KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Modelling Relational Data Mining

Sergey Paramonov

Supervisors:
Prof. dr. Luc De Raedt
Prof. dr. Marc Denecker

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

2018

Modelling Relational Data Mining

Sergey PARAMONOV

Examination committee:
Prof. dr. ir. Hilde Heynen, chair
Prof. dr. Luc De Raedt, supervisor
Prof. dr. Marc Denecker, supervisor
Prof. dr. ir. Gerda Janssens
Dr. Matthijs van Leeuwen
Prof. dr. ir. Karl Meerbergen
Helmut Simonis
(University College Cork, Insight Centre for Data Analytics)
Prof. dr. Christian Bessiere
(CNRS, U. Montpellier, LIRMM)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

2018

© 2018 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Sergey Paramonov, Celestijnenlaan 200A, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Abstract

The goal of data mining is to discover new knowledge in the data. This thesis studies a number of relational data mining problems and demonstrates how they can be modelled and solved. Relational data mining involves dealing with complex and interconnected data, such as spreadsheets or relational tables in databases.

Specifically, we follow a general, *declarative*, view on a class of relational data mining problems using the principles of declarative programming. In a declarative approach one specifies *what* the problem is, instead of *how* to solve it.

The contributions of this thesis are: 1) it shows how Boolean Matrix Factorization can be generalized to the relational setting of *Relational Data Factorization*, and demonstrates how Relational Data Factorization can model a wide range of data mining problems; 2) it introduces the novel problem of *Tabular Constraint Learning*, where one recovers Excel-like formulae in spreadsheets; 3) it introduces the problem of *Sketched Answer Set Programming*, which allows one to mark parts of a logic program as *uncertain*, or *open*; then *recover*, or *synthesize*, the program using a number of positive and negative examples; 4) it demonstrates how our approach can be applied to model various structured and unstructured pattern mining problems under constraints.

To summarize, we investigate how a number of important relational data mining problems can be modelled using Answer Set Programming and FO(.).

Beknopte samenvatting

Het Modelleren van Relationele Datamining

Het doel van datamining is om nieuwe kennis te ontdekken in data. Er werden vele modellen, systemen en technieken voor ontwikkeld. Opdat deze efficiënt zouden werken, moet aan een aantal veronderstellingen voldaan zijn. Een gebruikelijke veronderstelling is dat de records in de data onafhankelijk zijn van elkaar. In complexe data zoals rekenbladen en relationele gegevensbanken klopt deze veronderstelling echter niet. Het modelleren van relationele-datamining is dus niet triviaal. Dit doctoraatsproefschrift bestudeert een aantal problemen in relationele-datamining en toont hoe ze kunnen worden gemodelleerd en opgelost.

In het bijzonder volgen we een algemene, declaratieve kijk op een klasse van problemen in relationele-datamining. Een declaratieve benadering houdt in dat men het probleem zelf beschrijft, in plaats van hoe het op te lossen.

De eerste bijdrage in dit proefschrift is dat het aantoont hoe het klassiek probleem van de factorisatie van Booleaanse matrices kan worden veralgemeend naar de factorisatie van relationele data, en toont dat hiermee een breed spectrum aan problemen kan worden gemodelleerd. De tweede bijdrage in dit proefschrift zit in het nieuwe probleem van het leren van constraints in tabellen, waarbij een algoritme de formules moet reconstrueren in rekenbladen waarvan men enkel de getallen nog heeft. De derde bijdrage is de introductie van 'programmeren met geschetste antwoordverzamelingen' (sketched answer set programming). Hierbij kan men delen van een logisch programma markeren als onzeker, of open, en het volledige programma laten reconstrueren met behulp van voldoende positieve en negatieve voorbeelden. Tenslotte toont het hoe onze aanpak kan worden toegepast op allerlei gestructureerde en ongestructureerde patroonherkenningsproblemen.

Samenvattend: we onderzoeken hoe een aantal belangrijke problemen in relationele datamining kunnen worden gemodelleerd met Answer Set Programming en FO(.).

Acknowledgements

I would like to say a big THANK YOU to everyone who has been involved in making this happen. To all my friends and colleagues who have been an enormous support to me.

First and foremost, I would like to thank Luc De Raedt, my promoter. My scientific perception is shaped by his vision and expertise that he shared with me over the years. This work would not have been possible without his help and ideas. Working with him has taught me to see the bigger picture, and find aspirations worth spending my time and energy on. From him, I learned what it is like to be a researcher.

A special thank you goes to Marc Denecker. It might seem that Marc is always in the office, but actually, he is usually in a beautiful mathematical world that we do not get to see. This work would not have been possible without his many presents – ideas from that world – which he has given me. I am extremely grateful to him for sharing this with me.

I am also grateful to the members of my jury. I am glad that I worked with Matthijs van Leeuwen at the beginning of my PhD. He introduced me the field of data mining, and his comments and advice on the design and presentation of experiments have been priceless. Many insights in modeling have come from working with Gerda Janssens; I am grateful for all the discussions we had on IDP and graph mining. The work on sketching would not have been possible without great help and advice on constraint modelling from Christian Bessiere. Thanks as well to Helmut Simonis: my knowledge of constraint learning and my thesis have improved a lot. I am also grateful to Karl Meerbergen for his insights and comments on relational factorization. And last but not least, I would like to thank Hilde Heynen for being my chairman.

I have had the great pleasure of working with wonderful people at the Computer Science Department. They were not just colleagues, but friends as well. Irma and Anton were the main forces driving our small community of friends at the

department. Jessa is the person binding it all together in Leuven. I am really honoured to be her friend. Francesco – kernels. Thanks Evgeniya for all the wonderful weekend coffees we had. Sebastijan and Nikolina for the summer school and all the fun events they have organized for us. I am grateful to Kurt for his help that always came when needed. To Samuel, for all the great work we have done together, and for making it fun to be in the office. To Ondrej and Gitte, for showing that you can always come back after you have disappeared. To Pedro, for telling us all the stories that happen to him. To Antoine, for his enthusiastic support of all our events in Leuven. To Anna, for showing us geocaching. To Behrouz for teaching us about introverts. To Vladimir, for being the first person to welcome me in Leuven and show me around, and to Leen for taking care of him. To Stefano for showing me the most random object in the world. I still completely accidentally remember it from time to time. To Joana, for suddenly appearing and saving Anton. To Mohit, Nitesh, Arrchit, Guy, Tim, Vincent, Tom, Jonas, Killian, Ellia, Robin, Bogdan for all the coffees we had at the department. A special thank you is to Tias, for being simultaneously colleague, advisor, friend, FOSDEM org and circus teacher. To Ben, for showing that you can complain all the time, and *still* look like a happy person while doing it. To Siegfried, for guiding me through the pattern mining world while playing volleyball and telling all his conference travelling stories.

Ever since I started joining volleyball on Thursday evenings, it has had a special place in my heart. Not only because of the game, but because of all the wonderful people I met there: Luc, Wim, Bertram, Inge, Hanne, Marteen, Vera, Frank, Bram, Irena, Cyrielle, Arturo, and everyone else. I plan to continue this tradition. (But not every Thursday – Antwerp is far!)

I am really grateful to my EMCL friends who have been my support for many years already. To Alina for all her help, and our shared journeys, and stories. It is always great to have friends like her. To Peter and Ronald: we might not see each other for years, but once we meet again, it feels like we only saw each other yesterday.

Thanks to the people at Sentiance who showed great interest in the research I have done. To Vincent and Bertrand, who believed in me even before I finished this thesis. And to Jean, for being a great help, especially with the random things.

To my Phystech friends, who keep in touch after all these years. Maxim, Michael, Konstantin, Nick, Ilya, Alexandra, Nastya and Yuri: every time I travel to Russia, I am always happy to see you.

Throughout my PhD, I have met a lot of interesting people, especially at various events around the world. I am grateful to Tamara from the Summer

School of Science for showing me that chemistry can be fun, and supporting me through my thesis writing frustrations. To Raffael and Agnieszka for showing me London during my conference stay. To Aleksandra for talking and being there for me when I really needed it. To Daria and Pauli for believing in my ideas in structured pattern mining. To Anastasia for all the workshops and hackatons she has organized. To Nastya, for our walks in Saint-Petersburg, and organizing a wonderful wedding party. To Wouter and Seraphine for making me feel at home in Leuven.

When I was six years old, I met Kirill and Olga. You have always been a great support in everything I do, and you still are.

To Alexander Isaakovich, my first real math teacher, for introducing me to the world of mathematics.

Last, but not least, I would like to thank my family. Irina made me the person I am today. She is my constant support, and I could not have done this without her. Thanks to Olga for supporting and talking to me. To Anja and Andrei, for being wonderful visitors and great hosts. And my infinite gratitude goes to my parents for creating an environment in which I could grow and realize what I wanted in life.

List of Abbreviations

ASP Answer Set Programming. 9–15, 23, 177

CP Constraint Programming. 9, 11, 12, 14, 27

CSP Constraint Satisfaction Problem. 28–30

DCA Domain Closure Assumption. 23

FOL First Order Logic. 19, 21

IDP Imperative-Declarative Programming. 10–14

ILP Inductive Logic Programming. 177

ReDF Relational Data Factorization. 36

TaCLe Tabular Constraint Learner. 79, 177

UNA Unique Name Assumption. 23

Contents

Abstract i

Acknowledgements v

Contents xi

List of Figures xvii

List of Tables xxi

1 Introduction 1

1.1 Data mining 1

1.2 Propositional and relational data 2

1.3 Types of relational mining problems 5

1.4 Declarative approach 9

1.5 Logic programming 9

1.6 Constraint programming 11

1.7 Common traits 13

1.8 Contributions 14

1.9 Datasets, code and experimental results 17

2 Background **19**

2.1 First order logic 19

2.2 Answer Set Programming 23

2.3 Constraint programming 27

2.4 FO(·) and the IDP System 30

3 Relational data factorization **36**

3.1 Introduction 37

3.2 Relational data factorization 38

3.2.1 An example 39

3.2.2 Problem statement 39

3.2.3 Approach 40

3.3 Application to data mining problems 42

3.3.1 Tiling 42

3.3.2 The discrete basis problem (DBP) and boolean matrix factorization (BMF) 46

3.3.3 Discriminative k-pattern set mining 47

3.3.4 Block-diagonal matrix form 48

3.4 Beyond classic problems 50

3.5 Implementation 51

3.5.1 General computation methods: greedy and sampling approaches. 51

3.5.2 Data mining problems expressed in the framework . . . 53

3.5.3 Purely relational data factorization 54

3.6 Experiments 55

3.6.1 Solving existing tasks 56

3.6.2 Purely relational data factorization 61

3.6.3 Runtime discussion 65

3.7 Evaluating solver performance . . . 65
3.8 Visualizations for an application of purely relational ReDF . . 68
3.9 Maximum k set coverage analysis of greedy strategy . . . 68
3.10 Related work . . . 71
3.10.1 General models for pattern mining . . . 71
3.10.2 Decomposition of databases, tensors, and real-valued matrices . . . 72
3.10.3 Relational learning . . . 73
3.10.4 Declarative data mining . . . 74
3.11 Conclusions . . . 75

4 Learning constraints in tabular data **77**

4.1 Introduction . . . 78
4.2 Related work . . . 80
4.3 Formalization . . . 82
4.3.1 Terminology . . . 82
4.3.2 Constraint templates . . . 87
4.3.3 Problem definition . . . 89
4.3.4 Other considerations . . . 89
4.4 Approach to tabular constraint learning . . . 90
4.4.1 Step 1: Table extraction . . . 92
4.4.2 Step 2: Block detection with *BlockDetect* . . . 93
4.4.3 Step 3: Constraint learning algorithm . . . 93
4.4.4 Optimizations . . . 96
4.5 Evaluation . . . 100
4.5.1 Experimental setup . . . 100
4.5.2 Results on the running example . . . 101
4.5.3 Effectiveness on synthetic data . . . 102

4.5.4 Effectiveness on real spreadsheets . . . 105
4.6 Applications . . . 108
4.6.1 Auto-completion . . . 108
4.6.2 Error detection . . . 109
4.6.3 Spreadsheet domains and use-cases . . . 110
4.7 Future Work . . . 111
4.7.1 Pindakaas . . . 111
4.7.2 Richer type system, header labels and advanced formatting 115
4.8 Conclusions . . . 116
4.9 Spreadsheet dataset overview . . . 117

5 Constraint sketching 119
5.1 Introduction . . . 119
5.2 ASP and sketching . . . 122
5.3 Rewriting schema . . . 123
5.4 Sketched aggregates . . . 126
5.5 Experimental evaluation . . . 128
5.6 Related work . . . 136
5.7 Discussion and conclusions . . . 138

6 Relational Pattern Mining 140
6.1 Introduction . . . 141
6.2 Preliminaries . . . 143
6.2.1 Patterns . . . 144
6.2.2 Condensed pattern representations under constraints . . . 148
6.3 Hybrid ASP-based mining approach . . . 149
6.3.1 Encoding local constraints . . . 151
6.3.2 Encoding global constraints . . . 151

6.4 Covering with approximate patterns 156
6.5 Evaluation 161
6.6 Related work 171
6.7 Conclusion 173

7 Conclusions and future work **175**

7.1 Summary and conclusions 175
7.2 Contributions and key questions 176
7.3 Discussion and future work 178

A Appendix **181**

A.1 ReDF additional materials 181
A.1.1 Correctness proof of encoding 3.10 181
A.1.2 Additional ASP encodings 183
A.1.3 Precision-recall curves 185
A.1.4 Additional tiling experiments with sampling 185

Bibliography **187**

List of publications **199**

List of Figures

1.1 The famous Fisher's Iris dataset with first four columns being numeric properties of irises and the last column being its class. A typical problem then is to predict the class in the last column based on the given numeric values in the first four columns . . 3

1.2 A spreadsheet is an example of a relational dataset: it has multiple connected tables, constraints between tables, formulae over columns and rows, numeric and non-numeric entries 4

1.3 An example of a relational database: music school 7

2.1 Graph coloring of the Petersen's graph using three colors . . . 20

2.2 An ASP guess-and-check paradigm modelling example on the n-queen problem 27

3.1 Example of Boolean tiles and their coverage 43

3.2 Relational tiling: two relational tiles (right) in a toy dataset (left) concerning cars. 43

3.3 Example of a 0/1 database with a tiling consisting of two overlapping tiles (darkest shaded area corresponds to the intersection of the two tiles), due to Geerts et al. 2004 45

3.4 Re-arranging a matrix in block-diagonal form (Animals dataset): (a) regular, (b) with penalties, (c) with noisy blocks and penalties 48

3.5 Tiling comparison (runtime, coverage) with LTM-k (Mushroom dataset) 58

3.6 Boolean matrix factorization on datasets Animals. Runtime and coverage are depicted for different factorization ranks. 59

3.7 Discriminative pattern set mining summary: runtime (left) and coverage (right) 60

3.8 Discriminative pattern set mining (Tic-tac-toe dataset): precision (left) and recall (right) for different α, i.e., for varying weights of covering negative transactions 60

3.9 Clustering in topics by ReDF. Red nodes represent co-authors, blue their university cities, white nodes venues and green topics that bind them together. If there is an edge between a topic and a node, then there is a corresponding element in the relation (i.e., *interestedIn*, *specializedIn* or *inField*) 62

3.10 Discriminative learning in the purely relational setting; precision (left) and recall (right) for different α, i.e., for two weights of covering negative atoms. 65

3.11 Determining the best ASP solver parameters for tiling with the model of Listing 3.10: (a) the runtime (in s) needed for mining each subsequent individual tile, (b) runtime summary (in s) for first 25 tiles 66

3.12 An application of purely relational factorization: intersection of publication data (in red) and latent topics (in blue) of Bart Goethals with publications of his co-author Floris Geerts (highlighted in pink) in the graph. A latent topic belongs to an intersection if both have the same co-author, university and venue in the publication records. 69

3.13 An application of purely relational factorization: intersection of publication data (in red) and of latent topics (in blue) of Bart Goethals with publications of Pauli Miettinen (not his co-author). A latent topic belongs to an intersection if both have the same co-author, university and venue in the publication records. . . . 70

4.1 Running example 83

4.2 Dependency graph of Table 4.1; an arrow from s_1 to s_2 indicates that s_2 depends on s_1: its signature includes s_1. 97

4.3 Loglog plots of the experiments with synthetic (random) data. In plot *a* the number of vectors is increased; in plot *b* the size of the vectors is increased; and in plots *c* and *d* the number of blocks is increased (the block size is decreased). Missing values for the no-CSP baseline indicate that it timed out. 102

4.4 This figure visualizes the data from Table 4.3, i.e., the average properties and performance of spreadsheets within every benchmark category. Values are scaled relatively to the *maximum value across all individual spreadsheets*. Green bars indicate standard deviation within the category. 107

4.5 Auto-completion works by learning constraint on a snapshot (green tables T_1 and T_2) and combining those constraints with new data ΔD (purple). For example, in this figure a user types a new value *West*. The system can then suggest values (blue) for the other cells in that row. 109

4.6 We clearly see that FlashFill is highly sensitive to the noise (the figure is due to Devlin et al. (2017)) 112

4.7 An example of a nested spreadsheet formula 113

4.8 Possible visual interface for Pindakaas 113

5.1 Collection of sketches and an example of rewriting used in the paper 121

5.2 Addressing experimental questions $\mathbf{Q}_1$-$\mathbf{Q}_2$. $\mathbf{Q}_1$ and $\mathbf{Q}_2$: convergence without preferences (a), across the dataset (b), for each problem (c). Log-scale. 131

5.3 $\mathbf{Q}_3$: the effect of different number of sketched variable on the number of solutions and precision, FAP (a, b); N-queens (c). Log-scale. 132

5.4 Addressing experimental questions $\mathbf{Q}_4$ and $\mathbf{Q}_5$: generalization accuracy on Latin Square (a), the effect of the encoding variations on the number of solutions (b), on precision (c). 133

6.1 Graph examples 145

6.2 Isomorphic subgraphs for graphs in Figure 6.1 147

6.3 Examples of tiles in a matrix 160

6.4 Investigating $\mathbf{Q}_1$: comparison with pure ASP model (6.4a) and maximal (6.4b) sequence mining on JMLR, Unix Users, and iPRG datasets. 163

6.5 Investigating $\mathbf{Q}_1$: closed (6.5a), and skyline (6.5b) sequence mining on JMLR, Unix Users, and iPRG datasets. 164

6.6 Investigating $\mathbf{Q}_2$: comparison with DP (Négrevergne et al. 2013) (6.6b, 6.6a 165

6.7 Investigating $\mathbf{Q}_2$: comparison with DP (Négrevergne et al. 2013) (6.7a); and $\mathbf{Q}_3$: the effect of local constraints on runtime (6.7b) 166

6.8 $\mathbf{Q}_4$: comparing ASP solvers performance – WASP and Clasp on itemset and sequence mining tasks 167

6.9 $\mathbf{Q}_4$: comparing ASP solvers performance – WASP and Clasp on itemset and sequence mining tasks 168

6.10 Investigating $\mathbf{Q}_5$: evaluation on standard graph datasets. Log-scale 169

6.11 Investigating $\mathbf{Q}_5$: evaluation on standard graph datasets and comparison to the declarative ILP system for graph mining. Log-scale 170

A.1 Precision recall curves for discriminative learning for different α, i.e., for two weights of covering negative atoms. For different number of learned rules, indicated on each point (starting from zero). 185

A.2 Tiling comparison (runtime, coverage) with LTM-k (Mushroom dataset) 186

List of Tables

3.1 Dataset properties. For each dataset, we specify whether the attributes have Boolean or categorical domains, the number of tuples and attributes, and the average number of distinct values per attribute 55

3.2 Maximum k-Tiling 56

3.3 Maximum k-Tiling with overlap. The maximal allowed overlap is limited by parameter N 58

3.4 Experimental summary for pure relational factorizations from Subsection 3.4 64

3.5 Maximum k-tiling: grounding and solving – avg time per tile (s), and their time ratio 67

4.1 Constraint templates implemented in *TaCLe*. Only blocks in **bold** may contain more than one vector; $discrete(x)$ is a shortcut for: $textual(x) \vee numeric(x)$. Templates with $*$ are *structural*, i.e. they are not functional and not available in most spreadsheet software. For templates with † (called *aggregate* and *conditional aggregate*), the table shows variants for *SUM* but *TaCLe* also supports *MAX*, *MIN*, *AVERAGE*, *PRODUCT* and *COUNT*, e.g., $B_2 = MAX_{row}(\mathbf{B}_1)$ or $B_2 = COUNTIF(B_{fk}, B_{pk}, B_1)$ 86

4.2 Translation of signature requirements to input block constraints. The following need not be relaxed: $length(B) = x$, $orientation(B) = x$, $table(B) = x$ and $size(B) \geq x$. 94

4.3 Summary of properties and results per spreadsheet category (time in s) 106

4.4 Feature comparison between Pindakaas, TaCLe and FlashFill . 114

4.5 Constraint occurrence in the collected dataset in absolute values by category 118

5.1 Dataset summary: the number of sketched variables, of rules, of particular types of sketched variables, e.g., "# ?not", indicates how many atoms with the sketched negation are in the program. 129

5.2 Key feature comparison between SkASP and ILASP 137

6.1 Feature comparison between various ASP mining models and dominance programming ("–" : "not designed for this datatype", ✓* : only maximal is supported) 144

6.2 Transaction database 145

6.3 Sequence database 145

6.4 Addressing $\mathbf{Q}_6$: approximate tiling problem solution using specialized native algorithm for tile candidate generation and ASP encoding for selecting the resulting tiling (Divorce dataset) 171

6.5 Addressing $\mathbf{Q}_6$: approximate tiling problem solution using specialized native algorithm for tile candidate generation and ASP encoding for selecting the resulting tiling (Glass dataset) . 172

A.1 An empiricial reconstruction of β, given the values of α 186

Chapter 1

Introduction

- Homer, is it the way you pictured PhD life?
- Yeah, pretty much, except we drove in a van solving mysteries.

Could be Homer Simpson

This thesis is situated in the research fields of data mining, constraint learning and logic programming. We introduce each of them in turn and provide an overview of the contributions of the thesis and its general structure.

1.1 Data mining

The ability to write down a mathematical model and to simply push a button to get the answer is an appealing idea. It is one of the holy grails of Artificial Intelligence, called Declarative Programming, where one specifies what the problem is and not how to solve it. Today in the age of data, we would like to state what type of knowledge we are looking for, feed the data in and, as a result, (almost magically) find new insights into the data. What we do not want is to write a lot of tedious code. Unsurprisingly, this is a hard challenge.

The *knowledge* extracted from the data can manifest itself in multiple forms. For example, an interesting piece of data, *typically a substructure*, is referred to as a *pattern* (Han 2005). The most known pattern learning task is *frequent*

pattern mining, or pattern mining for short (Han et al. 2007). At a higher and more general level, often referred to as *theory mining*, it can be formalized as follows:

> **Definition** (Theory Mining)**.** Let D be a dataset, P be the language of allowed patterns and ϕ be a predicate of interest, the mining problem is then the problem of finding all patterns p in P such that $\phi(p, D)$ holds.

Simply speaking, we enumerate objects that have a certain property of interest. Various measures of relevance or "interestingness" of a pattern exist (Guns et al. 2013a).

The rest of the chapter is structured as follows. In Section 1.2 we introduce two types of data used in data mining. Then, in Section 1.3 we present the types of relational pattern mining problems that we investigate in this thesis. In Section 1.4, we propose an approach to these problems, while in Sections 1.5 and 1.6, we present techniques for this approach to work. In Section 1.7, we demonstrate the similarities and common traits of the problems we investigate. We conclude with contributions in Section 1.8 and links to experimental data and code in Section 1.9

1.2 Propositional and relational data

There are two common types of data in data mining: propositional and relational. Propositional data is in a, so called, single-tuple single-table format, where each row is an independent record and each column is an attribute. It is commonly used in machine learning and data mining for training models. Relational data, on the other hand, consists of multiple tables and there are constraints and dependencies between tables. This format is typically more complex.

To illustrate propositional data, let us consider one of the most classic datasets in statistics and machine learning – Fisher's Iris data set[1]. Figure 1.1 depicts a sample of data in the dataset. Looking carefully at this data, we can establish the following: every row is an example and every column is an attribute. We can also easily see what kind of problem it suggests: based on the number of values (in the first four columns), a tuple from R^n, we need to predict the class of Iris (in the last column). Many machine learning algorithms are predictive models. In this thesis we do not work with predictive models.

[1] https://en.wikipedia.org/wiki/Iris_flower_data_set

Sepal length	Sepal width	Petal length	Petal width	Species
5.1	3.5	1.4	0.2	*I. setosa*
4.9	3.0	1.4	0.2	*I. setosa*
4.7	3.2	1.3	0.2	*I. setosa*
4.6	3.1	1.5	0.2	*I. setosa*
5.0	3.6	1.4	0.3	*I. setosa*
5.4	3.9	1.7	0.4	*I. setosa*
4.6	3.4	1.4	0.3	*I. setosa*
5.0	3.4	1.5	0.2	*I. setosa*
4.4	2.9	1.4	0.2	*I. setosa*
4.9	3.1	1.5	0.1	*I. setosa*
5.4	3.7	1.5	0.2	*I. setosa*
4.8	3.4	1.6	0.2	*I. setosa*
4.8	3.0	1.4	0.1	*I. setosa*

Figure 1.1: The famous Fisher's Iris dataset with first four columns being numeric properties of irises and the last column being its class. A typical problem then is to predict the class in the last column based on the given numeric values in the first four columns

Classical machine learning algorithms such as the induction of decision trees (Quinlan 1986) or Bayesian networks (Pearl 1988) work with this kind of propositional data and representations. Propositional representations do not allow to represent naturally problems involving multiple complex objects together with the relationships that hold between them. This limitation has been pointed out by a number of researchers, e.g., in the seminal work of Plotkin (1971), who has introduced the use of first order logic for machine learning. Logical and relational learning overcomes this limitation by focusing on expressive knowledge representation languages such as first order logic to handle relational data (De Raedt 2008a). This approach allows to represent a variable amount of complex objects together with their relationships – we refer to this problem setting as *relational*.

Contrary to this classic machine learning setting with propositional data, relational learning is looking into a more complex and interconnected relational data. A real-world example and, probably very well familiar to the reader, a *spreadsheet* with its formulae is an excellent example of relational data: it contains a number of relations or *tables*, and their number is never fixed in advance, since a spreadsheet might contain an arbitrary number of them. Also,

spreadsheets combine both textual and numeric data. Furthermore, formulae and constraints over the tables play the role of relationships.

Product	State	Sep	Oct	Nov	Sub-total		Product:	Sales:
Cherries	CA	200	360	230	790		Apples	2070
Bananas	AZ	350	230	150	730		Bananas	2420
Apples	TX	180	270	200	650		Lemons	740
Bananas	KS	400	240	310	950			
Lemons	AL	250	360	130	740			
Apples	FL	120	120	380	620			
Bananas	LA	330	270	140	740			
Apples	KY	110	320	370	800			

Figure 1.2: A spreadsheet is an example of a relational dataset: it has multiple connected tables, constraints between tables, formulae over columns and rows, numeric and non-numeric entries

Figure 1.2 demonstrates this idea: there are multiple tables connected to each other, the table on the right has aggregated data over rows in the left table, which means that generally speaking rows cannot be seen as independent: adding a row to the first table should update the values in the right table. There are various data types present here: numeric, categorical and might have been textual as well. Furthermore, constraints that hold in a spreadsheet do not have to be functional, i.e., having a form of $f(\bar{x}) = y$, where given input $\bar{x}$ you can always compute y. An example of a non-functional constraint, as we called it here, is a foreign key. Simply speaking a foreign key constraint enforces the subset relation. In Figure 1.2, we see that the right table has an aggregated information over the left table. Namely there is a `SUMIF` constraint [2], then there is a subset relationship between the column *Product* in the right table and in the left – it cannot be the case that aggregated information contains a product not listed in the table on the left.

Depending on the data, one can consider different types of pattern mining tasks, which we now introduce.

[2] https://support.office.com/en-us/article/sumif-function-169b8c99-c05c-4483-a712-1697a653039b

1.3 Types of relational mining problems

In this section we discuss main relational data mining problems that we investigate in this thesis: relational pattern mining, constraint learning and relational data factorization.

Constraint learning Constraint Learning is the research field concerned with finding constraints that hold in data (Bessiere et al. 2005a, 2013, 2017b). Constraint learning in relational data is of a great importance to us as we will demonstrate here.

Constraint learning can be formulated in terms of theory mining from Section 1.1 as follows:

> **Definition** (Constraint Learning)**.** Let D be a dataset, the language of patterns P be the set of interesting constraints C, ϕ be the constraint satisfaction function, then the problem is to find all constraints c in C such that $\phi(D, c)$ is satisfied, i.e., $c(D)$ holds.

Simply speaking, constraint learning is pattern mining where patterns are constraints and the predicate of interest is that the constraints are satisfied by data.

Let us illustrate this idea on a spreadsheet in Figure 1.2. There are two constraints to be found: `ROWSUM` and `SUMIF`. The former sums values in rows in the left table (for the columns *Sep*, *Oct* and *Nov*) and the latter aggregates values (the column *Sales*) by category (the column *Product*) in the right table. Let us indicate why it might be an important practical problem. Imagine, if we only change one value, the whole dataset becomes inconsistent, since there are dependencies in the spreadsheet to be met, e.g., for example, the last column in the left table must be updated, as well the corresponding category value in the right table. Then, we would like to, first, discover all constraints in the data. Second, update the cell we are interested in and then, third, recompute the values. As a result, we would obtain an updated and consistent spreadsheet.

This practical problem is a clear constraint learning problem in a relational setting: we need to enumerate interesting properties, i.e., various spreadsheet constraints, having a form of first order statements and being able to carry out inference on top of a learned model and of relational data (multi-table, interconnected data containing multiple data types). And this is, indeed, the kind of problem that is problematic for classic data mining algorithms and require relational methods we develop and investigate here.

For technical details we refer to Chapter 4 and the work of Paramonov et al. (2017b).

Relational Pattern Mining Let us formulate relational pattern mining problem in terms of theory mining from Section 1.1.

> **Definition** (Relational Pattern Mining)**.** Let relational data be a relational database D, the language of patterns P be the set of relational queries Q, and ϕ be the query satisfaction relation, then the task is to find all queries q in Q such that $\phi(q(D))$ evaluates to true.

A special case of relational pattern mining is *frequent pattern mining*. The problem of frequent pattern mining has as the query satisfaction relation *the minimum frequency constraint*. Given a frequency threshold t and a predicate of interest, i.e., what we count, we need to find all *queries* that occur frequently enough, above the threshold t, in the database D. Often we are interested not only in frequent substructures, but also in substructures that have certain interesting properties, such as being maximal in size or restricted in its length or having certain topological properties.

For example, let us assume there is a school database D with three tables *pupil(PId,FName,LName)*, *class(PId,ClassId,Subject)*, *takes_test(PId,ClassId,Date,Result)*, which contains information about pupils, classes they are in and tests they have taken. Typically, in frequent query mining, we have a target value we count, in this case, it can be for example PId (Pupil Id), then our task would be to enumerate all queries that match D often enough. Let us give a couple of examples how a typical output might look like:

$q_1(PId) \leftarrow pupil(PId,FName,LName).$

$q_2(PId) \leftarrow class(PId,ClassId,Subject).$

$q_3(PId) \leftarrow pupil(PId,FName,LName),\ takes_test(PId,ClassId,Date,Result).$

$q_4(PId) \leftarrow class(PId,ClassId,Subject),\ takes_test(PId,ClassId,Date,Result).$

The first query q_1 is simply taking the table *pupil* and projecting on its first column *PId*, q_2 does the same for the *class* table and q_3 is making a join between *pupil* and *takes_test* on the column *PId*, q_4 is doing the same but with the tables *class* and *takes_test*.

Assume our school schema is populated with a data from a music school, as in Figure 1.3 and we have the frequency threshold $t = 4$, then the queries return

Pupil

PId	Fname	LName
p1	Nick	Mason
p2	David	Gilmour
p3	Syd	Barret
p4	Roger	Waters
p5	Richard	Wright

Class

PId	ClassId	Subject
p1	c1	drums
p2	c2	singing
p2	c3	guitar
p2	c4	bass
p3	c2	singing
p3	c3	guitar
p3	c5	piano
p4	c2	singing
p4	c4	bass
p5	c2	singing

Takes test

PId	ClassId	Date	Result
p1	c1	1965	passed
p2	c2	1965	passed
p2	c3	1965	passed
p5	c3	1965	passed

Figure 1.3: An example of a relational database: music school

corresponding values:

$$q_1(D) = \{p1, p2, p3, p4, p5\}$$

$$q_2(D) = \{p1, p2, p3, p4, p5\}$$

$$q_3(D) = \{p1, p2, p5\}$$

$$q_4(D) = \{p1, p2, p5\}$$

Then the frequencies of q_1 and q_2 are above the threshold $t = 4$ and the frequencies of q_3 and q_4 are below. That makes q_1 and q_2 frequent queries, while q_3, q_4 are not frequent.

Looking in Figure 1.3 at the relational data represented as a relational database, we can contrast this representation of relational data with a spreadsheet in Figure 1.2. They are both examples of relational data but their representations are different. Relational databases have a strict schema and structure, while spreadsheets might have records in columns and rows, it does not enforce type consistency, constrains and dependencies can be across multiple tables mixing columns and rows, etc.

Relational Data Factorization Another type of problems that is important for this thesis: is that of matrix and tensor factorizations. Factorizing matrices has a long history (Golub and Van Loan 1996), in this thesis we do not focus on classical continuous decompositions such as SVD or LU, but on its Boolean version, called *Boolean Matrix Factorization* (Miettinen 2009). The main reason for that is we are going to deal with relational data and non-numeric data in this thesis. We work with the relational version of BMF under constraints, we call this problem *Relational Data Factorization*. Simply speaking, for a relation q, we need to find a join of multiple relations $q_1, q_2, \ldots, q_n$ under constraints such that this join is close to the original relation.

For example, we can be given a relation *coauthor(Name,Uni,Venue)* (shortened below as (N, U, V)) and we would like to decompose it into three binary relations joined by a latent (hidden) variable, that can be interpreted as a potential topic (T) that binds them together:

$$\mathit{decomposed}(N, U, V) \leftarrow \mathit{interestedIn}(N, T), \mathit{specializedIn}(U, T), \mathit{inField}(V, T).$$

Of course, the relations we obtain can be subject to constraints such as "a university cannot specialize more than in n subjects" or "there must be at least two venues in a topic", etc. For details we refer to Chapter 3.

In general and not only in the examples presented in this section, the properties of interconnectedness and expressivity of the constraints make relational problems hard to model using traditional propositional approaches, while they can be clearly seen as a first order problem. This gives us confidence that for a number of important problems such as recovering formulae in spreadsheets, factorizing relational data and mining frequent structures in relational data, it is essential to look into richer knowledge representation languages for machine learning.

1.4 Declarative approach

Many Data Mining problems, and especially pattern mining problems, are **combinatorial search problems**, which in turn have two main components, namely a model and search, which can be conceptually written as:

$$\text{Combinatorial problem} = \text{Model} + \text{Search} \tag{1.1}$$

Therefore, it seems natural to model pattern mining problems using systems designed to perform search in a declarative manner, such as Constraint Programming (Rossi et al. 2006) or Answer Set Programming (Lifschitz 2008).

The declarative approach can be schematically depicted in Eq. 1.1, the key idea here is the following: when we deal with the imperative approach, even a slight change in the specification of a problem typically leads to a significant change in the implementation of the algorithm solving this new problem variation. Contrary to that in the declarative paradigm, we translate problem specification into a model in an executable declarative language that can handle various rules and constraints. As a result, a slight change in the specification translates, ideally, into an additional constraint being added or a constraint being changed. The key advantage of this approach is it is easy to solve closely related problems by varying the constraints. In our setting, we can think of data mining problem formulated using mathematical constraints as a model and Constraint Programming (CP) or Answer Set Programming (ASP) engines as general solvers. In the two following sections we introduce both formalisms ASP and CP and their corresponding systems respectively.

1.5 Logic programming

In this section we introduce Logic Programming and Answer Set Programming that we have discussed in the context of the declarative approach.

In Eq. 1.1, we indicate that *a solver* is going to take care of computing a specification describing the problem. Also, we already know that **data** is relational, i.e., represented as relations. We know that *Logic* naturally can represent relations and relational data in general, then it is logical and natural to use *Logic Programming* as the solving technique in our declarative approach.

In his seminal work Robert Kowalski (1979) proclaimed:

$$\text{Algorithm} = \text{Logic} + \text{Control}.$$

The intuition behind this statement is that you can turn logical rules into a computational process. Why is this important for us? Logic deals with relations and relational structures, it is naturally designed to work with relational data. Therefore, if we can combine logic with computation, we obtain a computational engine that can work in our relational learning setting: computationally it means that an executable model we obtain, following the schema in Eq. 1.1, is close to the original representation of the data and to the specification of the problem.

Ideally, we would like to have such a high level logic programming language that we can write human-like specifications and directly execute them in the system (Blockeel et al. 2012).

Let us demonstrate how logic programming can be used to model and solve problems.

Example 1.1 (Modelling Sushi preferences with logic)**.** There are three people: Ronald, Peter and Alina. We know that people who are not allergic to fish and not vegans like sushi; furthermore, we know that Peter is allergic to fish and Ronald is vegan. Who likes sushi then? We can model this puzzle as a simple logical implication:

$$\textit{likes_sushi}(X) :- \textit{person}(X),\ \text{not}\ \textit{allergic}(X),\ \text{not}\ \textit{vegan(X)}.$$

$$\textit{allergic(peter)}.$$

$$\textit{vegan(ronald)}.$$

The following notation in Prolog (Programming in logic) (Colmerauer and Roussel 1996) mimics the logical implication:

$$\forall X : \textit{person}(X) \wedge \text{not}\ \textit{allergic}(X) \wedge \text{not}\ \textit{vegan}(X) \implies \textit{likes_sushi}(X) \quad (1.2)$$

Therefore, we can think of Logic Programming as an executable Logic, that we can use to model and solve relational problems.

In this thesis we use modern Logic Programming engines, such as Answer Set Programming (ASP) (Gebser et al. 2012; Lifschitz 2008) and Imperative-Declarative Programming (IDP) (De Cat et al. 2014) that naturally support constraints and various types of logical inference.

ASP and IDP are declarative logic programming formalisms. Contrary to Prolog, which is essentially a full programming language based on logical inference and provides full control on the execution flow, both ASP and IDP systems fully decouple the problem specification from the execution flow. In this sense they are closer to Constraint Programming. This makes them an ideal candidate for our general solver in Eq. 1.1, since both of them support relational data natively and can solve problems declaratively based on a form of first order logic. We often refer to both paradigms (and to the systems, associated with them) as *modern logic programming*.

Let us briefly demonstrate the power of ASP on the graph colouring problem (or the map coloring problem, as it is known in the popular culture) (Gebser et al. 2012). We assume that `node/1` encodes nodes of the input graph and `edge/2` encodes edges of the given graph. First, we declare colours to be used for graph colouring as

```
\texttt{colour(red). colour(green). colour(blue).},
```

the colors to be used in the `colouring/2` predicate, then we introduce the following constraint (called a *choice rule*):

```
1 {colouring(X,C) : colour(C)} 1 :- node(X).
```

The rule states that for each node `X` the predicate `colouring(X,C)` has one and only one atom to be true for values of `C` among the colours defined in the predicate `colour(C)`, simply speaking, this constraint enforces a one-to-one mapping from the nodes (the countries) to the colours.

Then, we need to define a constraint that is saying what mapping is acceptable, in ASP, it is done by defining what is NOT an acceptable solution:

```
:- edge(X,Y), colouring(X,C), colouring(Y,C).
```

If there is an edge between two nodes (countries) `X` and `Y` and they are mapped to the same colour `C`, then it is not a valid solution. As a result, we obtain solutions, where all neighboring nodes of a graph (or neighboring countries) have different colours.

1.6 Constraint programming

Constraint Programming is a declarative methodology for solving combinatorial problems that also can be used in Eq. 1.1. The key difference is that logic

programming is typically focused on rules and inference, while CP is solely focused on constraint satisfaction and usually based on a pre-defined set of constraint primitives (such as comparison, sets operations, etc). In this thesis we prefer constraint programming over logic programming for numeric-heavy problems and in all other cases we use logic programming ASP or IDP.

The essence of the CP methodology consists of writing a set of constraints, such as $X + Y > Z$, and inferring a solution to them, such as $X = 1, Y = 2$ and $Z = 0$. This set of constraints is referred as a *specification* or a *model*; for this, multiple Constraint Programming languages exist, such as MiniZinc (Nethercote et al. 2007) or Essence (Frisch et al. 2008). A *solution* for a model is an assignment, as for example, the one we just indicated, that satisfies *all* constraints. This general methodology is declarative, since a user writes a set of constraints corresponding to the properties of her problem and a CP solver is looking for a solution: this clearly demonstrates that it is the task of the solver to figure out *how* to find a solution, while it is the user's responsibility to specify *what* the problem is in terms of writing these constraints.

Example 1.2 (MiniZinc CP program)**.** Let us illustrate how constraint programming can solve problems on a small example of the subset sum problem using the MiniZinc system.[3]

The problem is the following: given a list of numbers and a target value, we need to find a subset of these numbers that sum up to the target value.

```
1  array[int] of int: number;
2  int: target;
3
4  set of int: NUMBER = index_set(number);
5  var set of NUMBER: selected;
6
7  constraint sum(i in selected)
8                 (number[i]) = target;
9  solve satisfy;
10
11 output [show(selected)];
```

Listing 1.1: An example of a MiniZinc program

The most important part of program in the `constraint` part, where we state that the sum of selected numbers is equal to the target value. The rest is variable declaration and the index set computation – technical details at this point.

[3] https://github.com/MiniZinc, see Examples

1.7 Common traits

There is a number of common traits and assumptions that the problems, we investigate in the thesis, share. Typical machine learning algorithm make use of the, so called, single-tuple single-table assumption. This assumption states that the data can be represented using attribute-value pairs. As we have demonstrated the problems we consider break this assumption. This indicates, the first, common feature – they are all relational data mining problems.

Second, the pattern mining problems we investigate are a form of combinatorial search and enumeration problems: we are looking for certain interesting patterns that hold in the data under constraints, e.g., all maximal (in size) conjunctive queries that hold in the data frequently enough or in case of tabular constraint learning we enumerate all spreadsheet formulae (which can be viewed as constraints, see Theory Mining in Section 1.1) that hold between spreadsheet tables.

Third, all of these problems, we investigate, are relational problems under a broad variety of constraints, e.g., discovering constraints in tabular data is not limited to only the list of specified constraints but it is designed to support the whole *language* of tabular constraints, i.e., new constraints can be added there in a declarative manner in the accordance with the *elaboration tolerance* principle by McCarthy (1999). In the same way relational pattern mining is not limited only to its core version of finding frequent patterns (such as graphs, queries, sequences or itemsets) but also in finding its *condensed* (such as maximal, closed, free, etc) representations under various constraints (such as length, size, or topological constraints)

Fourth, as we have just indicated, our solutions are declarative in nature, as indicated in Eq. 1.1 Once we modify the problem by adding an extra constraint or feature, we do not have to modify the algorithm completely (as so often happens in the imperative approach (Yan and Han 2002; Yan et al. 2003)).

Last but not least is the way we approach them, our approach is based on declarative constraint and logic programming. We demonstrate how our pattern mining problems can be modelled using modern logic and constraint programming systems such as ASP and IDP. In addition to the mathematical models of the relational problems, we provide an evaluation of the performance of the resulting systems using an extensive experimental evidence on the behaviour of such systems. To the best of our knowledge, this is the first empirical study of such scale investigation applications of modern logic programming to relational data mining problems.

1.8 Contributions

Given the abundance of relational data (such as spreadsheets, graphs, databases and logical structures) for various machine learning and data mining problems, and the success in the development of declarative solvers (such as ASP, IDP and CP) a natural question arises: *"How can we model and solve relational data mining problems using declarative programming techniques?"*.

The **main contribution** of this thesis is in the application of modern logic programming and constraint programming languages, such as ASP and IDP, to these novel relational data mining problems.

Concretely, we investigate how the following data mining problems can be modelled and solved using a declarative approach based on logic and constraint programming.

- $\mathcal{Q}_1$: What are the challenges and advantages of generalizing classic data mining problems, such as the Boolean Matrix Factorization problem, into the relation setting? We address this question by proposing **Relational Data Factorization.**
- $\mathcal{Q}_2$: How can the constraint learning problem be formulated and modelled in the relational setting, where data is represented as spreadsheets, i.e., connected relations, and constraints are tabular Excel-like formulae? We propose as an answer **Tabular Constraint Learning.**
- $\mathcal{Q}_3$: Can we model and automatically complete a partially specified logic program? A partially specified program is a program where certain parts are marked as uncertain. We propose a declarative approach to this problem, called **Sketched Answer Set Programming**.
- $\mathcal{Q}_4$: How can we apply declarative relational mining techniques, such as logic programming, to structured frequent pattern mining, such as sequence, graph and query mining? We introduce an **ASP-based Relational Pattern Mining Model** as a possible solution to this problem.

Addressing $\mathcal{Q}_1$, **Chapter** 3 presents the novel problem of Relational Data Factorization. It demonstrates how this problem generalizes various data mining problems such as Database Tiling and Boolean Matrix Factorization. We start with the most basic setting and demonstrate how various problems can be modelled within the framework by adding or modifying the constraints. Then, we show how the framework can be used for new and interesting applications and propose Answer Set Programming as a method to solve the problem

in the general case. For each problem we provide extensive experimental evidence, including solver parameter tuning. The chapter consists of the research previously published in the following paper:

> Sergey Paramonov, Matthijs van Leeuwen, Luc De Raedt: Relational data factorization, Machine Learning, Springer, December 2017, Volume 106, Issue 12, pp 1867–1904.

With regards to $\boldsymbol{\mathcal{Q}}_2$, **Chapter** 4 introduces the novel problem of Tabular Constraint Learning and the system, called TaCLe (pronounced as the word "tackle"). The problem setting is straightforward to explain: we have an Excel file with formulae. The file is imported as CSV. The formulae are lost. Can we reconstruct them? We can already see that the problem setting is different from the standard machine learning problem. In a spreadsheet, columns are no longer variables and rows are no longer records. Textual and numeric data are mixed. Spreadsheet functions, such as Fuzzy-lookup, are unorthodox and unseen in the constraint programming and constraint learning research communities.

The chapter consists of the research previously published in the following papers:

> Samuel Kolb (*), Sergey Paramonov (*), Tias Guns, Luc De Raedt: Learning constraints in spreadsheets and tabular data. Machine Learning 106 (9-10), pages 1441-1468, 2017.
>
> Sergey Paramonov (*), Samuel Kolb (*), Tias Guns, Luc De Raedt: TaCLe: Learning constraints in tabular data. Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, pages 2511-2514, Sheridan Communications, Conference on Information and Knowledge Management (CIKM), Singapore, 6-10 Nov 2017.
>
> (*) – equal contribution.

Addressing $\boldsymbol{\mathcal{Q}}_3$: **Chapter** 5 introduces the novel problem of Sketched Answer Set Programming. The idea of sketching takes root in imperative programming, when a function in C or Java has a part of it, typically a constant, left unspecified. A user then provides a number of examples for this sketched constant to be filled in with a correct value satisfying the examples. Inspired by this approach, we introduce Sketched Answer Set Programming, where a user writes a regular program but with certain parts such as constants, predicates or operators are left uncertain. Then, we rewrite this sketched program into a regular Answer Set Programming program, i.e., we use a standard ASP solver to complete sketches.

The chapter consists of the research previously published in the following papers:

Sergey Paramonov, Christian Bessiere, Anton Dries, Luc De Raedt: Sketched Answer Set Programming. 30th International Conference on Tools with Artificial Intelligence, IEEE, 2018, International Conference On Tools With Artificial Intelligence, Volos, Greece

Finally, with respect to $\boldsymbol{\mathcal{Q}}_4$, **Chapter** 6 introduces a logic based approach to structured pattern mining. The idea to apply general solvers to pattern mining takes root in the work of De Raedt et al. (2008), where Constraint Programming is applied to Itemset Mining. Contrary to itemset mining, where objects do not have any particular structures, structured mining studies mining patterns in sequences, trees, graphs, etc. First, we introduce a general mathematical model of general frequent pattern mining based on the logical formulation of the constraints. Second, we show that the model can be hybridized by using our generic framework together with specialized solvers developed for particular pattern mining problems.

The chapter consists of the research previously published in the following papers:

Sergey Paramonov, Daria Stepanova, Pauli Miettinen: Hybrid ASP-based Approach to Pattern Mining. Theory and Practice of Logic Programming, 2018.

Sergey Paramonov, Matthijs van Leeuwen, Marc Denecker, Luc De Raedt: An Exercise in Declarative Modelling for Relational Query Mining. Lecture Notes in Computer Science, volume 9575, pages 166-182, Springer, Inductive Logic Programming, Kyoto, 20-22 August 2015.

Sergey Paramonov, Daria Stepanova, Pauli Miettinen: Hybrid ASP-Based Approach to Pattern Mining. Rules and Reasoning - International Joint Conference, RuleML+RR 2017, Springer Lecture Notes in Computer Science (LNCS), pages 199-214, Springer, 2017, RuleML+RR 2017: International Joint Conference on Rules and Reasoning, July 12-15, London, UK.

