rule[, Rule, ...]>

M[i] = size(F(i)); // rule count for given data object

Step 2. Calculate coverage coefficient (weight) for each rule in rule set R:

For each rule *r* that covers *s* data rows:

s = number of data objects that r covers

set coverage coefficient *c* for *r*:

$$c = \frac{\sum_{i=1}^{s} M[ID\ of\ ith\ object]}{s}$$

Step 3. For each data row from 1 to n

If data row is already covered, skip and continue with next row

If related rule set F[i] exists and contains at least one rule

Sort rule set F[i] by coverage coefficient in ascending ordering

Add first item F[1] to final rule set R2

END

Notes: variable M can be replaced with size(F(i)), it is added for clarity.

3.3.4 Unique coverage algorithm

This is not an independent algorithm, but is meant to be used in conjunction with either one of the other algorithms. This algorithm selects all rules that cover at least one object which is not covered by any other rule. Hence it is efficient for problems where such rules exist and can reduce running time of full algorithms significantly. It has especially good impact on greedy

algorithm, but it also reduces running time for approximation algorithm.

Definitions

n – number of objects (= rows in dataset)

R – rule set after DSR, containing rules. Each rule covers >= 1 object. Rule set covers <= *n* objects

R2 – rule set for final selection of rules, initially empty set.

N – vector of size *n* to store the status of object, all values initially 0. If object (data row) has been covered by rule set R2, object's value in N is set to 1.

Algorithm

R2 = Ø (final set is initially empty)

N = int[n] (integer array of size n, all values initially 0)

Step 1. Build frequency table to mark how many times each data row is covered by rules.
E.g if data row is covered by 3 rules then its frequency is 3

Step 2. Iterate over unsorted rule set R, select rule r on each iteration

Step 2.1 Check if rule r covers any data row where frequency is 1

Step 2.2. If such row is found

* add rule r to final set R2 as no other rule can cover given data row
* mark all data rows that r covers in N, set N[r] = 1 (to show that it is covered)
* remove rule r from original rule set R

Pass R, R2 and N to full algorithm.

4 Experiments and comparative analyses of algorithms

4.1 Overview of experiments

This chapter is dedicated to experiments with both MONSA algorithms and cover algorithms implementations. Rationale behind the experiments is to evaluate the speed and quality of the output of different algorithms. We will also investigate how they handle different datasets, and compare similar algorithms against each other. First half of the chapter examines cover algorithms and the rest is dedicated to MONSA algorithms. For cover algorithms, we test three: greedy algorithm, approximation algorithm, and unique coverage algorithm combined with greedy. As combination of unique coverage algorithm and greedy is more efficient than a combination of unique coverage algorithm and approximation algorithm, we only test unique coverage algorithm combined with greedy in this work. From MONSA algorithms, all three, MONSAMAX, MONSABAN, and MONSAMIN, are tested for running time and memory consumption under different input.

4.2 Used datasets

This work uses two datasets of different characteristics to compare algorithms. First dataset is Nursery dataset describing application acceptance decisions for nursery schools in Slovenia (Rajkovic et al 1997). It is a multivariate dataset with 9 categorical attributes and 12960 rows. Any attribute has at most 5 different values. Second is Mushroom dataset describing physical characteristics of mushrooms (Schlimmer 1987). It has 8124 rows and 23 attributes with categorical values up to 12 different values. For testing purposes, we use first 18 attributes and a class value from this dataset[12]. Nursery dataset represents social science data for decision problem, certain objects categorise well, but there are no dominant patterns for other objects. Mushroom dataset is a natural sciences dataset with more attributes where data is much easily categorised with fewer rules than in Nursery dataset. These datasets were selected to test algorithms with inputs of different complexity.

12 To achieve reasonable running time on given hardware and have comparable results. 18-column dataset was largest that can be run with MONSAMAX on a computer with 4 GB of RAM without excessive garbage collection overhead.

4.3 Test environment

All tests were conducted on the same PC with following characteristics:

- Processor Intel Core i5-2410M 2.3 GHz
- 4 GB DDR3 memory
- Windows 7 64-bit operating system
- Java version 1.7.0_07
- Java HotSpot 64-Bit Server VM (build 23.3-b01, mixed mode)

One every test, a program ran single-threaded and used maximum 1 processor core, but there were no restrictions for Java Virtual Machine to use as much processor power as necessary for background tasks (e.g. garbage collection). Each program was allocated maximum 2800 MB of memory (java was invoked with *-Xmx2800M* flag).

4.4 Comparison of covering algorithms

This chapter tests covering algorithms against two datasets to see how efficiently they can cover rule sets. Efficiency is measured both in terms of running time and size of the solution. All results for Nursery dataset are shown as average over 100 runs and for Mushroom dataset over 10 runs[13]. Tests were ran in cycles, using a new program instance for each run. Java virtual machine garbage collector was explicitly executed before each run to minimise chances that it stops the execution of program during the test. Each program was additionally executed once before the test cycle. This ensures all classes are loaded into computer memory before the test. At the end of this comparison the algorithms will also be run using raw rule set[14] from MONSAMAX algorithm instead of DSR. Purpose of this is to test how efficient the algorithms are and how stable the solution is.

13 Running Mushroom 18-column dataset for 100 runs with MONSAMAX would take too much time. It does not significantly affect the reliability as the differences between runs are proportionally much smaller than for Nursery dataset, e.g. 0.25 seconds difference of MONSAMAX running time would be 25% for Nursery dataset while 20 seconds (largest recorded value) for Mushroom dataset is only 6%.

14 This work uses the term “raw rule set” to indicate unprocessed rule set that MONSA algorithm outputs.

	Nursery dataset	Mushroom dataset reduced (18 columns + class)
No of rows in dataset	12960	8124
No of columns in dataset	9	19
No of rules MONSAMAX	8188	861756
No of rules after DSR	638	1634
Time spent for algorithm	1.247 sec	352.64 sec
Time spent for DSR	0.258 sec	133.53 sec
Approximation coverage		
Time	0.006 sec	0.113 sec
No of rules in cover	556	40
Greedy coverage		
Time	0.016 sec	0.323 sec
No of rules	557	21
Unique + Greedy coverage		
Time	0.015 sec	0.42 sec
No of rules	555	21

Table 7: Benchmark results for coverage algorithms with different datasets

4.4.1 Speed

Empirical evaluation demonstrates that approximation algorithm is always running in shortest time. This has also been case for all the datasets which were used when developing the algorithm, regardless of dataset properties. Approximation algorithm is almost three times faster than greedy or greedy and unique coverage algorithm combined. Still we observe that running time for every coverage algorithm is reasonably low and unnoticeable when compared to the running times of algorithm and DSR.

4.4.2 Coverage

For Nursery dataset where coverage includes many small rules, all algorithms perform equally well. For Mushroom dataset, coverage includes few rules that each cover many objects. In this case greedy algorithms offer much smaller cover (21 rules) than approximation algorithm (40 cover).

If we take a closer look at the rules (see Appendix 1), we see that greedy algorithms select rules that cover large number of objects. Only tree rules cover less than 100 objects. Approximation algorithm, on the other hand, selects more rules, but each rule covers less objects.

Over-coverage object count is 16938 for approximation and 14870 for greedy algorithm, so each data object is covered 2.1 or 1.8 times on average. Difference between object count is 2068 objects which is 13.9%. Therefore we can tell that while approximation algorithm finds almost 50% more rules, it only find about 14% larger over-coverage. This demonstrates a fundamental difference between two approaches. Greedy algorithms select rules that are universal while approximation algorithm selects rules that are more unique. From a mathematical perspective, greedy algorithm performs better in this case, but one may also judge the result by business value. Approximation algorithm might offer better insight into the characteristics of data. From business perspective greedy algorithm answers the question "What is common?" and approximation algorithm "What is different?".

4.4.3 Solving rule set optimisation problem on raw rule set

Rule set optimisation step is meant to be third and final step of selecting final selection of rules to describe the dataset. Rationale behind this is that rule set optimisation by finding minimal cover is a NP-complete problem which is hard to solve, so minimal input is preferred. But as this implementation is not looking for exact minimal cover, but a solution that is reasonably close or otherwise acceptable, it works in a decent time frame as seen from previous experiments. Hence it is feasible to investigate how the performance of covering algorithms is affected if instead of DSR they get a raw rule set from MONSAMAX as input. Results are given in Table 8.