Tias Guns, Sergey Paramonov, Benjamin Negrevergne: On declarative modelling of structured pattern mining, AAAI Workshop on Declarative Learning Based Programming, Phoenix, Arizona USA, 13 February 2016.

Sergey Paramonov, Chen Tao, Tias Guns: Generic mining of condensed pattern representations under constraints, Young Scientist's Second International Workshop on Trends in Information Processing (YSIP2), pages 168-177, CEUR Workshop Proceedings, Young Scientist's International Workshop on Trends in Information Processing, Dombai,

Russia, 16-20 May 2017.

Matthias van der Hallen, Sergey Paramonov, Michael Leuschel, Gerda Janssens: Knowledge representation analysis of graph mining, Bogaerts, Bart; Harrison, Amelia (eds.), Proceedings of ASPOCP 2016, pages 55-76, Workshop on Answer Set Programming and Other Computing Paradigms, New York, 16 October 2016.

Chapter 7 summarizes the thesis and discusses future work possibilities.

1.9 Datasets, code and experimental results

In order to make results repeatable and to follow the Open Source spirit, we have opened and published the datasets, use-cases, meta-data and other useful information. They can be found in the following GitHub repositories:

- `https://github.com/SergeyParamonov/TaCLe` with materials for Chapter "Learning Constraints in Tabular Data"
- `https://github.com/SergeyParamonov/sketching` with materials for Chapter "Sketched Answer Set Programming"
- `https://github.com/SergeyParamonov/StructuredMiningASP` with materials for Chapter "Relational Pattern Mining"
- `https://github.com/SergeyParamonov/RelationalDataFactorization` with materials for Chapter "Relational Data Factorization"

Chapter 2

Background

> From now on you shall be called Brian that is called Brian.
>
> The Life of Brian

We introduce commonly used formalisms and definitions. We start with classic First Order Logic, then proceed by introducing Answer Set Programming, Constraint Programming and the FO(.) language.

Each chapter introduces the formalization, terms and definitions that are specific to that chapter, here we present only the definitions that are common to all of them.

2.1 First order logic

In this section, we describe the syntax and semantics of FOL, for an extensive overview of FOL, we refer to Enderton (2001).

A formal language is a triple: *vocabulary*, *syntax* and *semantics*. The vocabulary is the set of symbols that can be used. The syntax is the set of rules on how these symbols can be combined together. And the semantics is how we interpret the statements. We omit the vocabulary, when it is clear from the context.

For each predicate p and each function symbol f in the vocabulary, we assume a natural number n called *arity* to be given, written as p/n and f/n. This

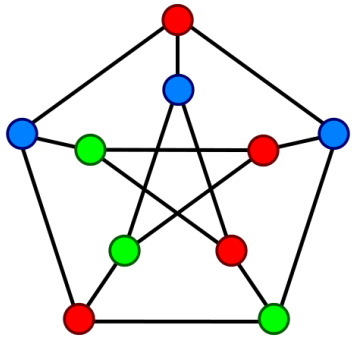


Figure 2.1: Graph coloring of the Petersen's graph using three colors

number indicates the number of parameters it takes; we often omit the arity when it is clear from the context. Propositional symbols are zero-arity predicates and constants are zero-arity functions. We assume propositional symbols *true* and *false*, whose domains are $\top$ and $\bot$ correspondingly, to be always in the vocabulary.

Example 2.1 (Predicates and functions)**.** Consider the famous problem of coloring a map, as in Figure 2.1. The vocabulary would contain a predicate symbol *border/2* (indicating if two countries share a border) and a function *coloring/1* (indicating the color of a country), together with a set of constants representing countries, such as *belgium*, *netherlands*, etc.

A *structure* associates values with the predicates and functions from the vocabulary. A structure I consists of

- A domain D^I, which we also refer to as the *universe*. The domain defines the possible values to be used.
- For each predicate p/n, the set of its values $p^I \subseteq \overbrace{D^I \times \cdots \times D^I}^{n}$
- For each function symbol f/n, f^I the mapping $\overbrace{D^I \times \cdots \times D^I}^{n} \mapsto D^I$

We call the set of values p^I for a predicate p and the mapping f^I for a function f an *interpretation*.

We assume the set of inequalities $\{=, \neq, <, >, \leq, \geq\}$, arithmetic $\{+, -, *, /\}$ and true $\top$, false $\bot$ to be interpreted by any interpretation in a standard way.

Example 2.2 (Interpretaion of map coloring)**.** Consider the predicate and the function from Example 2.1. Then, I can be the following:

- Domain $D^I = \{red,\ green,\ blue,\ belgium,\ netherlands,\ germany,\ france\}$
- The interpretation $border^I = \{(belgium,\ netherlands), \dots\}$
- The interpretation $coloring^I = \{belgium \mapsto green, \dots\}$

Syntax The syntax of First Order Logic, such as valid terms and formulae, is defined inductively.

A *term* is defined as follows

- a *variable* is a term
- if $t_1, \dots, t_n$ are terms and f/n is a function, then $f(t_1, \dots, t_n)$ is term

We say a term t is a *domain term*, if t is in the domain.

A *formula* is defined as follows

- if p/n is a predicate and $t_1, \dots, t_n$ are terms, then $p(t_1, \dots, t_n)$ is a formula, called an *atom*
- if ϕ is a formula, then $\neg\phi$ is a formula
- if ϕ is a formula and x is a variable, then $\forall x : \phi$ and $\exists x : \phi$ are formulae
- if ϕ and ψ are formulae, then $\phi \wedge \psi$ and $\phi \vee \psi$ are formulae

We call a *literal* an atom or the negation of an atom. An expression of the form $a \to b$ is a shorthand for $\neg a \vee b$, and $a \leftrightarrow b$ is a shorthand for $a \to b \wedge b \to a$. A *sentence* is a formula without free, i.e., non-quantified, variables. A set of sentences is called a *theory*.

Example 2.3 (Formula)**.** As in previous examples, we consider the problem of map coloring. The following formula:

$$\forall X, Y\colon border(X, Y) \to coloring(X) \neq coloring(Y)$$

has an intended meaning of enforcing the colours of adjacent countries to be different.

Semantics The semantics is defined inductively over the structure of the terms and of the formulae.

Let I be a structure and t be a domain term, then the *value* of t is $t^I = d$, where d is the domain element. If t is an expression of the form $f(t_1, \ldots, t_n)$, then its value is $f^I(t_1^I, \ldots, t_n^I) = d$, where d is the domain element.

The truth assignment of a structure I over formulae is defined inductively as follows:

- $p(t_1, \ldots, t_n)^I$ is true, iff $p^I(t_1^I, \ldots, t_n^I)$ is true
- $(\neg\phi)^I$ is true, iff ϕ^I is false
- $(\phi \wedge \psi)^I$ are true, iff ϕ^I and ψ^I are true
- $(\phi \vee \psi)^I$ are true, iff ϕ^I or ψ^I are true
- $\forall X : \phi$ is true, iff for each d in D^I, $\phi[X/d]^I$ is true[1]
- $\exists X : \phi$ is true, iff there is d in D^I such that $\phi[X/d]^I$ is true

We say that I is a *model* (in the logical sense) of or I *satisfies* a sentence ϕ iff ϕ^I is true, written as $I \models \phi$.

Example 2.4 (Formula evalution)**.** Assume, that the domain contains only two countries *belgium* and *netherlands* and two colors *green* and *red*. Let us show that I with *border* being true for (*belgium,netherlands*) and *coloring* mapping *belgium* to *green* and *netherlands* to *red* is a model of the formula in Example 2.3.

$(\forall X, Y\colon \mathit{border}(X, Y) \rightarrow \mathit{coloring}(X) \neq \mathit{coloring}(Y))^I$	iff
for all $d_1, d_2 \in D$ holds $(\mathit{border}(d_1, d_2))^I \rightarrow (\mathit{coloring}(d_1) \neq \mathit{coloring}(d_2))^I$	iff
for $d_1 \neq$ *belgium* and $d_2 \neq$ *netherlands* the formula holds trivially	and if
for $d_1 =$ *belgium* and $d_2 =$ *netherlands*,then (*belgium,netherlands*) in border^I	and
$f(\mathit{belgium})^I = \mathit{green} \neq \mathit{red} = f(\mathit{netherlands})^I$ holds	since
green $\neq$ *red* is true	

Herbrand interpretation The *Herbrand Universe* is the set of all terms over the vocabulary without variables. A *Herbrand Interpretaion* has the Herbrand

[1] $\phi[X/d]$ is a formula ϕ, where each occurrence of X is substituted with d

Universe as its domain and interprets each symbol and function syntactically, e.g., a constant a is interpreted by an interpretation I as $a^I = a$. For any model that contains Unique Name Assumption (UNA) (Reiter 1984) and Domain Closure Assumption (DCA) (Reiter 1980), there exists an equivalent Herbrand model.

2.2 Answer Set Programming

Answer Set Programming (ASP) is a logic programming paradigm for solving combinatorial and constraint optimization problems (Lifschitz 2008).

Contrary to the programming language Prolog, which is based on a proof-theoretic approach to answer queries, ASP follows a model generation approach. It has been shown to be effective for a wide range of constraint satisfaction problems (Gebser et al. 2012).

The remainder of this section introduces the essentials of ASP in a rather informal way. ASP is a rich (and technical) research area, so we do not focus on technical issues as these would complicate the presentation, but rather refer the interested reader to Eiter et al. (2009a), Gebser et al. (2012), Leone et al. (2002), and Lifschitz (2008) for more details on this. For the actual implementation, we will use the Clasp system (Brewka et al. 2011; Gebser et al. 2012).

Definition 2.2.1 (Disjunctive datalog program)**.** A disjunctive datalog program is a finite set of rules of the form:

$$a_1 \vee a_2 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k,\ \mathit{not}\ c_1, \ldots,\ \mathit{not}\ c_h$$

where $a_1, \ldots, a_n, b_1, \ldots, b_k, c_1, \ldots c_h$ are atoms of a function-free First Order language L. Each atom is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate name and t_i is either a constant or a variable. We refer to the head of rule r as $H(r) = \{a_1, \ldots, a_n\}$ and to the body as $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{b_1, \ldots, b_k\}$ is the positive part of the body and $B^-(r) = \{c_1, \ldots, c_h\}$ the negative.

If a disjunctive datalog program P has variables, then its semantics are considered to be the same as that of its grounded version, written as *ground(P)*, i.e. all variables are substituted with constants from the Herbrand Universe H_P (the constants occurring in the program). The semantics of a program with variables is defined by the semantics of the corresponding grounded version.

Example 2.5. Let us demonstrate a simple example of an ASP program, which has four rules with variables and four facts. All its predicates are binary, except

for the `drive` predicate, which is an unary predicate. The rule in Line 3 makes the connections undirected, while Lines 6 and 7 compute the transitive closure of the `road` predicate that are not blocked. The last route defines the unary predicate `drive` as all cities reachable from *berlin*.

```
1 road(berlin,potsdam).
2 road(potsdam,werder).
3 road(werder,brandenburg).
4 road(X,Y)  :- road(Y,X).
5 blocked(werder,brandenburg).
6 route(X,Y) :- road(X,Y), not blocked(X,Y).
7 route(X,Y) :- route(X,Z), route(Z,Y).
8 drive(X)   :- route(berlin ,X).
```

Listing 2.1: An example of an ASP program (from Gebser et al. (2012)); demonstrate possible route calculations for train connections around Berlin

An interpretation I w.r.t. to a program P is a set of ground atoms of P. Let P be a positive disjunctive datalog program (i.e. without negation), then an interpretation I is called closed under P, if for every $r \in \mathit{ground}(P)$ it holds that $H(r) \cap I \neq \emptyset$ whenever $B(r) \subseteq I$.

Definition 2.2.2 (Answer set of a positive program (Eiter et al. 2009a))**.** An answer set of a positive program P is a minimal (under set inclusion) interpretation among all interpretations that are closed under P.

Example 2.6. An answer set of the program in Example 2.1 would be the set: {

road(berlin,potsdam)	road(potsdam,werder)	road(werder,brandenburg)
road(potsdam,berlin)	road(werder,potsdam)	road(brandenburg,werder)
route(berlin,potsdam)	route(potsdam,werder)	route(brandenburg,werder)
route(werder,potsdam)	route(potsdam,berlin)	route(berlin,berlin)
route(werder,berlin)	route(potsdam,potsdam)	route(brandenburg,potsdam)
route(berlin,werder)	route(werder,werder)	route(brandenburg,berlin)
drive(potsdam)	drive(berlin)	drive(werder)
blocked(werder,brandenburg)		

}

Let us introduce the concept of a reduct (Lifschitz 2008).

Definition 2.2.3 (Gelfond-Lifschitz reduct)**.** A reduct of a ground program P w.r.t. an interpretation I, written as P^I, is a positive ground program P^I obtained by:

- removing all rules $r \in P$ for which $B^-(r) \cap I \neq \emptyset$;
- removing the literals "*not* a" from all remaining rules.

Intuitively, the reduct of a program is a program where all rules with bodies contradicting I are removed and in all non-contradicting all negative ones are ignored. The interpretation I is a guess as to what is true and what is false.

Definition 2.2.4 (An answer set of a disjunctive program)**.** An answer set of a disjunctive program P is an interpretation I such that I is an answer set of positive ground program $\mathit{ground}(P)^I$.

Example 2.7. Consider the following disjunctive Datalog program P.

$$a \vee c \leftarrow b. \qquad b \leftarrow a,\ \mathit{not}\ c. \qquad a.$$

If we take the interpretation $I = \{a, b\}$ of P as candidate answer set, then the reduct P^I is

$$a \vee c \leftarrow b. \qquad b \leftarrow a. \qquad a.$$

and it is easily seen that I is a minimal interpretation closed under P^I, and therefore an answer set.

We also use a special form of disjunctive rules called *choice rules* (Gebser et al. 2012):

$$v_1\ \{a_1, a_2, \ldots a_n\}\ v_2 \leftarrow b_1, \ldots, b_k,\ \mathit{not}\ c_1, \ldots,\ \mathit{not}\ c_h$$

where v_1 and v_2 are integer constants. The semantics are as follows: if the body is satisfied, then the number of true atoms in $\{a_1, a_2 \ldots a_n\}$ is from v_1 to v_2.

An aggregate atom is an atom that has the following form: $l\#\{a_1, \ldots, a_n\}u$ where l and u are constant numbers, each a_i is a literal. The atom is true in an answer set A iff there are between l and u literals a_i that are true in A.

Maximization is another construct used for modelling in ASP (Gebser et al. 2012; Leone et al. 2002) (*minimization* is defined analogously), written as $\#\mathit{maximize}\{a_1 = k_1, \ldots, a_n = k_n\}$, where $a_1, \ldots, a_n$ are classic literals and $k_1, \ldots, k_n$ are integer constants (possibly negative). The semantics of this constraint are as follows: a model I is selected if the weighted sum of $[a_i] * k_i$ is maximal in I, where $[\cdot]$ are Iverson brackets, i.e. $[a]$ is equal to 1 iff a is true in I and 0 otherwise.

Example 2.8. Let us demonstrate how the *minimization* statement can be used in an ASP program. We use a standard problem of finding a Hamiltonian cycle, where the predicate `cost(X,Y,C)` is added and `C` represent the cost of picking a particular edge to be present in the cycle. Then the program not only finds a Hamiltonian cycle, but also minimizes its costs. In actual ASP encodings, we use ":-" for logical implication "←", as for the optimization statement a weight C of an atom `cycle(X,Y)` is indicated as `cycle(X,Y) = C`.

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
reached(Y) :- cycle(a,Y).
reached(Y) :- cycle(X,Y), reached(X).
:- node(Y), not reached(Y).
#minimize [ cycle(X,Y) = C : cost(X,Y,C) ].
```

Listing 2.2: ASP encoding of the Hamiltonian cycle problem (due to Gebser et al. (2012))

We assume that arithmetic operators $\{+, -, \times, /\}$ are present in the language and have the standard interpretation.

Example 2.9 (Map Coloring in ASP)**.** Let us demonstrate how the problem of map coloring discussed before can be expressed in ASP.

The first constraint ensures that the binary predicate *coloring/2* is a function from nodes to colors and the second that if there are two nodes of the same color joined by an edge, then it is not a model.

```
1 { coloring(X,C) : colors(C) } 1 :- node(X).
2
3 :- edge(X,Y), coloring(X,C), coloring(Y,C).
```

Listing 2.3: ASP encoding of map coloring constraints

ASP systems: Clasp and Wasp In this thesis we make use of two existing ASP systems Clasp (Gebser et al. 2007) and Wasp (Alviano et al. 2013). We mostly focus on the former and use the second to provide an experimental comparison on how two different systems can execute the models with respect to runtime and memory usage.

Clasp extends the standard ASP theory described here with a number of features such as incremental grounding, preferences, satisfiability modulo theories, various solving parameters such as brave and cautious reasoning, parallel executions, heuristics and meta-programming (Gebser et al. 2012).

A typical modelling approach in ASP and in actual development with Clasp and Wasp follows guess and check paradigm. Where at first we define potential stable model candidates (typically through non-deterministic constructs) and then eliminate invalid candidates (typically through integrity constraints) (Gebser et al. 2012; Gebser et al. 2007; Lifschitz 2008), which in a nutshell allows us to write the following formula for ASP modelling:

$$\textbf{ASP Program} = \textbf{Data} + \textbf{Generator} + \textbf{Tester} \; (\; + \textbf{Optimizer}) \; [2]$$

We follow this modelling paradigm throughout the thesis. To demonstrate it, consider as an example, the n-queen modelling problem, where one needs to put a number of queens on the board such that no queen attacks another. This can be modelled as in Listing 2.2. first we specify data as facts, in Lines 2 and 3, then we define the set of possible answer sets using a choice rule in Line 5 and finally we validate them in Lines 7,8 and 9. In greater detail, we require the number of queens to be equal to k by using a cardinality constraint in Line 5 then right after this "guess" is verified with integrity constraints: no queen is on the same row (Line 7), no queen is in the same column (Line 8) or on the same diagonal (Line 9). An answer set that has a set of k queens which satisfies the integrity constraints is therefore a solution to k-queen problem.

```
1 % Data specifying the board
2 col(1..k).
3 row(1..k).
4 % Choice rule to specify possible stable models
5 k { queen(Row,Col) : col(Col), row(Row) } k.
6 % Integrity constraint to validate the candidate models
7 :- queen(Rw,Cw), queen(Rb,Cb), Rw  = Rb, Cw != Cb.
8 :- queen(Rw,Cw), queen(Rb,Cb), Rw != Rb, Cw  = Cb.
9 :- queen(Rw,Cw), queen(Rb,Cb), Rw != Rb, | Rw - Rb | = | Cw - Cb |.
```

Listing 2.2: An ASP guess-and-check paradigm modelling example on the n-queen problem

2.3 Constraint programming

A detailed introduction to Constraint Programming(CP) can be found in the "Handbook of Constraint Programming" (Rossi et al. 2006). A basic concept

[2] https://www.cs.uni-potsdam.de/~torsten/potassco/slides/asp.pdf

here is that of a variable v which always has an associated domain d, a domain here is simply a set of constants. An example domain is the Boolean domain { True, False }. A constraint c is an expression of the form *name(vars)*, where *vars* is a list of variable names and *name* is the type of constraints from a predefined set of constraint classes. Examples of classes of constraints are comparisons, arithmetic operations, list and set constraints (all-different, permutations, etc). We denote the set of all variables as V, of all domains as D and of constraints as C, then a triple (V, D, C) is a constraint programming model and a Constraint Satisfaction Problem (CSP) is the problem of finding an assignment A for V over D satisfying all constraints C.

Let us demonstrate how it is possible to model the map coloring problem in MiniZinc[3]

```
% number of countries
int: n;
% [belgium,denmark,france,germany,netherlands,luxembourg]
array[1..n] of var 1..4: country;
% the map (as a binary matrix)
array[1..n,1..n] of 0..1: connections;

forall(i,j in 1..n where i < j /\ conn[i,j] = 1) (
                country[i] != country[j]
      )
```

The first lines define variables together with their types, for example `country` is an array, each element of which can take value from one to four. And the constraint at the bottom of the listing verifies that for each value of indices i, j in the range of 1 to n such that (part: `where`) i is smaller than j (part: `i < j`) and there is a connection between i and j (part: `conn[i,j] = 1`) it holds that colors of the countries are different, i.e., the part: `country[i] != country[j]`.

Let us introduce the key definitions from constraints programming formally.

Definition 2.3.1 (Constraint Satisfaction Problem(CSP))**.** A CSP is a triple (V, D, C) where

- V is a finite set of variables (a set of distinct symbols)
- D is a domain that maps every variable v in V to a set of its possible values D_v (a set of district symbols, disjoint from the variable symbols)
- C is a finite set of constraints (language dependent)

[3]MiniZinc is a popular constraint programming language http://www.minizinc.org/. Based on the encoding from www.hakank.org/minizinc/map.mzn.

Algorithm 1 Simple constraint propagate-and-search algorithm: search(D)

```
1:  D ← propagate(D)
2:  if ∃v : D_v = ∅ then
3:      return ⊥
4:  if ∃v ∈ V : |D_v| > 1 then
5:      v = select_variable(V)          ▷ pick the most promising variable
6:      D_s = split(D_v)                ▷ D_v is the domain of v
7:      search(D ∪ {v ↦ D_s})           ▷ recursive DFS-like search
8:      search(D ∪ {v ↦ D_v/D_s})
9:  else
10:     return assignment
```

A solution A is a mapping function from V to D such that for every constraint it holds that for every constraint c in C, the constraint c^A, which is the constraint c where each variable v is replaced with $A(v)$, holds.

In case of the problem above, `n` is the constant that represents the number of countries, `conn[i,j]` represents the fixed connections between countries (is there or not) and each `country` represent an array of variables that can get assigned a colour, each country gets a colour. The domain is essentially specifying that `conn[i,j]` can be either one or zero and `country[i]` can be mapped to a color value (and the list of colours itself is restricted by the domain itself), which is represented as a number from 1 to 4 here. Assuming that for example there is only one connection `conn[1,2]` and two countries one and two, then solution assignment A maps `country[1]` to one and `country[2]` to two (connection `conn[1,2]` is indeed also mapped to one as the given data).

A constraint optimization problem is a CSP (V, D, C) extended with an objective function f_o from V to R^1, then the problem is to find a solution to the original CSP such that the score defined by f_o is maximal (or minimal).

A typical approach to solving a constraint satisfaction problem is to use a form of search over the variable values in their domains, we present here an overview over a possible searching procedure, for details we refer to Guns (2012).

Algorithm 1 demonstrates how we can solve a CSP. First, we propagate all possible deterministic assignments. Then we make a decision which variable we are going to split, i.e., we investigate two cases separately when it can take certain values and when it cannot. For each of this cases we call the procedure itself recursively. If in certain moment, we have derived that all variables have only one value, then it is a solution, if in certain point of propagating values using given constraints we established that no value is left for a variable, we return a failure for this particular case.

Let us elaborate on both key steps of the Algorithm 1.

Search Many popular CP frameworks follow the Depth First Search (DFS) strategy as described in the presented algorithm (Rossi et al. 2006; Schulte and Stuckey 2008). Other techniques exist as well (Van Hentenryck et al. 2000) but we focus only the most popular DFS approach described here.

For a constraint satisfaction problem, its search tree represents an order from more general to more specific domains. The tree root has the the most general (yet unrestricted) domain. The leaves of a search tree represent actual solutions to the given CSP: each $v \in V$ is mapped to a single value d from D_v. Each node represents branching of the search by domain splitting. If a constraint is violated, then Line 2 starts the backtracking mechanism (DFS is a form of backtracking search).

Propagation is a search speed up mechanism. In Algorithm 1 it appears in Line 1. The key idea behind this mechanism is in the domain reduction and so called, *local consistency*, i.e., satisfaction or actually violation of a subset (even a single one) constraints. This removes values that definitely cannot be in the solution since they violate one or more constraints. Contrary, *global consistency* ensures that all constraints in the model are satisfied. The latter is indeed hard to achieve, the former is easy to check, since you can check one or several constraints in the isolation, or in the other words, locally. If a certain value violates this constraint, then it definitely cannot be in the solution set. Many various methods to check local consistency exist (Bessiere 2006): node consistency, arc consistency and path consistency, etc.

To keep local consistency for constraints, *propagation rules* and *propagators* are applied. Such a rule, takes an original domain D and outputs a domain D' from which all impossible values are removed.

2.4 FO(·) and the IDP System

The knowledge based language FO(·) is the extension of First Order Logic with a number of features, such as, inductive definitions and aggregates. The IDP system (De Cat et al. 2013) is the reasoning system that accepts as input the FO(·) language and is able to perform a number of reasoning tasks: such as finding logical models, model checking and theorem proving. IDP and its language FO(.) can be seen as the executable logic (i.e., a form of constraint logic programming) we have discussed before in Section 2.1.

Let us illustrate an inductive definition and an IDP program by example, see Listing 2.4, where Reach(x,y) is an example of an inductive definition, defined via the base case and the inductive step (see the Theory part).

```
vocabulary V{
  type Node
  Edge(Node, Node)
  Reach(Node, Node)
}

theory T:V{
  {
    Reach(x,y) <-       Edge(x,y).
    Reach(x,y) <- ?z : Edge(x,z) & Reach(z,y).
  }
}

structure S:V{
  Node = {1;2;3}
  Edge = {1,2; 2,3; 3,1}
}
```

Listing 2.4: An example of an inductive definition in FO(.) and of an IDP program

An IDP program has three parts: *Vocabulary* that defines the terminology and concepts of a problem, *Structure* that sets certain values to true or false, and *Theory* introduces constraints.

For example the sushi problem can be encoded in IDP as in Listing 2.5. First, we define the vocabulary, which corresponds to vocabulary in first order logic, except for the extra type information. Facts and domains are specified in the structure section, as we see for example with the declaration of the domain for the type person, for which actual values are assigned in the structure section . The theory section defines the actual clauses or constraints, which indeed corresponds to the first order sentences, the only constraint here maps one-to-one to Eq. 1.2, which a classical logical implication. The procedure `main` is a directive to the system to extend the given structure satisfying the constraints in the theory.

```
vocabulary V{
  type person
  allergic(person)
  vegan(person)
  likes_sushi(person)
}
theory T:V{
  !x[person]: ~allergic(x) & ~vegan(x) => likes_sushi(x).
}

structure S:V{
  person   = {peter; alina; ronald}
  vegan    = {ronald}
  allergic = {peter}
}

procedure main(){
  M = modelexpand(T,S)[1]
  print(M)
}
```

Listing 2.5: Encoding sushi preferences problem

Therefore, when we write a first order model for a problem, we can always map it to the language of FO(.). Furthermore provided language extensions make modelling of relational problems more natural and convenient.

FO(.) and the system implementing it, IDP, can be seen a form of constraint logic programming, where one models a problem using a set of constraints expressed as logical formulae. Let us demonstrate this on an example.

Example 2.10 (Constrained Pizza Selection Problem)**.** Peter, Alina, Ronald and Sergey decided to order three pizzas in a trattoria: margarita, veggie and meat-lovers – each pizza can be shared by at most two people and everyone specified pizzas they can eat, together with their preferences for the selected pizzas on the scale from one to ten. Can we help them to share the pizzas in an optimal way?

We start with modelling the domain of the problem, there are four *persons*, which is captured by a type declaration, `type person`, in the same way we capture the domain of available *pizzas*, see first lines of `Vocabulary` in Listing 2.6. Then, once these two types are declared, in `Structure` we assign actual values to them, like

```
pizza  = {margaritha; veggie; meat}
```

IDP interprets this as the truth assignment, only specified constants belong to the type `pizza` and nothing else. Similarly, we define who can eat what pizza, e.g., Alina can eat veggie pizza and meat-lovers pizza, we see that clearly it's a binary relation between a `person` and a `pizza`, then we declare it as `can_eat(person,pizza)` and its actual domain as

```
can_eat = {
          (alina,veggie); (alina,meat);
          (sergey,margaritha); (sergey,meat);
          (ronald,margaritha); (ronald,veggie);
          (peter,meat)
          }
```

which specifies all legitimate pairs of *(person,pizza)* to be used.

To specify preferences over the selection, we introduce the partial function `partial preferences(person,pizza):score` (and the type `score` which is a subtype of integers) as the following mapping:

```
ronald,veggie -> 10;    ronald,margaritha -> 1;
alina,veggie  -> 5;     alina,meat  -> 7;
sergey,margaritha -> 2; sergey,meat -> 7;
peter,meat -> 10
```

First, we use a partial function because we already know we should only define this function on the pairs specified in `can_eat`. Second, we introduce score because we know exactly the range of values it takes and therefore helps the IDP solver to look for a solution more efficiently.

Now we need to define what we are looking for: the binary predicate `eats(person,pizza)`. This predicate determines who eats what and based on it, we are going to compute the overall *satisfaction* score. Simply speaking, what we are looking for is the best values (in terms of preferences) for `eats` that satisfy all specified constraints (on sharing pizzas, etc).

What are the constraints and how do we model them? First, we know that each person eats a pizza and each pizza is eaten by a person. These are constraints of the form: for each A there is a B,

```
// each person eats a pizza
!s[person]: ?p[pizza]: can_eat(s,p) & eats(s,p).
```

Since IDP supports types, this is essentially a direct formalization of: for each person, there must exist a pizza such that this person can eat this pizza and

he eats it. Similarly, we formulate it for pizzas. The no-sharing constraint is modelled in a slightlyferent manner: we say that if two people eat a pizza, there cannot be a third person that is different from these two (see `Theory` in Listing 2.6).

Then, to obtain *an optimal* solution, we introduce an optimization statement:

```
term O:V{
 sum{prsn[person],pzz[pizza],score[score]:
 eats(prsn,pzz) & preferences(prsn,pzz)=score:score}
}
```

what it does is the following: it sums all preference scores for all pairs of person and pizza `(prsn,pzz)` that the binary predicate `eats` assigns to true, or simply speaking, it sums scores corresponding to the assigned values in `eats`, which gives to us a measure of quality of a found model. Then we ask the IDP system to find a model that maximizes this measure.

Let us demonstrate the full model solving this program in Listing 2.6.

```
vocabulary V{
 type person
 type pizza
 can_eat(person,pizza)
 eats(person,pizza)
 type score isa int
 partial preferences(person,pizza): score
}
theory T:V{
 !s[person]: ?p[pizza]: can_eat(s,p) & eats(s,p). // each
    person eats a pizza
 !p[pizza]: ?s[person]: can_eat(s,p) & eats(s,p). // each
    pizza is eaten by someone
 // no pizza is share by more than 2 people
 !person1[person], person2[person], p[pizza]: person1 ~=
    person2 & eats(person1,p) & eats(person2,p) => ~?person3[
    person]: eats(person3,p) & person1 ~= person3 & person2
    ~= person3.
}

structure S:V{
 person = {peter; alina; ronald; sergey}
 pizza  = {margaritha; veggie; meat}
 score  = {1..10}
 can_eat   = {
           (alina,veggie); (alina,meat);
           (sergey,margaritha); (sergey,meat);
           (ronald,margaritha); (ronald,veggie);
           (peter,meat)
         }
 preferences = {
             ronald,veggie -> 10; ronald,margaritha -> 1;
             alina,veggie  -> 5;  alina,meat-> 7;
             sergey,margaritha ->2 ; sergey,meat->7;
             peter,meat -> 10
              }
}
// we maximize overall preferences
term O:V{
    sum{prsn[person],pzz[pizza],score[score]:
    eats(prsn,pzz) & preferences(prsn,pzz)=score:score}
}

procedure main(){
    result = maximize(T,S,O)[1]
    print(result)
}
```

Listing 2.6: Encoding pizza selection problem

Chapter 3

Relational data factorization

This chapter is based on the paper: "Relational Data Factorization" by Sergey Paramonov, Matthijs van Leeuwen, Luc De Raedt[1]. My contributions are in the algorithm design, implementation, experiments, and writing the results.

Motivated by an analogy with matrix factorization, we introduce the problem of factorizing relational data. In matrix factorization, one is given a matrix and has to factorize it as a product of other matrices. In Relational Data Factorization (ReDF), the task is to factorize a given relation as a conjunctive query over other relations, i.e., as a combination of natural join operations. Given a conjunctive query and the input relation, the problem is to compute the *extensions* of the relations used in the query. Thus, relational data factorization is a relational analog of matrix factorization; it is also a form *inverse* querying as one has to compute the relations in the query from the result of the query. The result of relational data factorization is neither necessarily unique nor required to be a lossless decomposition of the original relation. Therefore, constraints can be imposed on the desired factorization and a scoring function is used to determine its quality (often similarity to the original data). Relational data factorization is thus a constraint satisfaction and optimization problem. We show how answer set programming can be used for solving relational data factorization problems.

[1] https://link.springer.com/article/10.1007/s10994-017-5660-6

3.1 Introduction

The fields of data mining and machine learning have contributed numerous effective and highly optimized algorithms for analyzing data. However, this focus on efficiency and scalability has come at the cost of generality. Indeed, while the algorithms are highly effective, their application range is often very restricted, and the algorithms are typically hard to change and adapt even to small variations on the problem definition. This observation has led to an interest in declarative methods for data mining and machine learning in which the focus lies on the use of expressive models that can capture a wide range of different problem settings and that can then be solved using off-the-shelf constraint solving technology; see Arimura et al. (2012), De Raedt (2012, 2015), and Guns et al. (2013c).

Motivated by this quest for more general and generic data analysis approaches, the present chapter introduces the problem of relational data factorization (ReDF). ReDF is inspired by matrix factorization, one of the most popular techniques in machine learning and data mining for which many variants have been studied, such as non-negative, singular value and Boolean matrix factorization. In matrix factorization, one is given an $n \times m$ matrix $\mathbf{A}$, and the problem is to rewrite it as the product of some other matrices, e.g., the product of an $n \times k$ matrix $\mathbf{B}$ and $k \times m$ matrix $\mathbf{C}$ such that $\mathbf{A}_{i,j} = \sum_k \mathbf{B}_{i,k} \cdot \mathbf{C}_{k,j}$. In relational data factorization, one is given a relation (i.e., a set of tuples over the same attributes) and asked to to rewrite it in terms of other relations. Consider, for instance, a relation *sells(Company, Part, Project)*, stating that companies sell particular parts to particular projects. While it is well-known that ternary relations, in general, can not be rewritten as the join of three binary relations (Heath 1971; Jones et al. 1996)[2], we might be interested in an approximation of the ternary relation. That is, we might approximate *sells(Company, Part, Project)* by the query *offers(Company,Part), needs(Project, Part), deliversto(Company, Project)* (we follow logic programming notation, where the same variable name denotes a natural join). The question is then how to determine the extensions for the relations *offers*, *needs*, and *delivers*. The found solution will generally be imperfect, so in ReDF we want to find the best approximation w.r.t. a scoring function and we allow the user to specify hard constraints. In the example these might specify, e.g., that only tuples in the target relation *sells* may be derivable from the query.

In this chapter, we develop a modelling and solving approach for ReDF using answer set programming (ASP) (Brewka et al. 2011). This is realized by

[2]Heath Theorem: a relation $R(x, y, z)$ satisfying a functional dependency $x \to y$ can always be losslessly decomposed into its projections $R_1 = \pi_{xy}R$ and $R_2 = \pi_{xz}R$; see (Jones et al. 1996, Table 5)

showing for a number of ReDF problems how they can be tackled with ASP. This leads to the identification of constraints and scoring functions, which we then abstract to an even higher-level declarative language. We show that the resulting ReDF framework is general and generic and is in line with the declarative modelling approach to machine learning and data mining as 1) it allows one to easily specify and solve a wide range of well-known data analysis problems (such as tiling, Boolean matrix factorization, discriminative pattern mining, matrix block diagonalization, etcetera), 2) it is effective for prototyping such tasks (as we show in our experiments), even though it cannot yet compete with optimized special purpose algorithms in terms of efficiency, and 3) the constraints and optimization criteria are specified in a declarative and flexible manner. Translating problem definitions in the ReDF framework to ASP models is straightforward, and small changes in the problem definitions generally result in small changes in the model.

Relational data factorization is a form of relational learning. That is, it is a relational analog of matrix factorization and is therefore relevant to inductive logic programming (De Raedt 2008a; Muggleton and De Raedt 1994a) and can also be seen as a form of large-scale abduction (Denecker and Kakas 2002). Moreover, the solution techniques that we adopt are based on answer set programming, which has also been adopted in some recent works and methods on inductive logic programming (Järvisalo 2011; Paramonov et al. 2015a). The implementation techniques we employ may also be used in more traditional inductive logic programming settings.

This chapter is structured as follows. Section 3.2 introduces the formal ReDF framework. Section 3.3 shows how a wide range of data mining problems can be expressed as ReDF problems. Section 3.4 introduces some novel problems that the framework can express. Section 3.5 discusses the encoding of the problems into ASP, while Section 3.6 reports on the experimental evaluation. In Section 3.7 we discuss solver's performance configurations. In Section 3.8 we discuss potential applications of ReDF. In Section 3.9, we analyze theoretically the greedy approximation strategy. In Section 3.10, we discuss related work, and we formulate some conclusions and directions for future work in Section 3.11.

3.2 Relational data factorization

Before we formalize the ReDF problem and approach in its full generality, we illustrate Relational Data Factorization on the *sells(Company, Part, Project)* example from the Introduction.

3.2.1 An example

Assume we are given 1) a set of tuples for the *database* relation *sells(Company, Part, Project)*, 2) a definite *shape* clause defining the predicate *approx(Company, Part,Project)*, e.g.,

approx (Com,Pa,Proj) ← *offers(Com,Pa),needs(Proj, Pa),deliversto(Com,Proj)*

which should approximate the database relation *sells(Company, Part, Project)* in terms of the (unknown) relations *offers(Company, Part)*, *needs(Project, Part)* and *deliversto(Com, Project)*, and 3) an *error* function *error*(*approx*, *sells*) that measures how different the database predicate and its approximation are, e.g., the number of tuples that is one in relation but not in the other. Then, the goal is to find sets of tuples for the unknown relations that minimize the *error*.

In practice, it is usually impossible to find a perfect solution (with $error = 0$) to relational data factorization problems, in this example because of Heath's theorem (Heath 1971) (as discussed in the Introduction). Therefore, it is often useful to impose further restrictions on the sets to be considered. One such constraint could specify that there is no *overcoverage*, i.e., that all tuples in *approx* must be in *sells*.

3.2.2 Problem statement

Using a logic programming formalism, we generalize the above example into the following ReDF problem statement.
Given:

- a dataset D: a set of ground facts for target predicate *db*;
- a factorization shape Q: $approx(\bar{T}) \leftarrow q_1(\bar{T}_1), \ldots, q_k(\bar{T}_k)$, where the q_i are factors and the $\bar{T}_i$ denote tuples of variables;
- a set of constraints C;
- an *error* function measuring difference between two predicates (i.e., between the corresponding sets of ground facts);

Find: the set of ground facts F for the factors q_i that minimizes *error*(*approx*, *db*) and for which $Q \cup F \cup D$ satisfies all constraints in C.

The factorization shape is a single non-recursive rule defining *approx*, the approximation of the target predicate *db*, where the predicates in the body are

the factors. If a variable occurs in a body atom $\bar{T}_i$ and not in $\bar{T}$ (the head), then it is called *latent*. The task is to find a set F of ground facts defining the factors q_i. Furthermore, each such set F uniquely determines a set of facts for *approx*. Notice that if a predicate q_i is already known and defined, then the task simplifies.

As in matrix factorization, it is quite likely that a perfect solution, with $error = 0$, cannot be obtained. Consider the following example: $db(X, Y) \leftarrow p(X), q(Y)$ and dataset $D = \{db(a, c), db(b, d)\}$. Then it is impossible to perfectly reconstruct the target D. If $F = \{p(a), p(b), q(c), q(d)\}$, the resulting program overgeneralizes as it entails facts not in D: $db(a, d) \in approx$ and $db(a, d) \notin D$; if, on the other hand, there are facts in D that are not entailed in *approx*, one undergeneralizes (e.g., when $F = \emptyset$).

The scoring function in relational factorization measures the *error* between the predicates *approx* and *db*. Instead of minimizing *error*, however, in some cases it is more convenient to maximize *similarity*. Since these two perspectives can be trivially transformed from one to the other, we will use both without loss of generality.

3.2.3 Approach

To make this setup operational, we represent ReDF problems at two different levels. First, at the high level, we characterize typical constraints of interest that are employed across different models. Further, all problems are formulated using the template shown in Listing 3.1. Second, at the low level, the high-level constraints and encodings are formulated in ASP. The high-level constraints can in principle be automatically transformed into low-level ones.

```
Input: a set of facts D for the db predicate
Shape:  approx(T̄) ← q1(T̄1), ..., qk(T̄k)
Find:  q1 ... qk
Satisfying:  C1(approx, db) ∧ ... ∧ Cn(approx, db)
Minimizing:  error(approx, db)
```

Listing 3.1: Prototypical template of a high level problem encoding

We next illustrate this on the *sells* example. The high-level description from which we start is given in Listing 3.2.

```
Input: sells(c1,pa1,proj1), sells(c2,pa1,proj2)
Shape:
    approx(C, Pa, Prj) ← offers(C,Pa), needs(Prj,Pa), deliversto(C,Prj).
Find:  offers(·), needs(·), deliversto(·)
Minimizing:  error(approx, sells)
```

Listing 3.2: Sells example encoding

Next, this high-level formulation can be encoded in and solved using the ASP program given in Listing 3.3 (here, an ASP program can be thought as a conjunction of logical rules, where implication is denoted by ":-").

```
1  %factorization shape
2  approx(Com,Pa,Proj) :− offers(Com,Pa), needs(Proj,Pa), deliversto(Com,Proj).
3  %relation generators
4  0 { offers(Com,Pa) } 1 :− sells(Com,Pa,Proj).
5  0 { needs(Proj,Pa) } 1 :− sells(Com,Pa,Proj).
6  0 { deliversto(Com,Proj) } 1 :− sells(Com,Pa,Proj).
7  %optimization function
8  overcoverage(Com,Pa,Proj) :− approx(Com,Pa,Proj), not sells(Com,Pa,Proj).
9  undercoverage(Com,Pa,Proj) :− not approx(Com,Pa,Proj), sells(Com,Pa,Proj).
10 error(Com,Pa,Proj) :− undercoverage(Com,Pa,Proj).
11 error(Com,Pa,Proj) :− overcoverage(Com,Pa,Proj).
12 #minimize[error(Com,Pa,Proj)].
```

Listing 3.3: Factorization of a ternary relation into three binary relations

We introduce ASP in more detail below, but this model is easy to understand if one is familiar with the basics of logic programming. The ASP model basically defines the necessary predicates in ASP using a set of clauses. In addition, the rule in Line 4 encodes the constraint that whenever a tuple holds for *sells(Com, Pa, Proj)* there should be 0 or 1 corresponding tuples for the predicate *offers(Com, Pa)*. Furthermore, the minimize statement specifies that we are looking for a model (a set of ground facts or tuples) that minimizes the error. The encoding in Listing 3.3 together with a set of facts for *sells* can be given to an ASP solver such as Clasp (Gebser et al. 2011).

Observe that the relational data factorization approach we propose perfectly fits within the declarative modelling paradigm for machine learning and data mining (De Raedt 2012). Indeed, the next sections will show that it naturally supports a wide range of popular and well-known factorization problems. modelling different problems corresponds to specifying different constraints, shapes and optimization functions. By doing so, one obtains a deep understanding of the relationships among the many variations of factorization, and one can easily design, prototype and experiment with new variations of factorization problems.

Furthermore, the models of factorization are in principle solver-independent and do not depend on a particular ASP solver implementation.

Notice that it would also be possible to use other constraint satisfaction and optimization approaches (such as, e.g., Integer Linear Programming), but given that we work within a relational framework, ASP is a natural choice. It is also declarative and has the right expressiveness for the class of problems that we will study, many of which are NP-complete such as BMF; see Subsection 3.3.2.

Finally, let us mention that there are many factorization approaches in both linear algebra, databases, and even in logic. We provide a detailed discussion of their relationship to ReDF in Section 6.6.

3.3 Application to data mining problems

In this section we show that the ReDF framework generalizes a wide range of data mining tasks and provides a truly declarative modelling approach for relational data factorization. We introduce a range of constraints and optimization criteria that can be used in practice. The data mining tasks studied include tiling (Geerts et al. 2004), Boolean Matrix Factorization (BMF) (Miettinen et al. 2008), discriminative pattern mining (Knobbe and Ho 2006), and block-diagonal matrix forms (Aykanat et al. 2002).

3.3.1 Tiling

Data mining has contributed numerous techniques for finding patterns in (Boolean) matrices. One fundamental approach is that of *tiling* (Geerts et al. 2004). A tile is a rectangular area in a Boolean matrix represented by set of rows and columns such that all values on the corresponding rows and columns in the matrix are equal to 1.

One is typically not interested in *any* tile, but in maximal tiles, i.e., tiles that cannot be extended. For instance, Figure 3.1 shows a binary dataset and two tiles. The first tile consists of the first and second column together with the first and second row. All entries for these rows and columns are 1s. Furthermore, it cannot be expanded as adding the third column or row would also include 0 values. The second tile consists of all three columns and the third row. Together these two tiles "cover" the whole dataset, that is, all cells with value 1 in the matrix belong to one of the tiles. The area of a set of tiles, denoted as $area(\mathcal{T}, D)$, is the number of cells (7 in our example) in the (union of the) tiles $\mathcal{T}$ occurring in the dataset D

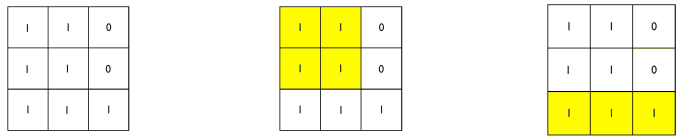


Figure 3.1: Example of Boolean tiles and their coverage

Definition 3.3.1 (Maximum k-tiling)**.** Given a binary dataset D and a positive integer k, find a tiling $\mathcal{T}$ consisting of at most k tiles and maximizing $area(\mathcal{T}, D)$.

We now formalize tiling as a relational data factorization problem and then solve it using ASP. Rather than restricting ourselves to Boolean values as in the traditional formulation, we consider the relational case. The standard way of dealing with tables in attribute-value datasets was to expand them into a sparse Boolean matrix (with one Boolean for every attribute-value). In contrast, our formulation employs the attribute-value format directly.

Given a relation *db*(*Value*, *Attr*, *Transct*), denoting that transaction *Transct* has *Value* for *Attr*, the task is to find a set of tiles that can be applied to the transactions to summarize the dataset *db*. Here, a tile is a set of attribute-value pairs.

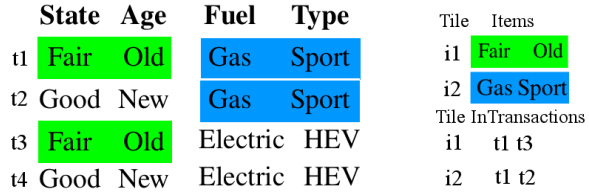


Figure 3.2: Relational tiling: two relational tiles (right) in a toy dataset (left) concerning cars.

In Figure 3.2, for example, we can see the initial dataset, in which *State* is an attribute and *Fair* and *Good* are values for this attribute. Moreover, the blue and green areas indicate two relational tiles occurring in particular sets of transactions.

The two example tiles can be expressed as

$$tile(i_1, fair, state).\ tile(i_1, old, age).\ in(i_1, t_1).\ in(i_1, t_3).$$

$$tile(i_2, gas, fuel).\ tile(i_2, sport, type).\ in(i_2, t_1).\ in(i_2, t_2).$$

where the first argument of each *tile* is the index of the tile, the second is the value of the attribute, and the third argument is the name of the attribute. When tile I is applied to a transaction T (i.e., it occurs in the transaction), this is denoted by $in(I, T)$. We call a set of tiles a *tiling*.

We would like to factorize the initial dataset, represented as a set of $db(fair, state, t_1)$, $db(old, age, t_1)$, ..., using the following *shape* query:

$$approx(Attr, Value, Transct) \leftarrow tile(Indx, Value, Attr), in(Indx, Transct). \quad (3.1)$$

To reason about the coverage of the shape, i.e., which transactions and attributes are covered in the dataset (indicated by color in Figure 3.2), we use the following definition:

$$covered(Transct, Attr) \leftarrow approx(Attr, Value, Transct).$$

For instance, $covered(t_1, age)$ holds because $tile(i_1, old, age)$ and $in(i_1, t_1)$ hold.

To specify the maximum k-tiling problem, we need the following constraints.

`one-value-attribute:` for every attribute of a tile there is at most one value:

$$\leftarrow tile(Indx, Val_1, Attr), tile(Indx, Val_2, Attr), Val_1 \neq Val_2. \quad (3.2)$$

`no-tile-intersection:` tiles do not overlap in the same transaction

$$\leftarrow in(I_1, T), in(I_2, T), tile(I_1, V, A), tile(I_2, V, A), I_1 \neq I_2. \quad (3.3)$$

`no-overcoverage:` tiles cannot "overcover" the transaction, that is, they are only allowed to cover tuples that are in the dataset;

$$\leftarrow tile(Indx, Value, Attr), in(Indx, Transct), not\ db(Value, Attr, Transct). \quad (3.4)$$

`number-of-patterns(K):` there are at most k-tiles (numbered from 1 to k):

$$Indx = 1 \vee Indx = 2 \vee \ldots Indx = k \leftarrow tile(Indx, Value, Attr).$$

Furthermore, the maximum k-tiling problem searches for the k tiles that maximize the area. This leads to an instance of the *similarity* score defined by

$$\texttt{coverage}: \quad \#\{(T, A) : \mathit{covered}(T, A)\}. \tag{3.5}$$

The statement above correspond to the standard mathematical function optimization notation, that reads as follows: count (#) the cardinality of the set ($\{\cdot\}$) of tuples (T, A) such that (:) $\mathit{covered}(T, A)$ holds. When we translate this statements into ASP formulation we have to use special syntax of ASP (*#maximize*) to capture this mathematical formulation.

We specify the high-level model for maximum k-tiling in Listing 3.4.

Input: dataset *db* and constant K
Shape: $\mathit{approx}(\mathit{Attr}, \mathit{Value}, \mathit{Transct}) \leftarrow \mathit{tile}(\mathit{Indx}, \mathit{Value}, \mathit{Attr}), \mathit{in}(\mathit{Indx}, \mathit{Transct})$
Find: the set of ground facts $\mathit{tile}(\cdot), \mathit{in}(\cdot)$
Satisfying: `no-tile-intersection` $\wedge$ `no-overcoverage`
$\wedge$ `number-of-patterns(K)` $\wedge$ `one-value-attribute`
Maximizing: `coverage`

Listing 3.4: Maximum k-Tiling ReDF Model

To illustrate the advantages of our declarative and modular approach, let us consider a small variation of the tiling task, in which tiles may overlap.



Figure 3.3: Example of a 0/1 database with a tiling consisting of two overlapping tiles
(darkest shaded area corresponds to the intersection of the two tiles), due to Geerts et al. 2004

Overlapping tiling Figure 3.3, taken from Geerts et al. 2004, illustrates a Boolean dataset with two overlapping tiles. We investigate and present two new variations of maximum k-tiling: overlapping and noisy tiling. The

first investigates the global pattern mining task, when the overall coverage is optimized, allowing overlaps between tiles. The second investigates the task when, in k-maximum tiling, a tile can have a number of mismatches as covering a transaction. It is straightforward to change the assumption in our ReDF framework (and the corresponding ASP implementation). For the first task, it only involves replacing the constraint `no-tile-intersection` by the following constraint.

`overlapping-tiles(N):` two tiles in one transaction can intersect only on at most N attributes:

$$\leftarrow in(I_1, T), in(I_2, T), tile(I_1, V, A_1), tile(I_2, V, A_2), I_1 \neq I_2, \#\{A_1 = A_2\} > N.$$

To model the variation that tolerates some noise in the data, we can replace constraint `no-overcoverage` by

`noisy-overcoverage(N):` every tile I can overcover at most N attributes in every transaction T where it occurs:

$$\leftarrow tile(I, V, A), in(I, T), not\ db(V, A, T), \#\{A\} > N.$$

Both variations show that a slight change of the formulation of property of a solution leads to a small change in the modelling and to a small change in the implementation.

3.3.2 The discrete basis problem (DBP) and boolean matrix factorization (BMF)

BMF has been extensively studied by Miettinen 2012, resulting in the well-known ASSO algorithm. Let us now show how it can be expressed as ReDF problem. As a starting point we take the same shape (Eq. 3.1) as in the tiling example in Subsection 3.3.1. However, we need to change the constraints to reflect the different properties of the desired solutions: tiles may now overlap, since one is not interested in tiles per se, but in good coverage of the dataset. That is why we remove the `no-tile-intersection` and `no-overcoverage` constraints, and introduce a notion of 'overcoverage' instead, by means of the following definition:

$$overcovered(T, A) \leftarrow approx(V, A, T), not\ db(V, A, T).$$

In the Discrete Basis Problem, the scoring function maximizes the number of covered elements, while minimizing the overcovered ones. The latter term can be simply defined as:

overcoverage: $\#\{(T, A) : \mathit{overcovered}(T, A)\}$.

We specify the high-level DBP model in Listing 3.5.

Input: dataset db and constants K, α
Shape: $\mathit{approx}(\mathit{Attr}, \mathit{Transct}) \leftarrow \mathit{tile}(\mathit{Indx}, \mathit{Attr}), \mathit{in}(\mathit{Indx}, \mathit{Transct})$
Find: the set of ground facts $\mathit{tile}(\cdot), \mathit{in}(\cdot)$
Satisfying: `number-of-patterns(K)`
Maximizing: `coverage` $- \alpha \times$ `overcoverage`

Listing 3.5: ReDF Model for the Discrete Basis Problem

This formulation mimics *The Discrete Basis Problem* (Miettinen et al. 2008). That is, K plays the role of the basis size and α mimics the bias towards rewarding covering and penalizing overcovering (the flags `-bonus-covered` and `-penalty-overcovered` in ASSO).

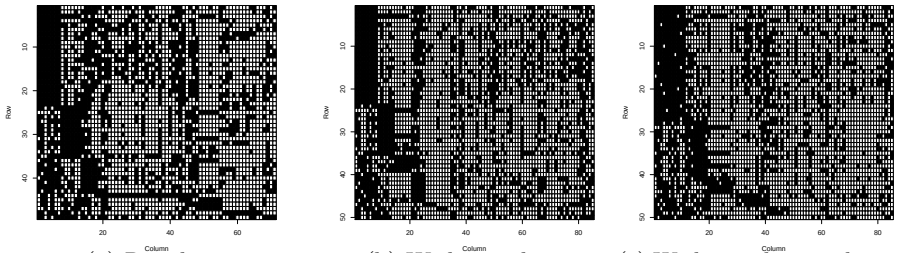
It is well-known that tiling and *Boolean matrix factorization* (BMF) are closely related (Miettinen 2012). Hence, let us also briefly show how BMF can be realized in our framework. It corresponds to an instance of DBP where only binary values (true and false) are possible and the `no-overcoverage` constraint applies. Hence, it is required that the factorization undercovers the initial dataset, i.e., if there is a 0 in a position in the original dataset, then there cannot be a 1 in the approximation. Therefore, the optimization criterion of DBP is further simplified and we obtain the following BMF model, without overcovering, in Listing 3.6.

Input: dataset db and constant K
Shape: $\mathit{approx}(\mathit{Attr}, \mathit{Transct}) \leftarrow \mathit{tile}(\mathit{Indx}, \mathit{Attr}), \mathit{in}(\mathit{Indx}, \mathit{Transct})$
Find: the set of ground facts $\mathit{tile}(\cdot), \mathit{in}(\cdot)$
Satisfying: `number-of-patterns(K)` $\wedge$ `no-overcoverage`
Maximizing: `coverage`

Listing 3.6: BMF without overcovering

3.3.3 Discriminative k-pattern set mining

A common supervised data mining task is that of discriminative pattern set mining (Knobbe and Ho 2006). Let $db(\mathit{Value}, \mathit{Attr}, \mathit{Transct})$ be a categorical dataset, $\mathit{positive}(T)$ ($\mathit{negative}(T)$) be the set of positive (negative) transactions, and k the number of tiles. Then, the task is to find extensions of the relations $\mathit{tile}(\mathit{Indx}, \mathit{Value}, \mathit{Attr})$ and $\mathit{in}(\mathit{Indx}, \mathit{Transct})$ such that positive and negative transactions are discriminated. A standard interpretation is to find tiles that



(a) Regular (b) With penalties (c) With penalties and noise

Figure 3.4: Re-arranging a matrix in block-diagonal form (Animals dataset): (a) regular, (b) with penalties, (c) with noisy blocks and penalties

cover as many positive and as few negative ones as possible (Liu et al. 1998). The only required change in the model concerns the scoring function (and assigning some weight to the errors):

$$\#\{T : \mathit{covered}(T), \mathit{positive}(T)\} - \alpha\#\{T : \mathit{covered}(T), \mathit{negative}(T)\}, \qquad (3.6)$$

where α is a constant that represents the weights for the errors made. It is typically a domain specific parameter (the cost of covering a negative example by a rule, i.e., the false positive cost or a weight of a negative example). Let us denote the coverage of the positive transactions as $\texttt{coverage}^+$ (left set term in Eq. 3.6) and the coverage of negative as $\texttt{coverage}^-$ (right set term in Eq. 3.6).

Given that we have no `no-overcoverage` constraint and negative transactions can be covered, the optimization criterion is given by

$$\mathit{similarity}(T) = \texttt{coverage}^+ - \alpha \times \texttt{coverage}^-.$$

This corresponds to the high-level model in Listing 3.7.

Input: dataset db and constants K, α
Shape: $\mathit{approx}(\mathit{Attr}, \mathit{Value}, \mathit{Transct}) \leftarrow \mathit{tile}(\mathit{Indx}, \mathit{Value}, \mathit{Attr}), \mathit{in}(\mathit{Indx}, \mathit{Transct})$
Find: the set of ground facts $\mathit{tile}(\cdot), \mathit{in}(\cdot)$
Satisfying: `number-of-patterns(K)` $\wedge$ `no-overcoverage` $\wedge$ `one-value-attribute`
Maximizing: $\texttt{coverage}^+ - \alpha \times \texttt{coverage}^-$

Listing 3.7: ReDF Discriminative Patter Set Mining Model

3.3.4 Block-diagonal matrix form

Aykanat et al. 2002 introduced the problem of and an algorithm for permuting the rows and columns of a sparse matrix into block diagonal form. They relate

this problem to other combinatorial and classical linear algebra problems. The underlying block-diagonal structure of a matrix can be used to parallelize certain matrix computations. An illustration of block-diagonalization (several variants) of the Animals dataset is depicted in Figure 3.4.

We reduce it to a form of tiling. The shape query is the same as in tiling but the constraints are different: if a tile I_1 has an attribute A, then a tile I_2 cannot use the same attribute. A similar constraint is imposed on the *in* predicate and transactions stating that each item A can belong to only one tile

`item-blocking`: $\leftarrow \mathit{tile}(I_1, A), \mathit{tile}(I_2, A), I_1 \neq I_2$.

Only one tile can occur in a transaction T.

`transaction-blocking`: $\leftarrow \mathit{in}(I_1, T), \mathit{in}(I_2, T), I_1 \neq I_2$.

We also modify the optimization criterion to take into account elements not covered by a tile but blocked by this tile. Every tile that selects attributes and transactions prohibits other tiles to use these attributes and transactions by means of the `item-blocking` and `transaction-blocking` constraints. We penalize excessive usage of attributes and transactions by a single tile. We do this to improve the block form of the matrix, since in this task we are not just interested in a tiling with maximal coverage, but in a tiling that maximizes the number of elements on the diagonal and minimizes the number of elements everywhere else. To enforce this we introduce two functions:

`item-penalty`: $\#\{(T, A) : \mathit{approx}(T, A'),\ \mathit{not\ covered}(T, A)\}$

`transt-penalty`: $\#\{(T, A) : \mathit{approx}(T', A),\ \mathit{not\ covered}(T, A)\}$

Then, the whole problem is formulated in Listing 3.8.

Input: dataset *db* and constants N, α, β
Shape: *approx*(*Attr*, *Value*, *Transct*) $\leftarrow$ *tile*(*Indx*, *Value*, *Attr*), *in*(*Indx*, *Transct*)
Find: the set of ground facts $\mathit{tile}(\cdot), \mathit{in}(\cdot)$
Satisfying: `transaction-blocking` $\wedge$ `item-blocking`
`one-value-attribute`$\wedge$`noisy-overcoverage(N)`$\wedge$`no-tile-intersection`
Maximizing: `coverage` $- \alpha \times$ `transt-penalty` $- \beta \times$ `item-penalty`

Listing 3.8: ReDF Block-Diagonal Model

If we omit `item-penalty` and `transt-penalty`, we obtain the standard optimization function for tiling. In the experimental section we evaluate the effect of the presence of this penalty.

3.4 Beyond classic problems

So far we have focused on matrix-like representations of the data, in which the dataset was represented by instances of $db(T, A, V)$, for a transactions T having a value V for an attribute A. This representation is independent of the number of attributes and values, it allows one to easily specify constraints over all attributes and to access the data using the predicate db only. We will now show that it is also possible to use other, purely relational representations, such as the *sells* example from the Introduction.

Section 3.2 already provided the *sells* example for decomposing a ternary relation into three binary ones. In the shape for the *sells* example in Listing 3.3 there is no latent variable: there are only attributes from the original dataset. Since there is no latent variable, there is no "pattern" to be found for which the optimization criterion needs to be optimized, which allowed us to use a simple error function using only one type of atom.

However, latent variables can also be useful in a purely relational setting. Let us illustrate this on an example inspired by the ArXiv community analysis example of Gopalan and Blei 2013. Assume we are given a relation *publishedIn* with attributes *Author*, *University*, and *Venue*, specifying that an author belonging to a particular university publishes in a particular venue. Furthermore, assume we want to factorize this relation into the relation *approx (A,U,V)* by introducing a latent attribute *Topic*, denoted as T. The latent topic variable clusters authors, universities and venues together in such a way that their join results in publications.

We obtain the following high-level model in Listing 3.9, where α is the constant that indicates the relative cost of overcovering an element and the integer constant k is the number of value that the latent variable (T) can take:

$$approx(A, U, V) \leftarrow interestedIn(A, T), specializedIn(U, T), inField(V, T).$$

Input: dataset db and constants K, α
Shape: $approx(A, U, V) \leftarrow interestedIn(A, T), specializedIn(U, T), inField(V, T)$.
Find: the set of ground facts $interestedIn(\cdot), specializedIn(\cdot), inField(\cdot)$
Satisfying: `number-of-patterns(K)`
Maximize: `coverage` $- \alpha \times$ `overcoverage`

Listing 3.9: ReDF Purely Relational

The corresponding model without latent variables would be different only in the decomposition shape, i.e., it would look like

$$approx(A, U, V) \leftarrow worksAt(A, U),\ publishesAt(A, V),\ knownAt(U, V).$$

Discriminative relational learning In the spirit of discriminative pattern mining, described in Subsection 3.3.3, we can also do discriminative learning in the purely relational setting. To do so, we assume that the relation has an extra argument Co-Author and we would like to discriminate the dataset by a particular Co-Author c^+, i.e.,

$$\begin{aligned}&\texttt{coverage}^+(A,U,V) \leftarrow \mathit{approx}(A,U,V), \mathit{publishedIn}(A,U,V,c^+).\\&\texttt{coverage}^-(A,U,V) \leftarrow \mathit{approx}(A,U,V), \mathit{publishedIn}(A,U,V,C), C \neq c^+.\end{aligned} \tag{3.7}$$

Then, the optimization criterion remains the same as in Subsection 3.3.3. Intuitively, if we have only information about an author of a paper (together with his or her university affiliation and a venue), we use this to 'predict' his or her co-author using the patterns we obtain in this discriminative setting.

3.5 Implementation

This section describes how ReDF models can be implemented in ASP. We do this for the basic problem of tiling, as well as for the purely relational data factorization presented before. Implementations of the other variations are included in Appendix A.1.2. Our primary implementation is written in Clasp, can be used with the Clasp system (Brewka et al. 2011; Gebser et al. 2012).

3.5.1 General computation methods: greedy and sampling approaches.

In all described problems, the goal is to find k patterns or tiles, where a pattern is interpreted as a set of facts corresponding to a particular value of the latent variable. We will follow an iterative approach to finding these patterns, in which the discovery of the next pattern or tile will be encoded in ASP. We will consider both a *greedy* and a *sampling* algorithm for realizing this. The sampling approach is intended for better scalability and will be evaluated in Section 3.6.1.

Greedy model. The greedy approach is described in Algorithm 2. Essentially, when the next best *pattern* has been computed (where *pattern* is a set of facts associated with the *pattern* identifier, e.g., in tiling a pattern is a set of transactions and attributes), it is added to the current set of *patterns*. The specific part for each tile is represented by `executeProgram` and is encoded separately in ASP. Note that this greedy, iterative approach to finding k patterns

is very common in pattern mining. Theoretical bounds on the solution quality of the greedy approach have been studied in the context of the maximum k-set coverage problem (Feige 1996; Hochbaum and Pathria 1998); more details can be found in Appendix 3.9.

Algorithm 2: Greedy execution model

input : *data* is the dataset
output: *patterns* is the set of patterns
patterns $\leftarrow \emptyset$;
for $i \in [1, k]$ **do**
    pattern $\leftarrow$ `executeProgram`(*data*, *patterns*, i);
    patterns $\leftarrow \{pattern\} \cup patterns$;

Column sampling execution model. To improve scalability, we employ a sampling approach. Interestingly, our approach is different from most existing sampling techniques in data mining: instead of sampling a rows or patterns, we sample columns. Algorithm 3 presents the column sampling approach we propose. The key difference with the greedy approach is that instead of determining the next best pattern on the *overall* dataset in each iteration, this approach samples N subsets of the data and determines the next best pattern for all of these subsets. The best among these is then fixed, and the process is repeated. We empirically evaluate the effects of sampling on the quality of the computed patterns and on the runtime in the experiment section. Quality bounds for this type of greedy search have also been analyzed previously (Hochbaum and Pathria 1998); for more details we refer to Appendix 3.9.

Algorithm 3: Column sampling execution model

input : *data* is the dataset
input : N is the number of samples
input : α is the relative size of a sample
output: *patterns* is the set of patterns
patterns $\leftarrow \emptyset$;
for $i \in [1, k]$ **do**
    maxPattern $\leftarrow \emptyset$;
    for $j \in [1, N]$ **do**
        sample $\leftarrow$ `getColumnSample`(*data*, α);
        pattern $\leftarrow$ `executeProgram`(*sample*, *patterns*, i);
        if `score`(*pattern*) $>$ `score`(*maxPattern*) **then**
            maxPattern $\leftarrow$ *pattern* ;
    patterns $\leftarrow \{maxPattern\} \cup patterns$;

```
1  %one-value-attribute; it generates at most one value per attribute
2  0 { tile(current, Value, Attribute) : valid(Attribute, Value) } 1 :- col(Attribute).
3  %no-overcoverage
4  over_covered(current,T) :- not db(Val, Attr, T), tile(current, Val, Attr),
       transaction(T).
5  %no-tile-intersection
6  intersect(T) :- current != I, tile(current, Val, Attr), tile(I, Val, Attr), in(I,T).
7  %defines presence of tiles in transactions
8  in(current,T) :- transaction(T), not over_covered(current, T), not intersect(T).
9  %defines coverage function
10 covered(Transct, Attribute) :- in(Index,Transct), tile(Index, _, Attribute).
11 #maximize[covered(Transct, Attribute)].
```

Listing 3.10: Greedy maximum k-tiling formalization in answer set programming

3.5.2 Data mining problems expressed in the framework

The maximum k-tiling problem can be encoded in answer set programming as indicated in Listing 3.10. The code implements the greedy model, i.e., Algorithm 2, for the maximum k-tiling problem with a fixed number of tiles (Geerts et al. 2004). It assumes we have already found an optimal tiling for $n-1$ tiles, and indicates how to find the n-th tile to cover the largest area. The n-th tile is called *currentI* in the listing. Further, we have information about the names of the attributes and the possible values for each attribute through predicates *col*(*Attr*) and *valid*(*Attr*, *Value*). That is, *col*(A) is an unary predicate that encodes possible column indices, and *valid*(A, V) is a binary predicate that encodes which possible values V can occur in column A.

Let us explain the code in Listing 3.10. The constraint in Line 2 generates at most one value for each attribute. The constraints in Lines 4 and 6 compute the transactions where the current tile cannot occur, i.e., `intersect(T)` is the set of all transactions where the current tile overlaps with another tile and the current tile cannot cover these transactions. Similarly, `overcovered(currentI,T)` is the set of transactions that cannot be covered because there is an element in the current tile, with fixed index `currentI`, that is not present in transaction `T`. The constraint in Line 8 states that if the tile does not violate the overcovering and intersection constraints in a transaction, it occurs in the transaction. Line 10 defines the coverage and the optimization constraint in Line 11 enforces the selection of the best model.

Theorem 3.1 (Correctness of the greedy ASP tiling encoding)**.** The ASP program $\mathcal{P}$ defined by the Listing 3.10 computes the k-th largest tile w.r.t. the scoring function `coverage` (3.5) as extensions of the predicates *tile*($k, \cdot, \cdot$) and

$in(k, \cdot)$ in its answer set $\mathcal{A}$, provided that the dataset is represented extensionally through the predicates *db*, *valid*, and *col* and the $k-1$ already found tiles are represented extensionally through the predicates $tile(I, \cdot, \cdot)$ and $in(I, \cdot)$ for $I \in [1, k-1]$.

For the proof, see Appendix A.1.1. The Clasp encodings for the other models are sketched in Appendix A.1.2.

3.5.3 Purely relational data factorization

In Section 3.4 we presented a factorization of the *publishedIn* relation into three binary relations. It constitutes a proof-of-concept prototype model in ASP and could be improved by, e.g., incorporating heuristics.

The general structure of the ASP encoding is similar to the *sells* example in Listing 3.3: we indicate here only a possible optimization for the relation generators. We use the left-to-right order of the atoms in the schema (replicated below) while generating candidates for the factorization.