	Input: DSR	Input: raw rule set	Increase %
Nursery dataset			
Input rule count	638	8188	11834%
Approximation	0.006 sec	0.142 sec	2267%
	556	556	0%
Greedy	0.016 sec	2.42 sec	15025%
	557	557	0%
Unique + greedy	0.015 sec	2.67 sec	17700%
	555	557	0.36%
Mushroom 18-column dataset			
Input rule count	1634	861756	52639%
Approximation	0.113 sec	22.32 sec	19652%
	40 rules	41 rules	2,5%
Greedy	0.323 sec	54.47 sec	16764%
	21 rules	21 rules	0%
Unique + greedy	0.42 sec	45.80 sec	10804%
	21 rules	21 rules	0%

Table 8: Running times and results of coverage algorithms depending on input rule set

We observe that for Nursery dataset, where good coverage includes many rules, approximation algorithm performs best both in terms of total running time and increase of running time. While input rule set grows 118 times, running time of approximation algorithm grows only 22 times. Greedy algorithms perform much worse in sense of running time and it's growth. Growth of running time (150 times for greedy and 177 for unique and greedy combined) exceeds the growth of input (118 times).

For Mushroom dataset where good coverage includes less rules, results are almost contradictory to Nursery's results. Largest increase in running time is observed for approximation algorithm. Still, given that the input increases 526 times, running time for approximation algorithm increases 197 times, meaning it still retains efficiency. While running times for greedy algorithms increase 167 and 108 times, their total running time is still twice as much as that of approximation algorithm. We also notice that for Mushroom dataset greedy algorithm is faster than greedy and unique combined if DSR is used as input. When calculating cover based on raw rule set, combined algorithm outperforms greedy noticeably.

Evaluating the quality of cover from raw rule set, we notice only very slight and trivial changes, meaning that the quality is approximately same as if the coverage was calculated on DSR.

4.4.4 Reducing the size of rule set

Main motivation behind constructing rule set optimisation algorithms is a necessity to reduce the size of final rule set. Combination of unique coverage and greedy algorithm is used to assess the proportion of reduction of rule sets.

	MONSAMAX output	**DSR**	**Change**	**Coverage**	**Change**	**Total change**
Nursery	8188	638	-92.2%	555	-13.0%	-93.2%
Mushr. 18-c	861756	1634	-99.8%	21	-98.7%	-99.9%

Table 9: Number of rules after each step. Unique + greedy coverage.

	MONSAMAX	**DSR**	**Coverage**
Nursery	7.64	1.95	1.82
Mushroom 18-col	4928.28	43.08	1.83

Table 10: Average over-coverage per object after each step. Unique + greedy coverage.

Table 9 And Table 10 demonstrate that rule optimisation algorithm is effectively reducing the size of the rule set for both datasets, therefore working as expected. As a co-effect they also show reasonably good results for over-coverage.

4.4.5 Discussion of test results

We observed that approximation algorithm runs fastest for both datasets and retains reasonable running times even when the input is raw rule set. It is efficient for finding good coverage if cover includes many rules. It is outperformed by greedy-based algorithms if dataset can be covered with few rules. We also noticed the business value of approximation algorithm – it tends to cover dataset by selecting rules that contain information about smaller groups of data objects.

Greedy algorithms offer coverage with smaller number of rules and smaller over-coverage rates if a dataset contains larger universal patterns. While greedy algorithms are always slower than approximation algorithm, they return good value for all types of datasets and they should be preferred if it is important to minimise the number of rules in cover. Running greedy algorithm in combination with unique coverage algorithm appears to be faster than greedy alone for larger rule sets that contain rules that can be selected by unique coverage algorithm.

The solutions that cover algorithms output are very stable. Solutions that are calculated using DSR as input are almost the same as solutions that are calculated directly from the raw rule set of MONSAMAX algorithm. As DSR algorithm removes rules that are already contained in other rules, extra rules from raw rule set do not add any value. Therefore it is a sign of quality that the solution remains very close for both types of input.

It is noticeable that given the current implementation of DSR, it would be more efficient to skip finding DSR for datasets with large number of rules (e.g. Mushroom dataset) and compute the final coverage rule set directly from raw rule set. When raw rule set contains relatively small number of rules as in Nursery dataset, only approximation algorithm achieves better result when skipping DSR.

4.5 Running time and memory usage of MONSA algorithms

Complexity of the implementation of MONSA algorithms was tested empirically against number of rows in the dataset, number of columns in the dataset and maximum number of different attribute values in a dataset.

Mushroom dataset was used for testing against number of columns. Number of columns in the dataset was reduced by removing last columns. For experiments 8 different datasets were created – a range from 11-column to 18-column Mushroom dataset (not counting class column).

For testing against number of rows, an 11-column Mushroom dataset was selected as a basis. This dataset has 8124 rows. On each iteration, a number of rows was increased by 8124 by adding an original 11-column dataset to the end. This method ensures, that the number of rows increases, but the complexity of other parameters (number of extracts, number of rules,

size of frequency tables) stays constant.

4.5.1 Number of columns

MONSAMAX

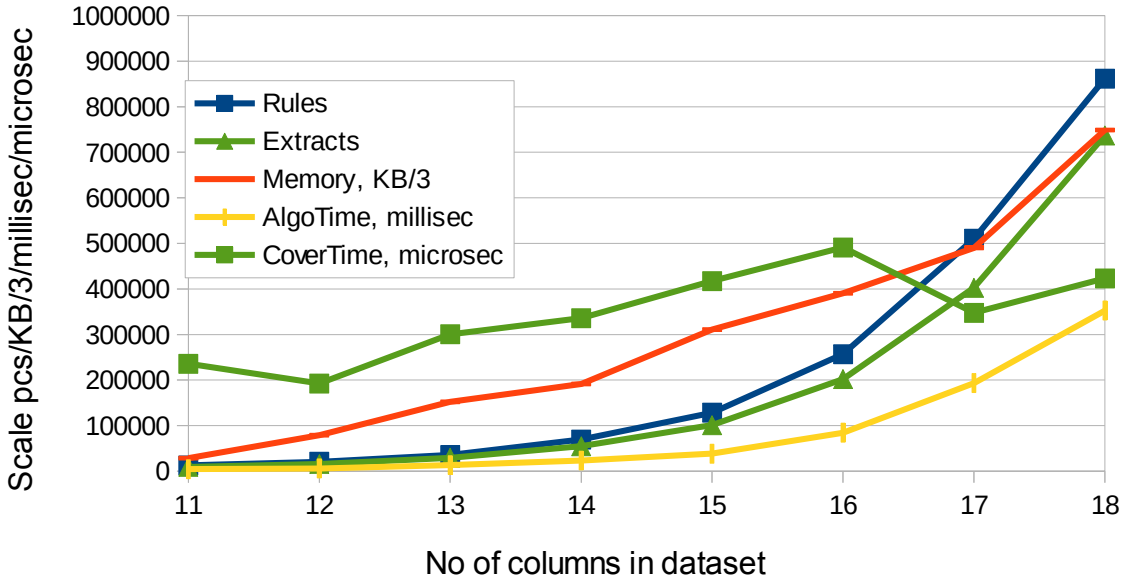

Illustration 1: Complexity growth of MONSAMAX by number of columns

Illustration 1 demonstrates how algorithm running time, memory usage and cover time are affected by number of columns (on X-axis), which increases number of extracts and number of rules. Units of different measures are scaled to fit into the same graph, please see Appendix 2 for original data. Numbers of rules and extracts are not a direct functions of number of columns. They are also dependent upon other properties of a dataset, i.e. data distribution. Number of extracts are perhaps one of the easiest ways to observe the complexity of task for any given number of columns. Observing graphed variables, we see that the bottleneck of this MONSAMAX implementation is memory usage. Memory usage seems to grow by much higher order polynomial term[15] than running time of algorithm.

Rule set covering time is relatively unaffected by column count as it depends on qualities of the rules rather than the size of dataset.

15 At least.

MONSABAN

Running same tests for implementation of MONSABAN, an algorithm which finds patterns of negations in dataset, we observe from Illustration 2 that algorithm running time is the only resource where usage increases noticeably[16]. Memory usage is unaffected by the number of columns in the dataset. Although we see that number of extracts increases very fast, it does not bring along a rise in memory usage as in the MONSAMAX algorithm.