```
1 0 { works_at(A,U)      } 1 :- published_in(A,U,V).
2 0 { publishes_at(A,V) } 1 :- published_in(A,U,V), works_at(A,U).
3 0 { known_at(U,V) } 1 :- published_in(A,U,V),works_at(A,U),publishes_at(A,V)
      .
```

Listing 3.11: Generators for the model without a latent variable into three binary relations

Implementation differences. When we generalize the factorization encoding with two relations to three relations, we observe a slight implementation difference between them. Factorization with the two relation shapes can be naturally implemented using the core ASP generate-and-test paradigm. Once we have guessed an extension for a certain value of the latent variable, we propagate it to the second relation and test against the constraints. This strategy is often deployed in specialized algorithms (Geerts et al. 2004; Miettinen et al. 2008). For a multiple relation shape we guess an extension of one relation, then we constrain the possible values we generate for the second value (e.g., see Line 2 in Listing 3.11). In general, we can search for one at a time using a greedy strategy (as in tiling). Theoretically, we can simultaneously search for values of a latent variable by replacing the fixed latent parameter by a variable and searching over the latent parameter as well. The work of Guns et al. 2013a provides evidence that this approach does not scale well, unless special propagators are introduced into the solver. This technique would allow extending the method to other shapes with more than three relations.

3.6 Experiments

The main goal of this section is to evaluate whether ReDF problems can be solved using a generic solver. In particular, we focus on solving the problem formulations as we specified them in ASP. We investigate whether the problems can be solved, and for a number of tasks compare the results and runtimes to those obtained by specialized algorithms. Since we here use generic problem formulations and generic solvers that have neither been designed nor optimized for the tasks under consideration, we cannot expect the approach to be as efficient as specialized algorithms. However, what is more important is that we demonstrate that all tasks formalized and prototyped using the ReDF framework can be solved using a unified approach.

Experimental setup and datasets. The ASP engine we use is 64-bit clingo (Clasp with the gringo grounder) version 3.0.5 with the parameter `-heuristic=Vmtf` (see Appendix 3.7 for details on the parameters) and all experiments are executed on a 64-bit Ubuntu machine with Intel Core i5-3570 CPU @ 3.40GHz x 4 and 8GB memory, except for Maximum k-tiling on Chess and Mushrooms datasets where Intel Xeon CPU with 128GB of memory (all single-threaded) has been used due to high memory requirements. For most experiments we use the datasets summarized in Table 3.1, which all but one originate from the UCI Machine Learning repository (Bache and Lichman 2013). The *Animals (with Attributes)* dataset was taken from Osherson et al. 1991. For the purely relational factorization task, the data and experiment results are described separately in the corresponding subsection.

Table 3.1: Dataset properties. For each dataset, we specify whether the attributes have Boolean or categorical domains, the number of tuples and attributes, and the average number of distinct values per attribute

Dataset	**Attributes**	**# tuples**	**# attributes**	**Avg # values per attribute**
Animals	Boolean	50	85	2
Solar flare	categorical	1 389	11	3.3
Tic-tac-toe	categorical	958	10	2.9
Nursery	categorical	12 960	8	3.4
Voting	categorical	435	17	3.0
Chess (Kr-vs-Kp)	categorical	3 196	36	2.1
Mushroom	categorical	8 124	22	5.6

Table 3.2: Maximum k-Tiling

(a) Runtime

Dataset	Number of tiles (k) 5	10	15	20	25
Animals	36s	1m4s	1m21s	1m32s	1m36s
Solar flare	6s	10s	13s	16s	18s
Tic-tac-toe	22s	31s	33s	34s	35s
Nursery	4m19s	6m32s	7m32s	7m56s	8m13s
Voting	52s	1m28s	1m42s	1m46s	1m49s
Chess	17h03m	22h31m	-	-	-
Mushroom	13h09m	19h44m	-	-	-

(b) Coverage

Number of tiles (k) 5	10	15	20	25
0.327	0.472	0.573	0.649	0.709
0.416	0.565	0.655	0.721	0.751
0.251	0.449	0.623	0.784	0.907
0.269	0.454	0.634	0.773	0.905
0.399	0.553	0.662	0.749	0.810
0.483	0.618	-	-	-
0.476	0.586	-	-	-

In Subsection 3.6.1 we show how ReDF formulations of existing data mining tasks (from Section 3.3) can be solved using the implementation presented in Section 3.5, afterwards in Subsection 3.6.2 we show the results of the purely relational data factorization task. The ASP solver parameters used in the experiments and a breakdown of individual solving steps and their runtimes determined by the meta-experiment are presented in Appendix 3.7.

3.6.1 Solving existing tasks

Maximum k-Tiling in Categorical Data We first consider the Maximum k-Tiling problem from Section 3.3.1 and present timing and coverage results in Table 3.2 obtained on all datasets from Table 3.1.

In all cases the problem specification given in Listing 3.10 was used to greedily mine $k = 25$ tiles. Since the problem becomes more constrained as the number of tiles increases, runtime decreases for each additional tile mined. We therefore report total runtime and coverage for different values of k, i.e., for different total numbers of tiles. Only $k = 10$ tiles were mined on Chess and Mushroom due to long runtimes.

Effect of sampling As we can see from Table 3.2a, runtimes are quite long on datasets like Mushroom. To address this issue, we use the sampling procedure of Algorithm 3 with the following parameters: $\alpha = 0.4$ and $N = 20$, i.e., 40% of all attributes were selected uniformly at random for each sample and 20 samples were used. Intuitively, the larger the sample size and the more samples, the better we approximate the exact result.

With the given parameters, we attain an order of magnitude improvement in runtime: instead of 19 hours with the regular algorithm, using sampling it takes only one hour to compute 10 tiles as indicated in Figure 3.5a. The effect of using sampling on coverage can be seen in Figure 3.5b: the first tiles that are mined have lower coverage than when sampling is not used, but after a while the difference in coverage with LTM-k remains more or less constant and even slightly decreases. LTM-k is the original, specialized tiling algorithm, to which we compare next.

Comparison to a specialized algorithm We now compare the performance of the ASP-based implementation of LTM-k greedy strategy to that of a specialized implementation[3]. Figures 3.5a and 3.5b present both runtime and coverage comparisons obtained on Mushroom, both for our approach (with and without sampling) and the specialized miner.

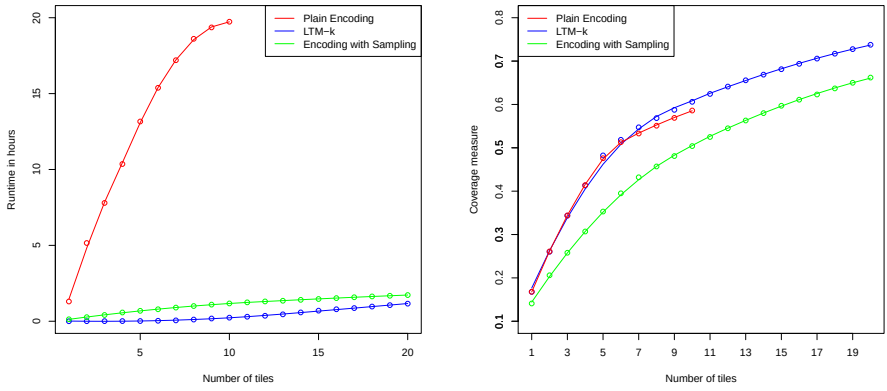
Without sampling, we can see that our approach gives the same results in terms of the coverage as the LTM-k algorithm. This is as expected though, since both LTM-k and our approach guarantee to find an optimal solution in each iteration. The slight difference between the two coverage curves in Figure 3.5b is caused by the fact that multiple tiles can have the same (maximum) area, and some choice between those has to be made. Although these choices are typically made deterministically, the different implementations make decisions based on different criteria, resulting in slightly different tilings.

Unfortunately, the ASP solver is not as efficient as the specialized miner as can be seen in Figure 3.5a, and the generality of the approach comes at the cost of longer runtimes. However, as already discussed, using a sampling approach can substantially decrease the runtime. Experiments on other datasets showed similar behavior to that depicted here.

Overlapping tiling To evaluate the overlapping tiling task from Subsection 3.3.1, we apply the model in Listing A.1 (ASP encoding in Appendix A.1.2) to the five smaller datasets from Table 3.1. We experiment with two levels of overlap, i.e., parameter N is set to either 1 or 2: tiles can intersect on at most one or two attribute(s). As the results in Table 3.3 show, allowing limited overlap can lead to a small increase in coverage, but runtimes also increase due to the costly aggregate operation in Line 1 of Listing A.1.

However, what is important to emphasize here is that only a small change in the problem formalization is sufficient to allow for overlap in the tilings, while the solver can still solve the problem without any further changes. And although

[3] http://people.mmci.uni-saarland.de/~jilles/prj/tiling/



(a) Runtime (b) Coverage

Figure 3.5: Tiling comparison (runtime, coverage) with LTM-k (Mushroom dataset)

Table 3.3: Maximum k-Tiling with overlap. The maximal allowed overlap is limited by parameter N

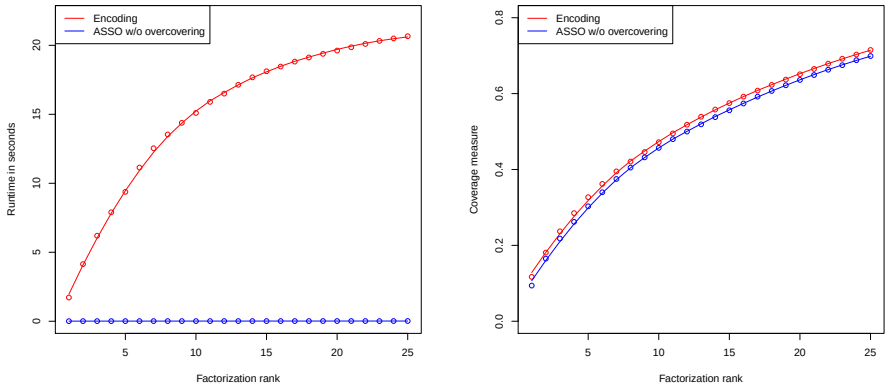
(a) Runtime

		Number of tiles (k)				
Dataset	**N**	**5**	**10**	**15**	**20**	**25**
Animals	1	1m10s	2m28s	3m46s	4m24s	4m47
	2	1m39s	4m10s	6m26s	7m40s	8m10s
Solar flare	1	8s	13s	17s	21s	24s
	2	8s	15s	20s	25s	29s
Tic-tac-toe	1	24s	41s	49s	52s	53s
	2	23s	43s	51s	55s	56s
Nursery	1	5m00s	8m19s	10m10s	10m48s	11m12s
	2	5m43s	9m32s	11m9s	11m50s	12m12s
Voting	1	1m10s	2m19s	2m53s	3m8s	3m15s
	2	1m39s	3m34s	4m35s	5m9s	5m33s

(b) Coverage

	Number of tiles (k)				
5	**10**	**15**	**20**	**25**
0.327	0.475	0.583	0.663	0.722
0.332	0.482	0.592	0.675	0.742
0.433	0.595	0.684	0.734	0.756
0.452	0.602	0.685	0.731	0.755
0.253	0.451	0.626	0.781	0.898
0.253	0.451	0.626	0.781	0.898
0.268	0.454	0.633	0.772	0.905
0.268	0.454	0.633	0.772	0.905
0.403	0.558	0.675	0.765	0.828
0.409	0.571	0.683	0.762	0.819

the runtimes are longer when more overlap is allowed, the difference with the basic, non-overlapping setting is moderate.



(a) Runtime (b) Coverage

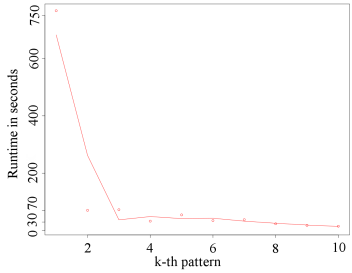
Figure 3.6: Boolean matrix factorization on datasets Animals. Runtime and coverage are depicted for different factorization ranks.

Boolean matrix factorization (BMF) We perform Boolean matrix factorization (Section 3.3.2) by applying the formalization of Listing A.2 and compare the results to those obtained by ASSO[4] (Miettinen 2012) with the no-overcoverage flag (`-P1000`). The factorization rank k is incremented by one in each iteration, and meanwhile coverage gain and runtime are measured. The results for Animals are presented in Figure 3.6 and show that coverage is almost identical to that obtained by ASSO. Again, this is unsurprising, as our implementation follows the same solving strategy. However, runtimes are several times higher, which is due to the usage of a general solver that is not optimized for this type of task. Results obtained on other datasets are very similar and are therefore not presented here.

Discriminative pattern set mining Here we demonstrate how the discriminative k-pattern mining model from Section 3.3.3 can be solved. For this we use Chess and Tic-tac-toe from Table 3.1, each of which has a binary class label indicating whether a game was won or not and can therefore be naturally used for this task.

We apply the encoding from Listing A.3 to both datasets, set $\alpha = 1$ to weigh positive and negative tuples equally, and summarize the results in Figure 3.7b. The results show that five patterns suffice to cover all positive examples of

[4] `http://www.mpi-inf.mpg.de/~pmiettin/src/DBP-progs/`

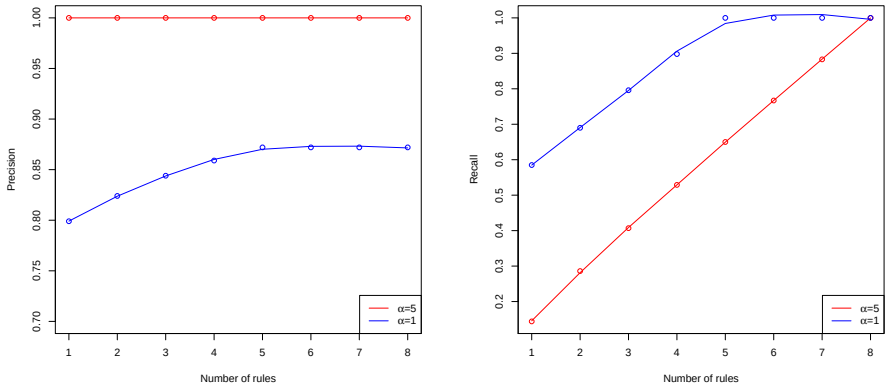


(a) Runtime (in s) to mine k-th discriminative pattern on Chess dataset ($\alpha = 1$, i.e., positive and negative tuples are weighted equally)

(b) Discriminative mining coverage on Chess and Tic-tac-toe datasets ($\alpha = 1$, i.e., positive and negative tuples are weighted equally)

	Tic-tac-toe ($k = 5$)	Chess ($k = 10$)
Covered −	92 (27.7%)	160 (7%)
Covered +	626 (100%)	864 (95.5%)
Difference	534	704
Runtime	0.52s	18m48s

Figure 3.7: Discriminative pattern set mining summary: runtime (left) and coverage (right)



(a) Precision (b) Recall

Figure 3.8: Discriminative pattern set mining (Tic-tac-toe dataset): precision (left) and recall (right) for different α, i.e., for varying weights of covering negative transactions

Tic-tac-toe, hence mining more than five patterns would be useless. 92 of the 718 covered tuples are negative, i.e., 12.8%, while 34.7% of the tuples in the complete dataset is negative. For Tic-tac-toe, the time needed to solve this task is very limited: about half a second.

Figure 3.7a shows the runtime needed to iteratively find subsequent patterns in the Chess dataset. Interestingly, it seems that the problem becomes substantially

easier (computationally) once the first few patterns have been found: the runtime per pattern drops heavily. This confirms that the search space shrinks when the problem becomes more constrained, i.e., the number of answer sets decreases with the addition of more constraints.

We next show the influence of the α parameter, i.e., the relative weight of covering positive and negative tuples in the optimization criterion. By increasing α, the 'penalty' for covering a negative tuple is increased and the algorithm can be forced to select more conservative rules. We investigate the effect of this parameter by measuring and comparing precision and recall of the obtained pattern sets for $\alpha = 1$ and $\alpha = 5$. Figure 3.8 shows that precision goes to 1 when α is increased, while recall is decreased but this can be compensated by mining a larger number of patterns[5].

This task differs from the previous one in its optimization criterion: positive coverage penalized by negative coverage allows for fast inference and discovery of the optimal solution, which results in shorter runtimes than for tiling.

Matrix block-diagonal form We apply three versions of the encoding to the Animals dataset (Osherson et al. 1991). The results presented in Figure 3.4 demonstrate that the Animals dataset can be re-arranged into block-diagonal form using our proposed framework. The runtime in all experiments are on the order of seconds. Parameters used in the experiments were $\alpha = \frac{3}{20}$ and $\beta = \frac{1}{20}$. Figure 3.4c demonstrates the model from Section 3.3.4, with the same α, β and $N = 1$. The low-level encoding of this model is given in Listing A.4.

3.6.2 Purely relational data factorization

In Section 3.4 we described how to model the factorization of *publishedIn(Author, University, Venue)* into three binary relations with a latent variable *Topic*. We now evaluate whether the standard ASP solver can solve this task. Unfortunately, we cannot expect a generic solver to handle enormous datasets such as the one from ArXiv as described by Gopalan and Blei 2013. Instead, we demonstrate a proof-of-concept of solving the model in Listing A.5 on a moderate dataset.

We constructed a dataset for a well-known colleague from the data mining community: Bart Goethals (Antwerp University). We collected his publication list from Microsoft Academic Search[6] and extracted for each paper the publication venue, and all co-authors together with their corresponding

[5] Appendix A.1.3 presents the data points of Figure 3.8 as a traditional precision-recall plot.

[6] http://academic.research.microsoft.com/Author/2266478/bart-goethals

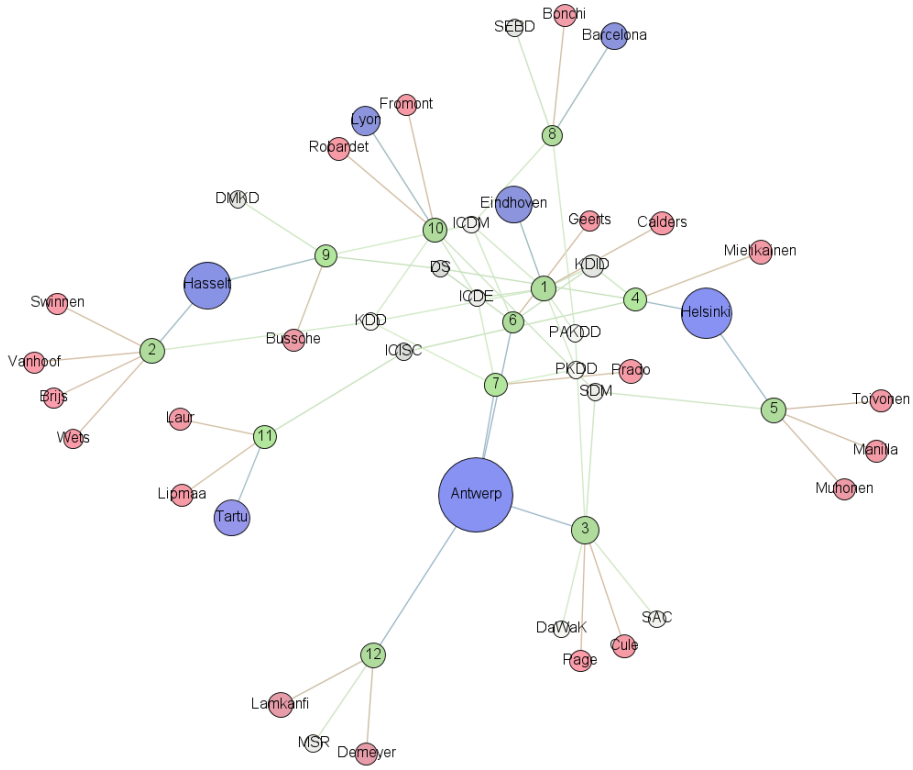


Figure 3.9: Clustering in topics by ReDF. Red nodes represent co-authors, blue their university cities, white nodes venues and green topics that bind them together. If there is an edge between a topic and a node, then there is a corresponding element in the relation (i.e., *interestedIn*, *specializedIn* or *inField*)

affiliations (i.e., the last known affiliation for each author in this list of papers). Each unique combination of venue, co-author, and affiliation resulted in a tuple in the *publishedIn* relation. The complete dataset contains 57 tuples over 19 universities, 38 authors, and 15 venues.

Intuitively, if a set of *authors* from a set of *universities* publish in a set of *venues*, then there must be an underlying research *topic* that unites them. Hence, by factorizing the relation into three separate relations, we cluster each of the entity types into a (fixed) number of topics, as indicated by the value of the latent *Topic* variable.

The results for factorization using $K = 12$ topics and $\alpha = \frac{1}{2}$ are presented in Figure 3.9, including co-authors (red), universities (blue), publication venues

(purple), and topics (green). To determine the number of topics K, we tracked the optimization criterion while increasing K and stopped when this no longer improved.

Since the task is of an exploratory character, we can only qualitatively evaluate the results. We observe that all data mining venues are located together in the center, connected to the same topics. SEBD, an Italian database conference, stands apart, and there is also a separate block for database and computing venues DaWaK and SAC. Manual inspection of the results indicates the topics (or clusters) to be coherent and meaningful: they represent different affiliations and groups of co-authors that Bart Goethals has collaborated with. For example, topic 5 contains the SDM conference, the University of Helsinki, and three co-authors specialized in Data Mining. Hence, this topic could be described as "Data mining collaboration with the University of Helsinki", which makes perfect sense as Bart Goethals was previously a researcher in Helsinki.

Not all authors are represented in the factorization. How much of the *publishedIn* dataset is covered depends on the number of topics K (which was chosen as described before). The higher the cardinality of the pattern set, the larger the total coverage. The *covered* elements positively contribute to coverage, whereas the *overcovered* elements contribute negatively. This implies that each pattern is chosen such that the number of *covered* and *overcovered* elements are balanced and the optimization criterion is maximized. In general, covering all authors with few patterns would lead to significant overcovering of the original dataset, while introducing too many patterns would create clusters with only one author (which is clearly undesirable, since these clusters would not be meaningful).

The decompositions, as the one depicted in Figure 3.9, could serve as a basis for new analyses. For example, we might visualize the intersection of common (latent) topics shared by two researchers. We outline possible examples in Appendix 3.8.

Relational factorization without a latent variable. In Section 3.4, we also described a factorization that does not use any latent variables (analog to the *sells* example in Listing 3.3 from the introduction section). We evaluate this model using Listing 3.11 on the same dataset as used in the previous experiment, i.e., the co-author relation *publishedIn(Author, Uni, Venue)* for Bart Goethals.

In general, factorizations do not perfectly match the original relation (i.e., $error \neq 0$), but in this particular case the system found a lossless solution. It is easy to see that this will not always be possible though. For instance, let us assume we keep multiple affiliations per author in the dataset. For example, apart from a fact *p(bonchi,barcelona,pakdd)*, there may be another

fact *p(bonchi, pisa, pakdd)* in *publishedIn*. Although the same factorization would be found by the solver, the found solution would be imperfect as the latter fact is not represented in the factorized relation.

Solving this task was computationally easy, since there is no latent variable to iterate over: the runtime was only $0.01s$. Table 3.4 presents a summary.

Table 3.4: Experimental summary for pure relational factorizations from Subsection 3.4

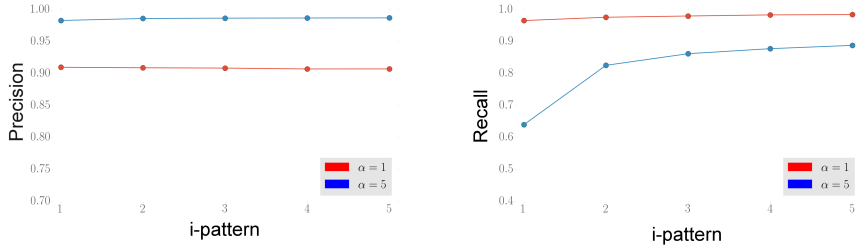
With a latent variable	**#Transactions**	**Overall runtime**	**Avg runtime**	**#Topics**	**Avg atoms per topic**
	58	14s	1.1s	12	5.4
Without a latent variable	**#Transactions**	**Overall runtime**	**Correct**	**#Incorrect**	**Avg factor size**
	58	0.01s	58	0	45

Relational discriminative learning Here we investigate discriminative learning in the purely relational setting, for which we use the discriminative optimization criteria described in Section 3.4. For this experiment we collected DBLP data for two well-known researchers in the field of data mining: Jiawei Han and Philip S. Yu. In this example all publications belonging to either researcher are considered a class. Since DBLP data does not have authors affiliations, we replace this attribute with the publication date converted to a categorical variable $M \in \{old, recent, new\}$, using the following rule: if the date is later than 2010, it is represented as "new"; if it is between 2005 and 2010, then it is "recent"; otherwise it is "old". The complete dataset contains around 6000 ground atoms of the following form: *paper(Author,Age,Venue,Han\Yu)*. The goal is to predict whether the paper is co-authored by Han or Yu based on author, venue and age using discriminative rules as defined in Eq. 3.7. In this experiment the shape is

$$approx(A, M, V) \leftarrow author(A, T), paperAge(M, T), venue(V, M),$$

where T is a latent variable. As for the previous discriminative experiment, in Figure 3.10 we present an overview of the dependency of precision and recall on the number of patterns and α, the penalty for covering the incorrect class. Runtimes are similar as in Figure 3.7a. ASP finds an optimal solution in half an hour and then spends a substantial amount of time to prove it is indeed the best solution. We therefore used a time limit of one hour per pattern to speedup the computation. This limit was only reached in the computation of the first pattern, for both values of α.

In Figure 3.10 we see that, depending on the number of patterns and penalty for covering the wrong class (α), we can obtain a classifier with different precision and recall[7].



(a) Precision (b) Recall

Figure 3.10: Discriminative learning in the purely relational setting; precision (left) and recall (right) for different α, i.e., for two weights of covering negative atoms.

3.6.3 Runtime discussion

In this section we have seen a number of experiments that solve ReDF problems using generic solving technology, i.e., answer set programming. As we can see in Figures 3.5a and 3.6, specialized algorithms are substantially faster than ASP. On datasets of moderate size, however, generic solvers obtain reasonable runtimes, as indicated by the results in Tables 3.2a, 3.3a, and 3.7b, and Figures 3.6 and 3.7a. For the purely relational data factorization task from Section 3.4 we present a summary of the experiments in Table 3.4. In these experiments, computation time ranged from several seconds to few minutes.

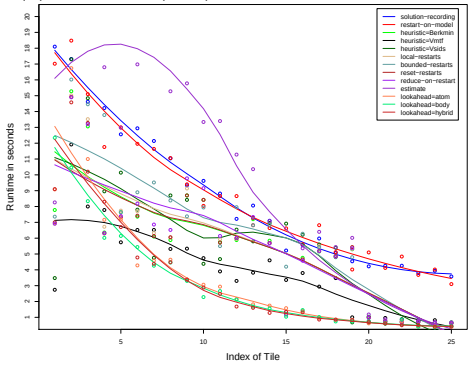
3.7 Evaluating solver performance

This appendix briefly evaluates the performance of the solver that was used for the experiments. We first describe how we tuned the system and then continue with an analysis of the solving process.

Solver tuning. The clingo system (Gebser et al. 2011) has a variety of parameters that affect runtime. We performed a series of preliminary experiments to

[7] Appendix A.1.3 presents the data points of Figure 3.10 as a traditional precision-recall plot.

(a) Runtime (in s) for individual tiles



(b) Runtime (in s) for N tiles

	Number of tiles				
Search option	**5**	**10**	**15**	**20**	**25**
solution-recording	76	131	168	193	213
restart-on-model	75	127	164	191	211
heuristic=Berkmin	50	86	112	133	137
heuristic=Vmtf	**36**	**64**	**83**	**95**	**100**
heuristic=Vsids	54	88	119	138	140
local-restarts	49	87	113	134	138
bounded-restarts	59	101	132	154	158
reset-restarts	48	85	111	132	136
reduce-on-restart	48	89	115	136	139
estimate	86	169	213	237	241
lookahead=atom	**51**	**71**	**81**	**86**	**88**
lookahead=body	**43**	**63**	**72**	**76**	**79**
lookahead=hybrid	**48**	**68**	**77**	**81**	**84**

Figure 3.11: Determining the best ASP solver parameters for tiling with the model of Listing 3.10: (a) the runtime (in s) needed for mining each subsequent individual tile, (b) runtime summary (in s) for first 25 tiles

determine the parameters that were used for all other experiments in this section. These tuning experiments mostly concerned maximum k-tiling on datasets of moderate size, but some test experiments on other tasks did not reveal any discrepancies.

Runtime was measured for a large number of search options while fixing the dataset and task. Figure 3.11 shows the results obtained for maximum k-tiling on the Animals dataset. Search options "heuristic=Vmtf" and the "lookahead" options result in shorter runtimes, but the "lookahead" options do not scale well; with these options the system is unable to handle bigger datasets like Mushroom and Chess. Therefore, all remaining experiments in this section have been performed with "heuristics=Vmtf". None of the other combinations of parameters gave any substantial improvement in runtime.

Grounding-solving analysis Besides measuring the time needed for the solver to provide an answer, it is also useful to look at the time needed for the individual execution steps. In case of answer set programming, solving a problem consists

Table 3.5: Maximum k-tiling: grounding and solving – avg time per tile (s), and their time ratio

Dataset	Grounding	Solving	Ratio – Solving/Grounding
Nursery	2.173	38.185	18
Voting	0.052	8.350	161
Animals	0.020	2.728	136
Tic-tac-toe	0.124	1.969	16
Flare	0.225	0.575	3

of two main steps: 1) grounding and 2) solving (or, alternatively, searching).

Table 3.5 presents results obtained for the maximum k-tiling problem, i.e., the time needed to compute a single tile, averaged over the first 15 tiles, split into grounding and solving time. For many ASP programs the grounding step is the bottleneck, but the results clearly demonstrate that this is not the case for our ReDF tasks: the ratio between solving and grounding time is generally large. (Due to long runtimes this experiment was not performed for Chess and Mushroom.)

The large ratios can be explained by the fact that we deal with optimization problems, as is usual in data mining. In the presence of an exponential number of answer sets and a binary predicate as the optimization criterion, it is natural to expect the solving part to be the computationally most intensive step.

The results suggest that, if we would like to speed-up ReDF in ASP, we would have to focus on making the general solving process more efficient, possibly by using properties of the ReDF framework. Specialized mining algorithms, e.g., for tiling, demonstrate that faster solving is possible, but translating these efficient mechanisms to an ASP solver, which works completely differently, seems far from trivial.

Alternatives to manual parameter tuning. Claspfolio is a portfolio system for Clasp ("A Portfolio Solver for Answer Set Programming: Preliminary Report"). However, our experiments indicated that none of the precomputed models of Claspfolio applied to the problems considered in this work and the system switches to the default solver. *hClasp* (Gebser et al. 2013) introduces heuristics into the reasoning process of Clasp. Since we consider a wide class of problems, it requires a nontrivial decision procedure to find the right heuristics for a particular task. We regard it as a possible direction for future research.

3.8 Visualizations for an application of purely relational ReDF

Figures 3.12 and 3.13 visualize an example application of purely relational factorization. Figure 3.12 shows the intersection of latent topics depicted in Figure 3.9 with publication data of Floris Geerts, while Figure 3.13 shows the intersection of the same latent topics with the publication data of Pauli Miettinen.

The figures show that we can relate not only particular data points such as venues and co-authors between scientists, but also common (latent) topics. They are identified and visualized in an interpretable manner. The intersection includes the publication records (indicated in red) of two authors (the other author is indicated in pink, if he/she belongs to co-authors of the original decomposition). If for a latent variable, there are a co-author, a university and a venue that belong to both authors, then the latent topic is labeled (in blue) as common.

3.9 Maximum k set coverage analysis of greedy strategy

We here elaborate on the bounds of the greedy and sampling strategies as described in Algorithms 2 and 3, when applied to the problems we consider in this work.

In the greedy case of Algorithm 2, the bound analysis is similar to LTM-k and based on the analysis of the greedy approach to general maximum k-coverage (Feige 1996; Hochbaum and Pathria 1998). Greedy search optimizes its result in each iteration, given the solutions from the previous iterations, until k solutions have been found. Hence, this algorithm follows the general maximum k-coverage strategy and the factor $\frac{greedy}{optimal} \leq 1 - \frac{1}{e}$ applies.

The key assumption highlighted by Hochbaum and Pathria 1998 is that for a set selected in the i^{th} iteration, no data point can be counted twice, i.e., in two different sets, towards the maximal weight summary. That is, if a data point is present in the sets selected in previous iterations, then the total weight should be the same as if we excluded this data point from one of the sets; the solutions should be non-overlapping. This is the case for the problems we consider (tiling, discriminative rule mining, tiling-based matrix diagonalization, etc).

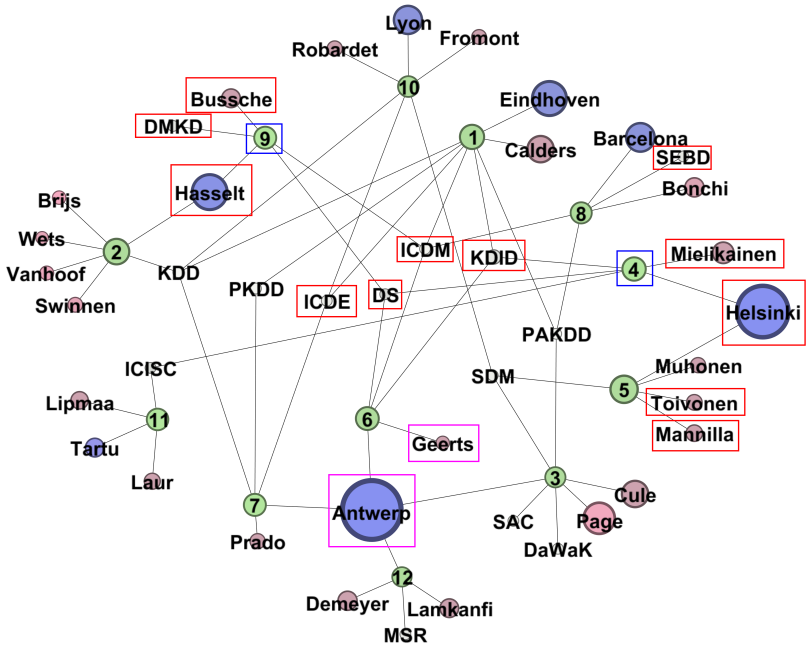


Figure 3.12: An application of purely relational factorization: intersection of publication data (in red) and latent topics (in blue) of Bart Goethals with publications of his co-author Floris Geerts (highlighted in pink) in the graph. A latent topic belongs to an intersection if both have the same co-author, university and venue in the publication records.

Furthermore, the maximum k-set cover bound is very general and in practice, the greedy solution is way closer to the optimum than suggested by the bound. For example, Guns et al. 2013b presented a constraint model of optimal tiling and based on this it was possible to compute the optimal k-tiling for $k = 2$ for the smallest datasets and for $k = 3$ for two datasets (a time-out was set to 6 hours). The average difference between optimal and actual area across all datasets was around 1%, while the bound suggests that it might deviate up to 36.7%.

The sampling strategy of Algorithm 3 follows the approximate greedy scheme: on each iteration it greedily finds an approximation of a 'greedily optimal' solution. As has been shown by Hochbaum and Pathria 1998, the approximate greedy schema with approximation parameter β (i.e., a solution found at i-th

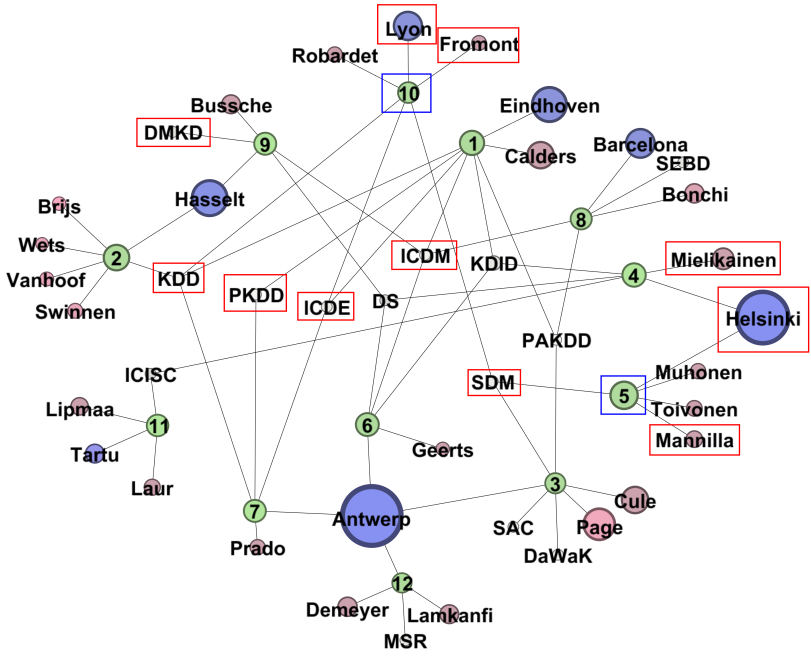


Figure 3.13: An application of purely relational factorization: intersection of publication data (in red) and of latent topics (in blue) of Bart Goethals with publications of Pauli Miettinen (not his co-author). A latent topic belongs to an intersection if both have the same co-author, university and venue in the publication records.

iteration is within a β constant factor of the greedy optimal one at i-th iteration) is within $1 - \frac{1}{e^{\beta}}$ from the optimal solution.

α, the parameter controlling the number of columns to sample, influences both runtime and coverage, i.e., solution quality. It is, however, extremely hard—if not impossible—to analytically derive the dependency between sample size α and approximation factor β, given the generality of the method. Many existing methods used to obtain sampling bounds rely on a) (model) distributions, or its forms, on the parameters and/or values (observed and/or latent) in the data, b) the structure of the constraints, c) the form of the optimization function. In our case, we consider a very generic sampling method and therefore determining an exact analytical form of the dependency between α and β is beyond the scope of this thesis.

It is possible, however, to empirically estimate the parameter β for a particular choice of α. We use the following approach: we estimate the average β for a particular value of α as the mean of n iterations. In the following the upper index denotes the iteration: $\hat{\beta} = \frac{\sum_i \beta^i}{n}$. Then, to estimate β^i for an arbitrary iteration i, we assume that we have computed an optimal solution G^i greedily and an approximation S^i via sampling as $\hat{\beta}^i = \frac{S^i}{G^i}$.

An empirical analysis using this approach can be found in Appendix A.1.4.

3.10 Related work

Our work is related to 1) previous work on generalizing problem definitions and solutions in factorization, 2) existing forms of relational decomposition, and 3) approaches in inductive and abductive logic programming, and 4) the use of declarative languages and solvers for data mining.

3.10.1 General models for pattern mining

Our work can be related to a number of approaches that have generalized some of the tasks addressed in Section 3.4. Lu et al. 2008 used BMF as a basis for defining several data mining tasks and modeled them using integer linear programming. While ibid. also used a general purpose solver, it is restricted to Boolean matrix products involving only two Boolean matrices. In a similar manner Li 2005 defined a *General Model for Clustering Binary Data*, using matrix factorization to model several well-known clustering methods. The framework supports only one possible factorization shape, a lower-level modelling language, and requires complete partitions as well as specialized algorithms for different problems. In our approach, the shape of the factorization is separated from the constraints and optimization criterion.

Biskup et al. 2004; Fan et al. 2012 investigated inverse querying and the problem of solving relational equations $e_1(D) = e_2(D)$ exactly under several assumptions, that could be used to compute exact solutions to a restricted form of ReDF. However, this approach does not seem to allow for approximations and the use of loss functions.

3.10.2 Decomposition of databases, tensors, and real-valued matrices

ReDF is related to several forms of *relational decomposition*, a term that has been heavily overloaded in the literature. Hence, it is imperative that we present an overview and contrast existing paradigms to our own work. Moreover, ReDF is also related to decomposition methods for real-valued matrices.

Relational decomposition in database theory. Ever since the seminal paper by Codd 1970, the decomposition of relations has been an important theme in database research (Date 2006; Koehler 2007). Key properties of this form of relational decomposition are (Elmasri and Navathe 2010): 1) a relational schema together with its constraints, e.g., functional dependencies, is assumed given; 2) decomposition is never based on the data (extension), but only on the schema (intension); 3) decomposition is always lossless, i.e., factorization is always exact for any possible extension, and never an approximation. An interesting exception is *Relational Decomposition via Partial Relations* (Berzal et al. 2002), where one is looking for partially satisfied dependencies in the data and then uses these partial dependencies to derive a normal form. It does take into account data, but only to mine additional schema constraints in the decomposition.

Relational decomposition in tensor calculus. Kim and Candan 2011 extend classical tensor factorization, CP decomposition, to deal with datasets composed of several relations, i.e., CP is generalized to multi-relational datasets. This requires adding relational algebra operations to CP. Key differences are: the data consists of several tables, with a schema to join them at the end; the shape is always the same and a tensor is decomposed into a sum of terms having the same structure; the optimization function is fixed; no user constraints are supported.

Decomposition of real-valued matrices. Let us start with SVD (Singular Value Decomposition) (Golub and Van Loan 1996), the best-known method in this area, which gives an optimal rank-k decomposition of a real-valued matrix A into a composition of three matrices $U\Sigma V^T$, where U and V are orthogonal real-valued matrices and Σ is a diagonal non-negative matrix with singular values of A. One of the key problems with SVD in the context of relational and Boolean factorization is that U and V may contain negative values, which make interpretation in the relational setting problematic. To overcome this issue NNMF (Non-Negative Matrix Factorization) has been introduced (Paatero and Tapper 1994). Still, there are two key issues with the usage of NNMF and SVD for relational and Boolean data.

First of all, for a Boolean matrix A there is no clear relation between its real value

rank, denoted $\text{rank}_{\mathcal{R}}(A)$, and its Boolean rank, denoted $\text{rank}_{\mathcal{B}}(A)$, ($\text{rank}_{\mathcal{R}_{\geq 0}}(A)$ denotes the non-negative rank). We know that the following inequalities hold (Miettinen 2009):

$$\text{rank}_{\mathcal{R}}(A) \leq \text{rank}_{\mathcal{R}_{\geq 0}}(A)$$

$$\text{rank}_{\mathcal{B}}(A) \leq \text{rank}_{\mathcal{R}_{\geq 0}}(A) \tag{3.8}$$

Furthermore, there are examples where $\text{rank}_{\mathcal{R}}(A) = n$ and $\text{rank}_{\mathcal{B}}(A) = \log(n)$ (where A is $n \times n$ matrix) (ibid.), which implies that there are cases where Boolean factorization could be preferable over real-valued matrix factorization, i.e., there are cases where we can obtain a smaller decomposition if we use discrete rather than real-valued methods. Also, if we look at approximate ranks (when the approximation is set within ϵ), Equation 3.8 above does not hold, i.e., there is no clear connection between NNMF and Boolean ranks in the approximate case.

Secondly, existing real-valued matrix factorization methods do not support multiple relations in the decomposition shape and extra constraints in the decomposition, which is at the core of the ReDF method. Furthermore, the constraints used in our method are hard constraints over discrete values. The latter problem has been addressed by Collective Matrix Factorization (Singh and Gordon 2008), which allows to handle multiple relations and optimization criteria. However, at its core the method relies on stochastic optimization over reals, which leads to the problems discussed at the beginning of the section.

Finally, as ReDF is defined over discrete values in the presence of the hard constraints, all the problems described above (rank inequalities, optimization over reals, uninterpretable values, etc) apply to the comparison of ReDF with real-valued matrix factorizations as well.

3.10.3 Relational learning

ReDF is also related to some well known techniques in inductive logic programming and statistical relational learning and even to abductive reasoning.

Several frameworks for *abduction* have been introduced over the years (Denecker and Kakas 2002; Flach and Kakas 2000). In abduction, the goal is to find a (minimal) hypothesis in the form of a set of ground facts that explains an observation. Abductive reasoning uses a rich background theory as well as integrity constraints; it also uses a set of clauses defining the predicate in the observation. The differences with ReDF are that ReDF uses a much simpler shape definition and no real background theory. On the other hand,

abductive reasoning proceeds in a purely logical manner, and typically does neither take into account multiple facts in the observation nor does it use complex optimization functions. There also exist similarities between ReDF and fuzzy abduction (Miyata et al. 1995; Vojtás 1999), but we differ in the core assumptions we make: all rules and constraints in our setting are deterministic, as well as the evidence that needs to be derived. Also, ReDF has the shape constraint, which allows to derive only specific explanations in a form of a factorization.

Meta-interpretive learning (Muggleton et al. 2015a) uses templates together with a kind of abductive reasoning to find a set of rules and facts in a typical inductive logic programming setting. While it can use much richer templates and background theory, it uses neither constraints nor optimisation functions like ReDF does.

Kok and Domingos 2007 introduced a probabilistic framework based on Markov logic together with the EM principle to realize statistical predicate invention. This captures what the authors call *multiple relational clustering* and addresses essentially the same task as the *infinite relational model* of Kemp et al. 2006. Statistical predicate invention shares several ideas with our approach: it employs a kind of query or schema to denote the kind of factorization one wants and also imposes some hard constraints on the possible solutions. On the other hand, its optimization criterion is built-in and based on the maximum likelihood principle, the framework seems restricted to a kind of block modelling approach, essentially clustering the different rows and columns into different blocks, and the approach is inherently probabilistic.

3.10.4 Declarative data mining

The idea of using generic solvers and languages for data mining is not new and has been investigated by, for instance, Guns et al. 2011, 2013c; Métivier et al. 2012, who used various constraint programming languages for modelling and solving itemset mining problems. The use of ASP for frequent item set and graph mining were investigated in Järvisalo 2011 and Paramonov et al. 2015a. Furthermore, the use of integer linear programming is quite popular in data mining and machine learning; e.g., Chang et al. 2008. While the choice of a particular framework for modelling and solving may lead to both different models and performances, it should be possible to use alternative frameworks, such as constraint programming or integer programming, for modelling and solving ReDF problems.

Aftrati et al. 2012 extended the typical structure of the mining problem using three-level graphs that represent a chain of relations in the multi-relational

setting: authors writing papers, and papers being about certain topics. The goal is to find the subgraphs that satisfy particular constraints and optimization criteria. E.g., an author is an authority if the number of topics he has written papers on is maximal. They provide various interesting discovery tasks and solve them using integer programming.

3.11 Conclusions

The key contribution of this chapter is the introduction of the framework of relational data factorization, which was shown to be relevant for modelling, prototyping, and experimentation purposes.

On the modelling side, we have formulated several well-known data mining tasks in terms of ReDF, which allowed us to identify commonalities and differences between these data mining tasks. One advantage of the framework is that small changes in the problem definition typically lead to small changes in the model. Furthermore, ReDF allowed us to model new types of relational data mining problems.

We have not only modeled problems, but also demonstrated that these models can be easily translated into concrete executable ASP encodings. The experiments have shown the feasibility of the approach, especially for prototyping, and especially with the sampling technique. The runtimes were typically not comparable with highly optimized and much more specific implementations that are typically used in data mining. Still they could be run on reasonable datasets of modest size (e.g., Mushroom and Chess have approximately 185 000 and 115 000 non-empty elements respectively).

Directions for future research include investigating the use of alternative solvers (such as constraint or integer programming), the study of heuristics and local search, and the expansion of the range of tasks to which ReDF can be applied. For example, a general ReDF framework is needed to factorize evidence for probabilistic lifted inference, where the shape of the factorization crucially affects the overall performance of the algorithm (Van den Broeck and Darwiche 2013).

Chapter 4

Learning constraints in tabular data

This chapter is based on the paper: "Learning constraints in spreadsheets and tabular data" by Samuel Kolb (*), Sergey Paramonov (*), Tias Guns, Luc De Raedt[1]. My contributions are in the algorithm design, implementation of the ASP prototype, data collection, and writing the results.

Spreadsheets, comma separated value files and other tabular data representations are in wide use today. However, writing, maintaining and identifying good formulas for tabular data and spreadsheets can be time-consuming and error-prone. We investigate the automatic learning of constraints (formulas and relations) in raw tabular data in an unsupervised way. We represent common spreadsheet formulas and relations through predicates and expressions whose arguments must satisfy the inherent properties of the constraint. The challenge is to automatically infer the set of constraints that is present in the data, without labeled examples or user feedback. We propose a two-stage generate and test method where the first stage uses constraint solving techniques to efficiently reduce the number of candidates, based on the predicate signatures. Our approach takes inspiration from inductive logic programming, constraint learning, and constraint satisfaction. We show that we are able to accurately discover constraints in spreadsheets from various sources.

[1] https://link.springer.com/article/10.1007/s10994-017-5640-x

4.1 Introduction

Millions of people across the world use spreadsheets every day. The tabular representation of the data is often intuitive, and the programming of functions in individual cells is quickly learned. However, large and complex sheets, possibly with multiple tables and relations between them, can be hard to handle. Many end-users lack the understanding of the underlying structures and relations in such sheets and the data they contain. This is especially the case when spreadsheets have been exported from other software such as Enterprise Resource Planning (ERP) systems. In this case, often a Comma Separated Values (CSV) format is used meaning that all formulas are lost, including inter-sheet formulas and relations. Even in manually created spreadsheets, it can be challenging to be consistent and correct with formulas across all tables. For example, the influential Reinhart-Rogoff economical paper "Growth in a Time of Debt" (Reinhart and Rogoff 2010) had some of its claims contested (Herndon et al. 2013) after an investigation showed that the used Excel sheets contained mistakes in formulas.

Such issues and mistakes could be overcome through the development of intelligent user-assisting tools that would be able to automatically discover what constraints and formulas hold in a spreadsheet. This is the problem we study in this chapter. We envision that automatic constraint discovery can enable: auto-completion, error checking, formula suggestion, rich import, and data compression. Auto-completion can help users quickly populate spreadsheets by suggesting values for unfilled cells based on constraints that have previously been detected within the spreadsheet. Error checking can identify erroneous values in spreadsheets, for example, while typing, by noticing that previously discovered constraint are violated by newly entered values, or offline by selectively leaving out values. Both of these applications can facilitate the use of spreadsheets even for users who do not explicitly use formulas at all. Furthermore, these applications are able to explain what constraints are responsible for suggesting or rejecting a value. Other potential applications include formula suggestion, which is a variant of auto-completion that does not suggest a *value* but provides a formula which computes the user provided value. This setting can help users in finding the formula to perform a certain task or finding the right syntax for a formula. Rich import denotes importing semi-structured data from another source such as CSV files and extracting the formula's that were used implicitly or explicitly. A similar approach could be used to compress the data: instead of storing all data in a file, one can identify rows or columns generated by formula and store just that formula instead of the values.

To this aim, we investigate whether machine learning and knowledge discovery techniques can be used to learn constraints (formulas and other relations) in

spreadsheet data in an unsupervised way. From a machine learning point of view this is an unconventional problem because the data is in tabular form, but the constraints we wish to learn can involve both rows and columns of the table. Being unsupervised, there is no labeled information either, although for every possible function one can verify whether a certain input satisfies the definition. From a data mining point of view, the data is relational on the one hand, since one can have multiple tables with relationships between them, but also consists of mixed textual and numeric types. More closely related are clausal discovery (De Raedt and Dehaspe 1997; Lallouet et al. 2010), learning CSP constraints (Beldiceanu and Simonis 2012; Bessiere et al. 2005a, 2013) and dependency discovery in databases (Savnik and Flach 2000). But an important difference remains that we want to learn constraints on both columns and rows over integer, floating point as well as textual data. Our inspiration comes from work on program synthesis, in particular Flashfill (Gulwani 2011), where the definition of a string-manipulation function is learned in spreadsheet data from very few examples.

The question that we ask in this chapter is: is it possible to learn the set of constraints present in a spreadsheet, from just tabular spreadsheet data? The main challenge is the number of possible constraints and combinations of rows and columns that need to be considered as input to the constraints. To answer this challenge we propose a general-purpose method and system, named TaCLe (from: Tabular Constraint Learner, pronounced "tackle"), for discovering row-wise and column-wise constraints. Constraints on contiguous rows and columns are most common in spreadsheets, with semantically related items stored in rows or columns and formula's usually ranging over contiguous ranges, as formulas parallel to a table are often *dragged* over its entire span (this suggests a potential way to generalize learned constraints: allow user to drag them along the axis). Our contributions are as follows:

- we define the tabular constraint learning problem, where the goal is to find constraints that range over entire rows or columns in an unsupervised way;
- we propose an effective two-stage generate-and-test method where the first stage reasons only over properties of contiguous blocks of rows/columns, and the second stage continues to investigate individual rows and columns and their content;
- furthermore, in the first stage we use a constraint solver to efficiently enumerate all combinations of maximally contiguous blocks compatible with the arguments of the candidate constraints

- experiments on different publicly available spreadsheets show that the system is able to extract constraints with high precision and recall.

This chapter is organized as follows. Section 4.2 discusses related work. Section 4.3 introduces the relevant concepts and defines the problem formally while Section 4.4 presents our approach. This is evaluated in Section 4.5 after which we show how to realize two of the motivating applications using our system in Section 4.6, and we conclude in Section 4.8.

4.2 Related work

TaCLe combines ideas from several existing approaches and different fields of research.

First, it borrows techniques from logical and relational learning (De Raedt 2008a), as it discovers constraints in multiple relations or tables. Unlike typical logical and relational learning approaches, it focuses on data in spreadsheets and constraints involving both columns and rows in the sheets. Furthermore, it derives a set of simple "atomic" constraints rather than a set of rules that each consist of a conjunction of literals as in clausal discovery (De Raedt and Dehaspe 1997; Lallouet et al. 2010). Nevertheless, several ideas from logical and relational learning proved to be useful, such as the use of a canonical form, a refinement graph, and pruning for redundancy. Tabular constraint learning also connects with the logical generalization of given formulas in a spreadsheet (Isakowitz et al. 1995), where the existing formulas are abstracted by introducing parameters in the place of particular cell references. The key difference here is that we discover constraints/formulas in the data, we do not have or use any prior information on the formulas in a spreadsheet.

Second, there exist a number of algorithms in the constraint programming community that induce constraints from one or more examples and questions. Two well-known approaches include ModelSeeker (Beldiceanu and Simonis 2012) and Quacq (Bessiere et al. 2013). The former starts from a single example in the form of a vector of values and then exhaustively looks for the most specific constraints that hold given a large constraint library. To this aim, it cleverly enumerates different de-vectorized tables; for instance, if the initial vector was of dimension 1×20, it would consider rearrangements of size 2×10, 4×5, 5×4, ... and Modelseeker would then look for both column, row and even diagonal constraints. Key differences with our work are that we use spreadsheets with *multiple* tables and that ModelSeeker's constraints are restricted to integer values and constraints that are typical for the constraint programming community, e.g., aimed at combinatorial optimization and scheduling. Conacq (Bessiere

et al. 2005a) acquires a conjunction of constraints using techniques inspired by Mitchell's version space algorithm to process examples; its successor Quacq uses active learning to query the user. Each of the examples/queries is processed for each constraint, hence an important issue is how fast the algorithm converges to the entire solution set (e.g. for Sudoku). The focus is again on integer variables and typical combinatorial optimisation constraints. In a similar spirit the field of equation discovery ("Data-Driven Discovery of Physical Laws"; Todorovski 2010) focuses, guided by heuristics based on the structure of physical equations and unit measures, on the discovery of physical formulas over numbers and times series. However, they focus specifically on noisy physical data and formulas, e.g., a single variable can take values from a series of measurements to get a regression parameter estimate or they apply specific data transformation such as numerical calculation of time and partial derivatives. In other words, the method works with approximations and the equation structures are heavily biased towards parameter estimation, while our goal is to discover hard constraints that hold over a subset of the rows and columns.

Third, our work borrows from the seminal work of Gulwani (2011) on program synthesis for string manipulation in tabular data, which is incorporated in Microsoft's Excel. In FlashFill, the end-user might work on a table containing the surnames (column A) and names (column B) of scientists. The first row might contain the values A1 = "Curie" and B1 = "Marie", and the user might then enter in cell C1 the value "M.C.". At that point FlashFill generates a program that produces the output "M.C." for the inputs "Curie" and "Marie", and also applies it to the other scientists (rows) in the table. While it might need an extra example to deal with names such as "Bell", "Alexander Graham" and determine whether it should output "A.B." or "A.G.B.", FlashFill learns correct programs from very few examples. Flashfill has been extended to, e.g., FlashExtract (Le and Gulwani 2014) to produce a set of tables starting from an unstructured document (e.g., a txt or HTML file) with a set of examples marked for extraction. However, unlike our approach, Flashfill requires the user to identify explicitly the desired inputs/outputs from the function as well as to provide explicitly some positive and sometimes also negative examples. In contrast, our approach is unsupervised, although it would be interesting to study whether it could be improved through user interaction and through user examples. Furthermore, FlashFill handles only single tables with textual data to obtain a string transformation program, while the examples we consider contain multiple tables with mixed numeric and textual data and multiple constraints in them. An interesting open issue for all of these techniques, with the exception of logical and relational learning, is how to deal with possibly incomplete or noisy data.

Fourth, also related to this line of research is the work on deriving constraints

in databases such as functional and multi-valued dependencies (Mannila and Räihä 1994; Savnik and Flach 2000) although that line of research has focused on more specialized techniques for specific types of constraints. Many of the discovery techniques rely on the database schema (Flach and Savnik 1999), which is not available in our case. Techniques that work directly on the data have been investigated for the discovery of functional dependencies between attributes (Huhtala et al. 1999). The constraints and formulas we wish to learn go beyond that, including arithmetic, conditional arithmetic and *fuzzy* lookups, for example.

Finally, worth mentioning is BayesDB ("BayesDB: A probabilistic programming system for querying the probable implications of data") and Tabular (Gordon et al. 2013), two recent probabilistic modelling languages that have been specifically designed for dealing with relational data in tabular form and that can, as FlashFill, also automatically fill out missing values in a table. However, unlike the other mentioned approaches, it is based on an underlying probabilistic graphical model that performs probabilistic inference rather than identify the "hard" constraints that hold in spreadsheets.

4.3 Formalization

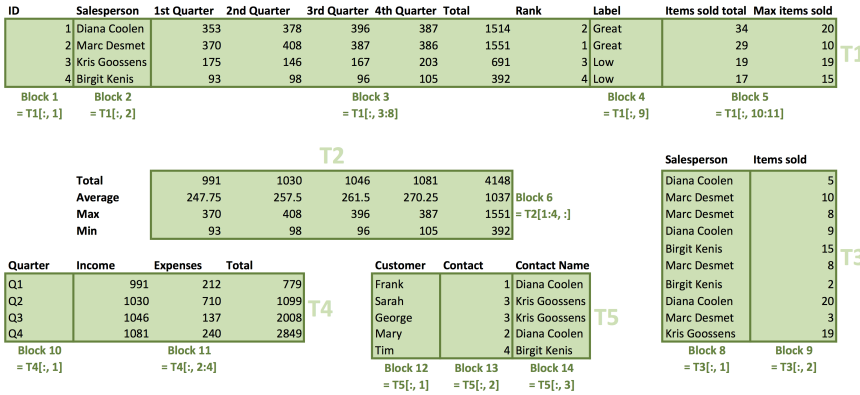
Our goal is to automatically learn the constraints that hold between the rows and/or columns in a spreadsheet. This is applicable not just to data from spreadsheets, but to any data in tabular form.

We first introduce some terminology and the concept of a *constraint template*, after which we define the problem and discuss some additional considerations.

4.3.1 Terminology

Spreadsheets and tabular data may conceptually consist of multiple tables, as in Figure 4.1a. Note that a table can contain one or more headers; however, we wish to reason over entire rows and columns of data, and hence we will consider **headerless tables** only. Headerless tables contain only data entries and are obtained by stripping away all contextual information such as headers.

Formally, a (headerless) table is an $n \times m$ matrix. Each entry is called a *cell*. A cell has a **type**, which can be numeric or textual. We further distinguish numeric types in subtypes: integer and float. We also consider *None* as a special type when a cell is empty; *None* is a subtype of all other types.



(a) Example spreadsheet, black words and numbers only. Green background indicates headerless tables, dark borders indicate maximal type-consistent blocks. Most tables only contain type-consistent columns while table T_2 also contains type-consistent rows. This example combines several Excel sheets based on exercises in the book "MS Excel 2010" (Broeck and Cuypers 2011).

$$SERIES(T_1[:,1]) \qquad T_2[1,:] = SUM_{col}(T_1[:,3{:}7])$$

$$T_1[:,1] = RANK(T_1[:,5])^* \qquad T_2[2,:] = AVERAGE_{col}(T_1[:,3{:}7])$$

$$T_1[:,1] = RANK(T_1[:,6])^* \qquad T_2[3,:] = MAX_{col}(T_1[:,3{:}7])$$

$$T_1[:,1] = RANK(T_1[:,10])^* \qquad T_2[4,:] = MIN_{col}(T_1[:,3{:}7])$$

$$T_1[:,8] = RANK(T_1[:,7]) \qquad T_4[:,2] = SUM_{col}(T_1[:,3{:}6])$$

$$T_1[:,8] = RANK(T_1[:,3])^* \qquad T_4[:,4] = PREV(T_4[:,4]) + T_4[:,2] - T_4[:,3]$$

$$T_1[:,8] = RANK(T_1[:,4])^* \qquad T_5[:,2] = LOOKUP(T_5[:,3], T_1[:,2], T_1[:,1])^*$$

$$T_1[:,7] = SUM_{row}(T_1[:,3{:}6]) \qquad T_5[:,3] = LOOKUP(T_5[:,2], T_1[:,1], T_1[:,2])$$

$$T_1[:,10] = SUMIF(T_3[:,1], T_1[:,2], T_3[:,2]) \qquad T_1[:,11] = MAXIF(T_3[:,1], T_1[:,2], T_3[:,2])$$

(b) Constraints learned for the tables above, except 19 *ALLDIFFERENT*, 2 *PERMUTATION* and 5 *FOREIGNKEY* and 5 *ASCENDING* constraints not shown. Constraints marked with $*$ were not present in the original spreadsheets.

Figure 4.1: Running example

A row or a column is **type-consistent** if all cells in that row or column are of the same base type, i.e., numeric or textual. We will use notation $T[a,:]$ to refer to the a-th row of table T and similarly $T[:,a]$ for the a-th column. For

example, in Figure 4.1a, $T_1[:,1] = [1,2,3,4]$ and $T_3[1,:] = [\text{'Diana Coolen'}, 5]$. $T_3[1,:]$ is not type-consistent while $T_1[:,1]$ is.

The most important concept is that of a **block**.

Definition 4.3.1. A **block** has to satisfy three conditions: 1) it contains entire rows or entire columns of a single headerless table; 2) it is contiguous; and 3) it is type-consistent. The rows or columns have to be contiguous in the original table, meaning that they must visually form a block in the table, and each of the rows or columns has to be of the same type. If a block contains only rows we say it has *row-orientation*, if only columns, *column-orientation*.

In line with this definition, we can use the following notation to refer to blocks: $B = T[a{:}b,:]$ for a row-oriented block containing rows a to b in table T; and similarly $B = T[:,a{:}b]$ for a column-oriented block. We will refer to the *vectors* of a block when we wish to refer to its rows/columns independently of their orientation.

Blocks are used to reason about multiple rows or columns simultaneously. Requiring blocks to be contiguous is quite natural in the spreadsheet setting, as they logically group neighboring rows or columns. This matches the way people commonly construct formulas, by selecting a contiguous ranges in a sheet. For example, constraints such as summation often occur over contiguous blocks.

A block has the following properties:

- *type*: a block is type-consistent, so it has one type;
- *table*: the table that the block belongs to;
- *orientation*: either row-oriented or column-oriented;
- *size*: the number of vectors a block contains;
- *length*: the length of its vectors; as all vectors are from the same table, they always have the same length;
- *rows*: the number of rows in the block; in row-oriented blocks this is equivalent to the size;
- *columns*: the number of columns in the block; in row-oriented blocks this is equivalent to the length.

Example 4.1. Consider the (headerless) table T_1 in Figure 4.1a. Its rows are not type consistent (i.e. they contain both numeric and textual data). However, the table can be partitioned into five column-oriented blocks b_1, b_2, b_3, b_4, b_5, as shown in the figure ($b_1 = T_1[:,1]$, $b_2 = T_1[:,2]$, $b_3 = T_1[:,3{:}8]$, ...).

Definition 4.3.2. Block containment $\sqsubseteq$. A block B' is contained in a block B, $B' \sqsubseteq B$, iff both are valid blocks (contiguous, type consistent) with the same orientation and table, and each of the vectors in B' is also in B. For row-oriented blocks it means that: $B' \sqsubseteq B \Leftrightarrow B = T[a{:}b, :] \wedge B' = T[a'{:}b', :] \wedge a \leq a' \wedge b' \leq b$ and similarly for column-oriented blocks.

We will sometimes write that B' is a *subblock* of B or that B is a *superblock* of B'. An example of block containment is $T_1[:, 3{:}6] \sqsubseteq T_1[:, 3{:}8]$, which contains the sales numbers of all employees for the four quarters.

Table 4.1: Constraint templates implemented in *TaCLe*. Only blocks in **bold** may contain more than one vector; *discrete*(x) is a shortcut for: *textual*(x) $\vee$ *numeric*(x). Templates with $*$ are *structural*, i.e. they are not functional and not available in most spreadsheet software. For templates with † (called *aggregate* and *conditional aggregate*), the table shows variants for *SUM* but *TaCLe* also supports *MAX*, *MIN*, *AVERAGE*, *PRODUCT* and *COUNT*, e.g., $B_2 = MAX_{row}(\mathbf{B}_1)$ or $B_2 = COUNTIF(B_{fk}, B_{pk}, B_1)$.

Syntax	Signature	Definition
$ALLDIFFERENT(B)^*$	$discrete(B)$	For values in B: $i \neq j$: $B[i] \neq B[j]$.
$PERMUTATION(B)^*$	$numeric(B)$ and $ALLDIFFERENT(B)$	The values in B are a permutation of the numbers 1 through $length(B)$.
$ASCENDING(B)^*$	$numeric(B)$	$i < j$: $B[i] \leq B[j]$.
$SERIES(B)$	$integer(B)$ and $PERMUTATION(B)$	$B[1] = 1$ and $B[i] = B[i-1] + 1$.
$FOREIGNKEY(B_{fk}, B_{pk})^*$	B_{fk} and B_{pk} (primary *key*) are *discrete*, from different *table*s, same *type* and $ALLDIFFERENT(B_{pk})$	Every value in B_{fk} also exist in B_{pk}.
$B_2 = LOOKUP(B_{fk}, B_{pk}, B_1)$	B_{fk} and B_{pk} are *discrete*; arguments $\{B_{fk}, B_2\}$ and $\{B_{pk}, B_1\}$ in the same set have the same *length*, *table* and *orientation*; B_1 and B_2 the same type; $FOREIGNKEY(B_{fk}, B_{pk})$.	$B_2[i] = B_1[j]$ where $B_{pk}[j] = B_{fk}[i]$.
$B_2 = LOOKUP_{fuzzy}(B_{fk}, B_{pk}, B_1)$	*Same as lookup*; except $ASCENDING(B_{pk})$ instead of $FOREIGNKEY(B_{fk}, B_{pk})$	$B_2[i] = B_1[j]$ where $B_{pk}[j] \leq B_{fk}[i]$, j maximal.
$B_3 = B_1 \times LOOKUP(B_{fk}, B_{pk}, B_2)$	$\{B_3, B_1, B_{fk}\}$ are *numeric*, $\{B_{pk}, B_2\}$ are *discrete* and within both sets all have the same *length*, *table* and *orientation*; also $FOREIGNKEY(B_{fk}, B_{pk})$.	$B_3[i] = B_1[i] \times LOOKUP(B_{fk}, B_{pk}, B_2)[i]$.
$B_3 = B_1 \times B_2$	B_3, B_1, B_2 are all *numeric* and have the same *length*	$B_3[i] = B_1[i] \times B_2[i]$.
$B_3 = B_1 - B_2$	Arguments $\{B_3, B_1, B_2\}$ are all *numeric* and have the same *length* and *orientation*	$B_3[i] = B_1[i] - B_2[i]$.
$B_3 = \frac{B_1 - B_2}{B_2}$	*Same as* $B_3 = B_1 - B_2$	$B_3[i] = (B_1[i] - B_2[i])/B_2[i]$.
$B_2 = PROJECT(\mathbf{B}_1)$	$\{B_2, \mathbf{B}_1\}$ have the same *length*, *orientation*, *table* and *type*; $\mathbf{B}_1$ contains at least 2 vectors; $B_2 = SUM_{orientation(\mathbf{B}_1)}(\mathbf{B}_1)$	At every position i in 1 through $length(B_2)$ there is exactly one vector v in $\mathbf{B}_1$ such that $v[i]$ is a non-blank value, then $v[i] = B_2[i]$.
$B_2 = RANK(B_1)$	$integer(B_2)$; $numeric(B_1)$; and $length(B_2) = length(B_1)$	The values in B_2 represent the rank of the values in B_1, including ties.
$B_3 = B_1 - B_2 + PREV(B_3)$	$\{B_3, B_1, B_2\}$ are *numeric* and have the same *length* ≥ 2, $PREV()$ denotes previous value	$B_3[i] = B_1[i] - B_2[i] + B_3[i-1]$.
$B_2 = SUM_{row}(\mathbf{B}_1)^\dagger$	B_2 and $\mathbf{B}_1$ are *numeric*; $columns(\mathbf{B}_1) \geq 2$; and $rows(\mathbf{B}_1) = length(B_2)$	$B_2[i] = \sum_{j=1}^{columns(\mathbf{B}_1)} row(i, \mathbf{B}_1)[j]$
$B_2 = SUM_{col}(\mathbf{B}_1)^\dagger$	B_2 and $\mathbf{B}_1$ are *numeric*; $rows(\mathbf{B}_1) \geq 2$; and $columns(\mathbf{B}_1) = length(B_2)$	$B_2[i] = \sum_{j=1}^{rows(\mathbf{B}_1)} column(i, \mathbf{B}_1)[j]$
$B_2 = SUMIF(B_{fk}, B_{pk}, B_1)^\dagger$	B_{fk}, B_{pk} are *discrete*; B_2, B_1 are *numeric*; within the sets $\{B_1, B_{fk}\}$ and $\{B_{pk}, B_2\}$ arguments have the same *length* and *orientation*; B_{fk} and B_1 have the same *table*; B_{fk} and B_{pk} must have different *table*s but the same *type*; and $ALLDIFFERENT(B_{pk})$	$B_2[i] = \sum_{j=1}^{length(B_1)} \begin{cases} B_{val}[j] & \text{if } B_{fk}[j] = B_{pk}[i] \\ 0 & \text{otherwise} \end{cases}$
$B_3 = SUMPRODUCT(B_1, B_2)$	Arguments $\{B_3, B_1, B_2\}$ are *numeric*; $length(B_1) = length(B_2) \geq 2$; and $rows(B_3) = columns(B_r) = 1$	$B_3[1] = \sum_{j=1}^{length(B_1)} B_1[j] \times B_2[j]$.

4.3.2 Constraint templates

The goal is to learn constraints over blocks in the data. The knowledge needed to learn a constraint is expressed through *constraint templates*. A *constraint template* s is a triple $s = (Syntax, Signature, Definition)$:

- *Syntax* specifies the syntactic form of the constraint $s(B_1, ..., B_n)$, that is, the name of the template together with n abstract arguments B_i; each of those arguments will correspond to a block. Thus, a constraint is viewed as a relation or predicate of arity n in first order logic.
  Functions can be represented in a similar way. Any function $B_{n+1} = s(B_1, ..., B_n)$ that has n arguments and computes B_{n+1} can be represented with an $(n+1)$-ary predicate $s'(B_1, ..., B_n, B_{n+1})$.
- *Signature* defines the requirements that the arguments of the predicate must satisfy. This can concern properties of individual blocks as well as relations between properties of arguments, for example, that the corresponding blocks must belong to the same table or have equal length. In terms of logical and relational learning (De Raedt 2008a), the Signature is known as the *bias* of the learner, it specifies a conjunction of the properties that the arguments must have for the constraint to be well-formed. We denote this bias for a template s using the predicate Sig_s.
- *Definition* is the actual definition of the constraint that specifies when the constraint holds. Given an assignment of blocks to its arguments, it can be used to verify whether the constraint is satisfied or not by the actual data present in the blocks. In logical and relational learning this is known as the background knowledge. We introduce the predicate Def_s to capture this background knowledge for a template s.

The constraint templates implemented in *TaCLe* are defined in Table 4.1.

Example 4.2. Consider, for example, the constraint template for the row-based sum:

- Syntax: $B_2 = SUM_{row}(\mathbf{B}_1)$, for arguments B_2 and $\mathbf{B}_1$.
- Signature: B_2 has to be a single vector ($size = 1$) while $\mathbf{B}_1$ can be a block ($size \geq 1$), which can be derived from the use of a normal or **bold** font. Both blocks have to be numeric. This constraint is orientation-specific, so it requires that the number of rows in $\mathbf{B}_1$ equals the length of B_2. Moreover, the bias specifies that the number of columns to sum over is at least 2. This avoids that $\mathbf{B}_1$ contains only a single column.

- Definition: each value in the vector B_2 is obtained by summing over the corresponding row in $\mathbf{B}_1$.

SUM_{row} and SUM_{col} are part of a family of constraints that we refer to as *aggregate constraints* that range over multiple rows or columns and are also available for aggregate operators *MIN*, *MAX*, *AVERAGE*, *PRODUCT* and *COUNT*. *SUMIF* is part of a family of constraints that we refer to as *conditional aggregate constraints*, which aggregate over a vector and only include those cells that satisfy a condition on a related cell in that table, e.g., for the spreadsheet in Figure 4.1a: $T_1[:, 10]$ and the data in T_5 for matching salesperson names. Conditional aggregates are also available for the above aggregate operators, except for product. Note that *SUMPRODUCT* is not considered part of the family of aggregate constraints in this chapter as it is not defined for other aggregate operators. See Section 4.4.4 for a more detailed discussion of the constraints used in *TaCLe*.

It is helpful to see the analogy of constraint templates with first order logic (FOL) and constraint satisfaction. From a FOL perspective, a constraint of the form $B_2 = RANK(B_1)$ can be seen as a predicate $RANK(B_2, B_1)$ where *RANK* is the name of the predicate and its arguments B_2 and B_1 are terms, which can be seen as either uninstantiated variables or as concrete values. This also holds in our setting, where an instantiation of a variable corresponds to a concrete block. For example, for the spreadsheet in Figure 4.1a, when we write $T_1[:, 8] = RANK(T_1[:, 7])$, then the value of B_2 is the 8th vector in T_1: $B_r = T_1[:, 8] = [2, 1, 3, 4]$ and the value of B_1 is the 7th vector: $B_x = T_1[:, 7] = [1514, 1551, 691, 392]$.

With this interpretation, we can speak about the signature and definition of a constraint template being *satisfied*. We say that a signature of a constraint template s with n arguments is satisfied by the blocks $(B_1, ..., B_n)$ if $Sig_s(B_1, ..., B_n)$. Likewise a definition is satisfied if $Def_s(B_1, ..., B_n)$ is. The template is hence satisfied if both the signature and definition are satisfied; in logic programming, we would define the predicate s using a Prolog like clause: $s(B_1, ..., B_n) \leftarrow Sig_s(B_1, ..., B_n) \wedge Def_s(B_1, ..., B_n)$. Under this interpretation, the term constraint and constraint template can be used interchangeably.

Definition 4.3.3. A **valid argument assignment** of a constraint template s is a tuple of blocks $(B_1, ..., B_n)$ such that $s(B_1, ..., B_n)$ is satisfied, that is, both the signature and the definition of the corresponding constraint template are satisfied by the assignment of $(B_1, ..., B_n)$ to the arguments.

4.3.3 Problem definition

The problem of learning constraints from tabular data can be seen as an inverse *constraint satisfaction problem* (CSP). In a CSP one is given a set of constraints over variables that must all be satisfied, and the goal is to find an instantiation of all the variables that satisfies these constraints. In the context of spreadsheets, the variables would represent blocks of cells, and one would be given the actual constraints and functions with the goal of finding the values in the cells. The inverse problem is, given only an instantiation of the cells, to find the constraints that are satisfied in the spreadsheet.

We define the inverse problem, that is the **Tabular Constraint Learning Problem**, as follows:

Definition 4.3.4. *Tabular Constraint Learning.*
Given: a set of instantiated blocks $\mathcal{B}$ over tables $\mathcal{T}$, and a set of *constraint templates* $\mathcal{S}$.
Find: all constraints $s(B'_1, ..., B'_n)$ where $s \in \mathcal{S}$, $\forall i : B'_i \sqsubseteq B_i \in \mathcal{B}$ and $(B'_1, ..., B'_n)$ is a satisfied argument assignment of the template s.

The input is a set of blocks, and in Section 4.4 we will discuss how these can be extracted from a spreadsheet. Figure 4.1b shows the solution to the tabular constraint learning problem when applied to the blocks of Figure 4.1a and constraint templates listed in Table 4.1.

4.3.4 Other considerations

Dependencies

In Table 4.1 one can see that for some constraints we used the predicate of another constraint in its signature, e.g. for *PERMUTATION*. This expresses a dependency of the constraint on that other *base* constraint. This can be interpreted as follows: the signature of the constraint consists of its own signature plus the signature of the base constraint, and its definition of its own definition plus the definition of the base constraint. In FOL, we can see that one constraint entails the other, for example if $PERMUTATION(B)$ holds for a block B, then $ALLDIFFERENT(B)$ also holds.

While dependencies are optional, they simplify the specification of signatures and definitions. Moreover, in Section 4.4 we will see how such dependencies can be used to speed up the search for constraints.

Redundancies

Depending on the application, some constraints in the solution to the tabular constraint learning problem may be considered *redundant*. This is because constraints may be logically equivalent or may be implied by other constraints, e.g., if you know $PERMUTATION(B)$, then $ALLDIFFERENT(B)$ is redundant.

Within the same constraint, there can be equivalent argument assignments if the order of some of the arguments does not matter. For example, for the product constraint, $B_3 = B_1 \times B_2 \equiv B_3 = B_2 \times B_1$, so one can be considered redundant to the other. Two different constraints can also be logically equivalent, due to their nature, e.g., for addition / subtraction and product / division: $B_3 = B_1 \times B_2 \equiv B_1 = B_3/B_2$.

Finally, when the data has rows or columns that contain *exactly* the same values, then any constraint with such a vector in its argument assignment will also have an argument assignment with the other equivalent vectors.

Dealing with redundancy is often application-dependent. Therefore, in Section 4.4 we first explain our generic method for finding all constraints, before describing some optimizations that avoid obvious equivalences (Section 4.4.4). In the experiments we will investigate the impact of redundancy further.

4.4 Approach to tabular constraint learning

The aim of our method is to detect constraints between rows and columns of tabular data. Recall that a valid argument assignment for a constraint is an assignment of a block to each of the arguments of the constraint, such that the signature and definition of the constraint template is satisfied.

Our proposed methodology contains the following steps (Algorithm 4):

The core of our method is step 3. In principle, one could use a generate-and-test approach by generating all possible blocks from the input blocks and testing each combination of blocks for each of the arguments of the constraints. However, each input block of size m has $m * (m + 1)/2$ contiguous subblocks, meaning that a constraint with n arguments would have to check $O(n^{m^2})$ combinations of blocks.

Instead, we divide this step into two parts: in step 3a, we will not reason over individual (sub)blocks and their data, but rather over the properties of the input blocks. Consider table T_1 in Figure 4.1a and the SUM_{row} constraint

Algorithm 4 The *TaCLe* approach

Step 1 Extract headerless tables from tabular data

Step 2 Partition the tables into contiguous, type-consistent blocks (*input blocks*) using the *BlockDetect* tool

Step 3 Generate for each constraint template all valid argument assignments in two steps:

(a) For each constraint template s, generate all assignments $(B_1, \ldots, B_n)$ where each B_i is an input block and the B_i are *compatible* (defined below) with the signature of s.

(b) For each generated assignment $(B_1, \ldots, B_n)$, find all constraints $s(B'_1, \ldots, B'_n)$ such that $B'_i \sqsubseteq B_i$ holds for all i and the signature and definition of s are satisfied.

template. Instead of considering all possible subblocks of T_1 (b_1, b_2, b_3, b_4, b_5), we first reason over the properties of the input blocks. For example, blocks b_2 and b_4 are not numeric so they can be immediately discarded. Input blocks b_1, b_3 and b_5 are valid candidates for the second argument of the constraint, i.e., the functionally defined block. However, since they all have length 4, the first argument, which is the block to sum over, needs at least 4 columns which is only satisfied by b_3. Hence, the assignments generated for SUM_{row} in step 3a would be $(b_1, b_3), (b_3, b_3), (b_5, b_3)$.

In step 3b, we start from an assignment $(B_1, \ldots, B_n)$ with input blocks B_i and generate and test all possible (sub)block assignments to the arguments, using the properties and actual data of the blocks. As we only have to consider the blocks $B'_i \sqsubseteq B_i$ contained in each input blocks, this is typically an easier problem to solve. For SUM_{row} in the above example and assignment (b_5, b_3), each of the vectors of b_5 will be considered as candidate for the left-hand side, and one can enumerate all subblocks of b_3 for the right-hand side, verify the signature (e.g. at least size 4) and test whether for each row the definition is satisfied.

We now describe in more detail how headerless tables are extracted (step 1, Section 4.4.1), how input blocks are generated from them (step 2, Section 4.4.2), and in Section 4.4.3 how the candidate input block assignments are generated (Step 3a) and how the actual assignments are extracted from that (Step 3b). In Section 4.4.4 we describe a number of optimizations designed to improve the effectiveness algorithm.

4.4.1 Step 1: Table extraction

Many spreadsheets contain headers that provide hints at where the table(s) in the sheet are and what the meaning of the rows and columns is. However, detecting tables and headers is a complex problem on its own (Fang et al. 2012). Furthermore, headers may be missing and their use often requires incorporating language-specific vocabulary, e.g. English.

Instead, our algorithm will assume tables are specified by means of their coordinates within a spreadsheet and optionally a fixed orientation of each table. The orientation can be used to indicate that a table should be interpreted as having only rows or only columns.

We developed two simple tools that can be used for the specification of the tables:

Automatic detection (*AutoExtract*) Under the following two assumptions, the table detection task becomes easy enough to be automated; 1) tables are rectangular blocks of data not containing any empty cells; and 2) tables are separated by at least one empty cell from other tables. The sheet is then processed row by row, splitting each row into ranges of non-empty cells and appending them to adjacent ranges in the previous row. Headers can be detected, for example, by checking if the first row or column contains only textual values. If a header row (column) is detected, it is removed from the table and the orientation of the table is fixed to columns (rows), otherwise, we assume there is no header and the orientation is not fixed. Our implementation of this approach is called the *AutoExtract* tool.

Visual selection (*VisualExtract*) The above assumptions do not hold for many tables. However, since the specification of tables is usually easy for humans, we employ a second approach which allows users to indicate tables using a visual tool. Screenshots are available in the accompanying GitHub repository with meta-data[2]. Users select tables excluding the header and optionally specify an orientation. The *VisualExtract* tool then generates the specification of the tables automatically.

[2] https://github.com/SergeyParamonov/TaCLe

4.4.2 Step 2: Block detection with *BlockDetect*

The goal of the block detection step is to partition tables into maximally type-consistent (all-numeric or all-textual) blocks. First, the *BlockDetect* tool preprocesses the spreadsheet data so that currency values and percentual values are transformed into their corresponding numeric (i.e. integer or floating point) representation (e.g. 2.75$ as 2.75 and 85% as 0.85). Then, each table is partitioned into maximal type-consistent blocks.

To find row-blocks, each row is treated as a vector and must be type-consistent; similarly for columns and column-blocks. Then, adjacent vectors that are of the same type are merged to obtain the maximally type-consistent blocks.

4.4.3 Step 3: Constraint learning algorithm

Our learning method assumes that constraint templates and input blocks are given and solves the Tabular Constraint Learning problem by checking for each template: what combination of input blocks can satisfy the signature (*input block assignment*) and which specific *subblock assignments* satisfy both the signature and the definition. The pseudo-code of this approach is shown in Algorithm 5.

This separation of checking the *properties* of input block assignments from checking the *actual data* in the subblock assignment controls the exponential blow-up of combinations to test. Furthermore, we use constraint satisfaction technology in the first step to efficiently enumerate input block assignments that are compatible with the signature[3].

Step 3a: Generating input block assignments

Given a constraint template s and the set of all input blocks $\mathcal{B}$, the goal is to find all combinations of input blocks that are compatible with the constraint signature. An argument assignment $(B_1, ..., B_n)$ is *compatible* with the signature of template s if for each block B_i there exists at least one subblock $B'_i \sqsubseteq B_i$ that satisfies the signature.

[3]Note, that input block assignment is dependent on the signature of the constraint template, therefore, an assignment of input blocks cannot be, generally speaking, re-used for other constraint templates. For this reason, we first pick a constraint template and then generate an input block assignment based on its signature. Also, an input block assignment might have a block used multiple times, subblocks in Algorithm 6, however, cannot overlap, e.g., B1 can be used for input and output of `SUM(X)=Y`, however, in the subblock assignments the first $n-1$ columns would be X and the n-th would be Y, and they cannot overlap.

Algorithm 5 Learn tabular constraints