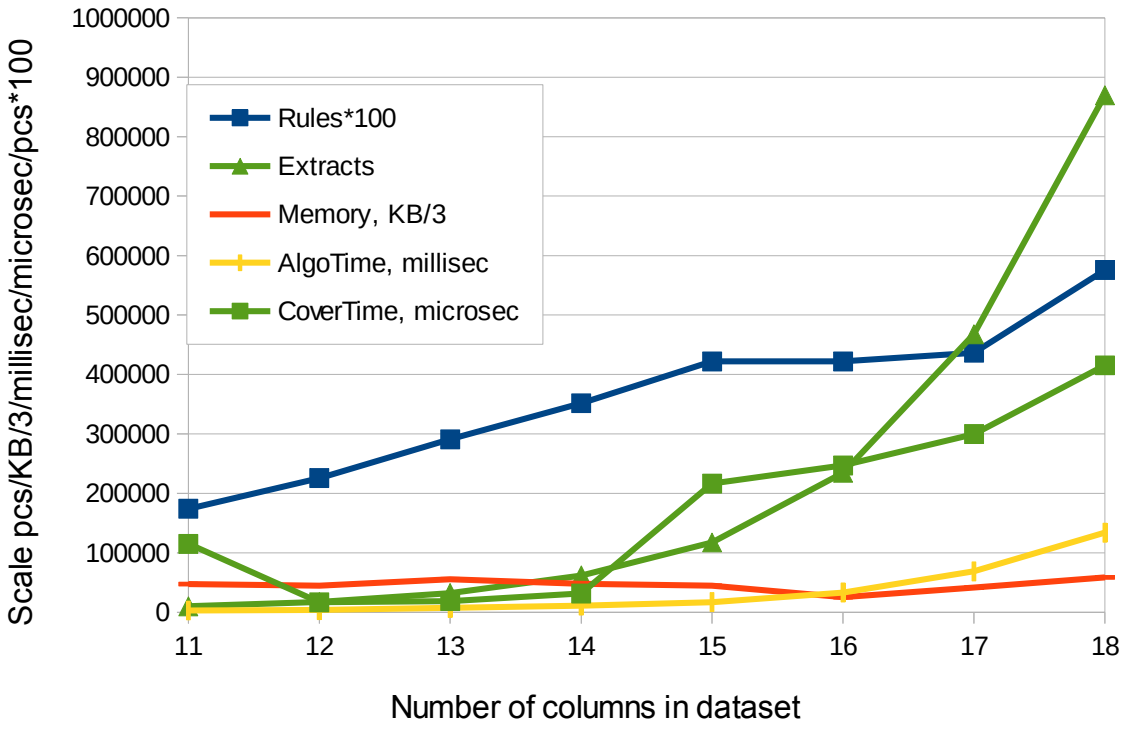

Illustration 2: Complexity growth of MONSABAN by number of columns

Combination of unique and greedy algorithms is used in this test to find cover. Although the graph displays a rise for running time of finding a cover, this change is rather irrelevant in absolute terms as cover time increases from 0.12 seconds for 11-column dataset to 0.42 seconds for 18-column dataset[17]. For unscaled data, please consult Appendix 3.

16 When comparing results to MONSAMAX, please note that the scale for number of rules for MONSABAN is pcs*100, not pcs as for MONSAMAX.

17 This rise is due to more data that allows MONSABAN to extract different rules. For 11-column dataset there were 643 rules in DSR and 44 in cover, for 18-column dataset there is 1634 rules in DSR and only 21 in cover. As we see, while input has grown more than 250%, number of rules in coverage has fallen 50%.

MONSAMIN

Illustration 3 demonstrates that complexity growth for MONSAMIN is very similar to MONSABAN with 19-column Mushroom dataset. This is due to the fact that class column of this dataset has two different values. Hence the number of extracts and rules are the same for both algorithms. MONSABAN achieves 10-20% better results in running time and memory usage because of comparison method. MONSABAN compares frequencies in class frequency tables against constant 0, while MONSAMIN compares them against main frequency table. Compared to MONSAMAX, we see that MONSAMIN achieves much better results. It processes 18-column dataset in 175 seconds compared to 353 seconds of MONSAMAX, and uses 172 MB of memory against 2193 MB that MONSAMAX uses. It achieves this result because MONSAMIN processes smallest frequencies first. Smallest frequencies have shortest paths to the deepest level and less data must be kept in memory in any given time. MONSAMAX selects longest paths first and this requires much more memory.

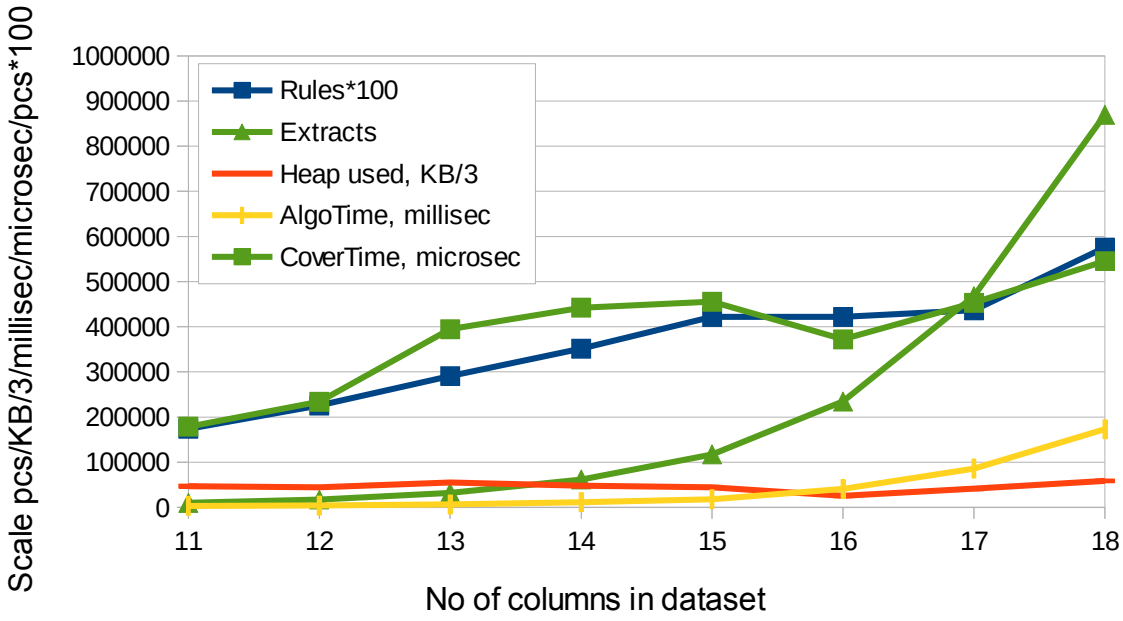

Illustration 3: Complexity growth of MONSAMIN by number of columns

4.5.2 Number of rows

Five different datasets that are based on 11-column Mushroom dataset were used to test how algorithms perform when the number of rows increases in the dataset while number of extracts and rules stay constant. At first iteration, 11-column Mushroom dataset is used and on

each iteration the number of rows increases by the original dataset size. Memory usage was measured with Virtual VM software. Results are given on illustrations 4 and 5.

Observing the growth in running time and memory usage, we notice that the results are much more promising than for the number of columns. Usage of both resources grows very reasonably for MONSAMIN and MONSABAN. Although the growth is higher for MONSAMAX, it is still linear growth for both memory and running time. MONSAMIN performs best in memory usage. Data for experiments is given in Appendix 5.

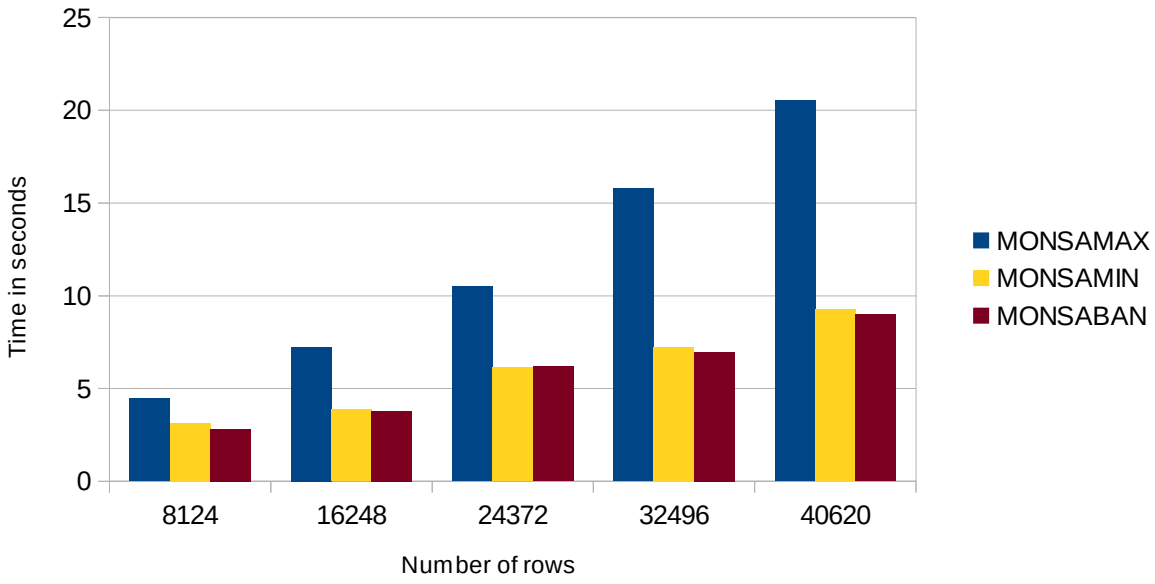

Illustration 4: Running time of MONSA algorithms by number of rows in the dataset

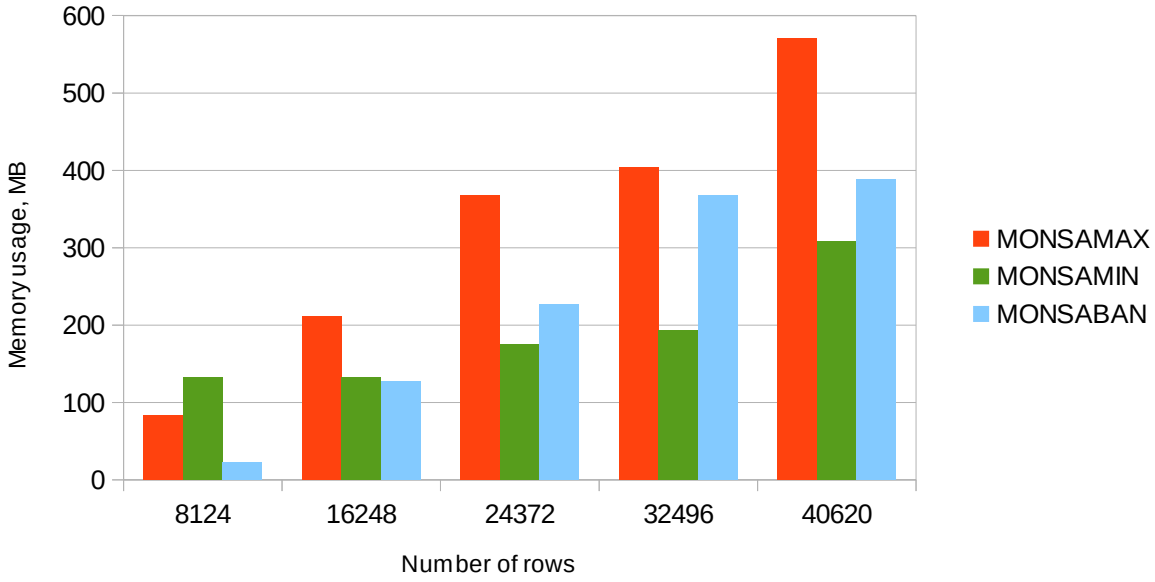

Illustration 5: Memory usage of MONSA algorithms by number of rows in the dataset

4.5.3 Number of different values

The size of frequency tables is affected by the number of different values in the dataset. This part of work tests how algorithms perform when the size of input (number of data rows) stays constant, but the number of different variables inside the dataset increases. Nursery dataset is used to evaluate this. Dataset was modified in two steps for this test. First, it was multiplied 5 times – same set of data was added 5 times to form a new dataset. New dataset has 5 times as many rows as the original. Second, 5 different versions of the dataset were made, each replacing additional 20% of the original dataset with duplicate data with same structure, but with different value range. For example, original dataset had values in range from 1 to 5. Second dataset has 20% of them replaced by values 6 to 10 accordingly (1 was replaced with 6; 2 with 7 etc). All replacements are shown in Table 11.

Maximum number of different values in this test was 25. This should cover many cases where categorical variables are used and is enough to capture the trends.

Range of values ↓	1st run	2nd run	3rd run	4th run	5th run
1-5	100 %	80%	60%	40%	20%
6-10	-	20%	20%	20%	20%
11-15	-	-	20%	20%	20%
16-20	-	-	-	20%	20%
21-25	-	-	-	-	20%

Table 11: Replacement of values in a dataset

Results from Illustration 6 show that number of different values in the dataset do increase running time for every algorithm. It also demonstrates that number of different values is the weakest spot of MONSABAN, running time goes up from 0.26 seconds with 5 different values in range to 7.35 seconds when there are 25 different values in range.

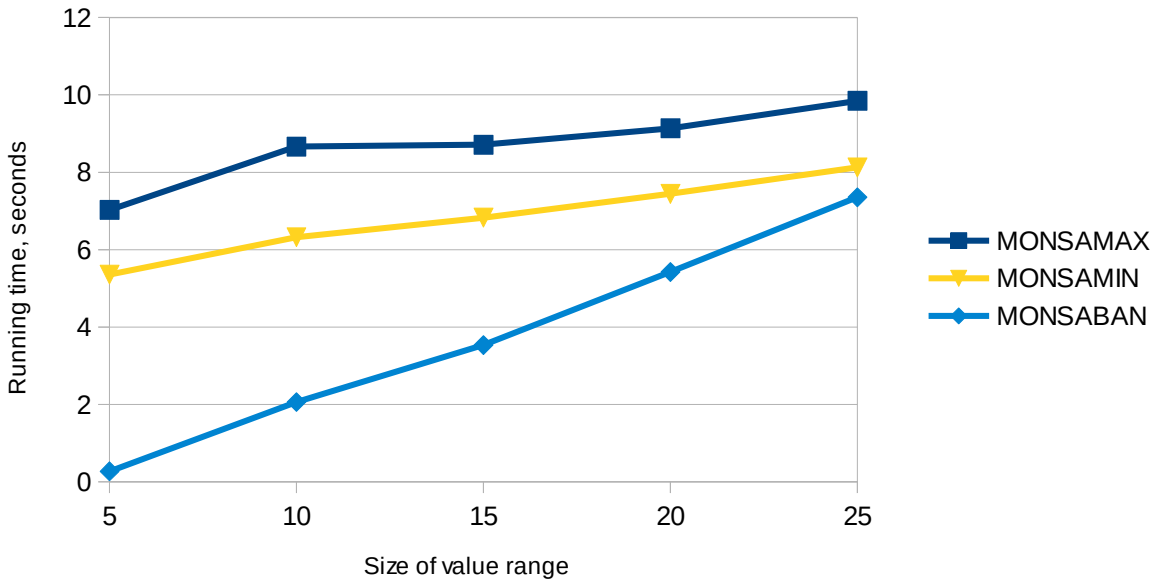

Illustration 6: Running time by number of different values

Increase in running time for MONSABAN means that for datasets with high range of values it loses it's advantage over other algorithms. Both MONSAMAX and MONSAMIN show good results as running time of those increases 40% and 52 % respectively when going from 5 different values to 25.

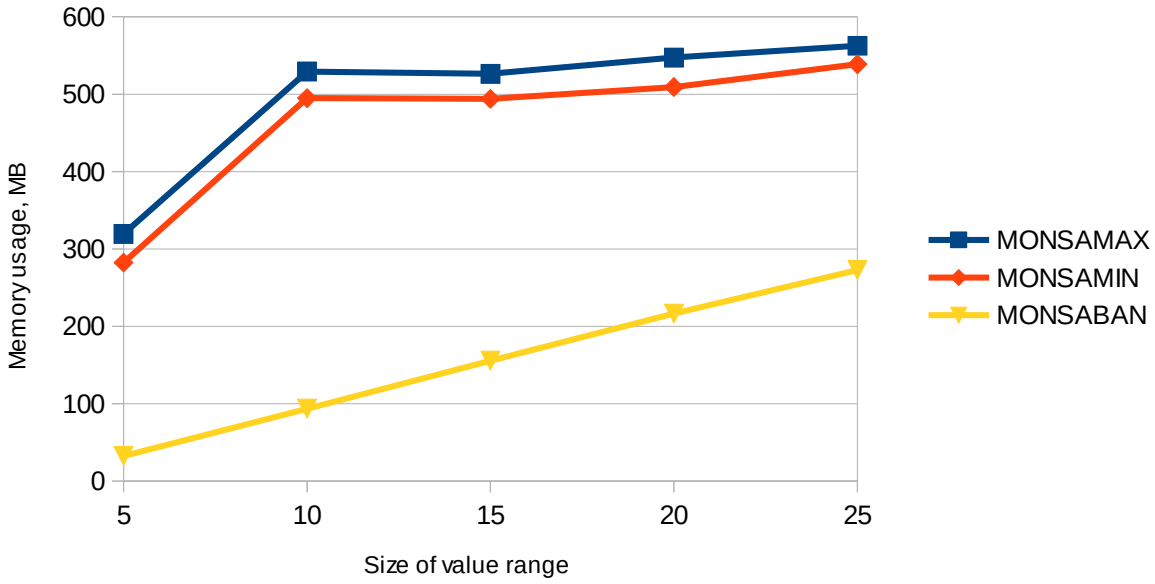

Illustration 7: Memory usage by number of different values in range

Illustration 7 demonstrates that while losing it's advantage in running time, MONSABAN is still the most efficient in memory consumption. On the other hand, memory usage also goes up fast, achieving half of the consumption for MONSAMAX and MONSAMIN. Latter two also perform very well. While there is a significant increase in memory usage when going from 5 different values to 10, further growth is slow and very stable for both MONSAMAX and MONSAMIN.