```
1: Input: S – constraint templates, B – maximal blocks
2: Output: C – learned constraints
3: procedure LearnConstraints(B, S)
4:     C ← ∅
5:     for all s in S do
6:         A ← InputBlockAssignments(s, B)
7:         for all (B_1, ..., B_n) ∈ A do
8:             A' ← Subassignments(s, (B_1, ..., B_n))
9:             for all (B'_1, ..., B'_n) ∈ A' do
10:                C ← C ∪ {c_s(B'_1, ..., B'_n)}
11:    return C
```

Table 4.2: Translation of signature requirements to input block constraints. The following need not be relaxed: $\mathit{length}(B) = x$, $\mathit{orientation}(B) = x$, $\mathit{table}(B) = x$ and $\mathit{size}(B) \geq x$.

Requirement	**Input block constraint**
$\mathit{type}(B) = x$	$\mathit{basetype}(\mathit{type}(B)) = \mathit{basetype}(x)$
$\mathit{size}(B) = x$	$\mathit{size}(B) \geq x$
$\mathit{columns}(B) = x$	*if* $\mathit{orientation}(B) = \mathit{column} : \mathit{columns}(B) \geq x$
	if $\mathit{orientation}(B) = \mathit{row} : \mathit{columns}(B) = x$
$\mathit{rows}(B) = x$	*if* $\mathit{orientation}(B) = \mathit{column} : \mathit{rows}(B) = x$
	if $\mathit{orientation}(B) = \mathit{row} : \mathit{rows}(B) \geq x$

The choice of one argument can influence the possible candidate blocks for the other arguments, for example, if they must have the same length. Instead of writing specialized code to generate and test the input block assignments, we make use of the built-in reasoning mechanisms of constraint satisfaction solvers.

A Constraint Satisfaction Problem (CSP) is a triple (V, D, C) where V is a set of *variables*, D the domain of possible values each variable can take and C the set of constraints over V that must be satisfied. In our case, we define one variable V_i for each argument of a constraint template. Each variable can be assigned to one of the input blocks in $\mathcal{B}$, so the domain of the variables consists of $|\mathcal{B}|$ block identifiers.

To reason over the blocks, we add to the constraint program a set of background facts that contain the properties of each input block, namely its type, table, orientation, size, length and number of rows and columns.

The actual CSP constraints must enforce that an input block assignment is *compatible* with the requirements defined in the signature. Table 4.2 shows how the conversion from requirements of the signature to CSP constraints works: Requirements on the lengths, orientations and tables of blocks can be directly enforced since they are invariant under block containment ($\sqsubseteq$). Typing requirements need to be relaxed to check only for base types, namely numeric and textual. Minimum sizes can be directly enforced, but exact size requirements are relaxed to minimum sizes, since blocks with too many vectors contain subblocks of the required smaller size. Finally, restrictions on the number of rows or columns behave as length or size constraints based on the orientation of the block they are applied to.

The *InputBlockAssignments*(s,$\mathcal{B}$) method will use these conversion rules to construct a CSP and query a solver for all solutions. These solutions correspond to the valid input block assignments for constraint template s.

Example 4.3. Consider the constraint template $B_2 = SUM_{row}(\mathbf{B}_1)$, then the generated CSP will contain two variables V_2, V_1 corresponding to arguments B_2 and $\mathbf{B}_1$. Given maximal blocks $\mathcal{B}$, the domain of these variables are $D(V_1) = D(V_2) = \{1, \ldots, |\mathcal{B}|\}$. Finally, the signature of $B_2 = SUM_{row}(\mathbf{B}_1)$ (see Table 4.1) will be translated into constraints:

$$numeric(V_2) \wedge numeric(V_1) \wedge columns(V_1) \geq 2$$

$$(orientation(V_1) = column) \Rightarrow (rows(V_1) \geq length(V_2))$$

$$(orientation(V_1) = row) \Rightarrow (rows(V_1) = length(V_2))$$

Step 3b: Generating subblock assignments

Given an input block assignment $(B_1, \ldots, B_n)$ from the previous step, the goal is to discover valid *subassignments*, i.e., assignments of subblocks $(B'_1, \ldots, B'_n)$ with for all i, $B'_i \sqsubseteq B_i$, that satisfy both the signature and the definition of template s.

Example 4.4. Consider the sum-over-rows template $B_2 = SUM_{row}(\mathbf{B}_1)$ again; an example input block assignment from Figure 4.1a is $(B_2, \mathbf{B}_1) = (T_1[:, 3{:}8], T_1[:, 3{:}8])$. Note that this assignment does not satisfy the signature yet, because B_2 contains more than one vector. This step aims to generate subassignments $(B'_2 \sqsubseteq B_2, \mathbf{B}_1{}' \sqsubseteq \mathbf{B}_1)$ that do satisfy the signature and test whether they satisfy the definition: $\forall i, B_2[i] = \sum_{j=1}^{length(\mathbf{B}_1)} row(i, \mathbf{B}_1)[j]$. In Figure 4.1a there is exactly one such subassignment: $(T_1[:, 7], T_1[:, 3{:}6])$.

In this step, we could also formulate the problem as a CSP. However, few CSP solvers support floating point numbers, which are prevalent in spreadsheets. Furthermore, in a CSP approach we would have to *ground* out the definition for each of the corresponding elements in the vectors. This is inefficient, as such blocks may not even satisfy the signature or just the first element in the data may already not satisfy the definition.

Algorithm 6 Generate-and-test for *Subassignments*

procedure SUBASSIGNMENTS(s, $(B_1, \ldots, B_n)$)
  $n \leftarrow$ number of arguments of template s
  $A_{sub} \leftarrow \emptyset$
  for all $(B'_1, \ldots, B'_n)$ **where** $\forall i : B'_i \sqsubseteq B_i \wedge disjoint_{j \neq i}(B'_i, B'_j)$ **do**
    if $Sig_s(B'_1, \ldots, B'_n) \wedge Def_s(B'_1, \ldots, B'_n)$ **then**
      $A_{sub} \leftarrow A_{sub} \cup \{(B'_1, \ldots, B'_n)\}$
  return A_{sub}

On the other hand, a simple generate-and-test approach, as illustrated in Algorithm 6, will usually suffice. Given an input block assignment, all disjoint subassignments (in Algorithm 6 this is enforced by the constraint *disjoint*) are generated taking into account the required size, columns or rows. For every disjoint subassignment the exact signature (e.g. subtypes will not have been checked yet) and definition will be tested and all satisfying subassignments are returned.

Based on the assumptions that most subassignments will not satisfy the definition of the constraint template, the implementation of the definition check is geared to *fail-fast* when possible. Specifically, for many constraints the entries of every subblock are fetched one by one, and as soon as one does not hold failure is returned. Often, the first entry will already not satisfy the definition (e.g. if $B_3[0] \neq B_1[0] \times B_2[0]$ then $B_3 = B_1 \times B_2$ does not hold). As a result, many subassignments can be discarded by looking at just one or a few entries, and considering all subassignments that will be checked, the runtime of the checks will typically not be influenced much by the length of the vectors.

4.4.4 Optimizations

This section discusses various design decisions and optimizations aimed at improving the extensibility and effectiveness of our method, as well as reducing the redundancy in the output.

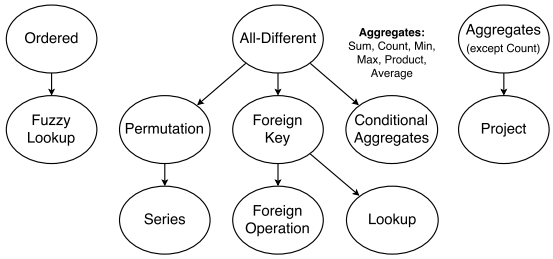


Figure 4.2: Dependency graph of Table 4.1; an arrow from s_1 to s_2 indicates that s_2 depends on s_1: its signature includes s_1.

Template dependencies

As discussed in Section 4.3.4, some constraint templates depend on others by including templates they depend on in their signature (see Table 4.1).

In Inductive Logic Programming, one often exploits implications between constraints to structure the search space (De Raedt 2008a). Our approach uses the dependencies between templates to define a dependency graph. Dependencies are provided to the system as part of the specification of the constraint templates. We assume that signatures do not contain equivalences or loops, and hence the resulting graph is a directed acyclic graph (DAG). Figure 4.2 shows the dependency graph extracted from the signatures in Table 4.1. Constraint templates that have no dependencies are omitted.

Using the dependency graph ($\mathcal{D}$) we can reorder the templates such that a template occurs after any template it depends on. Concretely, in Algorithm 5 line 5 the input templates S are handled following an ordering that agrees with the partial ordering imposed by $\mathcal{D}$ (e.g. *ALLDIFFERENT* before *FOREIGNKEY*). When using this ordering, templates later in the ordering can use the learned constraints of the templates they depend on. To enable this reuse of constraints, the set of learned constraints C is added to the argument of *InputBlockAssignments* on line 6.

The previously learned constraints are used to speed up the *InputBlockAssignments* step. For constraint templates where some arguments are also part of a constraint that it depends on, it is not needed to search for input blocks and subblocks from scratch. Instead, one can start from all and only the actual argument assignments of the base constraint, and only search matching input blocks for the remaining arguments. Therefore, the number of candidate assignments for a constraint template can be reduced by adding a dependence on a lower arity constraint.

Example 4.5. Consider $FOREIGNKEY(B_{fk}, B_{pk})$, which states that every value in B_{fk} also exists in B_{pk}; its signature includes $ALLDIFFERENT(B_{pk})$. There are 18 $ALLDIFFERENT$ constraints to be found in Figure 4.1a, hence, the *InputBlockAssignments* only needs to check which input blocks for B_{fk} are compatible (different table, same type) with these 18 assignments to B_{pk}.

In this case, instead of generating one CSP to find all assignments, a CSP is generated for every known assignment of the depending constraint, which then searches for all assignments completing this partial assignment. The procedure to generate CSPs remains the same otherwise.

Redundancy

We consider two types of redundancy that we aim to eliminate during search. Specifically, these redundancies are taken into account in the implementation for finding subassignments for a constraint. As noted in Section 4.3.4, for some constraint templates there are *symmetries* over the arguments that lead to trivially equivalent solutions, for example, $B'_3 = B'_1 \times B'_2 \Leftrightarrow B'_3 = B'_2 \times B'_1$. Such duplicates are avoided by defining a *canonical* form for these constraints. In practice, we define an arbitrary but fixed block ordering and require those blocks that are interchangeable to adhere to this ordering. Moreover, among the semantically equivalent constraints product ($B'_3 = B'_1 \times B'_2$) and division ($B'_1 = B'_3 / B'_2$) we only added product, as well as only difference ($B_3 = B_1 - B_2$) and not sum ($B_1 = B_3 + B_2$). However, division and sum could easily be added, and they could equivalently be added in post-processing based on the matching product and difference constraints.

A last type of redundancy we chose to avoid during search is that there may be multiple *overlapping* subblocks that satisfy the definition of a constraint template. For example, consider the constraint $B_2 = SUM_{col}(T[:, 1{:}n])$ where $T[:, k{:}n]$ consists of only zeros, then $B_2 = SUM_{col}(T[:, 1{:}j])$ will be true for all $k - 1 \leq j \leq n$. In *TaCLe* we have, therefore, chosen to only consider *maximal* subblocks. This can be seen as a bias of our system.

In some cases a maximal subblock might falsely include irrelevant columns. Consider input block $\mathbf{B}_1 = T[:, 1{:}3] = [[200, 300], [200, 150], [1, 2]]$ and $B_2 = [200, 300]$, then *TaCLe* will find the constraint $B_2 = MAX(T[:, 1{:}3])$. Should the target constraint be $B_2 = MAX(T[:, 1{:}2])$, then *TaCLe* is only able to find exactly that constraint if $T[:, 3]$ is split off into a separate block in the block detection phase or by post-processing the detected constraints.

Constraints

The 33 constraints that are currently supported in our system are shown in Table 4.1. We included most formulas that we encountered in tutorial spreadsheets including the popular *SUM* and *LOOKUP* constraints.[4] We also added four structural constraints (*ALLDIFFERENT*, *PERMUTATION*, *ASCENDING* and *FOREIGNKEY*) so that they can be used in the signature of other constraint templates; they are also popular in constraint satisfaction (Beldiceanu and Simonis 2012). In our current system, the signature for these structural constraints was chosen to be *strict*, i.e. only support those types that are required by depending constraints. Therefore, *ALLDIFFERENT* does not support floating point numbers and *ASCENDING* does not support strings.

For *LOOKUP*, aggregate constraints and *PROJECT* we use optimized implementations to find subassignments instead of generic generate-and-test.

Given an input block assignment, the implementation for $B_2 = LOOKUP(B_{fk}, B_{pk}, B_1)$ first generates candidate vectors for B_{pk} and B_1, then populates a hash table that allows it to quickly look up keys, afterwards it generates candidate vectors for the remaining arguments and, finally, checks which subassignments are valid.

For aggregates (e.g. *SUM*) and *PROJECT*, a custom implementation is used that tries to find maximal subblocks that satisfy the constraint templates. Instead of generating all possible subblocks, given an input block assignment, these implementations start from an input block and repeatedly generate and test smaller subblocks. Therefore, once a match is found, all smaller subblocks do not need to be considered anymore.

Finally, for *FOREIGNKEY* and conditional aggregates, generate-and-test is used, but in combination with caches that store intermediate results and a preparation step to populate some of the caches. For example, for conditional aggregates, input block assignments are analyzed to find vectors with overlapping values. These results are then cached and used to quickly reject subassignments where there is no overlap between B_{pk} and B_{fk}.

Limited precision

One of the challenges of discovering mathematical functional constraints, such as *product*, is that values in spreadsheets often have limited precision and might have been rounded. Testing whether such a constraint holds for a given

[4] https://support.office.com/en-us/article/Excel-functions-by-category-5F91F4E9-7B42-46D2-9BD1-63F26A86C0EB

subassignment, e.g., $B_3 = B_1 \times B_2$, can be done by calculating the *expected* result ($B_1 \times B_2$) first and testing if it corresponds to the actual result (B_3). For the values $B_1 = [1.7], B_2 = [1.8], B_3 = [3.1]$, we would compute first $B_1 \times B_2 = [3.06]$ and conclude $[3.06] \neq [3.1]$ unless the expected result is first rounded. Therefore, *TaCLe* analyzes how long the fractional part of an actual result is and rounds the expected result accordingly.

4.5 Evaluation

In this section we experimentally validate our approach. We first explain the experimental setup and illustrate it on the example in Figure 4.1a, we then investigate the effectiveness of our method on synthetic spreadsheets, after which we evaluate our method on spreadsheets from various sources.

4.5.1 Experimental setup

All spreadsheets used in the experiments are in CSV format and for each spreadsheet the tables are specified. The tables were obtained by running the table extraction tool, *AutoExtract*. Manual intervention of the simple automatic detection was needed only in the case of *None* values (e.g. to merge split-up tables caused by the *None*'s) and in cases where the headers are ambiguous (e.g. headers containing textual and numeric cells). In these cases the tables specifications were generated using the *VisualExtract* tool.

Blocks are then detected automatically using the block detection algorithm, *BlockDetect*. Manual intervention was required for tables that contain *None* values (e.g. if the type of empty cells cannot be inferred).

All spreadsheets also have a set of ground-truth of constraints, we call these the **intended** constraints, that are expected to be (re-)discovered. These were determined by us using either the formulas of the original sheet if present, or inferring them based on, for example, the header names in the sheet. Five spreadsheets contained rows or columns that were entirely identical in which case results can be computed in multiple ways and there are multiple valid constraints. For these spreadsheets, we used the original formulas when present and otherwise inferred the intended constraints based on the headers and the location within the sheet. However, some of the intended constraints are currently outside of the scope of *TaCLe*, in particular nested mathematical or nested logical formulas. We denote by **supported** constraints the subset of intended constraints that the system can find in theory.

In the following experiments, we focus on spreadsheets specifically and hence only include *functional* constraints, constraints that can be expressed as formulas. Therefore, we filter out the structural constraints *ALLDIFFERENT*, *FOREIGNKEY*, *PERMUTATION* and *ASCENDING* in the output. All constraints are stored in their canonical form.

We will use recall and precision to measure how well our tool is performing. Recall is the fraction of intended constraints actually discovered by the system: $\frac{\text{intended discovered}}{\text{all intended}}$, while precision is the fraction of constraints found by the system that are indeed intended: $\frac{\text{intended discovered}}{\text{all discovered}}$.

We include a *no-CSP* baseline to compare our approach to. This baseline follows a similar approach as *TaCLe*, however, it does not prune input blocks in step 3a and generates subassignments directly. Therefore, the signature is also only tested on the subassignment level. While the modified approach still benefits from the reuse of earlier solutions, it does not generate partial assignments for dependent constraint templates.

All experiments were run using Python 3.5.1 on a Macbook Pro, Intel Core i7 2.3 GHz with 16GB RAM. The constraint solver that is used by the input block assignment phase (Step 3a) is python-constraint ().

4.5.2 Results on the running example

For the example presented in Figure 4.1a, *TaCLe* takes a few seconds to find the constraints listed in Figure 4.1b. These include 5 spurious *RANK* constraints, e.g. $T_1[:,1] = RANK(T_1[:,5])$, that are true by accident. Moreover, there is one *LOOKUP* constraint that was not intended in the original spreadsheet (looking up ID based on Salesperson) and which is symmetric to the intended one (Salesperson based on ID).

Hence, for this example we achieve a recall of 100%, that is, all intended constraints are discovered, and a precision of $12/18 = 67\%$. All intended constraints are supported. The recall is perfect, but the precision is rather low. This is mostly due to the spurious RANK constraints. We note that the tables are quite short, which increases the chance of a constraint like *RANK* to be true by chance; we expect less spurious constraints on larger datasets.

We now investigate three main questions that influence the quality of the solutions found by a constraint learning method:

- $\boldsymbol{Q}_1$: the method may fail to find intended constraints if it takes too much time to find them; what are the factors that influence the runtime of the

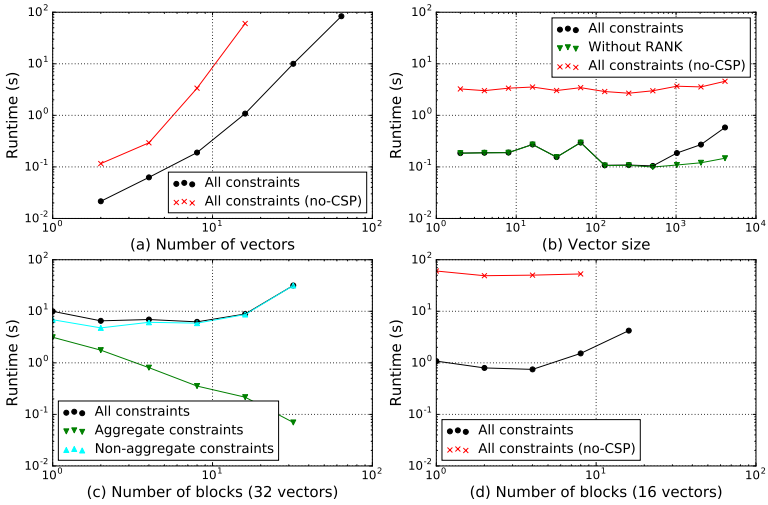


Figure 4.3: Loglog plots of the experiments with synthetic (random) data. In plot *a* the number of vectors is increased; in plot *b* the size of the vectors is increased; and in plots *c* and *d* the number of blocks is increased (the block size is decreased). Missing values for the no-CSP baseline indicate that it timed out.

method most?

- $\boldsymbol{Q}_2$: the method may fail to find intended constraints because it does not support such constraints; how does our method perform on real spreadsheets?
- $\boldsymbol{Q}_3$: the method may find non-intended constraints; how many and what type of non-intended constraints are found on real spreadsheets?

We first investigate $\boldsymbol{Q}_1$ on synthetic spreadsheets and investigate $\boldsymbol{Q}_1$, $\boldsymbol{Q}_2$ and $\boldsymbol{Q}_3$ on a collection of real spreadsheets.

4.5.3 Effectiveness on synthetic data

In this section we investigate the factors that influence the runtime of *TaCLe* the most; we do this by investigating the impact on the running time of 1) the number of vectors; 2) the length of the vectors; and 3) the size of the blocks.

This analysis is accomplished by testing our system on synthetic spreadsheets. The timeout for all experiments was set to 200 seconds.

Spreadsheet generation. The synthetic sheets should trigger the checking of many different constraints, both in the first phase (input block assignment) and in the second phase (subblock assignment), in order to clearly study the effect on the constraint checking. As many constraints require numeric or discrete data, we will use integers in the synthetic sheets as these are both numeric and discrete.

We generate spreadsheets with random data. The result is that there are no intended constraints in the data. This is reasonable, as even in real data most of the constraints will not hold. Additionally, for vectors in random data the *ALLDIFFERENT* constraint is likely to hold, which is a requirement for many other constraint templates.

The synthetic spreadsheets contain two tables, because a number of constraints require that their arguments are picked from different tables. For one table the number of vectors and block size is fixed to two, while for the other table these properties are varied according to the experimental setup. The reported number of vectors and the block size in the experiments refer to those of the second table. The length of the vectors can also be adapted and applies to both tables, since equal length vectors are a required property of many constraints. Unless otherwise mentioned, the 'adaptable' table consists of one block of 8 vectors of length 8, and hence 2 vectors of length 8 in the 'fixed' table.

Effect of number of vectors Figure 4.3 (a) shows the results of our experiments, in which *TaCLe* was run on synthetic spreadsheets containing an increasing number of equally sized vectors. As shown in the figure, the number of vectors in a spreadsheet can have a significant impact on the running time. Moreover, *TaCLe* performs favorably to the no-CSP baseline whose running time not only starts off slower but also grows faster.

The number of vectors is expected to impact the running times, especially when few input block assignments can be discarded in the first phase. This occurs, for example, when most vectors belong to the same block or all vectors have the same type and size as we artificially enforced in this experiment. In this case, for a constraint template that requires n single vector arguments and a spreadsheet with v vectors, up to $\frac{v!}{(v-n)!}$ subassignments are generated and checked by the algorithm.

Typically, we expect the number of generated subassignments to be much smaller in real world spreadsheets and, therefore, the running time to be less sensitive to the number of vectors. This is because in the first phase entire blocks can be disqualified that do not match the signature requirements, e.g. blocks of vectors with different lengths.

Effect of length of vectors To measure the effect that the length of the vectors has on the running time, we ran our system on spreadsheets in which the length of the vectors was gradually increased. Figure 4.3 (b) summarizes our results and shows that the length of the vectors has a small impact on the performance for most templates. The no-CSP baseline performs similarly, however, its runtime is higher by a large margin.

As discussed in Section 4.4.4, our implementation attempts to detect as fast as possible if a subassignment does not satisfy a constraint template. Our experiments satisfy the assumption that the large majority of possible subassignments do not satisfy most constraints. Therefore, the results confirm that our implementation is able to severely limit the impact of the length of vectors.

Most of the increase in running time can be attributed to the *RANK* constraint. The implementation is often not able to quickly detect that subassignments do not satisfy *RANK*, meaning that it will have to analyze many if not all entries in the vector.

Effect of block size To measure the effect of changing the block size, the number of vectors is set to 32 and the vectors are explicitly split into varying number of blocks/tables (up to 32). Figure 4.3 (c) shows the running times of *TaCLe* for various settings. As the blocks become smaller, the total running time first decreases before increasing again. Looking at the running times for aggregate constraints and non-aggregate constraints separately reveals that they are affected differently by the block size. For aggregate constraints (green line) the running time decreases with the block size, while for non-aggregate constraints (red line) the running time first stays more or less constant but then increases more strongly as the size of the blocks becomes small.

Aggregate constraints, e.g. sum or max, have an argument that allows subblocks of varying sizes while non-aggregates only allow single vector arguments. Hence, for aggregate constraints our method will try to find any subblock in the given input block that satisfies the constraint. Since a input block of size m has

$m(m+1)/2$ contiguous subblocks, the search for aggregate constraints becomes faster if there are more but smaller blocks[5].

Non-aggregate constraints have only single vector arguments. By splitting a set of vectors into many blocks, however, there will be a larger number of input blocks given as input to the first phase. In the synthetic spreadsheets, the first phase, which looks at properties of input blocks alone, can disqualify few or no input block assignments, leading to some increase in runtime as more blocks are added.

In order to compare to the baseline, we had to lower the number of vectors from 32 to 16. Figure 4.3 (d) shows that here, too, our system performs better. Both approaches profit from lower running times for aggregates and the no-CSP baseline can offset some of the slowdown as the block size grows small. Note that the synthetic data has been generated to test step 1 only: therefore it is not important if constraints itself are satisfied since this accounts for the runtime in step 2. Our goal here was to measure and determine if there is a significant gap between CSP and no-CSP solutions.

These experiments with synthetic data have given us a better understanding of what effect the properties of vectors and blocks can have on runtime and hence on the ability of the algorithm to finish within reasonable time. We next investigate the behavior of our method on real spreadsheets.

4.5.4 Effectiveness on real spreadsheets

In order to test our approach on real data we assembled a benchmark of spreadsheets from three sources: 1) spreadsheets from an exercise session for teaching Excel at the Brussels Business School based on the book (Broeck and Cuypers 2011), covering the most popular Excel formulas; 2) spreadsheets from online tutorials on Excel formulas; and 3) publicly available *data* spreadsheets such as real-world crime statistics (FBI:UCR) or financial reports (US BEA). The data and its extended description are available in Appendix 4.9 in Table 4.5 and at the same repository as before with the links to all original publicly available spreadsheets used in the experiments.

Table 4.3 gives an overview of the spreadsheets in the different categories and summarizes the results of our experiments. The data is also visualized in Figure 4.4.

[5]Consider, for example, v vectors distributed over b blocks of size v/b, the number of subblock candidates then is $O(b * v^2/b^2) = O(v^2/b)$

Table 4.3: Summary of properties and results per spreadsheet category (time in s)

	Exercises (9)		**Tutorials** (21)		**Data** (4)	
	Overall	Sheet avg	Overall	Sheet avg	Overall	Sheet avg
Tables	19	2.11	48	2.29	4	1
Cells	1231	137	1889	90	2320	580
Intended constraints	34	3.78	52	2.48	6	1.50
Recall (all)	0.88	0.85	0.88	0.87	1.00	1.00
Recall (supported)	1.00	1.00	1.00	1.00	1.00	1.00
Precision	0.94	0.96	0.69	0.90	1.00	1.00
Runtime *TaCLe*	0.91	0.10	0.87	0.04	0.79	0.20
Runtime no-CSP baseline	58.36	6.48	36.84	1.75	19.73	4.93

Q_1: Effectiveness Our first question, $\mathbf{Q}_1$, is about the inability to find intended constraints due to excessive runtime. As Table 4.3 (last row) shows, the runtime of our method on these spreadsheets is always fast. The average runtime across all spreadsheets is $0.08s$. Across categories the average running times fluctuate in the range between $0.04s$ to $0.20s$. While most spreadsheets can be processed very quickly, running times can go up to around $0.61s$ for others. In the previous section we analyzed the effect of various factors that can help explain runtime behavior on data generated to allow many candidate subassignments (e.g. using only vectors of the same type and length). Our benchmark experiments show that our algorithm is able to perform well on real spreadsheets, even for larger numbers of vectors. Moreover, these experiments validate our two-step approach, since *TaCLe* is able to find constraints 20 to 60 times faster than the no-CSP baseline.

Looking at the runtime per constraint template individually, the slowest by far is fuzzy lookup using ±21% of the running time. It is followed by product (±7%), relative difference (±6%), running total (±5%), difference (±5%) and various row-aggregates (±5%).

Dependencies are optional and do not affect precision or recall. However, our method can exploit dependencies in order to find constraints incrementally and increase the systems efficiency. To illustrate the effect this can have, we

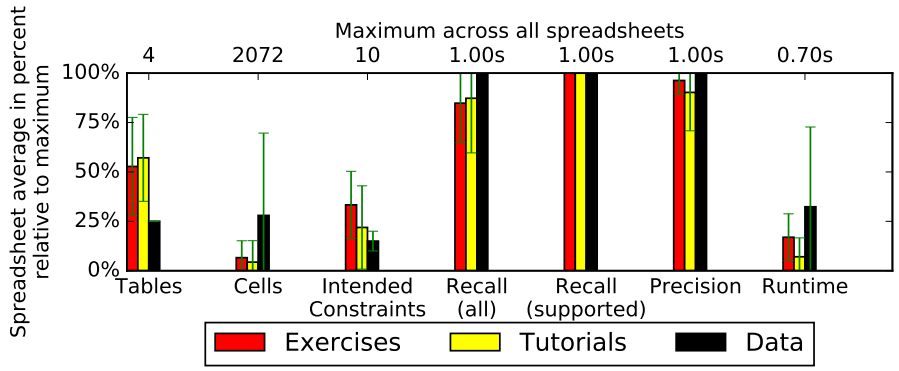


Figure 4.4: This figure visualizes the data from Table 4.3, i.e., the average properties and performance of spreadsheets within every benchmark category. Values are scaled relatively to the *maximum value across all individual spreadsheets*. Green bars indicate standard deviation within the category.

ran the benchmarks without using *ASCENDING* as base constraint for fuzzy lookup. The total running times per category increase by 74% (exercises), 84% (tutorials) and 13% (data). This shows that the use of dependencies can have a strong effect on running times by reducing the number of candidate assignments for depending constraints and sharing the computation and pruning done for one constraint with another.

Q_2: Intended and supported constraints The overall recall achieved by our system, *TaCLe*, is 0.88 (81/92). When looking at the recall per category, *TaCLe* is able to consistently obtain high values for both overall and average recall, as shown in Table 4.3. The intended constraints not discovered correspond to constraints that are not supported by the system, hence the supported recall is 1. Therefore, *TaCLe* is able to find all constraints for supported constraint templates, which shows that our method is able to learn all constraints using the bias chosen, that is, the formalism of signature and definition.

The intended constraints that are not discovered correspond with nested constraints (e.g., $R = \frac{W}{(L/100)^2}$), variants of supported constraints (e.g., series that do not start at 1) or more complex versions of supported constraints (e.g. conditional aggregates whose conditions are constant values).

Q_3: Precision The final question, Q_3, is about the amount and type of **non**-intended constraints found. Across all spreadsheets *TaCLe* achieves a precision

of 0.79 (81/103). While the average precision per spreadsheet is high, over 0.90 in every category, we observe that the overall precision is much higher for the exercises and data categories (0.94 and 1.00) than for the tutorials category (0.69).

Examining the tutorial spreadsheets confirms that there are a few spreadsheets that have a high number of additional constraints, e.g., one spreadsheet computes aggregates on inventory data, but copies the data column for every aggregate. Since every aggregate can be calculated based on any of the equal columns, our method will find all of these constraints. However, for every aggregate only one constraint is considered to be *intended*, the others will be considered as redundant and decrease the precision.

Such redundant or unintended constraints are the result of multiple ways to calculate the same result, e.g., if there is duplicate data as in the example above. One way to increase precision for this case is to post-process the constraints to detect equivalences and implications, and heuristically select one among multiple ways to compute the same vector.

4.6 Applications

In this section, we illustrate how two of the motivating applications, auto-completion and error checking, can build on our method for solving the tabular constraint learning problem.

4.6.1 Auto-completion

Auto-completion can be seen as using knowledge derived from the current data to predict new values before they are written by the user. We can decompose this process into three phases: 1) learn constraints on a snapshot of the spreadsheet; 2) new data (ΔD) is added by the user; and 3) the system uses the learned constraints to predict missing values.

Step 1 corresponds to running *TaCLe* on the snapshot to discover constraints in the spreadsheet. In step 2, the user extends one or more vectors in the spreadsheet with new values. These extended vectors now stretch beyond the original table they are situated in. Therefore, the remaining vectors and blocks in the same table are marked as *incomplete*, indicating they are missing some values required to complete the table. Finally, in step 3, *functional* constraints discovered in step 1, are used to predict missing values. A functional constraint, such as $B_r = SUM_{col}(\mathbf{B}_x)$, calculates a result ($B_r$) based on its *input* ($\mathbf{B}_x$).

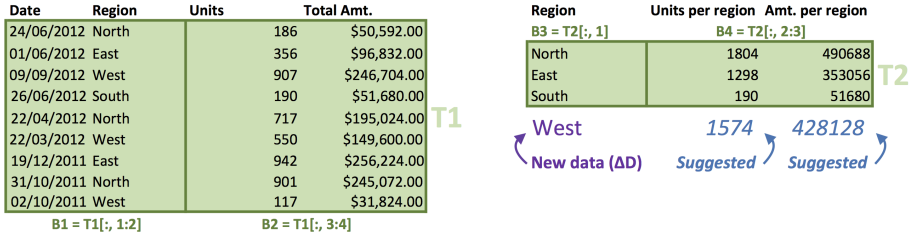


Figure 4.5: Auto-completion works by learning constraint on a snapshot (green tables T_1 and T_2) and combining those constraints with new data ΔD (purple). For example, in this figure a user types a new value *West*. The system can then suggest values (blue) for the other cells in that row.

Hence, a functional constraint can predict missing values if the result block is incomplete and none of the input blocks is incomplete.

Let us illustrate this application on the two-table spreadsheet shown in Figure 4.5. First, we run *TaCLe* on tables T_1 and T_2 (in green) and discover the constraints $T_2[:,2] = SUMIF(T_1[:,2], T_2[:,1], T_1[:,3])$ and $T_2[:,3] = SUMIF(T_1[:,2], T_2[:,1], T_1[:,4])$. Afterwards, vector $T_2[:,1]$ is extended with a new value *West* and vectors $T_2[:,2]$ and $T_2[:,3]$ are marked as incomplete. Finally, the *SUMIF* constraints are used to complete their incomplete result blocks, suggesting $T_2[4,2] = 1547$ and $T_2[4,3] = 428128$.

In an interactive setting, auto-completion can be performed whenever new values are entered, however, to update the learned constraints, *TaCLe* would need to consider only assignments containing modified vectors.

4.6.2 Error detection

We discuss two cases of error detection. The first setting is the *online* detection of errors and is similar to the auto-completion setting. Constraints are learned on a snapshot before new data is added. Functional constraints whose result block was modified are checked on the new data. If such a constraint does not hold anymore, it could indicate errors such as typing errors in the data just entered. For example, filling in $T_2[4,2] = 1572$ in Figure 4.5 would violate the corresponding *SUMIF* constraint and likely indicate an error.

The second setting is *offline* error detection, that is, on a given file. First, *TaCLe* is run to detect all constraints (S) in the given spreadsheet. Then, from this spreadsheet, one can repeatedly remove one or multiple rows or columns (v)

and detect all constraints (S_v) that hold in the modified spreadsheet. Every constraint in S_v that did not occur in S is violated by a value in the rows or columns v that were removed. Therefore, values in the rows or columns v that occur in such a violated constraint can be marked as potential errors to be reviewed by a user. This approach allows constraint discovery to be treated as a black-box, however, it requires a scheme for choosing which row(s) or column(s) are removed.

Alternatively, we propose to extend current constraint definitions to tolerate up to a given number of wrong values and report these. This approach would be more robust, however, it requires definition tests to be able to deal with wrong values.

4.6.3 Spreadsheet domains and use-cases

We can approximately divide the usage of spreadsheet into three use cases

- spreadsheets that are used for data exchange, flat files with little redundancy
- generated reports, with lots of aggregated information and formatting
- spreadsheets that are used by humans, containing lots of formulas

These offer different challenges and opportunities.

Data exchange spreadsheets Often systems communicate using textual CSV files that passed between systems. Sometimes a user might need to look into one of those intermediate results to verify or debug. In this case, some simple formulae might be expected to occur and if not detected by the systems, there is a bug present. Of course, TaCLe offers only limited language to detect most popular flat formulae.

In that case our system can be used to verify that values satisfy expected formulae and its future extensions would offer a richer language of formulae to detect, as well as described error-checking possibilities, see Subsection 4.6.2.

Generated reports spreadsheets Many interesting aggregated statistics are published online (for details, see Section 4.5.1) as flat comma-separated-files. Often, formulae were used to computed some of the columns or rows. Typically original formulae are not present in the report itself, which makes adding or

editing information challenging. That is where TaCLe offers an elegant solution by showing formulae and allowing to recompute the values, for details we refer to Paramonov et al. (2017b).

Human created and used spreadsheets That is the most complicated in terms of formulae used and relationships between tables. TaCle is not intended to handle very complex formulae and can have difficulties with non rectangular shapes of cells in a spreadsheet, which can be used by humans.

Furthermore, in this setting a user would typically be interested only in a part of a spreadsheet and maybe even more often in a particular formula used to compute a value the user is interested in.

Also, spreadsheets created and filled by humans, sometimes manually, might contain typing errors, which makes formula detection extremely hard for logical methods, see Figure 4.6.

To overcome these issues, we describe a potential new system, called Pindakaas, which should be able to learn complex nested formulae in presence of noise and would target specific parts or specific tables of a spreadsheet.

4.7 Future Work

4.7.1 Pindakaas

Here we outline an idea on how to expand TaCLe to work in the supervised setting. We also propose a language extension to handle nested functions and to make the system robust to noise. All features described in this section are not implemented in the current system and should be regarded as a future research direction.

If we recap the key features of FlashFill (Gulwani 2011) and of TaCLe (Kolb et al. 2017), then we would highlight the following:

FlashFill is designed to work with string transformation: one of its main capabilities is to auto-complete strings in a column. Each column must be marked clearly as input/output columns. One or more columns are used as input and only one columns is treated as the output of the string transformation function. The system performs a structured search over programs: internally it searches for a program that performs the matching transformation while minimizing its complexity.

Contrary to FlashFill, **TaCLe:** works with textual and numeric flat Excel formulae. It is not purely textual as FlashFill, but it handles both numbers and string data. Furthermore, it is unsupervised: the constraints we find do not have marked inputs and outputs columns. Even more, it is not necessary columns it might be rows as well or even a mix of them. While Flashfill finds a solution, TaCLe enumerates all constraints matching the data: TaCle does not search for a single solution but find the whole set of formulae (typically, it outputs its condensed representation).

We see that there indeed possibilities for creating a system that would take best from two worlds: supervised optimal constraint learning robust under noise.

Let us outline where we see a possibility to combine them. We expect the following setting in which the system can operate: the system would act in the supervised setting where a block (a list of columns or rows) denoted $\bar{x}$ stands for the input and the block y, stands for the output. To account for noise, we introduce the loss function L between formula $f(\bar{x})$ and the output data y. This gives us an optimization problem of searching for the best fitting formula, i.e., $L(f(\bar{x}), y) \mapsto \min$. However, as we know from machine learning research, this approach is prone to overfitting, therefore we penalize complexity of a formula. Here a functional $\mathcal{C}(f)$ is the measure of complexity. To clarify the idea, we have have depicted a possible visual interface in Figure 4.8.

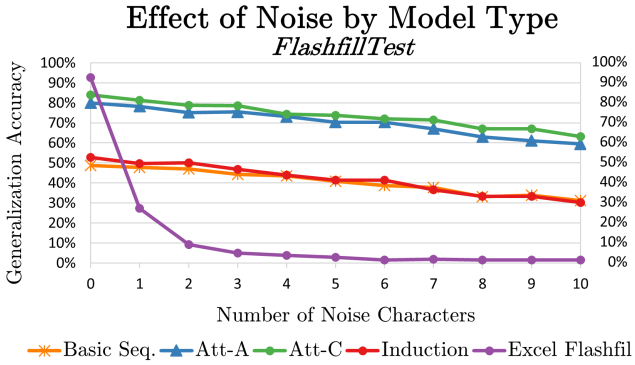


Figure 4.6: We clearly see that FlashFill is highly sensitive to the noise (the figure is due to Devlin et al. (2017))

Noise handling FlashFill is highly sensitive to noise, see Figure 4.6, and TaCle as well. If we want to make TaCLe robust to noise, we also need to adapt a form statistical methodology (i.e., going from a strict logical satisfaction of a

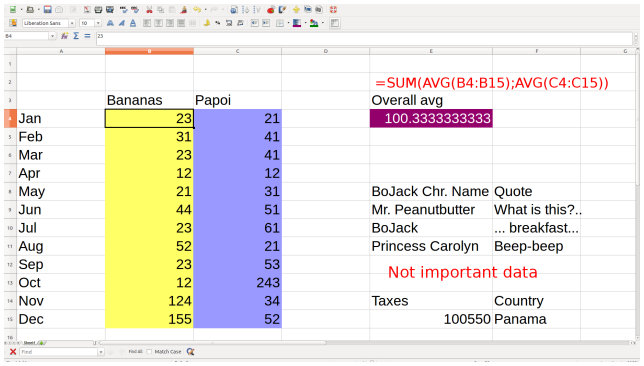


Figure 4.7: An example of a nested spreadsheet formula

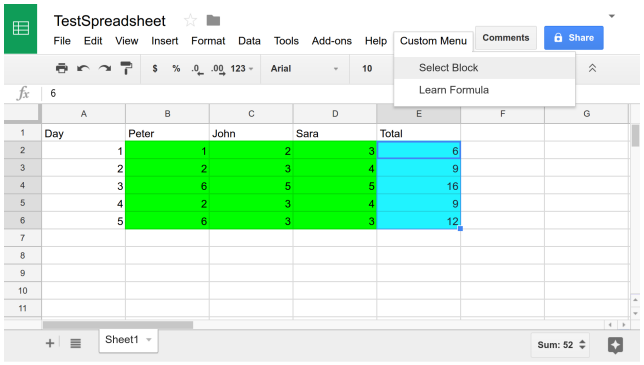


Figure 4.8: Possible visual interface for Pindakaas

formula to a form of decision making based on available statistics about input data): one way to do it is to introduce a loss function that would measure how close is the formula to the target output.

Nested formulae TaCLe is only able to handle flat formulae, if we want to extend it to nested formulae, as for example in Figure 4.7, we need to put preference of one formula over another. This can be done by means of regularizing the formula complexity. This is in fact necessary due potentially infinite amount of formulae satisfing given input-output pairs and once we have established a number of solutions, we would like to output the best one (and probably the most useful one) to a user.

This can be achieved by introducing the following optimization objective:

Property	**FlashFill**	**TaCLe**	**Pindakaas**
Data type	Text	Numeric and textual	Numeric and textual
Learns	String transformation	Set of flat excel formulae	Single best nested formula
Supervised	Marked columns + multiple examples	Unsupervised	Marked columns
Solving	Structured search	Constraint enumeration	Regularized optimization
Satisfaction	Exact	Exact	Best matching
Output	set of values	set of formulae	single formula
Anytime	No	No	Yes

Table 4.4: Feature comparison between Pindakaas, TaCLe and FlashFill

$$f^* = \min_{f \in F} L(f(\bar{x}), y) + \alpha \mathcal{C}(f)$$

where α is the penalty scaling constant, like, e.g., in Support Vector Machines.

How can we define complexity There are multiple ways in how we can define complexity. We take the following approach, for any flat formula f, it is a constant $\mathcal{C}(f) = c_f$ and for any nested formula it is defined as follows

$$\mathcal{C}(f(g(x_1), h(x_2))) = \mathcal{C}(f) + \mathcal{C}(g) + \mathcal{C}(h)$$

where the penalization constant α can be learned by cross-validation or set as a parameter. The constants for flat formulae might come from multiple sources: expert knowledge on what formulae are considered more useful/simple/etc, from statistical observations on what flat formulae are the most occuring and therefore more likely to happen and also these values can be interpreted as scalar parameteres and therefore learned in the same way as any other ML parameter via cross-validation, etc.

To better position Pindakass, let us contrast it with the features of TaCLe and FlashFill in Table 4.4, as we see it borrows features and ideas from both systems.

As a result, we can obtain a system that can on the one hand learn from a rich constraint language and, on the on the other hand, can handle noise and scale to large spreadsheets.

4.7.2 Richer type system, header labels and advanced formatting

Type system In many cases, the type system we propose might be simplistic. For many spreadsheets, symbols like $ or % provide an important indication about the input and output of a formula. For example, a numeric value formatted with a % sign points to a percentage formula. Also, type errors are very common in spreadsheets, especially human entered ones. Understanding that a string really is a date or a number is essential, as they are often mixed in randomly in columns of other types.

This useful information is currently thrown away. Therefore, we imagine that a new system can be extended with additional types such as: percentages, datetime and currencies, fixed precision numbers, integers, natural numbers, etc. This would allow better constraint propagation, fewer false positives, new constraint templates to be defined (that would take into account these types), improve error-detection and allow to generalize and handle rounding errors of fixed precision numbers in a systematic manner.

Header labels This is another useful piece of information that is currently thrown away. Indeed labels such as ”Total“ or ”Sum“ might indicate the usage of a certain type of formulae. These types of hints can be utilized by a future generation of formulae detection systems.

Indeed such a system would be language specific and utilize headers as an extra feature, which would make them closer to classic machine learning algorithms and systems, for details on how these systems might operate in spreadsheet settings see Devlin et al. (2017) and Subsection 4.7.1.

Advanced formatting Current systems relies on the blocks of rectangular shape to determine what can be fed into the system to discover constraints. Often a table might have a row, column or a subblock of missing values, as well as other not perfectly shaped cell structures.

There is a number of ways how we can overcome this problem of having a rectangular structure: one way is to ask user to specify input-output pairs, and then interpreting them in a standard way as Excel does, i.e.,

`SUM(1,2,3,NONE,4)=10` and `NONE + 5 = NONE`; the other way is to enhance the searching mechanics behind the first step of the algorithm.

Neither of them are intrinsically bad or good. The first follows the ideas described in Subsection 4.7.1 and the second would be a natural extension of the implemented TaCLe system, which would support less rigid input structures.

4.8 Conclusions

Our goal is to automatically identify constraints in a spreadsheet. We have presented and evaluated our approach, implemented as the system *TaCLe*, that is able to learn many different constraints in tabular data. The resulting method has high recall, produces a limited number of redundant constraints and is sufficiently efficient for normal and interactive use. Our approach also allows new constraint templates to be easily added by specifying their signature in terms of properties and providing an implementation that finds subassignments satisfying the definition.

The approach is designed to find constraints that hold over entire columns and rows. In future work we plan to extend this to learn arbitrary nested constraints (e.g. $B_4 = (B_1 + B_2)/B_3$), as well as constraints over only a subset of the vectors. This may give rise to more redundant and spurious constraints, which was not problematic up to this point.

Two promising ways to mitigate redundant and spurious constraints are heuristic filtering or post-processing of the constraint set and the integration of our approach in an interactive setting where users can receive and provide feedback. The latter is more similar to an active learning setting.

A final direction is that the current approach assumes consistent (noise-free) data, typically generated by some external system. We do support the detection of limited precision calculations as this is a type of noise generated by exporting data. Also, our approach can be used for error correction in case some noise/errors are assumed. However, in inductive logic programming there has been much work on how to handle constraints over noisy data such as allowing a tolerance level ϵ when a rule or a constraint $\sum_{i=1}^{n} b_i = c$ matches up to ϵ: $|\sum_{i=1}^{n} b_i - c| \leq \epsilon$, while in constraint satisfaction a popular alternative is to consider *soft* constraints that can be violated at a cost, which is a real value, and then the task is to find the top-k best matching constraints. Threshold bounded error tolerance can be supported by adapting constraint definitions to allow such noise. This could extend the approach to new application domains, beyond traditional spreadsheets.

Furthermore, once there is a large established collection of spreadsheets together with the ground truth, we can go from a fixed flat bias of supported constraint templates to learning syntax, levels of nesting, etc. This would follow a general program synthesis and machine learning ideas. To do so realistically, we have to create a certain "golden standard" dataset that would exhibit all necessary patterns, varieties of formatting and represent the general population of spreadsheets in the wild (current implementation of TaCLe relies on statistics provided for the most popular formulae used). The dataset we have created (see Section 4.9 is a first step towards this direction.

4.9 Spreadsheet dataset overview

We collected spreadsheets from three main sources.

- After identifying popular Excel functions, the MS Office web page has an overview of popular functions (see the link in Subsection 4.4.4), we searched for online tutorials about these functions and collected them under the category **Tutorials**. A link to each webpage found and used is provided in the accompanying GitHub repository.
- We have collected the exercises under the category **Exercises** from the introductory Excel book (Broeck and Cuypers 2011) that focused 1) on popular Excel functions and 2) on datatypes supported by our system.
- We have collected under the category **Data** spreadsheets reporting data. More specifically, economic data, crime reporting data and data from runtime experiments. The spreadsheets on economic data and crime reporting are publicly available and originate from the U.S. Bureau of Economic Analysis (BEA), the RWE annual report 2014 and the U.S. FBI Uniform Crime Reporting (UCR) Program. The links are also provided in the accompanying GitHub repository) and data summary spreadsheets.

The overview of constraint distributions per category is presented in Table 4.5.

The file *links.txt*, available in the accompanying GitHub repository, contains the links to all original publicly available spreadsheets used in the experiments.

Table 4.5: Constraint occurrence in the collected dataset in absolute values by category

Constraint	**Exercises**	**Data**	**Tutorials**
average (col)	0	0	2
average (row)	0	1	0
average-if	1	0	0
count (col)	0	0	1
count-if	1	0	0
difference	2	1	0
equal	0	0	10
foreign-product	1	0	0
fuzzy-lookup	2	0	1
lookup	1	0	0
max (col)	0	0	2
max-if	1	0	0
min (col)	0	0	3
min-if	1	0	0
percentual-diff	5	1	0
product	2	0	3
project	1	0	0
rank	1	0	0
series	2	1	0
sum (col)	5	0	10
sum (row)	2	2	3
sum-if	2	0	9
sum-product	0	0	2

Chapter 5

Constraint sketching

This chapter is based on the paper: "Sketched Answer Set Programming" by Sergey Paramonov, Christian Bessiere, Anton Dries, Luc De Raedt, ICTAI 2018[1]. My contributions are in the development of the algorithm, implementation, experiments, and writing the results.

Answer Set Programming (ASP) is a powerful modelling formalism for combinatorial problems. However, writing ASP models can be hard. We propose a novel method, called Sketched Answer Set Programming (SkASP), aimed at facilitating this proposal. In SkASP, the user writes partial ASP programs, in uncertain parts are left open and marked with question marks. In addition, the user provides a number of positive and negative examples of the desired program behaviour. SkASP then synthesises a complete ASP program by rewriting the SkASP program into another ASP program, which can then solved by traditional methods. We evaluate our approach on 21 well known puzzles and combinatorial problems inspired by Karp's 21 NP-complete problems and on publicly available ASP encodings.

5.1 Introduction

Many AI problems can be formulated as constraint satisfaction problems that can be solved by state-of-the-art constraint programming (CP) (Rossi et al. 2006) or answer set programming (ASP) techniques (Lifschitz 2008). Although

[1] https://arxiv.org/abs/1705.07429

these frameworks provide declarative representations that are in principle easy to understand, writing models in such languages is not always easy.

On the other hand, for traditional programming languages, there has been significant attention for techniques that are able to complete (Lezama 2013) or learn a program from examples (Gulwani et al. 2015). The idea of program sketching is to start from a sketched program and some examples to complete the program. A sketched program is essentially a program where some of the tests and constructs are left open because the programmer might not know what exact instruction to use. For instance, when comparing two variables X and Y, the programmer might not know whether to use $X < Y$ or $X \leq Y$ or $X > Y$ and write X ?= Y instead (while also specifying the domain of ?=, that is, which concrete operators are allowed). By providing a few examples of desired program behaviour and a sketch, the target program can then be automatically found. Sketching is thus a form of "lazy" programming as one does not have to fill out all details in the programs; it can also be considered as program synthesis although there are strong syntactic restrictions on the programs that can be derived; and it can be useful for repairing programs once a bug in a program has been detected. Sketching has been used successfully in a number of applications (Jeon et al. 2015; Lezama 2009; Singh et al. 2014) to synthesise imperative programs. It is these capabilities that this chapter brings to the field of ASP.

As a motivating example assume one needs to solve the Peacefully Coexisting Armies of Queens, a version of the n-queens problem with black and white queens, where queens of the same color do not attack each other. One might come up with the following sketched program (where R_w (C_b) stand for the variable representing the row (column) of a white (black) queen):