Results indicate that the growth of value range does not affect the running time and memory usage of MONSAMAX and MONSAMIN unreasonably. Resource consumption for them grows in significantly slower pace than the number of different attribute values in the dataset. For MONSABAN, increase in number of different values brings along much higher resource consumption. This is explained by the logic of algorithms. MONSAMAX and MONSAMIN are effectively pruning the search space each time a rule is found. If a rule is found they are blocking corresponding frequencies for all subsequent frequency tables (“bringing zeroes down” in Step 1 of algorithm). In most datasets there are less negating rules than there are positive rules, so MONSABAN does not benefit from such pruning and processes most values down to the deepest level. Data for experiments is given in Appendix 6.

4.6 Analyses of memory usage during the execution

Analyses of resource consumption of MONSA algorithms in Chapter 4.5 demonstrated that memory usage becomes a bottleneck for processing larger datasets. This chapter takes a closer look at the memory usage of MONSAMAX and MONSABAN implementations.

For analysing memory usage, we observe heap size and memory usage in Java Virtual Machine (JVM) during the execution of MONSAMAX and MONSABAN programs. A special software package Visual VM is used for analysis (Sedlacek and Hurka 2014). Tests and results show that memory consumption for MONSABAN and MONSAMIN are very similar, so only one of them is selected for comparison with MONSAMAX.

Heap is a portion of memory that JVM allocates to the program. This memory is available when a program needs it. JVM keeps heap larger than actual memory (heap) usage. This allows efficient garbage collection without considerably affecting the performance of execution of the program.

4.6.1 MONSAMAX

Illustration 8 demonstrates that peak memory usage occurs during MONSAMAX execution and that memory usage is lower for DSR and coverage algorithms. MONSAMAX is being executed from around 1:08:58 to 1:10:15 where peaks occur. Peaks rise when new extracts are made and fall when garbage collector removes redundant extracts that have went out of scope[18]. From 1:10:15 onwards a DSR processing occurs and coverage is calculated. As we see, this does not increase memory usage significantly and memory usage remains lower than during algorithm execution.

18 For example, when a frequency table extract is no longer used and there are no references to the extract in the program code, this extract is ready to be collected by garbage collector which frees up the memory that was used by the extract.

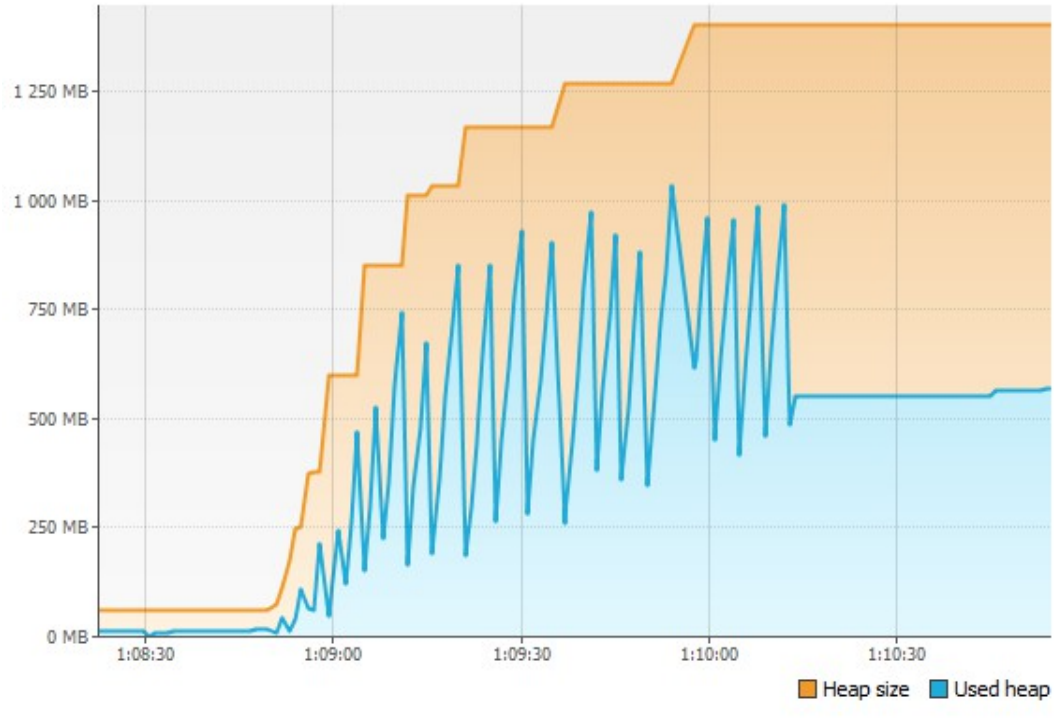

Illustration 8: MONSAMAX memory usage while processing Mushroom 16-column dataset (complete run: MONSAMAX, DSR, Unique + greedy coverage algorithm). Illustration is made using Virtual VM software.

4.6.2 MONSABAN

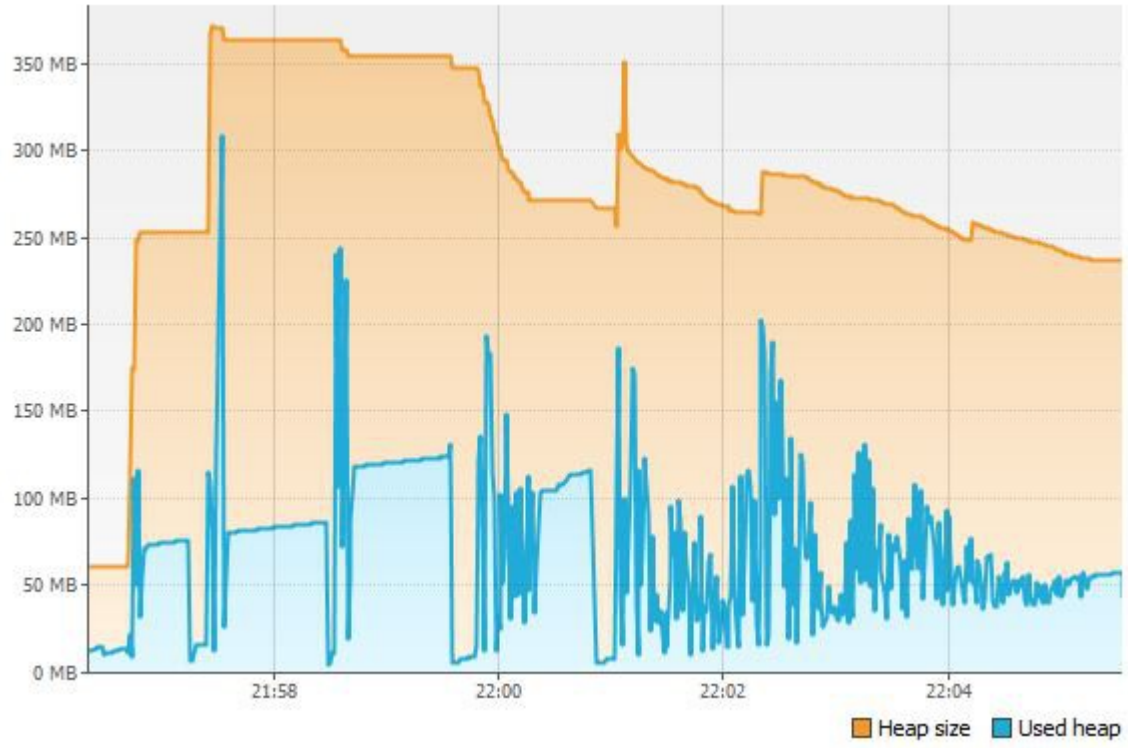

Illustration 9: MONSABAN memory usage while processing Mushroom 19-column dataset (complete run: MONSABAN, DSR, Unique + greedy coverage algorithm). Illustration is made using Virtual VM software.

Memory usage of MONSABAN differs much from that of MONSAMAX. As can be seen from Illustration 9, memory usage of MONSABAN is stable during the execution, even falling at the end. For MONSAMAX, it was constantly increasing. Also, memory usage for MONSABAN is much lower than for MONSAMAX. 19-column Mushroom dataset was used to record the memory usage here, while 16-column dataset was used for MONSAMAX[19].

19 Memory usage patterns for algorithms are the same for all lengths of Mushroom dataset extracts. Reason for selecting this configuration was that their graphical representation was clear and easier to understand than for very small or very large time frames.

4.6.3 Comparison of memory usage

Better memory efficiency of MONSABAN is caused by the sorting criteria it uses. Like sister algorithm MONSAMIN, it takes shortest path to scan all data. All new extracts are made by selecting smallest frequency first. MONSAMAX selects largest frequency first, which causes deeper recursions and larger extracts that must be kept in the memory.

Analysis of memory dump for MONSAMAX with Visual VM showed that for 16-column Mushroom dataset, at the time of peak memory usage, on average 65% of memory was allocated to extract data. For MONSABAN, maximum memory usage for extract data was only 25%.

Number of columns seems to increase resource consumption for every algorithm more than other factors. Number of rows also increases running time and memory usage, but the growth is very reasonable. MONSAMAX performs well, but is still most resource hungry, it's running time increases as much the input increases. MONSAMIN and MONSABAN have much better ratios.

Number of different values in dataset affects mostly MONSABAN, but does not increase memory usage significantly for others.

5 Implementation notes of MONSA family of algorithms

MONSA algorithms have been implemented two times, this chapter gives a short overview of the implementations.

5.1 Implementation of MONSAMAX using primitive data types

MONSAMAX was first fully implemented as software package in 2012 using C# programming language (Jõgiste, Vellamäe, and Rebane 2012). This implementation used data structures that closely mimicked the structures proposed in the original description of the algorithm (see Kuusik and Lind 2012). The implementation used multi-dimensional integer arrays to store frequency tables, and literally set used frequencies to zeroes just as described in the algorithm. It is a fine implementation as it offers good debugging options and each step is easy to follow. Algorithm was implemented using static methods to decrease object overhead in memory usage. It has a graphical user interface to control input parameters of algorithm.

On the other hand, such implementation had a few disadvantages that constrained its further development. Namely, it was necessary to create new data structure for each level of algorithm, while more general object oriented approach would have allowed to efficiently reuse existing data structures. Also, static methods had somewhat poorer interface for storing meta information about the rules. As we see in this thesis, information about objects that a rule covers is important in finding optimal rule coverage for the dataset. For those reasons, a new implementation of MONSA algorithms was developed using object oriented methods in Java programming language.

5.2 Object oriented implementation of MONSA family of algorithms

Main reasons for implementing MONSA algorithms in object oriented Java code were extensibility and better code reuse capabilities. It also uses different data structures to store data more efficiently.

5.2.1 Efficient handling of zeroes and NULL-values

Trivial implementation of a frequency table for MONSA algorithm would bring along unreasonable growth in running time and memory usage if the number of different attribute values in the dataset grows. This is because the size of the frequency table is a product of number of columns and number of different attribute values (Table 2 demonstrates this). Should there be a dataset with m columns where all attributes use maximum k different values and one uses $3k$ values, the size of frequency table would be $m * 3k$. This unreasonably increases the running time of algorithm, because all those unnecessary rows are carried along and checked on each iteration. Even if impossible combinations are set to NULL, they still cause noticeable overhead both in terms of running time and memory usage. This implementation of MONSA family of algorithms avoids this pitfall by saving only relevant pairs of attribute-value combinations.

This implementation also introduces a novel concept for *bringing zeroes down* in the first step of MONSA family of algorithm. Instead of literally setting values to zeroes, this implementation saves a list of banned frequencies. When the frequency table is accessed, it will efficiently return only valid frequencies. This also allows to distinguish between cells where frequency is indeed zero[20], and cells where frequency has been set zero by algorithm. Such distinction was necessary for implementing and testing MONSABAN algorithm on the same code base.

5.2.2 Using abstract data types

In sense of memory usage it is efficient to store data using primitive data types. This uses less memory than complex data types. On the other hand, operating with primitive data types becomes increasingly difficult when one needs to add, sort and access data in different ways. It would take too much time to implement accessors and sorting algorithms with reasonable running time for primitive data types. It is much reasonable to use existing data structures. Although they bring along somewhat higher memory usage, built-in structures let programmer concentrate on the main task of implementing algorithm instead of reinventing data structures.

20 As noted this implementation does not save such frequencies but computes them on demand as it is more efficient. Zero-frequencies are relevant for class frequency tables when compared against main frequency table.

Also, object oriented approach is better suited for testing and implementing different versions of algorithms that are sharing much of the code. Eventually there is more time to improve the algorithm as refactoring of program code takes less time. Performance that is initially lost by using complex data structures, is eventually regained by well-crafted code which uses efficient accessors and sorting algorithms.

Illustration 10: Class diagram demonstrating the differences in implementation for MONSA algorithms. Private methods are omitted.

Class diagram on Illustration 10 demonstrates how different algorithms from MONSA family share most of the methods and override only a small amount of them. As an example, implementing different sorting algorithms for frequency tables, is as easy as passing relevant Comparator object for data structure in method createEmptyFrequencySet.

Such approach easily allows to override certain methods to implement and test different algorithms.

Additionally, when a programmer is familiar with one MONSA algorithm, he only needs to examine a concrete implementation to understand the differences between algorithms.

6 Ideas for future research

6.1 Implementation

Results indicate that there are many ways to improve both MONSA family of algorithms and rule set optimisation algorithms, especially in terms of memory consumption. Also, results suggest that further integration of MONSA algorithms and rule set optimisation algorithm could offer better performance.

Program code of current implementations of both MONSA algorithms and cover algorithms are designed to be easily expandable and modifiable. It suits well for testing different versions of algorithms quickly and can be used in future research.

This work identified that number of column affects running time of MONSA algorithms considerably. Using matrix calculations could be one possible option to overcome that difficulty and improve speed of both MONSA family of algorithms and cover set optimisation problem and is worth further investigation. Basis of such solution is introduced by Zakrevskij et al (2008:67).

6.2 Integration of MONSAMAX and MONSABAN

MONSAMAX is very efficient in terms of finding a set of rules that describe the dataset completely, it's drawback is memory consumption. MONSABAN finds rules that negate existence of patterns, it describes combinations that are not possible in the dataset. It is efficient in resource consumption, requiring much less memory and running time than MONSAMAX. Still, it's results are not as informative as those of MONSAMAX. Therefore it would be fruitful to investigate possibilities to combine those two algorithms to achieve the descriptive rule set of MONSAMAX while using less resources like MONSABAN. One approach would be to run MONSABAN before MONSAMAX and prune the search space for the latter by using information that MONSABAN gathers.

6.3 Rule set optimisation

This work concentrated on minimising the number of rules in cover set. Alternative approach

would be to minimise over-coverage instead, to see if it yields results with different qualities. Additionally, current cover algorithms work to find one best solution. It might be fruitful to test if finding *k* best solutions would offer better business value.

Much effort in this thesis was directed towards finding mathematically good coverage, future research might broaden the scope of coverage algorithms by considering different business use cases.

6.4 Parallel algorithms

MONSA algorithms and cover algorithms would benefit a lot from parallelism. In parallel implementation a dataset could be parsed by several or evens hundreds of parallel threads either in one computer or in a grid of computers. While it is largely an implementation problem, it also requires analyses of algorithms to determine if and how the results would be affected.

6.5 Live algorithm

Currently MONSA algorithms take a finite dataset as input and they output the number of rules. While there is large number of potential use cases for such algorithms, the potential of MONSA algorithms is even wider. It would be worthwhile to consider if live MONSA algorithms are also possible. Such algorithms would accept initial input like current algorithms, but they would then continue to accept new incremental input and recalculate the rule set based on new data. Current implementation could do it by parsing all the data again, but a live implementation would recompute only relevant parts that are affected by new data. Such algorithm could be used in big data applications or in embedded devices. For example, to detect changes in system environment.

6.6 Plug-ins for statistical computing packages

MONSA family of algorithms would be more accessible to scientific community if they were developed and offered as plug-ins for larger statistical computing packages like R, Stata, Octave, MathLab or similar.

7 Conclusion

Central research question for this thesis was to solve rule set optimisation problem of MONSA family of algorithms. Along the process, a new implementation of MONSA family of algorithms in Java programming language was developed. Thesis then concentrated on analysing the properties of MONSA and cover algorithms both in terms of quality of the solution and resource consumption of the implementation.

Rule set optimisation problem was reduced to set cover problem in this thesis. Two different cover algorithms for solving rule set optimisation problem were proposed. Greedy algorithm takes a mathematical approach to minimise the number of rules that are selected to the final cover. It offers stable cover solutions for different types of datasets. Output rule set tends to describe universal trends in the dataset. Main deficit of the algorithm is somewhat greater running time than the alternative solution, approximation algorithm. Approximation algorithm attacks the problem from the other side, selecting the best rule for each data object. The result is different from that of greedy algorithm as it covers dataset with rules that tend to describe unique properties of the dataset. Such rule set offers different business value for the user. Running time of approximation algorithm is much faster than running time for greedy algorithm.