```
1 :- queen(w,Rw,Cw), queen(b,Rb,Cb), Rw ?= Rb.
2 :- queen(w,Rw,Cw), queen(b,Rb,Cb), Cw ?= Cb.
3 :- queen(w,Rw,Cw), queen(b,Rb,Cb), Rw ?+ Rb ?= Cw ?+ Cb.
```

Listing 5.1: Peacefully Coexisting Armies of Queens

This program might have been inspired by a solution written in the constraint programming language Essence available from the CSP library (Özgür 2015). [2] These sketched rules indicate the general shape of constraints such as no two queens of different colors, i.e., black and white can not be on in the same row, column or diagonal. And at the same time it expresses uncertainty in the operators used in these constraints: should it be plus, minus, multiplication, etc? Intuitively, the sketched ASP specifies constraints on the relationship between

[2] A full sketched and ASP program can be found in https://github.com/SergeyParamonov/sketching

two queens on the rows (first rule), columns (second rule) and diagonals (third rule), but it expresses also uncertainty about the particular operators that should be used between the variables through the built-in alternatives for ?= (which can be instantiated to one of $=, \neq, <, >, \leq, \geq$) and for ?+ (for arithmetic operations). When providing an adequate set of examples to the ASP, the SkASP solver will then produce the correct program.

The key contributions of this chapter are the following: 1) we adapt the notion of sketching for use with Answer Set Programming; 2) we develop an approach (using ASP itself) for computing solutions to a sketched Answer Set Program; 3) we contribute some simple complexity results on sketched ASP; and 4) we investigate the effectiveness and limitations of sketched ASP on a dataset of 21 typical ASP programs.

```
[SKETCH]
reached(Y) :- cycle(a,Y).
reached(Y) :- cycle(X,Y),
              reached(X).
    :- ?p(Y), ?not ?q(Y).
[EXAMPLES]
positive: cycle(a,b).
          cycle(b,c)....
negative: cycle(a,b).
          cycle(b,a).
[SKETCHEDVAR]
?p/1 : node, reached
?q/1 : node, reached
[FACTS]
node(a). node(b). node(c).
[EXAMPLES]
positive: cycle(a,b).
   cycle(b,c). cycle(c,a).
negative: cycle(a,b).
          cycle(b,a).
```

(a) Hamiltonian Cycle (ASP encoding due to Gebser et al. 2012)

```
%%%%% EXAMPLES AND DECISIONS %%%%%%
positive(0).cycle(0,a,b).cycle(0,b,c).cycle(0,c,a).
reified_q_choice(c_node).reified_q_choice(c_reached).
1 {decision_q(X) : reified_q_choice(X)} 1.
1 {decision_not(pos) ; decision_not(neg)} 1.
%%%%% INFERENCE RULES %%%%%%
reified_q(E,c_node,X0) :- node(X0),examples(E).
reified_q(E,c_reached,X0) :- reached(E,X0).
reached(E,Y) :- cycle(E,a,Y),examples(E).
reached(E,Y) :- cycle(E,X,Y),reached(E,X),
                            examples(E).
reified_not(E,pos,Q,Y) :- reified_q(E,Q,Y).
reified_not(E,neg,Q,Y) :- not reified_q(E,Q,Y),
                                  dom1(Y),...
%%%%% POSITIVE/NEGATIVE SKETCHED RULES %%%%%%
:- reified_p(E,P,Y),reified_not(E,Not_D,Q,Y),
                decision_p(P), decision_q(Q),
                decision_not(Not_D), positive(E).
neg_sat(E) :- reified_p(E,P,Y),
          reified_not(E,Not_D,Q,Y), decision_p(P),
decision_q(Q), decision_not(Not_D), negative(E).
:- not neg_sat(E), negative(E).
```

(b) ASP core of rewritten Listing 5.1a (only predicate q shown, domains, facts, etc omitted)

```
[SKETCH] %constraints on squares,rows..
:- cell(X,Y,N), cell(X,Z,M), Y ?= Z,
                             N ?= M.
:- cell(X,Y,N), cell(Z,Y,M), X ?= Z,
                             N ?= M.
insquare(S,N) :- cell(X,Y,N),
                        square(S,X,Y).
:- num(N),squares(S),?not insquare(S,N).
```

(c) Sudoku (core). ASP code from Hölldobler and Schweizer 2014

```
[SKETCH]%constraints on rows and columns
:- cell(X,Y,N), cell(X,Z,M), Y ?= Z,
                             N ?= M.
:- cell(X,Y,N), cell(Z,Y,N), X ?= X,
                             N ?= M.
[EXAMPLES]
positive: cell(1,1,a). cell(1,2,b).
                       cell(1,3,c)...
```

(d) Latin Square (based on Sudoku; core)

Figure 5.1: Collection of sketches and an example of rewriting used in the paper

5.2 ASP and sketching

We extend the syntax of ASP with *sketched* language constructions. Instead of allowing atoms of the form $p(t_1, ..., t_n)$, where p/n is a predicate and the t_i are terms (variables or constants), we now allow to use *sketched atoms* of the form $?q(t_1, ..., t_n)$ where $?q$ is a *sketched predicate variable* with an associated domain d_q containing actual predicates of arity n. The meaning of the sketched atom $?q(t_1, ..., t_n)$ is that it can be replaced by any real atom $p(t_1, ..., t_n)$ provided that $p/n \in d_q$. It reflects the fact that the programmer does not know which p/n from d_q should be used. Sketched atoms can be used in the same places as any other atom.

We also provide some syntactic sugar for some special cases and variants, in particular, we use a *sketched inequality* $X\ ?{=}\ Y$, a *sketched* arithmetic operator $X\ ?{+}\ Y$ (strictly speaking, this is not a sketched predicate but an operator, but we only make this distinction where needed), and *sketched negation* $?not\ p(X)$. The domain of $X\ ?{=}\ Y$ is the set $\{=, \neq, <, >, \geq, \leq, \top\}$, where $\top$ is the atom that is always satisfied by its arguments, the domain of $X\ ?{+}\ Y$ is the set $\{+, -, \times, \boldsymbol{\nabla}\cdot, dist\}$ where $dist(a, b)$ is defined as $|a - b|$, and the domain of $?not$ is $\{\emptyset, not\}$. An example of sketched inequality can be seen in Line 3 of Figure 5.1c, examples of sketched predicates and negation in Line 4 of Figure 5.1a and sketched arithmetic in Line 5 of Figure 5.1.

A sketched variable is a sketched predicate, a sketched negation, a sketched inequality or a sketched arithmetic operator. The set of all sketched variables is referred to as S. Predicate p *directly positively (negatively) depends* on q iff q occurs positively (negatively) in the body of a rule with p in the head or p is a sketched predicate and q is in its domain; p *depends on* q iff q is in the transitive closure of predicates on which p directly depends. A sketch is *stratified* iff there is no negative cyclic dependency. We restrict programs the stratified case. An *example* is a set of ground atoms.

A *preference* is a function from Θ (possible substitutions) to $\mathbb{Z}$. A substitution θ is *preferred* over θ' given preferences f iff for all $s_i \mapsto d_i \in \theta$ and $s_i \mapsto d'_i \in \theta'$ it holds that $f(s_i \mapsto d_i) \geq f(s_i \mapsto d'_i)$ and at least one inequality is strict. First, when $f(\theta)$ is constant, all substitutions are equal and there are no preferences. Because specifying preferences might impose an extra burden on the user, we also provide default preferences for the built-in sketched variables (like inequality, etc), cf. the experimental section.

The Language of Sketched Answer Set Programming (SkASP) supports some of the language features of ASP. The language of SkASP has the following characteristics:

- it allows for a set of rules of the form $a \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m.$;
- atoms (such as a predicate p/n or comparison $\leq$) and operators (such as arithmethic $+, -, \times$, etc) in these rules can be sketched;
- aggregates can be used in the body of the rules as well (stratified; see Extension Section 5.4);
- the SkASP program has to be stratified;
- choice rules are not allowed.

The key idea behind our method is that the SkASP program is rewritten into a normal ASP program (with choice rules, etc.) in order to obtain a solution through the use of an ASP solver. That explains also Theorem 5.2: the language of SkASP stays within the complexity bounds of normal ASP, which makes the rewriting possible (SkASP$\mapsto$ASP).

Let us now formally introduce the problem of SkASP.

Definition 5.2.1 (The Problem of Sketched Answer Set Programming)**.** Given a sketched answer set program P with sketched variables S of domain D and preferences f, and positive and negative sets of examples $\mathbf{E}^+$ and $\mathbf{E}^-$, the *Sketched Answer Set Problem* is to find all substitutions $\theta : S \mapsto D$ preferred by f such that $P\theta \cup \{e\}$ has an answer for all e in $\mathbf{E}^+$ and for no e in $\mathbf{E}^-$. The decision version of SkASP asks whether there exists such a substitution θ.

5.3 Rewriting schema

One might consider a baseline approach that would enumerate all instances of the ASP sketch, and in this way produce one ASP program for each assignment that could then be tested on the examples. This naive grounding and testing approach is, however, infeasible: the number of possible combinations grows exponentially with the number of sketched variables. E.g., for the sketch of the Radio Frequency Problem (Cabon et al. 1999) there are around 10^5 possible assignments to the sketched variables. Multiplied by the number of examples, around a million ASP programs would have to be generated and tested. This is infeasible in practice.

The key idea behind our approach is to rewrite a SkASP problem $(P, S, D, f, \mathbf{E}^+, \mathbf{E}^-)$ into an ASP program such that the original sketching program has a solution iff the ASP program has an answer set. This is achieved by 1) inserting *decision*

variables into the sketched predicates, and 2) introducing *example identifiers* in the predicates.

The original SkASP problem is then turned into an ASP problem on these decision variables and solutions to the ASP problem allow to reconstruct the SkASP substitution.

The rewriting procedure has four major steps: *example expansion*, *substitution generation*, *predicate reification* and *constraint splitting*. (Here we follow the notation on meta-ASP already used in the literature (Eiter and Polleres 2006; Law et al. 2014).)

Example Identifiers To allow the use of multiple examples in the program, every relevant predicate is extended with an extra argument that represents the example identifier. The following steps are used to accommodate this in the program, denoted as $metaE(P, S, \mathbf{E}^+, \mathbf{E}^-)$.

1. Let SP be the set of all predicates that depend on a predicate occurring in one of the examples.
2. Replace each literal $p(t_1, ..., t_n)$ for a predicate $p \in SP$ in the program P by the literal $p(E, t_1, ..., t_n)$, where E is a variable not occurring in the program.
3. Add the guard $examples(E)$ to the body of each rule in P.
4. For each atom $p(t_1, ..., t_n)$ in the i-th example, add the fact $p(i, t_1, ..., t_n)$ to P.
5. For each positive example i, add the fact $positive(i)$ to P, and for each negative one, the fact $negative(i)$.

E.g., the rule in Line 2 of Figure 5.1a becomes Line 9 of Figure 5.1b, and the example in Line 17 is rewritten as in Line 2.

Substitution Generation We now introduce the decision variables, referred as $metaD(S, D)$:

1. For each sketched variable s_i with domain d_i

$$1\ \{decision_s_i(X) : d_i(X)\}\ 1.$$

2. For each value v in d_i, add the fact $d_i(v)$.

This constraint ensures that each answer set has exactly one value from the domain assigned to each sketched variable. This results in a one-to-one mapping between decision atoms and sketching substitution θ. An example can be seen in Lines 4 and 5 of Figure 5.1b.

Predicate Reification We now introduce the reified predicates, referred as *metaR(S, D)*

1. Replace each occurrence of a sketched atom $s(t_1, ..., t_n)$ in a rule of P with the atom $\mathit{reified_s}(D, t_1, \ldots, t_n)$, and add $\mathit{decision_s}(D)$ to the body of the rule.
2. For each sketched variable s and value d_i in its domain, add the following rule to P:

$$\mathit{reified_s}(d_i, X_1, \ldots, X_n) \leftarrow d_i(X_1, \ldots, X_n).$$

   where the first argument is the decision variable for s.

Thus, semantically $\mathit{reified_s}(d_i, X_1, \ldots, X_n)$ is equivalent to $d_i(X_1, \ldots, X_n)$ and $\mathit{decision_s}(d_i)$ indicates that the predicate d_i has been selected for the sketched variable s. Notice that we focused here on the general case of a sketched predicate $?p(\ldots)$. It is straightforward to adapt this for the sketched inequality, negation and arithmetic. Examples of reification can be seen in Lines 7 of Figure 5.1b for the sketched $?q$ of the sketch in Figure 5.1a and in Lines 12, 14 for reified negation.

Integrity Constraint Splitting (referred as *metaC(P)*)

1. Replace each integrity constraint $\leftarrow \mathit{body}$ by:

$$\leftarrow \mathit{body}, \mathit{positive}(E)$$

$$\mathit{negsat}(E) \leftarrow \mathit{body}, \mathit{negative}(E)$$

2. And add the rule to the program:

$$\leftarrow \mathit{negative}(E), \mathit{not}\ \mathit{negsat}(E).$$

This will ensure that all positives and none of the negatives have a solution. For example, the constraint in Line 5 of Figure 5.1a is rewritten into a positive constraint in Lines 16, 17,18 and into a negative in Lines 19, 20, 21.

Another important result is that the preferences do not affect the decision complexity. Proofs can be found in the supplementary materials.

Theorem 5.1 (Sound and Complete Sketched Rewriting)**.** A sketched ASP program $(P, S, D, f, \mathbf{E}^+, \mathbf{E}^-)$ has a satisfying substitution θ iff the meta program $T = \mathit{metaE}(P, S, \mathbf{E}^+, \mathbf{E}^-) \cup \mathit{metaD}(S, D) \cup \mathit{metaR}(S, D) \cup \mathit{metaC}(P)$ has an answer set.

Interestingly, the sketched ASP problem is in the same complexity class as the original ASP program.

Theorem 5.2 (Complexity of Sketched Answer Set Problem)**.** The decision version of propositional SkASP is in NP.

Proof. Follows from the encoding of SkASP into a fragment of ASP which is in NP. □

Dealing with preferences Preferences are, as we shall show in our experiments, useful to restrict the number of solutions. We have implemented preferences using a post-processing approach (which will also allow to apply the schema to other formalisms such as CP or IDP (De Cat et al. 2014)). We first generate the set of all solutions O (without taking into account the preferences), and then post-process O. Basically, we filter out from O any solution that is not preferred (using tests on pairs (o, o') from $O \times O$). The preferences introduce a partial order on the solutions. For example, assume $?p$ ($?q$) can take value p_1 (q_1) with preference of 1 and p_2 (q_2) with 2. If (p_1, q_2) and (p_2, q_1) are the only solutions, they are kept because they are incomparable – $(1, 2)$ is not dominated by $(2, 1)$ (and vice versa). If (p_1, q_1) is also solution, (p_1, q_2) and (p_2, q_1) are removed because they are dominated by (p_1, q_1).

While the number of potential Answer Sets is in general exponential for a sketched ASP, the number of programs actually satisfying the examples is typically rather small (in our experiments, below 10000-20000). If that is not the case, then the problem is under-constrained and it needs more examples. No user would be able to go over a million of proposed programs. Consequently, we implement the preference handling as a post-processing mechanism. This allows us to apply our method to Constraint Programming and Constraint Logic Programming.

5.4 Sketched aggregates

An aggregate $\#agg$ is a function from a set of tuples to an integer. For example, $\#count\{Column, Row : queen(Column, Row)\}$ counts the number of instances *queen(Column,Row)* at the tuple level. Aggregates are often useful for modelling.

However, adding aggregates to non-disjunctive ASP raises the complexity of an AS existence check, unless aggregate dependencies are stratified (Faber et al. 2011). It is possible to add aggregates into our system under the following restrictions: stratified case, aggregates occur in the body in the form $N = \#agg\{\dots\}$, sketched with the keyword `?#` , where $\#agg$ can be *max*, *min*, *count* and *sum*. This immediately allows us to model problems such as Equal Subset Sum (for details, see the repository), where one needs to split a list of values, specified as a binary predicate `val(ID,Value)` into two subsets, such as `subset1(ID)` (and `subset2(ID)` respectively), such that the sum of both subsets is equal. Essentially, we sketch the constraint of the form:

```
:- S1 != S2, S1 = ?#{V,X:val(X,V),subset1(X)}...
```

Formally, each aggregate can be seen as an expression of the form:

$$S = \#agg\{Z_1, \dots, Z_n : cond\overbrace{X_1, \dots, X_k}^{internal}, \overbrace{Y_1, \dots, Y_h}^{external}, \overbrace{Z_1, \dots, Z_n}^{aggregated})\}, external(Y_1, \dots, Y_h)$$

where S is an integer output, and $Y_1, \dots, Y_k$, shortened as $\bar{Y}$ ($\bar{X}$ and $\bar{Z}$ are the same kind of shortening) are bound to other atoms in the rule, to which we refer as *external($\bar{Y}$)* ("external" with respect to the condition in the aggregate; it is simply shortening for a conjunction of atoms, which share variables with the condition in the predicate).

To give an example of $\bar{X}, \bar{Y}, \bar{Z}$ in a simple context: if we were to compute an average salary per department in a company, we might have written a rule of the form:

```
avg_sal(S,Dept) :- S = #avg{Salary:salaries(Name,Salary,Dept)},
                                              department(Dept).
```

Then, $\bar{Z}$ consists of the variable `Salary`, `Dept` is the external variable (with respect to the condition in the aggregate), i.e., $\bar{Y}$ and $\bar{X}$ is composed out of the variable `Name`, since it is neither used in the aggregation, nor in the other atoms outside of the aggregate.

A sketched aggregate ?#, can be reified similarly to the regular sketched atoms, i.e.:

$$reifed(S, sum, \bar{Y}) \leftarrow S = \#sum\{\bar{Z} : cond(\bar{X}, \bar{Y}, \bar{Z})\}, external(\bar{Y}).$$

similarly for other aggregate functions; the same rules, e.g., the example extension, apply to aggregate reification.

With aggregates we can sketch a significantly larger class of problems. Consider the problem from the Functional Pearls Collection: “Finding celebrities problem”

(Bird and Curtis 2006) [3]. Problem statement: "Given a list of people at a party and for each person the list of people they know at the party, we want to find the celebrities at the party. A celebrity is a person that everybody at the party knows but that only knows other celebrities. At least one celebrity is present at the party." The sketch core looks as follows (names are shortened):