Additionally, a unique coverage algorithm was introduced. This algorithm is meant to be used in conjunction with other rule set optimisation algorithms and improves the speed of them if dataset contains objects that are only covered by one single rule.

Testing the performance of MONSA algorithms demonstrated that small details in algorithms bring along large differences in performance. We saw that MONSAMAX, MONSAMIN and MONSABAN are all rather efficient for datasets with small number of columns. This work identified that MONSAMAX, because it makes extracts by preferring larger frequencies, is currently not able to parse datasets with large number of columns. This thesis proposed two ideas for future research to attack this problem. It could be addressed by optimising the implementation of the algorithm or by modifying the algorithm by combining MONSAMAX and MONSABAN.

Finally, few concepts were introduced that allowed to develop MONSA algorithms more efficiently than would have been possible by using trivial implementation. Efficient handling of zeroes and null-values is the most notable. Complex data structures were developed to store frequencies and used values. This significantly reduced memory usage for algorithms.

All research questions in this thesis were successfully answered and new ideas for future research were proposed.

8 Kokkuvõte

Selle töö keskseks teemaks on MONSA algoritmiperekonna reeglisüsteemide optimiseerimisülesande lahendamine. Antud ülesande püstitus nõudis ka kogu MONSA algoritmiperekonna realisatsiooni loomist. Realisatsiooniks valisin Java programmeerimiskeele. Kolmanda olulise komponendina analüüsisin MONSA algoritmide ning reeglisüsteemide kattealgoritmide tööefektiivsust.

Reeglistiku optimiseerimisülesande taandasin töö käigus hulgakatte ülesandeks. Pakkusin välja kaks erinevat algoritmi probleemi lahendamiseks. Ahne (*greedy*) algoritm läheneb probleemile matemaatiliselt, püüdes lõplikus valikus olevate reeglite arvu miinimumini viia. See algoritm pakub stabiilselt head tulemust erinevate andmestike korral, kuid tema puuduseks alternatiivselt väljapakutud lähendusalgoritmiga (*approximation algorithm*) on suurem tööaeg. Lähendusalgoritm valib probleemi lahendamiseks teistsuguse lähenemise, valides iga konkreetse andmerea jaoks parimate näitajatega reegli. Tulemus erineb ahne algoritmi omast, sest katab andmestiku pigem selliste reeglitega, mis kirjeldavad väga hästi erisusi andmestiku sees, mitte ei otsi võimalikult universaalset lahendust. Selline lähenemine pakub kasutajale teistsugust ülevaadet ja võib teatud ärivajaduste korral paremini sobida. Lähendusalgoritmi suureks plussiks on ahnest algoritmist oluliselt kiirem tööaeg.

Lisaks eelnimetatutele tutvustab töö ka unikaalse katte algoritmi. Selle algoritmi tööpõhimõte on valida kattasse ainult sellised reeglid, mis ainsana katavad mõnda objekti ja on seega ainsaks võimaluseks antud reegli katmisel. See algoritm on mõeldud kasutamiseks koos teiste algoritmidega, lisades enne põhialgoritmi käivitamist kattesse kõik eelkirjeldatud kriteeriumile vastavad reeglid ja seega vähendades põhialgoritmi sisendi mahtu.

Töö testis ka realiseeritud algoritmide töökiirust ja mälukasutust, leides, et väikesed muudatused algoritmi tööpõhimõtetes toovad kaasa ootamatult suuri muutusi jõudluses. Väikeste andmestike puhul olid nii MONSAMAX, MONSAMIN kui MONSABAN ühtviisi efektiivsed. Samas selgus, et MONSAMAXi jõudlus väheneb oluliselt andmemahtude kasvades, eriti rängalt mõjutab MONSAMAXi mälukasutust veergude arvu suurenemine andmestikus. Selle põhjustab väljavõttude tegemise printsiip – kui MONSAMIN ja MONSABAN teevad väljavõttusid minimaalsed sageduse järgi, siis MONSAMAX teeb

maksimaalse sageduse järgi. See toob kaasa mahukamad rekursioonid ja suurema mälukasutuse. Töö lõpus pakun välja kaks ideed, kuidas MONSAMAXi jõudlust parandada. Üks võimalus oleks kasutusele võtta Zakrevskij pakutud maatriksarvutused (2008), teine võimalus on MONSAMAX ja MONSABAN algoritme kombineerides vähendada MONSABANi tulemuse abil MONSAMAXi sisendi mahtu.

Algoritmide realisatsiooni osas pakkusin välja uued võtted MONSA algoritmide tööefektiivsuse tõstmiseks, mis erinevad oluliselt triviaalse algoritmi meetoditest. Triviaalses käsitluses toovad nullid ja väärtuste puudumine kaasa olulise mahtude tõusu. Siinses realisatsioonis on see probleem lahendatud kasutades keerukamaid, kuid efektiivsemaid andmestruktuure.

Töö tulemusena leidsid vastuse kõik algselt püstitatud uurimisküsimused ja pakkusin välja uusi ideid edasiseks uurimistööks.

9 References

Barták, Roman. 1999. “Constraint Programming-What Is behind.” Pp. 7–15 in *Proceedings of the Workshop on Constraint Programming for Decision and Control*.

Chvatal, V. 1979. “A Greedy Heuristic for the Set-Covering Problem.” *Mathematics of Operations Research* 4(3):233–35.

Feige, Uriel. 1998. “A Threshold of Ln N for Approximating Set Cover.” *J. ACM* 45(4):634–52.

Jõgiste, Liisa, Madis Vellamäe, and Martin Rebane. 2012. *Realisation of MONSAMAX and DSR (software in C#)*. Tallinn: Tallinn Technical University.

Karp, Richard M. 1972. “Reducibility among Combinatorial Problems.” Pp. 85–103 in *Complexity of Computer Computations*, *The IBM Research Symposia Series*, edited by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Springer US. Retrieved May 8, 2014 (http://link.springer.com/chapter/10.1007/978-1-4684-2001-2_9).

Kuusik, Rein, and Grete Lind. 2011a. “New Developments of Determinacy Analysis.” Pp. 223–36 in *Advanced Data Mining and Applications*, *Lecture Notes in Computer Science*, edited by Jie Tang, Irwin King, Ling Chen, and Jianyong Wang. Springer Berlin Heidelberg.

Kuusik, Rein, and Grete Lind. 2011b. “New Solution for Extracting Inductive Learning Rules and Their Post-Analysis.” Pp. 121–26 in *IMMM 2011, The First International Conference on Advances in Information Mining and Management*.

Kuusik, Rein, and Grete Lind. 2012. “An Effective Inductive Learning Algorithm for Extracting Rules.” Pp. 339–44 in *Proceedings of the 2011 2nd International Congress on Computer Applications and Computational Science*, *Advances in Intelligent and Soft Computing*, edited by Ford Lumban Gaol and Quang Vinh Nguyen. Springer Berlin Heidelberg. Retrieved May 20, 2013 (http://link.springer.com/chapter/10.1007/978-3-642-28308-6_46).

Lensen, Harri, and Margus Kruus. 2012. *Diskreetne Matemaatika*. 4., parand. ja täiend. tr. Tallinn: [Tallinna Tehnikaülikooli] Kirjastus.

Lund, Carsten, and Mihalis Yannakakis. 1994. “On the Hardness of Approximating Minimization Problems.” *J. ACM* 41(5):960–81.

Praust, Valdo. 1996. *Keerukusteooria Alused*. Tallinn: Ülikoolide Informaatikakeskus.

Rajkovic, Vladislav et al. 1997. “Nursery Data Set.” Retrieved May 10, 2013 (http://archive.ics.uci.edu/ml/datasets/Nursery).

Roosmann, Peeter, Leo Võhandu, Rein Kuusik, Tarvo Treier, and Grete Lind. 2008. “Monotone Systems Approach in Inductive Learning.” *International Journal of Applied Mathematics and Informatics* 2(2):47–56.

Schlimmer, Jeff. 1987. “Mushroom Data Set.” Retrieved May 9, 2014 (https://archive.ics.uci.edu/ml/datasets/Mushroom).

Sedgewick, Robert, and Kevin Wayne. 2011. *Algorithms*. Addison-Wesley Professional.

Sedlacek, Jiri, and Tomas Hurka. 2014. *Visual VM*. Oracle Corporation. Retrieved May 14, 2014 (http://visualvm.java.net/).

Slavík, Petr. 1996. “A Tight Analysis of the Greedy Algorithm for Set Cover.” Pp. 435–41 in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, *STOC ’96*. New York, NY, USA: ACM. Retrieved May 7, 2014 (http://doi.acm.org/10.1145/237814.237991).

Zakrevskij, Arkadij. 2008. *Combinatorial Algorithms of Discrete Mathematics*. Tallinn: TUT Press.

10 Appendices

Appendix 1 – Coverage rule sets for 18-column Mushroom dataset

Approximation algorithm	Unique coverage + greedy combined
10.10 => class 1 (96)	4.3 & 6.7 => class 1 (32)
6.1 => class 1 (400)	4.4 & 6.7 => class 1 (1032)
4.7 => class 1 (16)	4.1 & 12.1 => class 1 (624)
6.2 => class 1 (400)	8.2 & 11.2 => class 1 (864)
2.6 => class 1 (32)	9.1 & 12.7 => class 1 (672)
16.7 => class 1 (96)	2.3 & 6.7 & 12.1 => class 1 (900)
15.7 => class 1 (96)	5.2 & 6.7 & 13.4 => class 1 (888)
16.5 => class 1 (192)	5.2 & 6.7 & 12.1 => class 1 (64)
4.3 & 6.7 => class 1 (32)	19.2 & 6.7 & 9.1 => class 1 (2688)
6.7 & 10.8 => class 1 (756)	19.2 & 9.1 & 11.1 & 13.4 => class 1 (896)
4.1 & 8.2 => class 1 (280)	10.3 => class 2 (1728)
4.4 & 6.7 => class 1 (1032)	10.6 => class 2 (24)
6.7 & 11.2 => class 1 (2496)	4.2 & 12.1 => class 2 (120)
6.7 & 10.1 => class 1 (216)	4.10 & 5.2 => class 2 (672)
4.1 & 12.1 => class 1 (624)	8.1 & 13.3 => class 2 (2228)
6.7 & 10.2 => class 1 (712)	10.5 & 12.1 => class 2 (504)
4.2 & 12.7 => class 1 (48)	2.1 & 5.1 & 6.7 => class 2 (38)
4.8 & 6.7 => class 1 (624)	5.1 & 9.2 & 11.1 => class 2 (264)
3.3 & 14.1 => class 1 (60)	4.9 & 11.1 & 12.1 => class 2 (96)
4.1 & 10.5 => class 1 (12)	3.4 & 19.2 & 8.1 & 12.1 => class 2 (336)
8.2 & 9.1 => class 1 (1056)	9.2 & 11.1 & 12.1 & 16.8 => class 2 (200)
5.1 & 12.7 => class 1 (192)	
2.3 & 19.3 => class 1 (180)	
6.5 => class 2 (2160)	
3.2 => class 2 (4)	
6.4 => class 2 (576)	
6.3 => class 2 (192)	

Approximation algorithm	Unique coverage + greedy combined
16.9 => class 2 (24) 10.3 => class 2 (1728) 6.8 => class 2 (256) 10.6 => class 2 (24) 2.1 & 9.2 => class 2 (12) 4.2 & 12.1 => class 2 (120) 4.10 & 6.7 => class 2 (24) 3.3 & 13.3 => class 2 (1132) 4.1 & 13.3 => class 2 (460) 10.5 & 12.1 => class 2 (504) 2.1 & 5.1 & 6.7 => class 2 (38) 3.3 & 5.1 & 8.2 => class 2 (4) 4.9 & 5.1 & 6.7 => class 2 (32)	

Letter “T” is omitted from the attributes, hence “T3.3” is shortened to “3.3” for better readability.

Appendix 2 – Data of MONSAMAX evaluation

Results of measuring the running time of MONSAMAX, DSR and cover algorithm (unique + greedy) as the number of columns increases in a dataset. Mushroom dataset is used as a base dataset, last columns omitted.

No of columns	11	12	13	14
Rules	12816	20735	35777	69414
DSR rules	643	786	982	1123
Cover rules	26	22	22	22
Extracts	9360	15843	29432	54860
Algo time, sec	4.554	6.103	13.491	23.432
DSR time, sec	0.721	1.460	3.314	7.614
Cover time, sec	0.236	0.193	0.301	0.336
Total time, sec	5.511	7.755	17.105	31.382
Max heap MB	215	347	673	887
Used heap MB	84	232	446	562

No of columns	15	16	17	18
Rules	128454	256878	510149	861756
DSR rules	1326	1326	1363	1634
Cover rules	22	22	22	21
Extracts	101069	202138	402766	736838
Algo time, sec	38.369	84.540	193.387	352.640
DSR time, sec	15.844	32.004	71.820	133.530
Cover time, sec	0.417	0.491	0.348	0.423
Total time, sec	54.631	117.036	265.555	486.593
Max heap MB	1062	1366	1920	2695
Used heap MB	911	1144	1436	2193

Appendix 3 – Data of MONSABAN evaluation

Results of measuring the running time of MONSABAN, DSR and cover algorithm (unique + greedy) as the number of columns increases in a dataset. Mushroom dataset is used as a base dataset, last columns omitted.

No of columns	11	12	13	14
Rules	1740	2256	2908	3516
DSR rules	643	786	982	1123
Cover rules	44	43	41	40
Extracts	9877	17293	32286	61788
Algo time	2.918	3.593	7.328	11.095
DSR time	0.082	0.128	0.166	0.306
Cover time	0.115	0.017	0.019	0.031
Total time	3.115	3.737	7.513	11.433
Max heap MB	119	223	237	213
Used heap MB	56	109	122	140

No of columns	15	16	17	18
Rules	4219	4219	4365	5756
DSR rules	1326	1326	1363	1634
Cover rules	22	22	22	21
Extracts	117312	234588	467714	869753
Algo time	16.911	32.981	68.612	133.715
DSR time	0.357	0.287	0.356	1.709
Cover time	0.216	0.247	0.300	0.415
Total time	17.483	33.515	69.267	135.839
Max heap MB	193	155	200	284
Used heap MB	131	74	121	172

Appendix 4 – Data of MONSAMIN evaluation

Results of measuring the running time of MONSAMIN, DSR and cover algorithm (unique + greedy) as the number of columns increases in a dataset. Mushroom dataset is used as a base dataset, last columns omitted.

No of columns	11	12	13	14
Rules	1740	2256	2908	3516
DSR rules	643	786	982	1123
Cover rules	26	22	22	22
Extracts	9877	17293	32286	61788
Algo time	3.387	4.028	6.483	11.442
DSR time	0.305	0.146	0.279	0.443
Cover time	0.179	0.234	0.395	0.442
Total time	3.871	4.407	7.156	12.328
Max heap MB	118	220	320	479
Used heap MB	62	113	140	267

No of columns	15	16	17	18
Rules	4219	4219	4365	5756
DSR rules	1326	1326	1363	1634
Cover rules	22	22	22	21
Extracts	117312	234588	467714	869753
Algo time	18.067	40.758	85.990	173.204
DSR time	0.514	0.724	0.530	1.004
Cover time	0.455	0.372	0.453	0.545
Total time	19.037	41.855	86.973	174.753
Max heap MB	467	416	365	362
Used heap MB	368	198	286	306

Appendix 5 – data of resource consumption depending on number of rows in dataset

Running time is given in seconds, memory consumption in megabytes.

No of rows in dataset	8124	16248	24372	32496	40620
MONSAMAX, time	4.46	7.23	10.51	15.78	20.52
MONSAMAX, mem	83.92	211.72	367.16	403.40	570.30
MONSAMIN, time	3.14	3.86	6.13	7.24	9.27
MONSAMIN, mem	132.56	132.56	175.48	192.64	308.04
MONSABAN, time	2.82	3.79	6.20	6.97	8.98
MONSABAN, mem	22.89	126.84	226.97	367.16	388.15

Appendix 6 – data of resource consumption depending on number of different values in the dataset

Running time is given in seconds, memory consumption in megabytes.

Time					
No of values	5	10	15	20	25
MONSAMAX	7.03	8.66	8.71	9.14	9.85
MONSAMIN	5.36	6.32	6.83	7.44	8.13
MONSABAN	0.27	2.06	3.53	5.42	7.35
Memory					
No of values	5	10	15	20	25
MONSAMAX	319.48	529.29	526.43	547.41	562.67
MONSAMIN	282.29	494.96	494.00	509.26	538.83
MONSABAN	32.42	93.46	155.45	216.48	272.75

Appendix 7 – Software and source code in Java (CD)

This appendix adds a CD to the thesis with following contents:

- Java executable JAR file with MONSA UI application for running all the algorithms that are implemented in this work
- Source code for the software, and SWT libraries

To run the software, please open command line, navigate to the folder of JAR file and type:

```
java -jar MONSA_UI_20140522.jar
```

Additional parameters can be used to change default settings. E.g. to allow more memory to Java, please use -Xmx parameter. For example, to allow maximum 1800 MB, use:

```
java -jar -Xmx1800M MONSA_UI_20140522.jar
```