```
1 n(N) :- N = ?#{ P : p(P) }.
2 :- c(C), p(C), n(N), S = ?#{P : k(P,C), p(P)}, S < N-1.
3 :- c(C), p(C), not c(P), k(C,P).
```

The last rule is an integrity constraint verifying that no celebrity, `c`, knows a person who is not a celebrity. The first line sketches a rule that should find what aggregation metric on the people (unary predicate person, `p`) should be used in the problem. The sketched rule in the second line makes use of this metric, denoted as `n`, and says that an aggregation should be performed on the binary "knows" predicate, `k`, (indicating that two persons know each other); so the outcome of the sketched aggregation on the connection between people should be compared to an overall metric on all people individually.

5.5 Experimental evaluation

For the experimental evaluation we have created a dataset consisting of 21 classical combinatorial problems among which most are NP-complete. For the problem list and precise sketch specifications used in the experiments, we refer to Table 5.1.

All problems, their code, and implementation details, can be found in the accompanying Github repository: https://github.com/SergeyParamonov/sketching

Dataset of Sketches. The key challenge in evaluating program synthesis techniques such as SkASP is the absence of benchmark datasets (as available in more typical machine learning tasks). At the same time, although there are many example ASP programs available in blogs, books or come with software, these typically employ advanced features (such as incremental grounding, optimization or external sources) which are not supported by SkASP as yet. Therefore we had to design our own dataset in a systematic way (and put it in the public domain). The dataset is based on a systematic concept (the 21 problems by Karp). When we could find encodings for these problem (such as Sudoku in Figure 5.1c from (Hölldobler and Schweizer 2014) and Hamiltonian Cycle in

[3] ASP code: hakank.org/answer_set_programming/finding_celebrities.lp4

Problem	# Sketched	# ?=	# ?+	# ?not	# ?p	# Rules
Graph Clique	3	1	0	0	2	4
3D Matching	3	3	0	0	0	1
Graph Coloring	7	4	0	0	3	2
Domination Set	3	0	0	1	2	5
Exact Cover	7	2	0	1	4	3
Sudoku	5	4	0	1	0	4
B&W Queens	5	3	2	0	0	3
Hitting Set	3	0	0	1	2	2
FAP	3	0	0	1	2	3
Feedback Arc Set	4	0	0	2	2	3
Latin Square	4	4	0	0	0	2
Edge Domination	3	0	0	1	2	5
FAP	5	3	2	0	0	3
Set Packing	4	2	0	0	2	1
Clique Cover	4	3	0	1	0	3
Feedback Set	5	0	0	5	0	3
Edge Coloring	3	3	0	0	0	3
Set Splitting	5	2	0	1	2	3
N Queens	6	4	2	0	0	3
Vertex Cover	3	0	0	1	2	4
Subgraph Isomorphism	5	2	0	1	2	4

Table 5.1: Dataset summary: the number of sketched variables, of rules, of particular types of sketched variables, e.g., "# ?not", indicates how many atoms with the sketched negation are in the program.

Figure 5.1a from (Gebser et al. 2012)) we took these problems, in all other cases we developed a solution according to the standard generate and test development methodology of ASP. Specifically of $\mathbf{Q}_5$ we looked for different encodings in the public domain of ASP's favorite – the N-queens problem (these encoding can tacle even its NP-complete version (Gent et al. 2017)).

After creating all the ASP programs, we turned them into sketched ASP programs by looking for meaningful opportunities to use sketched variables. We introduced sketched variables to replace operators (such as equalities and inequalities), to replace arithmetic (such as plus and minus) and to decide whether to use negated literals or not, and to make abstraction of predicates

for which another predicate existed with the same signature.

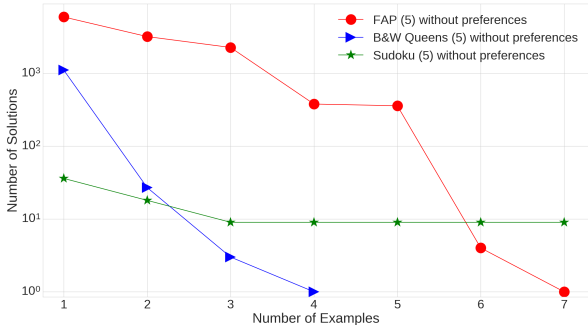
Finally, we had to select the examples in a meaningful way, that is, we selected examples that would be informative (as a user of SkASP would also do). Positive examples were actually selected more or less random, negative examples are meant to violate one or more of the properties of the problem. Furthermore, we also tried to select examples that carry different information (again as a user of SkASP would do). We selected between 4 and 7 examples for each model. Where relevant in the experiments, we sampled the sketched variables (e.g. $\mathbf{Q}_5$) or the examples (e.g. $\mathbf{Q}_3$)

Experimental questions are designed to evaluate how usable is SkASP in practice. Users want in general to provide only a few examples ($\mathbf{Q}_1$-$\mathbf{Q}_3$), to obtain a small number of solutions (ideally only one) ($\mathbf{Q}_1$-$\mathbf{Q}_2$), the examples should be small ($\mathbf{Q}_4$), the solutions should be correct (all), want to know whether and when to use preferences ($\mathbf{Q}_2$), and how robust the technique is to changes in the encoding ($\mathbf{Q}_5$) as it is well known in ASP that small changes in the encoding can have large effects. Finally, they are typically interested in how the learned programs change as the sketches becomes more complex ($\mathbf{Q}_3$). With this in mind, we have designed and investigated the following experimental questions:

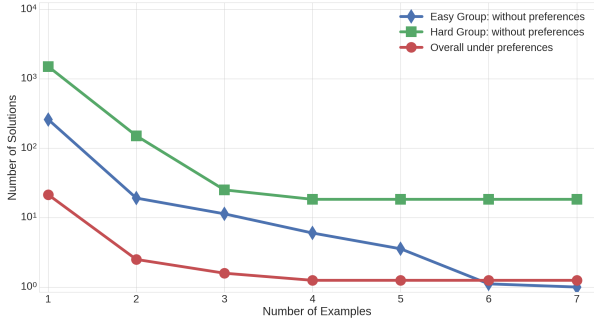
- $\mathbf{Q}_1$: What is the relationship between the number of examples and the number of solutions? How many examples does it take to converge (to a single solution or a small set of equivalent)?
- $\mathbf{Q}_2$: Are preferences useful?
- $\mathbf{Q}_3$: What is the effect of the number of sketched variables on the convergence and correctness of the learned programs?
- $\mathbf{Q}_4$: Do models learned on examples with small parameter values generalize to models with larger parameter values?
- $\mathbf{Q}_5$: What is the effect of encoding variations on the number of solutions and their correctness?

Implementation details and limitations. The SkASP engine is written in Python 3.4 and requires pyasp. All examples have been run on a 64-bit Ubuntu 14.04, tested in Clingo 5.2.0. The current implementation does not support certain language constructs such as choice rules or optimization.

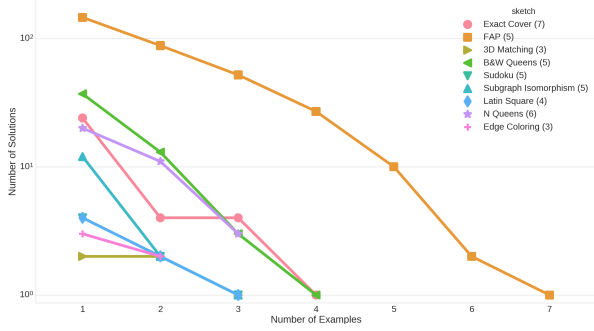
We use the *default preferences* in the experiments for the built-in inequality sketch $X \mathbin{?=} Y$: namely $=$ and $\neq$ have equal maximal preference. A user can



(a) Convergence without preferences (with 5 sketched variables): B&W Queens (Sketch 5.1) and FAP converge, while Sudoku does not

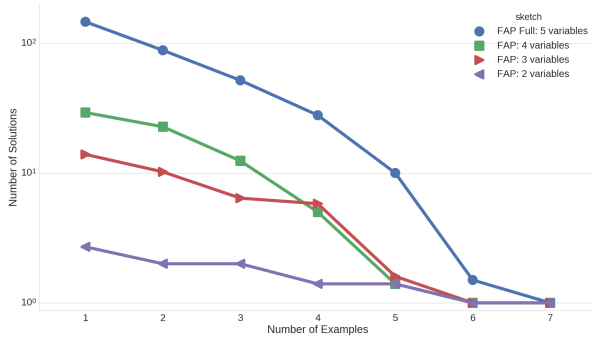


(b) Average number of solutions over the dataset, split into the *easy* group converging without preferences and *hard* not converging

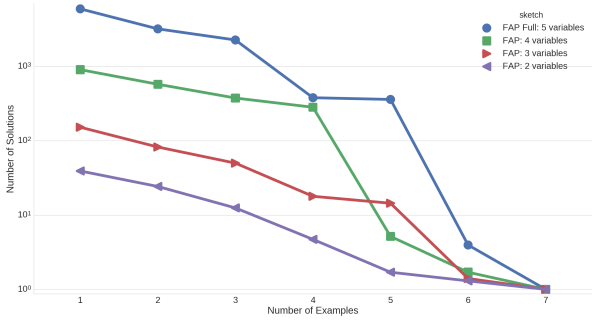


(c) Between 2 and 7 examples are needed to obtain a unique solution (or a small group of equivalent ones) under preferences.

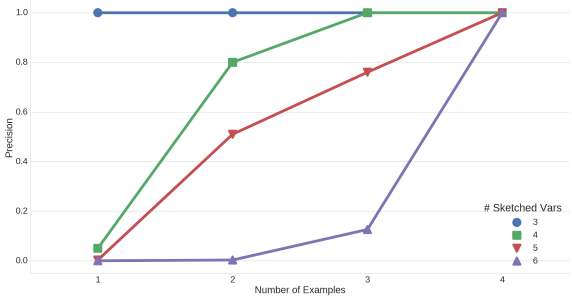
Figure 5.2: Addressing experimental questions $\mathbf{Q}_1$-$\mathbf{Q}_2$. $\mathbf{Q}_1$ and $\mathbf{Q}_2$: convergence without preferences (a), across the dataset (b), for each problem (c). Log-scale.



(a) The effect of the number of sketched variables on the solutions with preferences



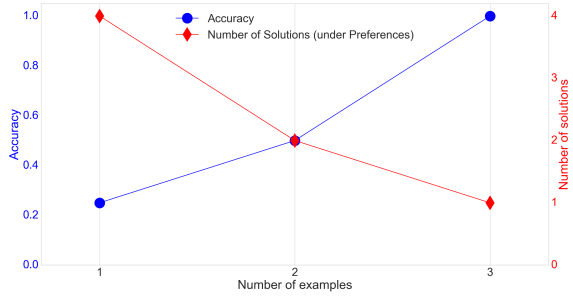
(b) The effect of the number of sketched variables on the solutions without preferences



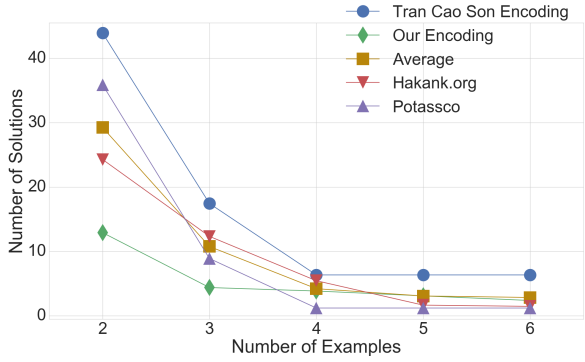
(c) Dependency between the number of sketched variables and precision

Figure 5.3: $\mathbf{Q}_3$: the effect of different number of sketched variable on the number of solutions and precision, FAP (a, b); N-queens (c). Log-scale.

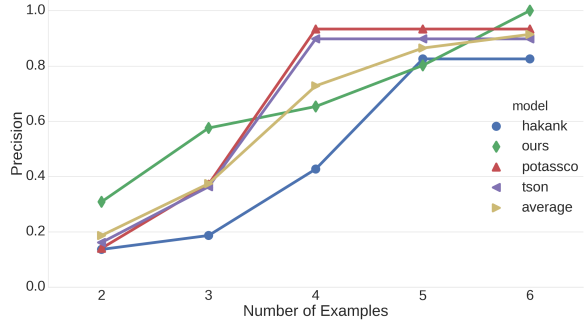
redefine the preferences by giving them weights. Our experiments indicate that for other sketched types no default preferences are needed.



(a) Measuring the generalized accuracy and number of solutions. Latin Square 4×4.



(b) The number of solutions (default preferences) for various N-queen encodings



(c) Correctness for N-queens various encodings (precision)

Figure 5.4: Addressing experimental questions $\mathbf{Q}_4$ and $\mathbf{Q}_5$: generalization accuracy on Latin Square (a), the effect of the encoding variations on the number of solutions (b), on precision (c).

We investigate $\mathbf{Q}_1$ by measuring the impact of the number of examples on the number of solutions of the 21 SkASP problems. An interesting observation is that even if the user wants to solve, say the Latin Square 20×20, she does not need to provide examples of size $20 \cdot 20 = 400$. She can simply provide 3×3 examples and our SkASP problem will learn the generic Latin Square program (see Figure 5.1d).

Figure 5.2a shows how the number of solutions for some of our 21 SkASP problems depends on the number of examples. In some cases, 7 examples are sufficient to converge to a single solution e.g., FAP, B&W Queens.

On some other problems, however, after 7 examples there still remain many solutions (on average 18 for problems that do not converge). Figure 5.2b reports the same information as Figure 5.2a for all 21 problems: the average number of solutions; the average on the 9 that converge within 7 examples, referred to as the *easy problems*; and the average on the 12 that still have several solutions after 7 examples, referred to as the *hard problems*. When SkASP does not converge to a unique solution, this leaves the user with choices, often amongst equivalent ASP programs, which is undesirable.

For problems that do not converge after a few examples, we propose to use preferences, as provided by our SkASP framework. We use the default preference described earlier.

We investigate $\mathbf{Q}_2$ by measuring again the impact of the number of examples on the number of solutions. In Figure 5.2c, we observe that all problems have converged in less than 7 examples (under default preferences). The impact of preferences on the speed of convergence is even more visible on the whole set of problems, as reported in Figure 5.2b. The number of solutions with preferences is smaller, and often much smaller than without preferences, whatever the number of examples provided. With preferences, all our 21 problems are learned with 7 examples.

To analyze the number of solutions in $\mathbf{Q}_3$, we look into the convergence of FAP by varying the number of sketched variables. The original sketched program of FAP contains 5 sketched variables. We vary it from 2 to 5 by turning 3, 2, 1, or 0 sketched variables into their actual value (chosen randomly and repeated over multiple runs). As expected, in Figure 5.3a, we observe that the more there are sketched variables in the SkASP, the slower the number of solutions decreases. Furthermore, the number of sketched variables has a greater impact on the convergence without preferences, as we see in Figure 5.3b. After 3-5 examples under preferences we have fewer than 10 solutions, while without preferences there are still dozens or hundreds of solutions.

To analyze *correctness* in $\mathbf{Q}_3$, we need first to define it. Informally, we mean

that the program classifies arbitrary examples correctly, i.e., positive as positive, etc. A typical metric to measure this is *accuracy*. However, there are no well defined arbitrary positive and negative examples for the most problems: what is an arbitrary random example for Feedback Arc Cover? Problems like Sudoku and N-queens do have standard examples because they are parameterized with a single parameter, which has a default value. Furthermore, for the standard 8-queens we know all solutions analytically, i.e., 92 combinations. Another issue is that the negative and positive classes are unbalanced. The usual way to address this issue is to use *precision*, i.e., $\frac{\text{True Positive}}{\text{True+False Positives}}$. (Recall is typically one because the incorrect programs produce way too many solutions that include the correct ones.) In Figure 5.3c, we see that in all cases we were able to reach the correct solution (here the locations of sketched variables were fixed as specified in the dataset); while increasing the number of sketched variables generally decreases the precision.

To investigate $\mathbf{Q}_4$, we have used the Latin Square from Listing 5.1d. We have used examples for Latin Square 3×3, and verified its correctness on Latin Square 4×4 (which can be checked analytically because all solutions are known). We have discovered, that there is an inverse dependency between number of solutions and accuracy, see Figure 5.4a. This happens because there are typically very few useful or "intended" programs while there are lot of incorrect ones.

To investigate $\mathbf{Q}_5$, we have focused on the N-queens problem and collected several encodings from multiple sources: Potascco, Hakank.org, an ASP course by Tran Cao Son[4] and our encoding. Whereas all the encodings model the same problem they show significant variations in expressing constraints. To reduce the bias in how the sketched variables are introduced and systematically measure the parameters, we pick sketched variables randomly (inequalities and arithmetic) and use the same examples from our dataset (randomly picking the correct amount) for all models.

In Figure 5.4b, we observe that while there is a certain variation in the number of solutions, they demonstrate similar behavior. For each encoding we have introduced 5 sketched variables and measured the number of solutions and the model's precision. In Figure 5.4c we see that there is indeed a slight variation in precision, with 3 out of 4 clearly reaching above 90% precision, one reaching 100% and one getting 82%. Thus, despite variations in encoding, they generaly behave similarly on the key metrics. The results have been averaged over 100 runs.

Overall, we observe that only few examples are needed to converge to a unique

[4] www.hakank.org/answer_set_programming/nqueens.lp
www.cs.uni-potsdam.de/~torsten/Lehre/ASP/Folien/asp-handout.pdf
www.cs.nmsu.edu/~tson/tutorials/asp-tutorial.pdf

or a small group of equivalent solutions. An example where such equivalent solutions are found is the edge coloring problem, where two equivalent (for undirected graphs) constraints are found:

$$\leftarrow color(X, Y_1, C), color(X, Y_2, C),\ Y_1 \neq Y_2.$$

$$\leftarrow color(X_1, Y, C), color(X_2, Y, C),\ X_1 \neq X_2.$$

For this problem these two constraints are equivalent and cannot be differentiated by any valid example.

An interesting observation we made on these experiments is that the hardness (e.g., in terms of runtime) of searching for a solution of a problem is not directly connected to the hardness of learning the constraints of this problem. This can be explained by the incomparability of the search spaces. SkASP searches through the search space of sketched variables, which is usually much smaller than the search space of the set of decision variables of the problem to learn.

5.6 Related work

The problem of sketched ASP is related to a number of topics. First, the idea of sketching originates from the area of programming languages, where it relates to so called self-completing programs (Lezama 2013), typically in C (Lezama 2009) and in Java (Jeon et al. 2015), where an imperative program has a question mark instead of a constant and a programmer provides a number of examples to find the right substitution for it. While sketching has been used in imperative programming languages, it has – to the best of the authors' knowledge – never been applied to ASP and constraint programming. What is also new is that the sketched ASP is solved using a standard ASP solver, i.e., ASP itself.

Second, there is a connection to the field of inductive (logic) programming (ILP) (De Raedt 2008b; Gulwani et al. 2015; Muggleton and De Raedt 1994b) as one induces programs from examples of their intended behaviour. An example of such approach is the work on meta-interpretive learning (Muggleton et al. 2014, 2015b) where a Prolog program is learned from a set of higher-order rules and examples. The higher-order rule acts as a kind of template that can be used to complete the program. However, meta-interpretive learning differs from SkASP in that it induces full programs and pursues as other ILP methods a search- and trace-based approach guided by generality, whereas SkASP using a constraint solver (i.e., ASP itself) directly. Furthermore, the target is different in that ASPs are learned, which include constraints. There is also a connection to meta-interpretation in ASP (Eiter and Polleres 2006), such as rule and decision

Feature	SkASP	ILASP
Problem type	Enumeration	Optimization
Input	Sketch, examples, preferences	Modes, examples
Intermediate	ASP (+post-processing)	Sets of pos. & violating solutions
Output	A set of preferred solutions	A minimal size program
Backg. knowl.	Yes	Yes
Single shot	Yes	No
Local uncertainty	Yes	No
Supports choices	No	Yes
Meta-ASP	Yes	Yes

Table 5.2: Key feature comparison between SkASP and ILASP

materialization. The purpose is, however, different: they aim at synthesizing a program of higher complexity (Σ_2^P) given programs of lower complexity (*NP* and *Co-NP*).

There are also interesting works in the intersection of ILP, program synthesis and ASP (Law et al. 2014, 2016). The ILASP system (Law et al. 2015) learns an ASP program from examples, and a set of modes, while minimizing a metric, typically the number of atoms. While there are similarities in the definitions and used ASP meta-programming techniques, the program, learned completely from scratch, is not necessarily the best program from the user's point of view and may limit the possibility to localize the uncertainty based on the user's knowledge of the problem. Indeed, if all sketched predicates are added in the modes with corresponding background knowledge, then the set of hypotheses of sketched ASP is a subset of ILASP. However, if we specify a sketched constraint `:- p(X),q(Y),X?=Y` with the negative example `{p(1),q(2)}` as modes for ILASP (ibid.), ILASP would learn a program like `:- p(X)`, since that is a minimal program in the number of atoms but that is clearly not the program intended by the sketch. Furthermore, we aim at computing the set of preferred programs instead of a single solution. For a more detailed feature comparison, we refer to Table 5.2. SkASP is also related to XHAIL (Ray 2009) an ILP abduction system, which, however, does not support constraints that are at the core of our system.

Third, there is also work on constraint learning, where the systems such as CONACQ (Bessiere et al. 2017a, 2005b) and QUACQ (Bessiere et al. 2013) learn a set of propositional constraints, and ModelSeeker (Beldiceanu and Simonis

2012) learns global constraints governing a particular set of examples. The subject has also been investigated in ILP setting (Lallouet et al. 2010). However, the idea in all these approaches is to learn the complete specification of CSPs from scratch. Our setting is probably more realistic from a user perspective as it allows to use the knowledge that the user no doubt possesses about the underlying problem, and also requires much less examples. On the other hand, SkASP also makes, as other sketching approaches, the strong assumption that the intended target program is an instance of the sketched one. This may not always be true, for instance, when rules are missing in the program. This is an interesting issue for further research.

Fourth, our approach is related to debugging of ASP (Gebser et al. 2008; Oetsch et al. 2011). Unlike SkASP such debuggers can be used to locate bugs, but typically do not provide help in fixing them. On the other hand, once a bug is identified, SkASP could be used to repair it by introducing a sketch and a number of examples. [5] The approach of (Li et al. 2015) is based on classical ILP techniques of generalization and specification and does not provide the freedom to indicate uncertain parts of the program.

5.7 Discussion and conclusions

Our contribution is four-fold: we have introduced the problem of sketched ASP; we have provided a rewriting schema for SkASP; we have created a dataset of sketches and we have evaluated our approach empirically demonstrating its efficiency and effectiveness.

User interaction is an interesting direction for future work, namely to suggest constraints and examples. For the former, if we are not able to reject a negative example, we can construct a constraint that would reject the negative examples and none of the positive examples. As for the examples, if we have two solutions to a problem, we can generate an example discriminating between them and ask user to clarify it, while this might not always be possible, since symmetric assignments might lead to semantically identical programs. In practice, however, this might be an important addition to simplify sketching for end users. Another interesting direction is to incorporate non-constant preference handling into the

[5] During the experiments, we stumbled upon a peculiar bug. One ASP encoding that we discovered in a public repository worked mostly by pure luck. The following constraint `:-queen(X1,Y1),queen(X2,Y2),X1<X2,abs(Y1-X1)==abs(Y2-X2).` works because `abs` is not actually absolute value but an uninterpreted function, essentially it checks $X == Y$, and that is indeed the found solution. (This kind of bugs would be extremely hard to find using traditional debuggers, since technically the encoding produced correct solutions.) Also, while working on the aggregate extension use-case, we discovered a subtle bug: the case of a single celebrity was not handled correctly. In both cases, the author has been contacted and models have been updated.

ASP model using the extensions of ASP for preference handling, such as *asprin* (Brewka et al. 2015).

Sketching in Constraint Programming and Constraint Logic Programming The rewriting schema that we have proposed here can be of course applied to other formalisms such as Constraint Programming, such as MiniZinc, and Constraint Logic Programming, such FO(.) (IDP). In that case, our schema follows the same ideas and structure, each constraint receives a decision variable and we search in the space of reified decisions, i.e., the reified constraints express the choice of the original constraints: in that case each constraint with a decision variable is essentially an OR between multiple ground constraints. Similar ideas apply to the negative example handling, aggregates and sketching multiple constraints.

Chapter 6

Relational Pattern Mining

This chapter is based on the papers: "An Exercise in Declarative Modelling for Relational Query Mining" by Sergey Paramonov, Matthijs van Leeuwen, Marc Denecker, Luc De Raedt[1] and "Hybrid ASP-Based Approach to Pattern Mining" by Sergey Paramonov, Daria Stepanova, Pauli Miettinen, Theory and Practice of Logic Programming 2018. My contributions are in the algorithm design, implementation, experiments, writing the results and the original idea of ASP model hybridization.

Detecting small sets of relevant patterns from a given dataset is a central challenge in data mining. The relevance of a pattern is based on user-provided criteria; typically, all patterns that satisfy certain criteria are considered relevant. Rule-based languages like Answer Set Programming (ASP) seem well-suited for specifying such criteria in form of constraints. Although progress has been made, on the one hand, on solving individual mining problems and, on the other hand, developing generic mining systems, existing methods either focus on scalability or on generality. In this chapter we make steps towards combining local (frequency, size, cost) and global (various condensed representations like maximal, closed, skyline) constraints in a generic and efficient way. We present a hybrid approach for itemset, sequence and graph mining which exploits dedicated optimized mining systems to detect frequent patterns and then filters the results using declarative ASP. To further demonstrate the generic nature of our hybrid framework we apply it to a problem of approximately tiling a database. Experiments on real-world datasets show the effectiveness of the proposed method and computational gains for itemset, sequence and graph mining, as well as approximate tiling.

[1] https://link.springer.com/chapter/10.1007/978-3-319-40566-7_12

6.1 Introduction

Motivation. The availability of vast amounts of data from different domains has led to an increasing interest in the development of scalable and flexible methods for data analysis. A key feature of flexible data analysis methods is their ability to incorporate users' background knowledge and different criteria of interest. They are often provided in the form of *constraints* to the valid set of answers, the most common of which is the frequency threshold: a pattern is only considered interesting if it appears often enough. Mining all frequent (and otherwise interesting) patterns is a very general problem in data analysis, with applications in medical treatments, customer shopping sequences, Weblog click streams and text analysis, to name but a few examples.

Most data analysis methods consider only one (or few) types of constraints, limiting their applicability. Constraint Programming (CP) (Guns et al. 2017; Négrevergne and Guns 2015) has been proposed as a general approach for (sequential) mining of frequent patterns (Agrawal et al. 1996), and Answer Set Programming (ASP) (Eiter et al. 2009a; Gelfond and Lifschitz 1988) has been proven to be well-suited for defining the constraints conveniently (see e.g., (Gebser et al. 2016; Guyet et al. 2014; Järvisalo 2011) for existing approaches on ASP-based frequent pattern mining) thanks to its expressive and intuitive modelling language and the availability of optimized ASP solvers such as Clasp (Gebser et al. 2007) and WASP (Alviano et al. 2013).

In general, all constraints can be classified into *local constraints*, that can be validated by the pattern candidate alone, and *global constraints*, that can only be validated via an comparison of the pattern candidate against all other candidates. Combining local and global constraints in a generic way is an important and challenging problem, which has been widely acknowledged in the constraint-based mining community. Although progress has been made, on the one hand, on solving individual mining problems and, on the other hand, on developing generic mining systems, existing methods either focus on scalability or on generality, but rarely address both of these aspects. This naturally limits the practical applicability of existing approaches.

State of the art and its limitations. Purely declarative ASP encodings for frequent and maximal itemset mining were proposed by Järvisalo (2011). In this approach, every item's inclusion into the candidate itemset is guessed at first, and the guessed candidate pattern is checked against frequency and maximality constraints. While natural, this encoding is not truly generic, as adding extra local constraints requires significant changes. Indeed, for a database, where all available items form a frequent (and hence maximal) itemset, the maximal ASP encoding has a single model. The latter is, however, eliminated once

restriction on the length of allowed itemsets is added to the program. This is undesired, as being maximal is not a property of an itemset on its own, but rather in the context of a collection of other itemsets (Bonchi and Lucchese 2006). Thus, ideally one would be willing to first apply all local constraints and only afterwards construct a condensed representation of them, which is not possible in the approach of Järvisalo (2011).

This shortcoming has been addressed in the recent work (Gebser et al. 2016) on ASP-based sequential pattern mining, which exploits ASP preference-handling capacities to extract patterns of interest and supports the combination of local and global constraints. However, both Gebser et al. (ibid.) and Järvisalo (2011) present purely declarative encodings, which suffer from scalability issues caused by the exploration of the huge search space of candidate patterns (existing solvers cannot yet take the full advantage of the stucture of the problem (Hallen et al. 2016)). The subsequence check amounts to testing whether an embedding exists (matching of the individual symbols) between sequences. In sequence mining, a pattern of size m can be embedded into a sequence of size n in $O(n^m)$ different ways, therefore, clearly a direct pattern enumeration is unfeasible in practice.

While a number of individual methods tackling selective constraint-based mining tasks exist (see Table 6.1 for comparison) there is no uniform ASP-based framework that is capable of effectively combining constraints both on the global and local level and is suitable for itemsets, sequences and graphs alike.

Contributions. The goal of our work is to make steps towards building a generic framework that supports mining of condensed (sequential) patterns, which (1) effectively combines dedicated algorithms and declarative means for pattern mining and (2) is easily extendable to incorporate various constraints. More specifically, the salient contributions of our work can be summarized as follows:

- We present a general extensible pattern mining framework for mining patterns of different types using ASP.
- In addition to the classical pattern mining problems, we demonstrate the generic nature of our framework by applying it to a problem of approximately tiling a database.
- We introduce a feature comparison between different ASP mining models and dominance programming (a generic itemset mining language and solver)

- We demonstrate the feasibility of our approach with an experimental evaluation across multiple itemset and sequence datasets using state-of-the-art ASP solvers.

This clearly indicates the key advantage of the approach: it preserves the generality of purely declarative methods with respect to the frequent pattern mining problems of the specified types, while provides an efficient system to develop prototypes, which can run on real-world datasets where often specialized algorithms are deployed. Our approach has indeed an advantage in the setting where a user considers a new variation of a pattern mining problem and needs to prototype a system to run on the standard real-world pattern mining datasets. Typically, even standard pattern mining datasets are too large for purely declarative systems and researchers have to experiment with modest (in size) datasets (for example, see experimental sections in (Gebser et al. 2016; Guns et al. 2013b; Négrevergne and Guns 2015; Négrevergne et al. 2013; Paramonov et al. 2015b)); contrary to this, developers of specialized algorithms have to significantly change their models to solve a new variation of a problem (for example, see separate algorithms and papers for gSpan (Yan and Han 2002) and cloSpan (Yan et al. 2003)). Our approach provides a middle ground between them, on the one hand, a researcher can model a new problem variation without changing the whole model, i.e., with slight changes and, on the other hand, she can experiment with large scale datasets, which indeed makes the declarative approach more practical and appealing.

Structure. After providing necessary background in Sec. 6.2 we introduce our approach in Sec. 6.3, and discuss approximate pattern mining in Sec. 6.4. Experimental results are described in Sec. 6.5, while related work and final remarks are provided in Sec. 6.6 and Sec. 6.7 respectively.

6.2 Preliminaries

In this section we briefly recap the necessary background both from the fields of pattern mining and Answer Set Programming (ASP).

Let D be a dataset, $\mathcal{L}$ a language for expressing pattern properties or defining subgroups of the data, and q a selection predicate. The task of pattern mining is to find $Th(\mathcal{L}, D, q) = \{\phi \in \mathcal{L} \mid q(D, \phi) \text{ is true}\}$, that is, to find all patterns $\phi \in \mathcal{L}$ that are selected by q (see, e.g, the seminal work of Mannila and Toivonen (1997)).

Datatype	Task	Järvisalo (2011)	Gebser et al. (2016)	Négrevergne et al. (2013)	**Our work**
Itemset	frequent pattern mining	✓	–	✓	✓
	condensed (closed, max, etc)	✓*	–	✓	✓
	condensed under constraints	–	–	✓	✓
Sequence	frequent pattern mining	–	✓	–	✓
	condensed (closed, max, etc)	–	✓	–	✓
	condensed under constraints	–	✓	–	✓
Graphs	frequent pattern mining	–	–	–	✓
	condensed (closed, max, etc)	–	–	–	✓
	condensed under constraints	–	–	–	✓

Table 6.1: Feature comparison between various ASP mining models and dominance programming ("–" : "not designed for this datatype", ✓* : only maximal is supported)

6.2.1 Patterns

Itemsets

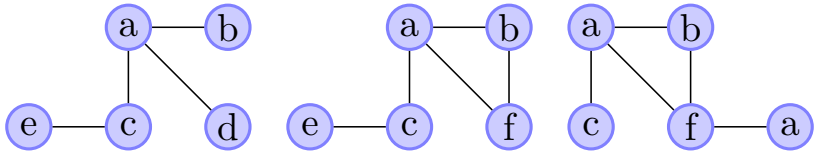
Itemsets represent the most simple setting of frequent pattern mining. Let $\mathcal{I}$ be a set of items $\{o_1, o_2, \ldots, o_n\}$. A nonempty subset of $\mathcal{I}$ is called an *itemset*. A *transaction dataset* D is a collection of itemsets, $D = \{t_1, \ldots, t_m\}$, where $t_i \subseteq \mathcal{I}$. For any itemset α, we denote the set of transactions that contain α as $D_\alpha = \{i \mid \alpha \subseteq t_i, t_i \in D\}$; we refer to D_α as the *cover* of an itemset α and to $|D_\alpha|$ as the *support (frequency)* of α in D, written $sup(\alpha)$. The *relative frequency* of α in D refers to the ratio between $sup(\alpha)$ and $|D|$. The *cardinality* (or *size*) $|\alpha|$ of an itemset α is the number of items contained in it.

ID	*a*	*b*	*c*	*d*	*e*
1	✓	✓		✓	✓
2		✓	✓		✓
3	✓				✓

Table 6.2: Transaction database

ID	Sequence
1	$\langle a\ b\ c\ d\ a\ e\ b \rangle$
2	$\langle b\ c\ e\ b \rangle$
3	$\langle a\ a\ e \rangle$

Table 6.3: Sequence database



(a) Graph G_1 (b) Graph G_2 (c) Graph G_3

Figure 6.1: Graph examples

Definition 6.2.1 (Frequent Itemset)**.** Given a transaction dataset D and a frequency threshold $\sigma \geq 0$, an itemset α is *frequent* in D if $sup(\alpha) \geq \sigma$.[2]

We illustrate the introduced notions by the following example.

Example 6.1. Consider a transaction dataset D from Table 6.2. We have $\mathcal{I} = \{a, b, c, d, e\}$ and $|D| = 3$. For $\sigma = 2$, the following itemsets are frequent: α_1=$\{a\}$, α_2=$\{b\}$, α_3=$\{e\}$, α_4=$\{a, e\}$ and α_5=$\{b, e\}$. Moreover, it holds that $D_{\alpha_4} = \{1, 3\}, D_{\alpha_5} = \{1, 2\}$, and the coverage for the rest of the itemsets can be analogously found. □

Sequences

A *sequence* is an ordered set of items $\langle s_1, \ldots, s_n \rangle$. The setting of *sequence mining* includes two related yet different cases: frequent substrings and frequent subsequences. In this work we focus on the latter.

Definition 6.2.2 (Embedding in a Sequence)**.** Let $S = \langle s_1, \ldots, s_m \rangle$ and $S' = \langle s'_1, \ldots, s'_n \rangle$ be two sequences of size m and n respectively with $m \leq n$. The tuple of integers $e = (e_1, \ldots, e_m)$ is an *embedding* of S in S' (denoted $S \sqsubseteq_e S'$) if and only if $e_1 < \ldots < e_m$ and for any $i \in \{1, \ldots, m\}$ it holds that $s_i = s'_{e_i}$.

[2] In *frequent pattern mining*, often, a *relative threshold*, i.e., $\sigma/|D|$ is specified by the user.

Example 6.2. For the dataset in Table 6.3 we have that $\langle b\,c\,e\,b\rangle \sqsubseteq_{e_1} \langle a\,b\,c\,d\,a\,e\,b\rangle$ for $e_1 = (2, 3, 6, 7)$ and analogously, $\langle a\,a\,e\rangle \sqsubseteq_{e_2} \langle a\,b\,c\,d\,a\,e\,b\rangle$ with $e_2 = (1, 5, 6)$. □

We are now ready to define an inclusion relation for sequences.

Definition 6.2.3 (Sequence Inclusion)**.** Given two sequences $S = \langle s_1, \ldots, s_m\rangle$ and $S' = \langle s'_1, \ldots, s'_n\rangle$, of size m and n, respectively, with $m \leq n$, we say that S is *included* in S' or S is a *subsequence* of S' denoted by $S \sqsubseteq S'$ iff an embedding e of S in S' exists, i.e.

$$S \sqsubseteq S' \leftrightarrow \exists e_1 < \ldots < e_m \text{ and } \forall i \in 1 \ldots m : s_i = s'_{e_i}. \tag{6.1}$$

Example 6.3. In Ex. 6.2 we have $\langle b\,c\,e\,b\rangle \sqsubseteq \langle a\,b\,c\,d\,a\,e\,b\rangle$ but $\langle a\,a\,e\rangle \not\sqsubseteq \langle b\,c\,e\,b\rangle$. □

For a given sequence S and a sequential dataset $D = \{S_1, \ldots, S_n\}$ we denote by D_S the subset of D s.t. $S \sqsubseteq S'$ for all $S' \in D_S$. The support of S is $sup(S) = |D_S|$. Frequent sequences are defined analogously to frequent itemsets.

Definition 6.2.4 (Frequent Sequence)**.** Given a sequential dataset $D = \{S_1, \ldots, S_n\}$ and a frequency threshold $\sigma \geq 0$, a sequence S is *frequent* in D if $sup(S) \geq \sigma$.

Example 6.4. For the dataset in Table 6.3 and $\sigma = 2$, it holds that $\langle b\,c\,e\,b\rangle$ and $\langle a\,a\,e\rangle$ are frequent, while $\langle b\,d\,b\rangle$ is not. □

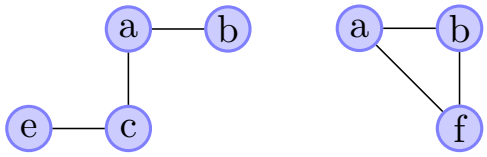
Note that $\sqsubseteq$ and $\subseteq$ are incomparable relations. Indeed, consider two sequences $s_1 = \langle a\,b\rangle$ and $s_2 = \langle b\,a\,a\rangle$. While $s_1 \subset s_2$, we clearly have that $s_1 \not\sqsubseteq s_2$.

Graphs

A *graph* G is a triple $\langle V, E, l\rangle$ where V is a set of vertices, E is a set of edges and l is a labeling function that maps each edge and each vertex to a label.

In this work we consider undirected graphs. Moreover, we primarily focus on two settings: a general one, where labels of the graph are not necessarily unique and a more restricted one, where unique labels are ensured.

General Case of Graph Mining. Here we consider the general case of mining graphs whose nodes have labels that are not necessarily unique. This setting is computationally harder than the special case of uniquely labelled graphs. In this general setting, a pattern is an arbitrary graph with labeled nodes and edges.



(a) Graph G'_1 (b) Graph G'_2

Figure 6.2: Isomorphic subgraphs for graphs in Figure 6.1

Uniquely Labelled Graphs. In this restricted case, the labeling function assigns unique labels to the nodes, i.e., each node has a unique label within this graph (in the setting, no labels is assumed on the edges). This restriction makes sub-pattern check computationally easier and allows certain reductions to other pattern mining problems as indicated below.

Example 6.5. The graphs in Figure 6.1a and 6.1b are unique-labeled ones, while the graph from Figure 6.1c is clearly not. □

Throughout the text we assume the general case unless stated otherwise, all experimental results have been provided for the general case of graph mining.

Definition 6.2.5 (Graph Isomorphism)**.** Given two graphs $G = \langle V, E, l\rangle$ and $G' = \langle V', E', l'\rangle$, we say that G is isomorphic to G' iff there is a bijective function f such that

- $v \in V$ iff $f(v) \in V'$ and for all $v \in V$ it holds that $l(v) = l'(f(v))$
- $e \in E$ iff $f(e) \in E'$ and for all $e \in E$ it holds that $l(e) = l'(f(e))$.

In the general setting, the problem of deciding whether a subgraph isomorphism exists is NP-complete. However, several restricted settings have been identified, for which the problem can be solved in polynomial time, e.g., unique-labelled undirected graphs (Neumann and Miettinen 2017).

Definition 6.2.6 (Graph Inclusion)**.** Given two graphs $G = \langle V, E\rangle$ and $H = \langle U, F\rangle$, such that $|V| \leq |F|$, we say that G is an *(isomorphic) subgraph* of H denoted by $G \sqsubseteq H$ iff there exists a subgraph $H' = \langle U' \subseteq U, F' \subseteq F\rangle$ such that G is isomorphic to H'.

Example 6.6. Graph G'_1 of Figure 6.2 is an isomorphic subgraph of graphs G_1 and G_2 of Figure 6.1; it is not isomorphic to graph G_3. Graph G'_2 is isomorphic to graphs G_2 and G_3, but it is not isomorphic to graph G_1. □

6.2.2 Condensed pattern representations under constraints

In data mining, constraints are typically specified by the user to encode domain background knowledge. Négrevergne and Guns (2015) distinguish four types of constraints: 1) constraints over the pattern (e.g., restriction on its size), 2) constraints over the cover set (e.g., minimal frequency), 3) constraints over the inclusion relation (e.g., maximal allowed gap in sequential patterns) and 4) constraints over the solution set (e.g., condensed representations).

Orthogonally, constraints can be classified into *local* and *global* ones. A constraint is *local* if deciding whether a given pattern satisfies it is possible without looking at other patterns. For example, minimal frequency or maximal pattern size are local constraints. On the contrary, deciding whether a pattern satisfies a *global* constraint requires comparing it to other patterns. All constraints from the fourth group are global ones. We are interested in global constraints related to condensed representations.

As argued in Sec. 6.1, the order in which constraints are applied influences the solution set (Bonchi and Lucchese 2006). Following Bonchi and Lucchese (ibid.), in this work we apply global constraints only after local ones.

We now present the notions required in our pattern mining framework. Here, the definitions are given for itemsets; for sequences and graphs they are identical up to substitution of $\subset$ with $\sqsubset$ (subsequence relation). First, to rule out patterns that do not satisfy some of the local constraints, we introduce the notion of validity.

Definition 6.2.7 (Valid pattern under constraints)**.** Let C be a constraint function (local constraint) from $\mathcal{L}$ to $\{\top, \bot\}$ and let p be a pattern in $\mathcal{L}$. Then the pattern p is called *valid* iff $C(p) = \top$; otherwise it is referred to as *invalid*.

Example 6.7. Let C be a constraint function checking whether a given pattern is of size at least 2. Then in Ex. 6.1, we have $C(\alpha_i) = \bot$, $i \in \{1, 2, 3\}$, and $C(\alpha_j) = \top$, $j \in \{4, 5\}$. □

For detecting patterns that satisfy a given global constraint, the notion of *dominance* is of crucial importance. Intuitively, a dominance relation reflects pairwise preference ($<^*$) between patterns and it is specific for each mining setting. In this work we primarily focus on global constraints related to maximal, closed, free and skyline condensed representations, for which $<^*$ is defined as follows:

(i) **Maximal.** For itemsets p and q, $p <^* q$ holds iff $p \subset q$

(ii) **Closed.** For itemsets p and q, $p <^* q$ holds iff $p \subset q$ and $sup(p) = sup(q)$

(iii) **Free.** For itemsets p and q, $p <^* q$ holds iff $q \subset p$ and $sup(p) = sup(q)$

(iv) **Skyline.** For itemsets p and q, $p <^* q$ holds iff

(a) $sup(p) \leq sup(q)$ and $size(p) < size(q)$ or

(b) $sup(p) < sup(q)$ and $size(p) \leq size(q)$

Dominated patterns under constraints are now formally defined.

Definition 6.2.8 (Dominated pattern under constraints)**.** Let C be a constraint function, and let p be a pattern, then p is called *dominated* iff there exists a pattern $p' \in \mathcal{L}$ such that $p <^* p'$ and p' is valid under C.

Example 6.8. In Ex. 6.1 for the maximality constraint we have that α_1 is dominated by α_4, α_2 by α_5, while α_3 both by α_4 and α_5. □

Exploiting the above definitions we obtain condensed patterns under constraints.

Definition 6.2.9 (Dominant (condensed) pattern under constraints)**.** Let p be a pattern from $\mathcal{L}$, and let C be a constraint function, then a pattern p is called *dominant* under constraints iff it is valid and not dominated under C.

Example 6.9. For the constraint function selecting maximal itemsets of size at most 2 and size at least 2, α_4 and α_5 from Ex. 6.1 are condensed patterns. □

Intuitively, a condensed representation is the smallest set of "good" patterns describing the data, i.e., there is no redundant or invalid patterns. By redundant here, we mean patterns dominated or preferred to another pattern. Formally speaking,

Definition 6.2.10 (Condensed representation)**.** The set of all *dominant* patterns is called a *condensed representation*.

Note that various names exist in the literature such the *dominating set* (Négrevergne et al. 2013). We use the terms *a condensed pattern* and *a dominant pattern* interchangeably.

6.3 Hybrid ASP-based mining approach

In this section we present our hybrid method for frequent pattern mining. Unlike previous ASP-based mining methods, our approach combines general declarative models and specialized algorithms in the two step procedure, where in the first

step, we apply highly optimized algorithms for frequent pattern discovery and, in the second step, we use a declarative ASP solver for their convenient post-processing. Intuitively, our algorithm can be seen as a pipeline where, at the first step, frequent patterns are being generated using highly efficient specialized algorithms and, at the second step, a condensed representation is extracted using a declarative approach based on ASP. Here, we mainly focus on itemsets, sequence and graph mining.

Given a frequency threshold σ, a dataset D and a set of constraints $\mathcal{C} = \mathcal{C}_l \cup \mathcal{C}_g$, where $\mathcal{C}_l$ and $\mathcal{C}_g$ are respectively local and global constraints, we proceed in two steps as follows.

Step 1. First, we launch a dedicated optimized algorithm to extract all frequent patterns from a given dataset, satisfying the minimal frequency threshold σ. Here, any frequent pattern mining algorithm can be invoked. We use Eclat (Zaki et al. 1997) for itemsets, PPIC (Aoga et al. 2016) for sequences and gSpan (Yan and Han 2002) for graphs.

Step 2. Second, the computed patterns are post-processed using the declarative means to select a set of *valid* patterns (i.e., those satisfying constraints in $\mathcal{C}_l$). For that the frequent patterns obtained in Step 1 are encoded as facts `item(i,j)` for itemsets and `seq(i,j,p)` for sequences, where `i` is the pattern's ID, `j` is an item contained in it and `p` is its position. Analogously, for graphs we have `graph(i,j,k,l)`, where `i` is the pattern's ID, `j` and `k` are node labels and the intentional meaning of this fact is that in the graph `i` the node `j` is connected to `k`, and `l` is its label. The local constraints in $\mathcal{C}_l$ are represented as ASP rules, which collect IDs of patterns satisfying constraints from $\mathcal{C}_l$ into the dedicated predicate `valid`, while the rest of the IDs are put into the `not_valid` predicate.

Finally, from all valid patterns a desired condensed representation is constructed by storing patterns `i` in the `selected` predicate if they are not `dominated` by other valid patterns based on constraints from $\mathcal{C}_g$. Following the principle of (Järvisalo 2011), for itemsets and sequences in our work every answer set represents a single desired pattern, which satisfies both local and global constraints (for graphs slight variations apply, as we discuss later in this section). The set of all such patterns forms a condensed representation. In what follows we present our encodings of local and global constraints in details.

6.3.1 Encoding local constraints

In our declarative program we specify local constraints by the predicate `valid`, which reflects the conditions given in Def. 6.2.7. For every constraint in $\mathcal{C}_l$ we have a set of dedicated rules, stating when a pattern is not valid. For instance, a constraint checking whether the cost of items in a pattern exceeds a given threshold N is encoded as

```
not_valid(I) :- #sum{C:item(I,J),cost(J,C)} > N, pattern(I).
```

A similar rule for sequences can be defined as follows:

```
not_valid(I) :- #sum{C:seq(I,J,P),cost(J,C)} > N, pattern(I).
```

Analogously, one can specify arbitrary domain constraints on patterns.

Example 6.10. Consider a dataset storing moving habits of young people during their studies. Let the dedicated frequent sequence mining algorithm return the following patterns: $S_1 = \langle bG\ mF\ ba\ mG\ ma\rangle$; $S_2 = \langle bF\ mG\ ba\ mF\ ma\rangle$; $S_3 = \langle bA\ ba\ ma\rangle$, where bG,bF, bA stand for born in Germany, France and America, ba, ma stand for bachelors and masters and the predicates mG, mF reflect that a person moved to Germany and France, respectively. Suppose, we are only interested in moving habits of Europeans, who got their masters degree from a German university. The local domain constraint expressing this would state that (1) bA should not be in the pattern, while (2) either both bG and ma should be in it without any mF in between or mG should precede ma. These constraints are encoded in the program in Listing 6.1. From the answer set of this program we get that both S_2 and S_3 are not valid, while S_1 is. □

To combine all local constraints from $\mathcal{C}_l$ we add to a program a generic rule specifying that a pattern `I` is valid whenever `not_valid(I)` cannot be inferred.

```
valid(I) :- pattern(I), not not_valid(I)
```

Patterns `i`, for which `valid(i)` is deduced are then further analyzed to construct a condensed representation based on global constraints from $\mathcal{C}_g$.

6.3.2 Encoding global constraints

The key for encoding global constraints is the declarative formalization of the dominance relation (Def. 6.2.8). For example, for itemsets the maximality

```
time(1..5).
% people born in Germany or France are Europeans
eu(I) :- seq(I,bG,P).
eu(I) :- seq(I,bF,P).
% collect those who moved to France before P
moved_before(X,P) :- seq(X,mF,P1), P > P1, time(P), time(P1).
% collect those who moved to France after P and before masters
moved_after(X,P) :- seq(X,mF,P1), seq(X,ma,P2), P < P1,
                    P1 < P2, time(P), time(P1), time(P2).
% keep Europeans who moved to Germany straight before masters
keep(X) :- seq(X,ma,P+1), seq(X,mG,P), eu(X).
% keep Germans who did not move before masters
keep(X) :- seq(X,bG,P1), seq(X,ma,P), not moved_before(X,P).
% keep Europeans whose last move before masters was to Germany
keep(X) :- seq(X,mG,P1), seq(X,ma,P2), P1 < P2,
           eu(X), not moved_after(X,P1).
% a pattern is not valid, if it should not be kept
not_valid(X) :- pattern(X), not keep(X)

```

Listing 6.1: Moving habits of people during studies

constraint boils down to pairwise checking of subset inclusion between patterns. For sequences this requires a check of embedding existence between sequences.

Regardless of a pattern type from $\mathcal{L}$ and a constraint from $\mathcal{C}_g$ every encoding presented in this section is supplied with a rule, which guesses (`selected/1` predicate) a single `valid` pattern to be a candidate for inclusion in the condensed representation, and a constraint that rules out `dominated` patterns thus enforcing a different guess.

```
1 {selected(I) : valid(I)} 1.

:- dominated.
```

In what follows, we discuss concrete realizations of the dominance relation both for itemsets and sequences for various global constraints, i.e., we present specific rules related to the derivation of the `dominated/0` predicate.

Itemset Mining. We first provide an encoding for maximal itemset mining in List 6.2. To recall, a pattern is *maximal* if none of its supersets is frequent. An itemset I is included in J iff for every item $i \in I$ we have $i \in J$. We encode

```
% I is not a subset of J if I has items that are not in J
not_superset(J) :- selected(I), item(I,V), not item(J,V),
                   valid(J), I != J.
% derive dominated whenever I is a subset of J
dominated :- selected(I), valid(J),
             I != J, not not_superset(J).
```

Listing 6.2: Maximal itemsets encoding

```
% support and size comparison among patterns
g_size_geq_fr(J) :- selected(I), valid(J), support(I,X),
                    support(J,Y), size(I,Si), size(J,Sj),
                    Si <  Sj, X <= Y.
geq_size_g_fr(J) :- selected(I), valid(J), support(I,X),
                    support(J,Y), size(I,Si), size(J,Sj),
                    Si <= Sj, X < Y.
% derivation of the domination condition
dominated :- valid(J), g_size_geq_fr(J).
dominated :- valid(J), geq_size_g_fr(J).
```

Listing 6.3: Skyline itemsets encoding

the violation of this condition in lines (1)–(3). The second rule presents the dominance criteria.

For closed itemset mining a simple modification of Listing 6.2 is required. An itemset is *closed* if none of its supersets has the same support. Thus to both of the rules from Listing 6.2 we need to add atoms `support(I,X), support(J,X)`, which store the support sets of `I` and `J` respectively (extracted from the output of Step 1).

For free itemset mining the rules of the maximal encoding are changed as follows:

```
not_superset(J) :- selected(I), item(J,V), not item(I,V),
                   valid(J), I != J.
dominated :- selected(I), valid(J), support(I,X),
             I != J, not not_superset(J), support(J,X).
```

Finally, the skyline itemset/sequence encoding is given in Listing 6.3, where the first two rules specify the conditions (a) and (b) for skyline itemsets as specified in Sec. 6.2.

```
1  % if V appears in a valid pattern I, derive in(V,I)
2  in(V,I) :− seq(I,V,P), valid(I).
3  % I is not a subset of J if I has V that J does not have
4  not_superset(J) :− selected(I), valid(J), I != J,
5                     seq(I,V,P), not in(V,J).
6  % if for a subseq <V,W> in I there is V followed
7  % by W in J then deduce domcand(V,J)
8  domcand(V,J,P) :− selected(I), seq(I,V,P), seq(I,W,P+1), I != J
9                    valid(J), seq(J,V,Q), seq(J,W,Q'), Q'>Q.
10 % if domcand(V,J) does not hold for some V in I
11 % and a pattern J then derive not_dominated_by(J)
12 not_dominated_by(J) :− selected(I), seq(I,V,P), seq(I,W,P+1),
13                       I != J, valid(J), not domcand(V,P,J).
14 % if neither not_dominated_by(J) nor not_superset(J)
15 % are derived for some J, then I is dominated
16 dominated :− selected(I), valid(J), I != J,
17             not not_superset(J), not not_dominated_by(J).
```

Listing 6.4: Maximal sequence encoding

Sequence Mining. The subpattern relation for sequences is slightly more involved, than for itemsets, as it preserves the order of elements in a pattern. To recall, a sequence S is included in S' iff an embedding e exists, such that $S \sqsubseteq_e S'$.

In Listing 6.4 we present the encoding for maximal sequence mining. A selected pattern is not maximal if it has at least one valid superpattern. We rule out patterns that are for sure not superpatterns of a selected sequence. First, `J` is not a superpattern of `I` if `I` is not a subset of `J` (lines (4)–(5)), i.e., if `not_subset(J)` is derived, then `J` does not dominate `I`. If `J` is a superset of `I` then to ensure that `I` is not dominated by `J`, the embedding existence has to be checked (lines (6)–(9)). `I` is not dominated by `J` if an item exists in `I`, which together with its sequential neighbor cannot be embedded in `J`. This condition is checked in lines (10)–(13), where `domcand(V,J,P)` is derived if for an item `V` at position `P` and its follower, embedding in `J` can be found.

The encoding for closed sequence mining is obtained from the maximal sequence encoding analogously as it is done for itemsets. The rules for free sequence mining are constructed by substituting lines (4)–(13) of Listing 6.4 with the following ones:

```
not_superset(J) :- selected(I), in(V,J),
                   not in(V,I), I != J.
domcand(V,J) :- selected(I), seq(J,V,P), item(J,P+1,W),
                item(I,V,Q), seq(I,W,Q'),Q'>Q, I != J.
not_dominated_by(J) :- selected(I), valid(J), I != J,
                       seq(J,V,P), seq(J,W,P+1),
                       not domcand(V,J).
```

Finally, the encoding for mining skyline sequences coincides with the skyline itemsets encoding, which is provided in Listing 6.3.

Graph Mining. Graphs represent the most complex pattern type. The subpattern relation between graphs amounts to testing subgraph isomorphism between them. In general, this problem is NP-complete; however, for some restricted graph types, it is solvable in polynomial time.

Uniquely Labelled Graphs. One of such graph types are undirected graphs (Neumann and Miettinen 2017), where every node has a unique label (and consequently, every edge (u, v) is labelled uniquely as $(l(u), l(v))$). For instance, the first two graphs in Figure6.1 fall into this category, while the third one does not. For this restricted graph type, we have that G is subgraph isomorphic to G' iff every edge in G is also present in G'. Therefore, the subgraph isomorphism test for these restricted graphs essentially boils down to subpattern test for itemsets.

We treat every edge in a graph as an item. Since the graph is allowed to have only unique labels, there cannot be any repetitions of the edge labels. The encoding of maximal frequent graph patterns is presented in Listing 6.5. A selected graph pattern is not maximal if at least one of its frequent supergraphs is valid. Similar to the case of itemsets and sequences we rule out the patterns that are guaranteed to be not maximal. A pattern `J` is not a superpattern of `I` if `I` contains an edge that `J` does not have.

The encoding for closed frequent graph patterns differs from the one for maximal graph patterns only in that in the second rule in Figure 6.5 the atoms `support(I,X), support(J,X)` are added. Encodings for other condensed representations are analogous.

Example 6.11. The edges of the graph G_1 in Figure 6.1 are represented with the following facts `edge(g1,e,c), edge(g1,c,a)`, etc. Given the unique-labeled graphs G_1 and G_2 and the frequency threshold $\sigma = 2$, we have that both `{(a,b)}` and `{(e,c),(c,a),(a,b)}` are frequent subgraphs. However, only the latter graph pattern is maximal. □

```
4 % I is not a subgraph of J if I has an edge (X,Y) that J does not have
5 not_supergraph(J) :- selected(I), edge(I,X,Y), valid(J),
6                      not edge(J,X,Y), not edge(J,Y,X), I != J.
7 % derive dominated, whenever I is a subgraph of J
8 dominated :- selected(I), valid(J), not not_supergraph(J), I != J.
```

Listing 6.5: Maximal graph encoding (unique labeled; ids represent labels as well; edge(I,X,Y) represents here a graph I with an edge between X and Y, whose values are their labels respectively)

General Case of Graph Mining The encoding for maximal (closed, etc) graph mining problem in the general case is slightly more complicated. For the maximal constraint we depict the encoding in Listing 6.6 (the rest of the constraints are treated analogously). Since the subgraph isomorphism check is an NP-complete problem, after obtaining a set F of frequent candidate graph patterns using a dedicated algorithm, we perform a dominance check using the ASP program in Listing 6.6 for each pattern separately (unlike in earlier presented encodings, where such a check was done for all frequent patterns jointly within a single ASP program).

More specifically, the solver receives as input a selected graph candidate G and the rest of the frequent patterns in F excluding G. The graph G is represented using two predicates: `selected_node(v,lv)` reflecting labelled vertices of G and `selected_edge(v,w,le)`, where `v,w` are nodes and the edge `(v,w)` is labeled with `le` in G. First, in (2) of Listing 6.6 the guess is performed on a graph in the set F of frequent patterns, to which G can be mapped, i.e., a dominating candidate pattern is guessed. Second, in (4) the guess on a mapping from G to the dominating candidate graph is done. Finally, in (6)-(9) the mapping is validated by two integrity constraints. If the solver returns an answer set, then the selected graph G is removed from the dataset as a dominated pattern, i.e., it is guaranteed to be not maximal.

6.4 Covering with approximate patterns

Let us demonstrate that the hybridization framework we have introduced can solve even more complex pattern mining problems than classical frequent pattern mining problems. We will consider a variation of the tiling problem that we have seen in Chapter 3. This variation, called approximate tiling, is indeed connected to the one we have modelled as a decomposition problem as it also asking to find a form of tiling. Approximate tiling introduces a more complex

```
% pick one graph to map to
1 { mapped_to(X) : graph(X)} 1.
% guess the mapping to the selected graph
1 { map(X,N) : node(_,N,_) } 1 :- selected_node(X,_).
% check mapping validity on the edges with labels
:- mapped_to(I), map(X1,V1), map(X2,V2),
   selected_edge(X1,X2,L),  not edge(I,V1,V2,L).
% check validity on the nodes
:- mapped_to(I), map(X,V), selected_node(X,L), not node(I,V,L).
```

Listing 6.6: Maximal graph encoding (core); general case

optimization function and appears to be highly computationally demanding (see details below). Here we aim at demonstrating how hybridization framework can be used to reach reasonable performance even for this highly computationally challenging problem.

So far, we have concentrated on exact patterns, that is, patterns that are present exactly in the data; for example, an itemset is present only in transactions that contain every item in the itemset. This is a standard assumption in pattern mining, and it is heavily utilized by the pattern mining algorithms. But it also means that the patterns are not robust against noise: a single item missing from a single transaction can turn a frequent itemset into an infrequent one. To make the patterns more robust to noise, we can consider approximate patterns, that is, patterns that are considered as contained in transactions even if the transaction does not contain all of the items in the pattern. In this section, we demonstrate how our hybrid approach can be used also with approximate patterns.

The approximate patterns we are interested in are based on the concept of *tiles* (Geerts et al. 2004). Let α be an itemset and D be a database. The *tile* corresponding to α is defined as $\tau(\alpha, D) = \{(t_{id}, i) \mid t_{id} \in D_\alpha, i \in \alpha\}$. When D is clear from the context, we write $\tau(\alpha)$. The area of $\tau(\alpha)$ is equal to its cardinality.

Example 6.12. The tile corresponding to the itemset $\{b, e\}$ from Table 6.2 is $\{(1, b), (2, b), (1, e), (2, e)\}$.

A *tiling* $T = \{\tau(\alpha_1), \ldots, \tau(\alpha_k)\}$ consists of a finite number of tiles. Its area is $area(T, D) = |\tau(\alpha_1) \cup, \ldots, \cup \tau(\alpha_k)|$. In *Maximum k-Tiling* the goal is to find a tiling of k tiles with the maximum area, whereas in the *Minimum Tiling* the goal is to find the least number of tiles such that for all $t_{id} \in D$ and for all $i \in t_{id}$, there exists at least one tile $\tau(\alpha)$ such that $(t_{id}, i) \in \tau(\alpha)$ (ibid.).

Another way to define tiles is via binary matrices. A transaction database D can be regarded as a binary matrix $\mathbf{D}$ of size $m \times n$, where m is the number of transactions in D and n is the number of items in $\mathcal{I}$. The (i, j)th element in the binary matrix corresponding to D is equal to 1 if the ith transaction contains item j, and it is equal to 0 otherwise. Hence, the size of D is equal to the number of ones in this matrix. In this framework, a tile is a *rank-1* matrix $\mathbf{T}$ that is *dominated* by the data matrix $\mathbf{D}$. Matrix $\mathbf{T}$ is rank-1 if it is an outer product of two binary vectors, that is, $\mathbf{T} = \mathbf{u}\mathbf{v}^T$ where $\mathbf{u} \in \{0,1\}^m$ and $\mathbf{v} \in \{0,1\}^n$. Matrix $\mathbf{T} = (t_{ij})$ is dominated by matrix $\mathbf{D} = (d_{ij})$ if $t_{ij} \leq d_{ij}$ for all i and j.

Example 6.13. The transaction database in Table 6.2 corresponds to the following binary matrix:

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} ,$$

and the tile from Example 6.12 corresponds to the matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \end{pmatrix} .$$

□

A tiling is an element-wise logical disjunction of the matrices corresponding to the tiles,

$$\mathbf{S} = \mathbf{T}_1 \vee \mathbf{T}_2 \vee \cdots \vee \mathbf{T}_k .$$

Assuming that $\mathbf{T}_i = \mathbf{u}_i \mathbf{v}_i^T$, we can write each element of $\mathbf{S} = (s_{ij})$ as

$$s_{ij} = \bigvee_{\ell=1}^{k} u_{i\ell} v_{j\ell} ,$$

that is, $\mathbf{S}$ is the *Boolean matrix product* of matrices $\mathbf{U} = [\mathbf{u}_1\ \mathbf{u}_2\ \cdots\ \mathbf{u}_k]$ and $\mathbf{V} = [\mathbf{v}_1\ \mathbf{v}_2\ \cdots\ \mathbf{v}_k]$ (see Miettinen (2009) for more discussion on the connections between pattern mining and Boolean matrix factorization).

The formulation of tiles as rank-1 binary matrices immediately suggests the concept of *approximate tiles* as non-dominated rank-1 matrices and *approximate tiling* as approximate Boolean matrix factorization. We will call both exact and approximate tiles simply as tiles from now on.

Example 6.14. Consider the following Boolean matrix:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

The areas with the red, blue and black border highlight the respective three approximate tiles of the matrix. □

A common approach to calculate an approximate Boolean matrix factorization (or tiling) is to first calculate a set of *candidate tiles* (or rank-1 matrices) and then select the final set from these (Miettinen et al. 2008; Tyukin et al. 2014). In what follows, we concentrate on a version of this problem where the user gives the target quality (the coverage constant, what percentage of the dataset should be covered), and we try to find a set of tiles that obtains this quality.

Definition 6.4.1 (Approximate Tile Selection Problem)**.** Given a binary matrix $\mathbf{D} \in \{0,1\}^{m \times n}$, a set T of tiles (rank-1 matrices) and an integer threshold σ, find a subset $T' = \{\mathbf{T}^{(1)}, \mathbf{T}^{(2)}, \ldots, \mathbf{T}^{(k)}\} \subseteq T$, such that when we write $\mathbf{S} = (s_{ij})$ with $s_{ij} = t_{ij}^{(1)} \vee t_{ij}^{(2)} \vee \cdots \vee t_{ij}^{(k)}$, the error

$$|\{(i,j) \mid d_{ij} = 1 \wedge s_{ij} = 0\}| + |\{(i,j) \mid d_{ij} = 0 \wedge s_{ij} = 1\}| \tag{6.2}$$

(i.e., the number of ones outside of the tiling plus the number of zeros inside the tiling) is smaller then the threshold σ.

It is worth noticing that (6.2) corresponds to the Hamming distance between $\mathbf{D}$ and $\mathbf{S}$. The decision version of the Approximate Tile Selection problem is NP-hard (Miettinen 2015), and the optimization version (finding the smallest possible error) is NP-hard to approximate to within $\Omega(2^{\log^{1-\epsilon}|\mathbf{D}|})$ for any $\epsilon > 0$, where $|\mathbf{D}|$ is the number of 1s in $\mathbf{D}$ (ibid.). The problem has two alternative characterizations, either as a variant of the famous Set Cover problem called *Positive-Negative Partial Set Cover* (Miettinen 2008) or as a form of *Boolean linear programming*: given a binary design matrix $\mathbf{A}$, and a binary target vector $\mathbf{b}$, find a binary vector $\mathbf{x}$ that minimizes the Hamming distance between $\mathbf{b}$ and $\mathbf{Ax}$, where the matrix-vector product is over Boolean algebra.

For computing a set of candidate tiles T to choose from, effective implementations exist (see, e.g., Tyukin et al. (2014)). From these candidates a subset needs to be selected, which will be a solution to the problem from Def. 6.4.1. In Figure 6.7 we provide an ASP program, whose answer sets exactly correspond to solutions to the above problem.

```
% for every tile store its ID and coordinates of 1s that it encodes
tileid(X) :- tile(X,Y,Z).
one(Y,Z)  :- tile(X,Y,Z).
% among tiles guess at least one to be included in the tiling
1 { intiling(X)  : tileid(X)}.
% collect and count ones that are outside the constructed tiling
incell(Y,Z) :- intiling(X), tile(X,Y,Z).
outsideone(Y,Z) :- tile(X,Y,Z), not incell(Y,Z).
numberofoutsideones(N) :- N = #count{Y, Z : outsideone(Y,Z)}.
% collect and count zeros that are inside the constructed tiling
insidezero(Y,Z) :- intiling(X), relitem(X,Y),
                   reltrans(X,Z), not one(Y,Z).
numberofinsidezeros(N) :- N = #count{Y, Z : insidezero(Y,Z)}.
% compute the total error
er(0,K) :- numberofinsidezeros(K).
er(1,K) :- numberofoutsideones(K).
totalerror(N) :- N = #sum{X, Y : er(Y,X)}.
% ensure that the total error does not exceed the threshold
:- totalerror(N), threshold(K), N > K.
```

Listing 6.7: Encoding for the approximate tiling problem from Def. 6.4.1

$$\left(\begin{array}{cccc} \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} \\ 0 & 0 & 0 & 0 \\ \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} \end{array}\right) \quad \left(\begin{array}{cccc} \boxed{1} & 0 & 0 & \boxed{1} \\ \boxed{0} & 0 & 0 & \boxed{0} \\ \boxed{1} & 0 & 0 & \boxed{1} \end{array}\right)$$

Figure 6.3: Examples of tiles in a matrix

In the input the ASP program gets a set of tile candidates to choose from. They all are of the form `tile(id,i,t)`, where `id` is a unique ID of the given tile, while `i` and `t` are the IDs of the items and transactions respectively, such that the given tile has 1 in the intersection of the column `i` and the row `t`. To be capable of fully reconstructing a tile, along with the positions of 1s in it, we also require IDs of relevant items and transactions. Indeed, Figure 6.3 represents two possible tiles that can be reconstructed based on the positions of 1s only. Therefore, in the input we are also provided with the facts `relitem(id,i)` and `reltrans(id,t)`, which respectively encode the relevant items and transactions for the tile `id` to ensure a unique interpretation of the tiles. Notice that if there exists a 1 in the original data that is not part of any tile, no method will be able to cover it, and they will contribute a constant error. Hence we can ignore such 1s in our setting.

Example 6.15. If the red tile from Ex. 6.14 has ID 1, its encoding is given by the following facts: `tile(1,1,1), tile(1,2,1), tile(1,1,2), tile(1,1,3), relitem(1..2), reltrans(1..3)`. Moreover, provided that the left and right tile in Figure 6.3 have IDs 2 and 3 respectively, they are represented using facts: `tile(2,), tile(2,), relitem(), relitem(), reltrans()`

In (1)–(3) of Listing 6.7 we collect tiles' IDs and locations of ones using the dedicated predicates `tileid` and `one` respectively. Than in (4)–(5) for every tile we perform a guess on whether to include it into the tiling. In lines (6)–(9), we count the number of 1s that are outside of the tiling, while in lines (10)–(13), the number of 0s inside the tiling is computed. Finally, the total error is determined in lines (14)–(17) and its admissibility based on the given threshold is checked in lines (18)–(19).

This ASP program can be exploited to solve Boolean matrix factorization, when used together with existing candidate generation approaches, or to solve Boolean linear programming instances. In fact, in Boolean linear programming, the design matrix $\mathbf{A}$ is given as a part of the problem instance, hence for that even a purely declarative programming solution will suffice.

6.5 Evaluation

In this section we evaluate the proposed hybrid approach by comparing it to the existing declarative pattern mining methods: ASP model for sequences (Gebser et al. 2016) and Dominance Programming (DP) (Négrevergne et al. 2013). We do not consider the itemset mining ASP model (Järvisalo 2011), since it focuses only on frequent itemset mining and is not applicable to the construction of condensed representations under constraints as explained and addressed in (Négrevergne et al. 2013). Moreover, we do not perform comparison with dedicated algorithms designed for a specific problem type; these are known to be more efficient (in terms of runtime and memory usage) than declarative mining approaches (Négrevergne and Guns 2015), yet obviously less flexible (in terms of modelling flexibility, i.e., how general the method is and how easily it can be modified to model a variaton of the problem).

More specifically, we investigate the following experimental questions.

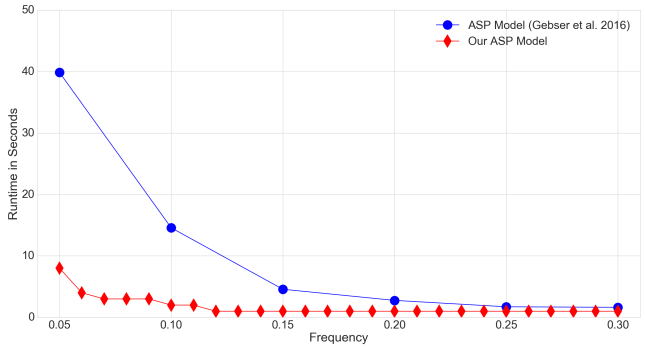
- $\mathbf{Q}_1$: How does the runtime of our method compare to the existing ASP-based sequence mining models?
- $\mathbf{Q}_2$: What is the runtime gap between the specialized mining languages such as dominance programming and our method?

- $\mathbf{Q}_3$: What is the influence of local constraints on the runtime of our method?
- $\mathbf{Q}_4$: How does the choice of an ASP solver influence the overall performance of the system?
- $\mathbf{Q}_5$: How does the hybrid system perform on highly structured graph mining task in comparison to other logic-based mining systems?
- $\mathbf{Q}_6$: How does the hybrid method perform on the approximate tiling problem?

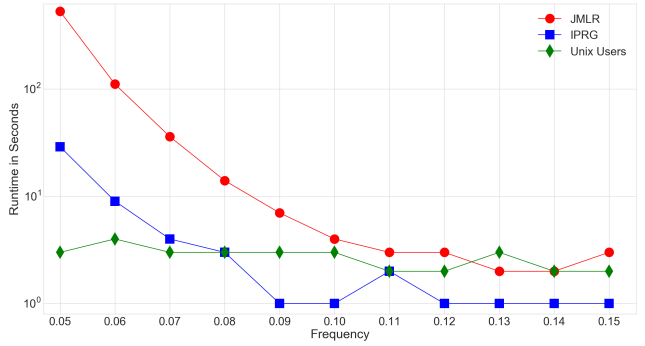
For $\mathbf{Q}_1$ we compare our work with the ASP-based model by Gebser et al. (2016). For $\mathbf{Q}_2$ we measure the runtime difference between specialized itemset mining languages (Négrevergne et al. 2013) and our ASP-based model. To address $\mathbf{Q}_3$ we estimate the runtime effect of adding local constraints. Moreover, for $\mathbf{Q}_4$ we compare the performance of our system with two state-of-the-art ASP solvers: Clasp and WASP. Regarding $\mathbf{Q}_5$, we perform the comparison of our hybrid mining system against the existing ILP-based method for graph mining. Finally, to tackle $\mathbf{Q}_6$ as a proof of concept we test the effectiveness of our hybrid method for solving the problem from Def. 6.4.1.

We report our evaluation results on 2 transaction datasets[3]: *Mushrooms* (8124 transactions/119 items) and *Vote* (435/48), 3 sequence datasets (full)[4]: *JMLR* (788 sequences/3847 distinct symbols), *Unix Users* (9099/4093), and *iPRG* (8628/21), 3 graph datasets[5]: *Yoshida* (265 graphs/20 avg. vertices/23 avg. edges/9 distinct labels), *Nctrer* (232/19/20/9) and *Bloodbarr*(413/21/23/9), and 2 datasets for the tiling problem: *Divorce*[6] (50 transactions/9 items) and *Glass*[7] (214/48). All experiments have been performed on a desktop with Ubuntu 14.04, 64-bit environment, Intel Core i5-3570 4xCPU 3.40GHz and 8GB memory using clingo 4.5.4 [8] C++14 for the itemset/sequence wrapper and python 2.7 for the graph wrapper. In the evaluation we used the latest available version 2 of WASP[9]. The timeout was set to one hour. Free pattern mining demonstrates the same runtime behavior as closed, due to the symmetric encoding, and is thus omitted.

[3] From https://dtai.cs.kuleuven.be/CP4IM/datasets/.
[4] From https://dtai.cs.kuleuven.be/CP4IM/cpsm/datasets.html.
[5] From https://github.com/amaunz/ofsdata
[6] https://sparse.tamu.edu/Pajek/divorce
[7] http://cgi.csc.liv.ac.uk/~frans/KDD/Software/LUCS-KDD-DN/DataSets/dataSets.html
[8] http://potassco.sourceforge.net
[9] https://github.com/alviano/wasp



(a) Comparing with ASP sequence model (Gebser et al. 2016) on the 200 generated sequences (closed)



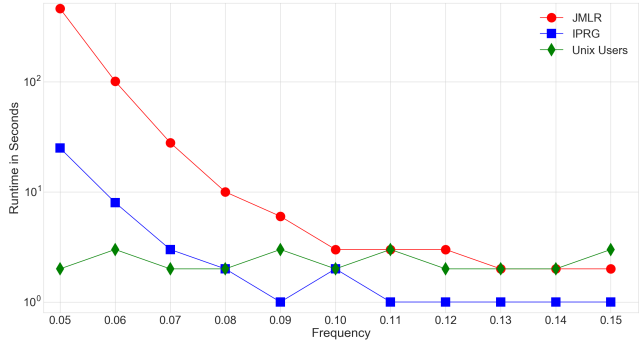
(b) Maximal sequence patterns

Figure 6.4: Investigating $\mathbf{Q}_1$: comparison with pure ASP model (6.4a) and maximal (6.4b) sequence mining on JMLR, Unix Users, and iPRG datasets.

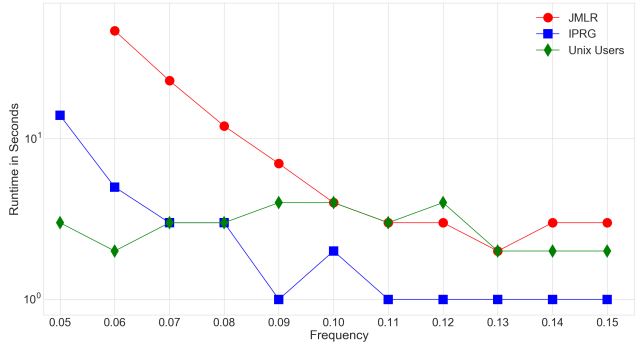
To investigate $\mathbf{Q}_1$, in Figure 6.4a, we compare the ASP model (Gebser et al. 2016) with our method on the default 200 sequence sample, generated by the tool[10] of Gebser et al. (ibid.). We performed the comparison on the synthetic data, as the sequence-mining model (ibid.) failed to compute condensed representations on any of the standard sequence datasets for any support threshold value within the timeout. One can observe that our method consistently outperforms the purely declarative approach of Gebser et al. (ibid.) and the advantage naturally becomes more apparent for smaller frequency threshold values.

In Figs. 6.4b, 6.5a and 6.5b (the point 0.05 for JMLR is a timeout), we present the runtimes of our method for *maximal, closed* and *skyline* sequential pattern mining settings on JMRL, Unix Users and iPRG datasets. In contrast to Gebser

[10] https://sites.google.com/site/aspseqmining



(a) Closed sequence patterns



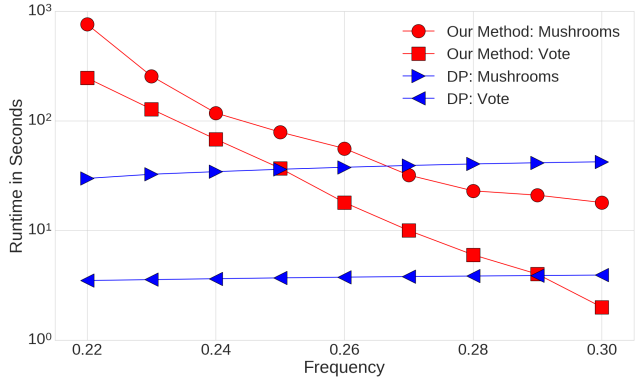
(b) Skyline sequence patterns

Figure 6.5: Investigating $\mathbf{Q}_1$: closed (6.5a), and skyline (6.5b) sequence mining on JMLR, Unix Users, and iPRG datasets.

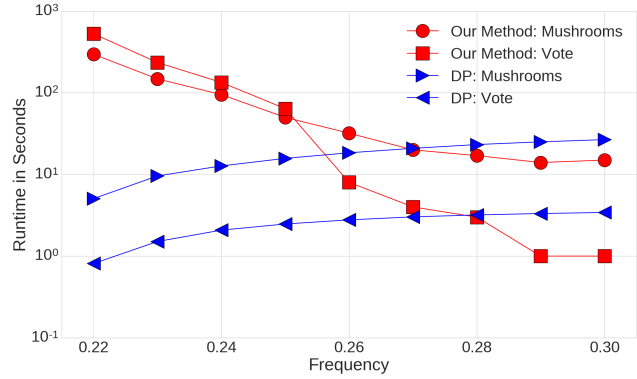
et al. (ibid.), our method managed to produce results on all of these datasets for reasonable threshold values within a couple of minutes.

To investigate $\mathbf{Q}_2$, we compare out-of-the-box performance of DP (Négrevergne et al. 2013) with our approach on maximal, closed and skyline itemset mining problems using standard datasets Vote and Mushrooms. As we see in Figs. 6.6a and 6.6b, on average, DP is one-to-two orders of magnitude faster; this gap is, however, diminishing as the minimum frequency increases. Surprisingly, our approach is significantly faster than DP out-of-the-box for skyline patterns (Figure 6.7a); this holds also for the Mushrooms dataset, not presented here.

Fine-tuning parameters of DP by changing the order in which operators are applied within the system (skyline+ option) allowed to close this gap. With this adaptation DP demonstrates one-to-two orders of magnitude better performance,



(a) Maximal itemset mining: comparing with DP on Vote and Mushrooms



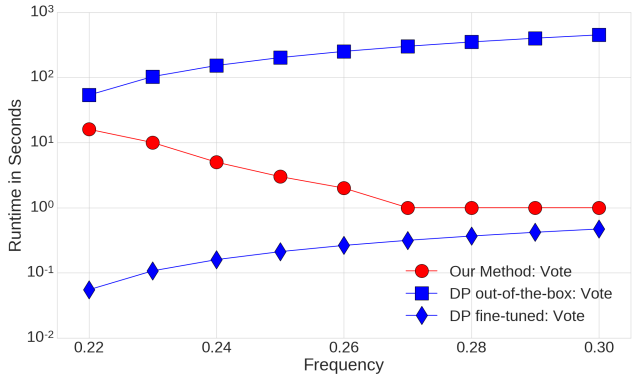
(b) Closed itemset mining: comparing with DP on Vote and Mushrooms

Figure 6.6: Investigating $\mathbf{Q}_2$: comparison with DP (Négrevergne et al. 2013) (6.6b, 6.6a

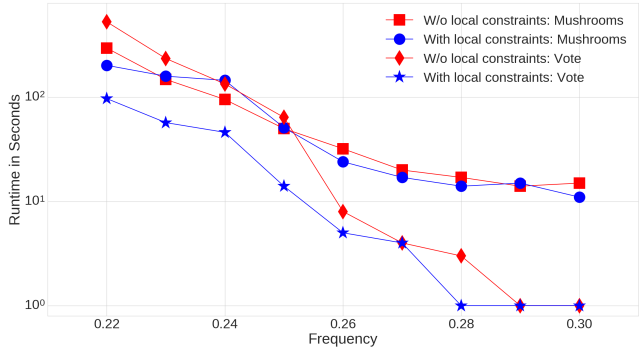
as can be seen in Figure 6.7a. However, fine-tuning such a system requires the understanding of its inner mechanisms or exhaustive application of all available options.

To address $\mathbf{Q}_3$ we introduced three simple local constraints for the itemset mining setting from $\mathbf{Q}_2$: two size constraints $size(I) > 2$ and $size(I) < 7$ and a cost constraint: each item gets weight equal to its value with the maximal budget of n, which is set to 20 in the experiments.

In Figure 6.7b, we present the results for closed itemset mining with and without local constraints (experiments with other global constraints demonstrate a similar runtime pattern). Local constraints ensure better propagation and speed



(a) Skyline itemset mining: comparing with out-of-the-box and fine-tuned DP on Vote

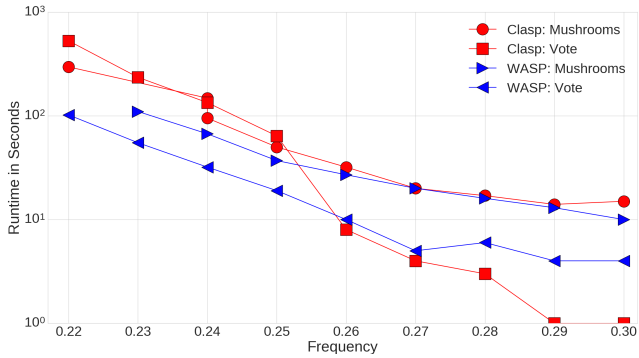


(b) Closed itemset mining: our method with (w/o) local constraints on Vote and Mushrooms

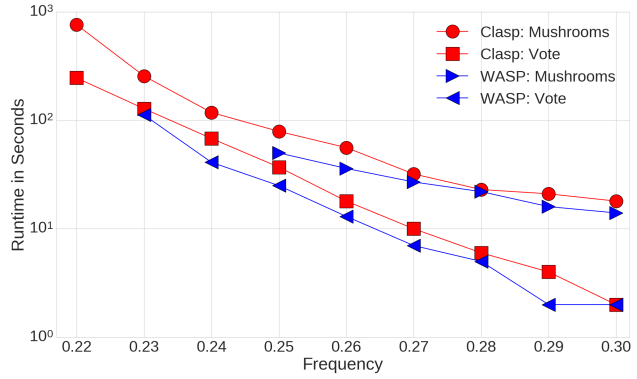
Figure 6.7: Investigating $\mathbf{Q}_2$: comparison with DP (Négrevergne et al. 2013) (6.7a); and $\mathbf{Q}_3$: the effect of local constraints on runtime (6.7b)

up the search. One of the key design features of our encoding is the filtering technique used to select candidate patterns among only valid patterns. Its effect can be clearly seen, e.g., for the Vote dataset in Figure 6.7b, where for certain frequencies the runtime gap is close to an order of magnitude.

To analyze $\mathbf{Q}_4$, we have replaced Clasp solver with WASP (Alviano et al. 2013) in our system. As we see in Figure 6.8a, for mining closed itemsets two systems perform on par except for a timeout of WASP on mushrooms with frequency 0.22. However, we already can observe significant difference in performance on maximal patterns for both itemsets and sequences. A similar behavior can be seen in the setting of closed sequences mining: there is at least an order of magnitude difference in performance. The runtime gap cannot be explained by



(a) Itemset mining: closed patterns

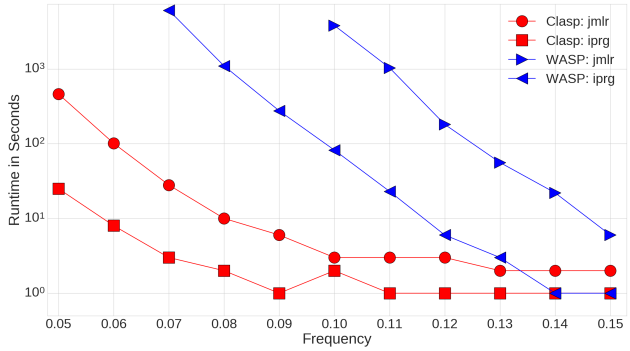


(b) Itemset mining: maximal patterns

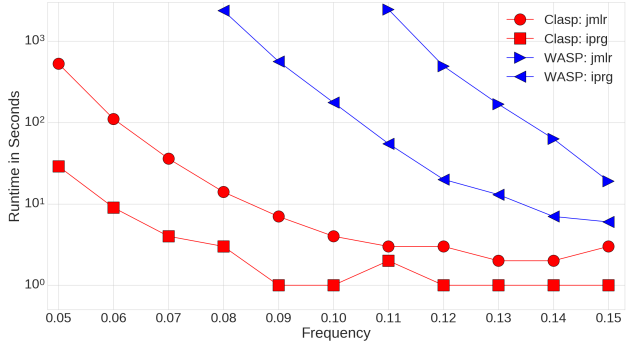
Figure 6.8: $\mathbf{Q}_4$: comparing ASP solvers performance – WASP and Clasp on itemset and sequence mining tasks

differences in grounding, since for both tasks Gringo has been used. However, we have noticed a difference in memory management: Clasp seems to be able to reason and keep track of significantly larger sets of patterns, while being economic and provident with memory. Since the task naturally allows one to generate problem instances of practically any size, we presume that our setting might be well-suited as one of the tests for ASP solvers' performance.

To address $\mathbf{Q}_5$, we consider the standard graph mining datasets in Figs. 6.10a, 6.10b and 6.11a. One can observe, that our system is indeed able to handle real-sized datasets and process hundreds or even thousands of graph patterns within the time-limit. Furthermore, the comparison with the purely declarative framework presented in Figure 6.11b demonstrates the two orders of magnitude speedup in runtime. In addition, note that our system is able to enumerate all



(a) Sequence mining: closed patterns

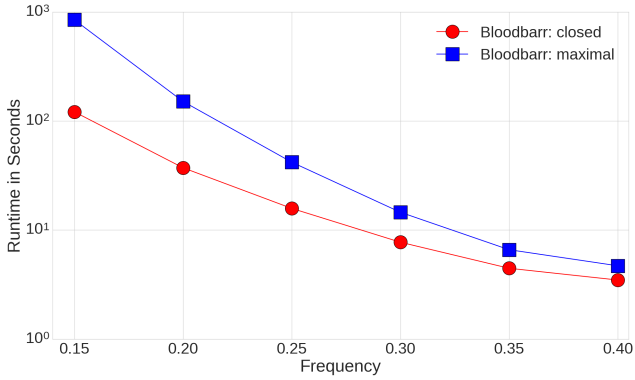


(b) Sequence mining: maximal patterns

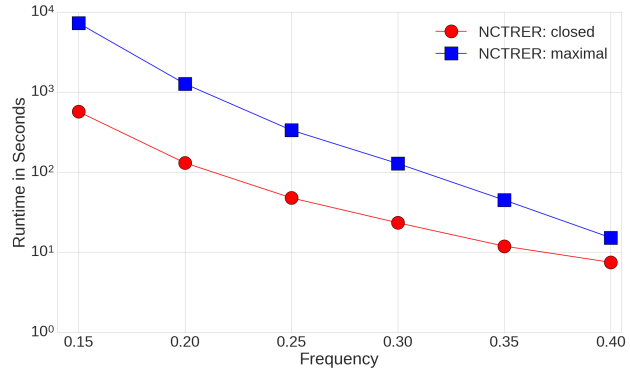
Figure 6.9: $\mathbf{Q}_4$: comparing ASP solvers performance – WASP and Clasp on itemset and sequence mining tasks

condensed graph patterns, while the purely declarative approach is not capable of doing that within the timeout.

To investigate $\mathbf{Q}_6$, we tested our hybrid approach on the approximate tile selection problem from Def. 6.4.1, exploiting Clasp reasoner for the ASP part of the algorithm. Since the considered problem is very computationally demanding, we did not set any timeout in this experimental setting. In Table 6.4 and Table 6.5 we report the results for the Divorce and Glass datasets respectively. We compute the candidate tiles using an approach based on association confidences (see Miettinen et al. (2008)). Given the set of candidate tiles, we use our ASP encoding from Listing 6.7 to find all tilings whose overall error is below a specified threshold. We compare this approach to a greedy method for finding *any* tiling that has error below the threshold. The greedy



(a) Bloodbarr dataset: maximal and closed graph patterns

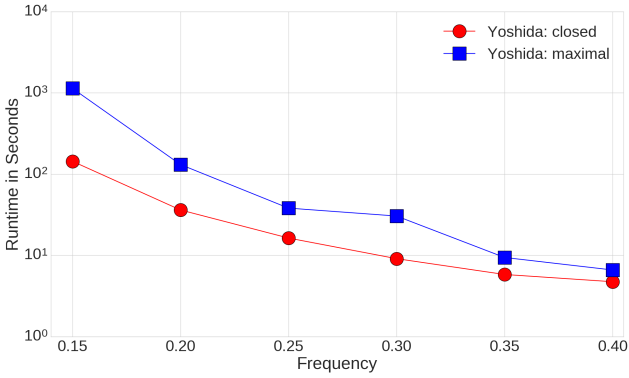


(b) Nctrer dataset: maximal and closed graph patterns

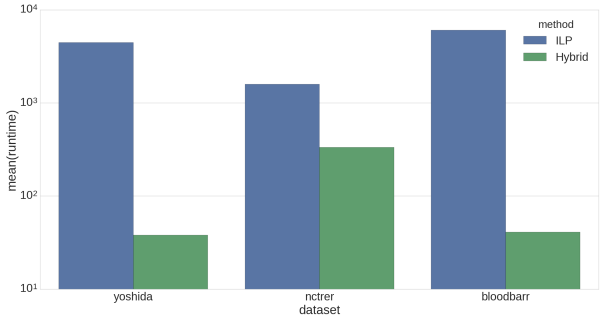
Figure 6.10: Investigating $\mathbf{Q}_5$: evaluation on standard graph datasets. Log-scale

tiling approach, proposed by Miettinen et al. (ibid.), will add the candidate tiles one-by-one until it has either found an admissible tiling, or it has exhausted all tiles. Note that, since the problem of finding the tiling is NP-hard, the greedy method is not guaranteed to find an admissible tiling even if one exists.

The first column in Table 6.4 (respectively Table 6.5) reports the number n of candidate tiles and the error threshold value σ. In the second column we present the details of the solution found by the greedy method, i.e., the number k of tiles in the solution tiling and its overall error. The computation of the tilings by the greedy algorithm takes less than a second. In the third column, the results of our hybrid approach are provided. More specifically, we present the number k of tiles and the overall error for the first solution found by the solver together with the running time, as well as the details of the optimal tiling.



(a) Yoshida dataset: maximal and closed graph patterns



(b) Comparison with logic programming ILP framework (Paramonov et al. 2015b). For the ILP framework time to find *a maximal solution* is indicated, while for our system time to enumerate all maximal graph patterns (frequency 0.2) is shown.

Figure 6.11: Investigating $\mathbf{Q}_5$: evaluation on standard graph datasets and comparison to the declarative ILP system for graph mining. Log-scale

To compute the optimal tiling we enforce the solver to find all models of the given ASP program (their total number is likewise reported).

The candidate tiles are computed by the dedicated algorithm within a fraction of a second, and the major computational efforts are done by the ASP solver to find the final tiling. Moreover, observe that since the grounding step is the most time-consuming, the time difference between the first found solution and all solutions including the optimal one is actually neglectable. Note that comparing the running time of the greedy and our hybrid algorithms is not entirely fair, as the former is not complete (i.e., it might not find any tiling, satisfying the conditions imposed by the error threshold even if one exists),

n/σ	greedy		hybrid						
	k	error	first			k opt	error opt	time all	|sols|
			k	error	time				
3/59	1	51	1	59	38.961	1	51	38.962	5
4/54	1	51	2	51	38.127	1	51	38.135	8
5/60	2	51	2	58	50.616	2	47	50.617	15
6/60	2	51	2	58	50.600	2	47	50.601	30

Table 6.4: Addressing $\mathbf{Q}_6$: approximate tiling problem solution using specialized native algorithm for tile candidate generation and ASP encoding for selecting the resulting tiling (Divorce dataset)

which is in contrast to our hybrid approach.

The greedy algorithm is capable of computing the optimal tiling only for small instances of the Divorce dataset. Starting from $n = 5$, our hybrid method outperforms the greedy one with respect to the quality of the found solutions. For the Glass dataset the benefit of our method is apparent even for smaller instances.

Summary. In all experiments, Step 1 of our method contributes to less than 5% of runtime. Overall, our approach can handle real world datasets for sequential pattern mining as demonstrated in $\mathbf{Q}_1$. In many cases its performance is close to the specialized mining languages, as shown in $\mathbf{Q}_2$. As demonstrated in $\mathbf{Q}_3$ various local constraints can be effectively incorporated into our encoding bringing additional performance benefits. The choice of an ASP solver plays a crucial role in the overall performance of the system, as discussed in $\mathbf{Q}_4$. In $\mathbf{Q}_5$, it has been established that our approach leads to a significant speed up (of orders of magnitude) for the mining tasks with a complex structure, such as graph mining. Finally, the results of $\mathbf{Q}_6$ have proved the applicability of our hybrid framework for other computationally intensive data mining tasks, e.g., approximate tile selection problem, where our method is able to find solutions of higher quality.

6.6 Related work

Pattern mining approaches, especially frequent (closed or maximal) itemset, sequence, and subgraph mining are amongst the foundational methods in data

n/σ	greedy		hybrid						
	k	error	first			k opt	error opt	time all	|sols|
			k	error	time				
3/214	1	184	1	196	294.172	3	88	294.173	7
4/168	3	142	2	154	1194.088	3	142	1194.093	2
5/222	3	180	3	192	2126.944	4	142	2126.948	11
6/224	3	182	3	194	2198.739	5	142	2198.743	25

Table 6.5: Addressing $\mathbf{Q}_6$: approximate tiling problem solution using specialized native algorithm for tile candidate generation and ASP encoding for selecting the resulting tiling (Glass dataset)

mining (see, e.g., Aggarwal (2015) for a recent textbook on the topic), and have been studied actively since the mid-nineties (Agrawal et al. 1993, 1996). These problems are considered local and exhaustive, as the goal is always to enumerate all patterns that satisfy the (local) frequency constraint (together with some global constraints, such as closedness). In addition, the patterns have to be exact, that is, they have to be present in the data. The exactness was relaxed in later work (Pensa and Boulicaut 2005), while Geerts et al. (2004) considered the problem of summarizing the data using closed itemsests, that is, tiling. These two approaches, non-exact patterns and summarization using them, were combined by Miettinen et al. (2008) in their work on Boolean matrix factorization.

The problem of enhancing pattern mining by injecting various user-specified constraints has recently gained increasing attention. On the one hand, optimized dedicated approaches exist, in which some of the constraints are deeply integrated into the mining algorithm (e.g., Pei and Han (2000)). On the other hand, declarative methods based on Constraint Programming (Métivier et al. 2013; Négrevergne and Guns 2015; Rojas et al. 2014), SAT solving (Jabbour et al. 2013, 2015) and ASP (Gebser et al. 2016; Guyet et al. 2014; Järvisalo 2011) have been proposed.

Techniques from the last group are the closest to our work. However, in contrast to our method, they typically focus only on one particular pattern type and consider local constraints and condensed representations in isolation (Pei et al. 2000; Yan et al. 2003). Négrevergne et al. (2013) and Guns et al. (2017) focused on CP-based rather than ASP-based itemset mining and did not take into account sequences like we do. Gebser et al. (2016) studied declarative sequence mining with ASP, but in contrast to our approach, optimized algorithms for

frequent pattern discovery are not exploited in their method. A theoretical framework for structured pattern mining was proposed by Guns et al. (2016), whose main goal was to formally define the core components of the main mining tasks and compare dedicated mining algorithms to their declarative versions. While generic, this work did not take into account local and global constraints and neither has it been implemented.

Järvisalo (2011) and Gebser et al. (2016) considered purely declarative ASP methods; unlike our approach, they do not admit integration of optimized mining algorithms and thus lack practicality. In fact, the need for such an integration in the context of complex structured mining was even explicitly stated by Paramonov et al. (2015b) and by Hallen et al. (2016), which study formalizations of graph mining problems using logical means. While the ASP (Gebser et al. 2016) and CP models (Négrevergne and Guns 2015) for frequent pattern mining cannot be hybridized completely out-of-the-box, due to dependencies between constraints and assumptions on the input-output structure, in principle, as we see from our work that it is possible to re-use ideas and the general modelling approach to turn them into hybrid models to reach more practical performance, for example, our ASP model for sequence mining is inspired by the non-hybrid ASP model of Gebser et al. (2016). The main observation here is that typically these systems have a number of constraint groups: to generate patterns, check patterns validity and perform dominance or group-property checks. Our hybridization idea suggests to replace one of these groups with a highly optimized solver, while keeping the other groups (with minor changes to adapt for the input-output structure) in the model to guarantee generality.

6.7 Conclusion

We have presented a hybrid approach for condensed itemset, sequence and graph mining, which uses optimized dedicated algorithms to determine the frequent patterns and post-filters them using a declarative ASP program. The idea of exploiting ASP for pattern mining is not new; it was studied for both itemsets and sequences. However, unlike previous methods we made steps towards optimizing the declarative techniques by making use of the existing specialized methods and also integrated the dominance programming machinery in our implementation to allow for combining local and global constraints on a generic level. Moreover, using the example of the approximate tile selection problem, we have demonstrated that our hybrid method can be further generalized to other data mining tasks.

One of the possible future directions is to extend the proposed approach to

an iterative technique, where dedicated data mining and declarative methods are interlinked and applied in an alternating fashion. More specifically, all constraints can be split into two parts: those that can be effectively handled using declarative means and those for which specialized algorithms are much more scalable. Answer set programs with external computations (Eiter et al. 2009b) could be possibly exploited in this mining context.

Chapter 7

Conclusions and future work

This chapter summarizes the thesis and discusses open problems and future work opportunities.

7.1 Summary and conclusions

Purely declarative machine learning and data mining, the holy grail of data analysis, is yet a long way ahead of us but today we have made a step.

First, we have introduced two types of data used in data mining: propositional and relational data. We have contrasted their differences and their use in various problems. Namely, we have indicated that propositional data is common for predictive models (that we do not investigate in this thesis), while relational data is common for problems like constraint learning or pattern mining. We have also demonstrated how the Boolean Factorization problem can be generalized to Relational Data Factorization, once we consider relational data. As we have demonstrated these problems can be clearly seen as relational data mining problems and require first order languages for modelling.

More specifically, we have showed that the problems such as tabular constraint learning are relational in its nature due to the data used in it and due to the constraints connecting multiple tables and breaking single-tuple single-table representation, when rows and columns are no longer independent.

To solve these relational problems we have used a declarative approach, where one specifies what the problem is and not how to solve it. For the problems we

have described this declarative approach is natural, since many data mining problems are combinatorial search problems, i.e., they can be phrased in terms of "Model + Search". To solve these relational models under a variety of constraints, we used general solvers based on constraint and logic programing such as Answer Set Programming and FO(.) IDP. Since logic is designed to work with relations and both ASP and IDP support first order statements, they both allow for a natural representation of relational data mining models and data associated with them. The experimental evidence, presented here, shows the efficiency of these systems in solving relational data mining problems.

Furthermore, as we have seen, relational data is so widespread that it is impossible to ignore. This implies that relational mining problems would affect many industries, where, for example, spreadsheets are an essential part of business processes, as in finance or auditing. This gives us confidence not only in that relational problems are a crucial part of data mining but also in that we should investigate applications of general solvers, such as ASP and IDP, for data mining and machine learning. In fact, it provides the solver developer community with important benchmarks and problems to improve solvers. The inverse process is also happening, once solvers get better, new types of problems can be tackled.

7.2 Contributions and key questions

In the introduction chapter we have identified the key research question and key problems we were going to apply our approach to. Now we are going to review these applications and to analyze the outcome.

$\mathcal{Q}_1$ ***Relational Data Factorization.*** *What are the challenges and advantages of generalizing classic data mining problems, such as the Boolean Matrix Factorization problem, into the relation setting?*

As we see in Chapter 3, the key advantage is the generality and flexibility of the approach, however, it often comes at the cost of longer runtimes of the method. There are a number of ways as has been indicated to handle this problem but it should not come as a surprise that general methods cannot compete the out-of-the-box in terms of the efficiency with a specialized method devoted to only one particular problem. It is also clear that specialized methods must be significantly modified in order to be used in a new setting. Our extensive experimental study also demonstrates that general relational methods are well-suited for system prototyping.

Namely, we have demonstrated that not only tiling and Boolean matrix factorization can be solved declaratively using ASP, but also factorizations with multiple tables under constraints, which allowed us to visualize relational groups of co-authors of a number of known data mining researchers. Finally, this showed that a broad variety of mining problems can be modelled using relational factorization and solved using modern logic programming techniques.

$\mathcal{Q}_2$ ***Tabular Constraint Learning.*** *How can the constraint learning problem be formulated and modelled in the relational setting, where data is represented as spreadsheets, i.e., connected relations, and constraints are tabular Excel-like formulae?*

As has been shown in Chapter 4, the problems related to spreadsheet data are intrinsically relational, since the data itself consists of multiple relations together with the dependencies between them. We have introduced the system called Tabular Constraint Learner, based on classic ILP ideas, to work in this setting. The systems enumerates the Excel-like constraints by working on them in the same manner as clause-learning systems do. It is able to detect the most popular Excel functions in spreadsheet data within a table as well as between multiple tables. This indicates Tabular Constraint Learner's applicability in practice, potentially, it can be turned into a plugin for Excel or Google Spreadsheets, which can help a broad audience of users.

Furthermore, we have introduced the first tabular constraint dataset and provided an experimental evidence on the efficiency of the TaCLe algorithm. Also, we have studied limitations of the method such as sensitivity to noise, nested constraints and richer type systems. As a result, we have proposed a modification of the existing system, a second generation TaCLe system, that would potentially overcome these issues.

$\mathcal{Q}_3$ ***ASP Sketching.*** *How can the problem of ASP Sketching, where one marks certain parts of the program to be uncertain, be modelled and solved using our declarative approach?*

As we have demonstrated in Chapter 5, given a partially complete specification of an ASP program, ASP can successfully learn relational programs based on a few examples provided. We have extensively analyzed the learned programs and discovered that on average only few, typically between 2-5, examples are needed to converge to a correct solution. Furthermore, we were able to detect errors in publicly available encodings, we have contacted the authors and the models were updated.

The general rewriting schema we have proposed is general and not limited to the ASP used in the chapter for solving. The same algorithm and schema can be applied to Constraint Programming or FO(.) IDP. In fact, this schema also allows for developing some other practical applications such as spreadsheet formula sketching.

$\mathcal{Q}_4$ ***ASP-based Relational Pattern Mining.*** *How can we apply declarative relational mining techniques, such as logic programming, to relational pattern mining, such as sequence, graph and query mining?*

As indicated in Chapter 6, we have applied logic programming methods to the structured mining problems. As the result, a purely declarative model of graph mining has been created. What we have discovered is that the systems cannot yet handle this type of models efficiently. Then, to overcome this issue, we have proposed a hybrid model that first applies a standard pattern mining algorithm and then uses a logic programming engine. This has led to an order of magnitude efficiency jump and, from the practical point of view, it seems to be a fruitful direction of future research in declarative pattern mining.

7.3 Discussion and future work

Let us introduce possibilities of future work here. Each chapter describes the future work specific for a particular area or a problem, while in this chapter we describe possible future directions from the perspective of the thesis as a whole.

Relational data mining in future years As indicated in Chapter 6, hybridization of declarative models is a promising and fruitful research direction. It allows one to combine highly efficient specialized algorithms with general solvers to create, on the one hand, efficient and scalable systems; and on the other hand, general systems that can handle not only particular restricted problems but suitable for the whole class of pattern mining problems. Similar to the dominance programming for itemset mining (Negrevergne et al. 2013) and our hybridization approach to structured pattern mining (Paramonov et al. 2017a), we can foresee a creation of general mining languages that handle various types of data and multiple types of condensed representations, while demonstrating close to state-of-art performance. These languages would allow a user to specify all needed building blocks to generate an executable model that can find all condensed patterns within reasonable time.

On the relational learning side, inspired by the results of TaCLe (Kolb et al. 2017) in Chapter 4 and Flashfill (Gulwani 2011), we envision an intelligent assistant that can help the user by analyzing tabular data in a spreadsheet and suggest possible cleaning transformations, usage of Excel-like formulae and data compression. The later idea of data compression, which can be seen as a form of cleaning, connects to the ideas of Relational Factorization in Chapter 3. Similarly, as we can think of *Spreadsheet Formula Sketching*, where the assistant can allow the user to specify some of the formulae or constraints, while leaving certain parts open. Then, the system can infer possible substitutions from the given tabular data.

Furthermore, the development of applications would drive the development of the solvers as well. We can imagine the support of sketching techniques from Chapter 5 being implemented on the solver level directly. Richer constraint languages might come into the learning process, when a model satisfies a negative example and the system might suggest a new constraint that the model is missing or find the largest set of constraints consistent with the examples.

Finally, we can imagine a fully automated tool able to analyze and clean data in a standard format, such as a spreadsheet. Such a system can work in the unsupervised setting and would not require user interaction or only to check if a suggestion looks ok, or which transformation looks better. We regard a creation of such a tool as a long term research direction for the future developments of the ideas, tools and models, presented in this thesis.

Appendix A

Appendix

A.1 ReDF additional materials

A.1.1 Correctness proof of encoding 3.10

We present the proof of Theorem 3.1 (Section 3.5).

Theorem (Correctness of the ASP tiling encoding)**.** The ASP program $\mathcal{P}$ defined by the listing 3.10 computes the k-th largest tile w.r.t. the scoring function `coverage` (3.5) as extensions of the predicates $tile(k, \cdot, \cdot)$ and $in(k, \cdot)$ in its answer set $\mathcal{A}$, provided that the dataset is represented extensionally through the predicates *db*, *valid*, and *col* and the $k-1$ already found tiles are represented extensionally through the predicates $tile(I, \cdot, \cdot)$ and $in(I, \cdot)$ for $I \in \overline{1, k-1}$.

Proof. We prove the theorem by means of mathematical induction. The encoding here implicitly refers to the encoding in Listing 3.10.

We first prove the statement for the case when $k = 1$, i.e., the program $\mathcal{P}$ computes the first largest tile by the coverage function in Eq. 3.5.

Step 1: k = 1

First of all, the answer set $\mathcal{A}$ always exists, since there is no cyclic dependency involving negation and an "empty" answer set w.r.t. to the predicates *tile* and *in* can be selected as a possible answer set.

Secondly, observe that the extension of the predicate $intersect(\cdot)$ is empty, since there is no other tile computed before and the rule in Line 6 never applies.

Then, let us show that $\mathcal{A}$ satisfies the other two conditions from the constraints Cs and it is the largest tile w.r.t. to coverage.

Assume that the following constraint from Cs is violated

$$\leftarrow \mathit{tile}(\mathit{Indx}, \mathit{Value}_1, \mathit{Attr}), \mathit{tile}(\mathit{Indx}, \mathit{Value}_2, \mathit{Attr}), \mathit{Value}_1 \neq \mathit{Value}_2.$$

Then there exist v_1, v_2, a such that $v_1 \neq v_2$, $\mathit{tile}(1, v_1, a) \in \mathcal{A}$, $\mathit{tile}(1, v_2, a) \in \mathcal{A}$. Then the body of the rule in Line 2 is satisfied, since a is a column in the dataset. Then, the head must be satisfied: v_1 and v_2 are valid values, so we derive that

$$0\ \{\mathit{tile}(1, v_1, a), \mathit{tile}(1, v_2, a)\}\ 1,$$

which contradicts the assumption that $v_1 \neq v_2$, $\mathit{tile}(1, v_1, a) \in \mathcal{A}$, $\mathit{tile}(1, v_2, a) \in \mathcal{A}$. Then, since v_1, v_2, a were fixed but arbitrary the constraint in Eq. 3.2 is satisfied by $\mathcal{A}$.

Assume that the following constraint from Cs is violated by $\mathcal{A}$

$$\leftarrow \mathit{tile}(\mathit{Indx}, \mathit{Value}, \mathit{Attr}), \mathit{in}(\mathit{Indx}, \mathit{Transct}),\ \mathit{not}\ \mathit{db}(\mathit{Value}, \mathit{Attr}, \mathit{Transct}).$$

There exist v, a, t such that $\mathit{tile}(1, v, a) \in \mathcal{A}$ and $\mathit{in}(1, t) \in \mathcal{A}$, but $\mathit{db}(v, a, t) \notin \mathcal{A}$. Then the rule in Line 4 applies, since the body of it satisfied by the assignment $\theta = \{T \mapsto t, \mathit{Value} \mapsto v, \mathit{Attribute} \mapsto a\}$ (and guess is equal to 1). Then, *overcover(t)* $\in \mathcal{A}$. Then the body of the rule in Line 8 is not satisfied and $\mathit{in}(1, t) \notin \mathcal{A}$. Then, we derived contradiction between the facts that $\mathit{in}(1, t) \in \mathcal{A}$ and $\mathit{in}(1, t) \notin \mathcal{A}$, therefore, $\mathcal{A}$ satisfies the constraint in Eq, 3.4.

Assume that $\mathcal{A}$ is not a maximal answer set by the coverage function, than there is an answer set $\mathcal{A}'$ such that it satisfies all the constraints from Cs and the cardinality of the extension of the predicate *covered* is higher than in $\mathcal{A}$. If so, then $\mathcal{A}$ violates the optimization criterion in Line 11. Then, $\mathcal{A}$ is maximal w.r.t. the coverage function.

Step 2: k-1. Assume, we have proven the theorem for the values up to $k - 1$.

Step 3: k. Let us show that theorem holds for the k-th tile, for $k > 1$. The proof is very similar to the reasoning presented above. We assume that $\mathcal{A}$ violates some of the contraints from Cs and demonstrate "reductio ad absurdum". Here we demonstrate the only different reasoning part from the previous case of the step one.

Assume that "intersection" constraint in Eq. 3.3 from Cs is violated:

$$\leftarrow \mathit{in}(I_1, T), \mathit{in}(I_2, T), \mathit{tile}(I_1, V, A), \mathit{tile}(I_2, V, A), I_1 \neq I_2.$$

Then, there exist i, v, t, a such that $i \neq k$ and $\mathit{in}(i, t) \in \mathcal{A}$, $\mathit{in}(k, t) \in \mathcal{A}$, $\mathit{tile}(i, v, a) \in \mathcal{A}$, $\mathit{tile}(k, v, a) \in \mathcal{A}$. Then the body of the rule in Line 6 is satisfied,

hence *intersect(t)* $\in \mathcal{A}$. Then, the body of the "in"-rule in Line 8 is not satisfied and $in(k,t) \notin \mathcal{A}$. We obtain contradiction between two facts: our assumption that $in(k,t) \in \mathcal{A}$ and the conclusion that $in(k,t) \notin \mathcal{A}$. □ □

A.1.2 Additional ASP encodings

Overlapping tiling. Constraint `overlapping-tiles(N)` is encoded in ASP as indicated in Listing A.1 (we indicate only the new lines in the listing). Let us explain how this new constraint works. The rule in Line 1 for a transaction T computes the attributes *Attr* on which the current tile intersects with the previous tiles. The constraint in Line 2 defines the transaction where the current tile cannot occur in the following way: if the number of attributes where the current tile intersect with the other is higher than *overlap_level*, then the tile cannot occur in this transaction.

```
1 intersect_N(T,Attr) :- guess != Indx, tile(currentI, V, Attr), tile(Indx, V, Attr), in(Indx,
      T).
2 intersect(T) :- overlap_level #count{ intersect_N(T,Indx) : col(Indx) }, transaction(T).
```

Listing A.1: Overlapping tiling

We see that a slight change of the formulation of property of a solution leads to a small change in the modelling and also to a small change in the implementation.

BMF. Boolean data is a particular case of attribute-value data, and corresponds to the situation when values can be either true or false. It is straightforward to realize the encoding of tiling for BMF by deleting all the arguments in Listing 3.10 corresponding to values; we encode only the true facts for *db(T,I)*. Also, as described in Subsection 3.3.2, we remove the rules corresponding to the intersection of tiles. The ASP code is in Listing A.2.

```
1 0 { tile(currentI,I) } 1 :- item(I).
2 over_cover(currentI, T) :- not db(T,I), tile(currentI,I), transaction(T).
3 in(currentI,T) :- not over_cover(currentI,T), transaction(T).
4 covered(T,I) :- tile(currentI,I), in(currentI,T), db(T,I).
5 #maximize[covered(T,I)].
```

Listing A.2: Boolean Matrix Factorization

The structure and functions of the constraints is the same as in Listing 3.10. Except for the modification described above, where we remove "value" argument from the predicates in the encoding and non-intersection constraint.

Discriminative k-pattern set mining. In case of discriminate k-pattern set mining, we need to change the optimization criterion and evaluation function to

capture positive and negative transactions. The changed optimization criterion 3.6 can be straightforwardly implemented in ASP, see Listing A.3.

```
in(currentI,T) :− transaction(T), not over_covered(currentI, T).
covered_plus(T)  :− in(Indx,T), tile(Indx, Value, Attribute), positive(T).
covered_minus(T) :− in(Indx,T), tile(Indx, Value, Attribute), negative(T).
#maximize[covered_plus(T) = 1, covered_minus(T) = −α].
```

Listing A.3: Discriminative k-pattern set mining

This encoding distinguishes between the positively and negatively covered transactions, and weights negative covered transactions by α.

Matrix block-diagonal form. The encoding mainly mimics the tiling encoding. We indicate here the constraints that are different in Listing A.4.

```
%generate possible tile using only not used items
0 { tile(currentI,I) } 1 :− item(I), not usedI(I).
%coverage takes into account noise level
over_covered(I,T) :− not db(T,I), code(currentI,I), transaction(T).
too_noisy(T)      :− noise_level #count{ over_covered(I,T) : item(I) }, transaction(T).
covered(T,I) :− in(C,T), code(C,I).
%current tile can occur only in not used transactions
in(currentI,T) :− transaction(T), not too_noisy(T), not usedT(T).
%penalty for used transactions
leftT(T,I) :− in(currentI,T), db(T,I), not covered(T,I).
%penalty for used columns
leftI(T,I) :− code(currentI,I), db(T,I), not covered(T,I).
%optimization takes into account penalties
#maximize[ covered(T,I)=1, leftT(T,I)=α, leftI(T,I)=−β].
```

Listing A.4: Encoding block diagonal form

The constraints are similar to the described above in tiling, overlapping tiling and discriminative k-pattern set mining. Rules in Lines 10 and 12 introduce the notation of "not used" but blocked by the tile transactions and items. In the optimization criterion 14 each of these elements is penalized.

Purely relational factorization with a latent variable. The factorization shape is defined as

$$approx(A, U, V) \leftarrow interestedIn(A, Topic), specializedIn(U, Topic), field(V, Topic).$$

The candidates are generated by the following rules:

```
0 { interestedIn(A,T)  } 1 :− p(A,_,_),topic(T).
0 { specializedIn(U,T) } 1 :− p(A,U,_),interestedIn(A,T),topic(T).
inField(V,T) :− p(A,U,V),interestedIn(A,T),specializedIn(U,T),topic(T).
approx(Author, Uni, Venue) :− interestedIn(Author, G), specializedIn(Uni, G),inField(
      Venue, G).
```

Listing A.5: Generator rules

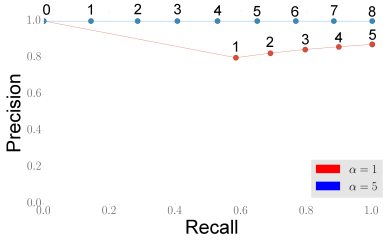
The idea is that we first consider for every author whether he is interested in the topic or not, and then this process is repeated for every university (having a corresponding author interested in the topic), and finally, it is determined whether the venue belongs to the topic based on the relations *interestedIn* and *publishedIn*. The optimization criterion is essentially the same as that of BMF.

```
1 covered(A,U,V) :− approx(A,U,V),     p(A,U,V).
2 overcovered(A,U,V) :− approx(A,U,V), not p(A,U,V).
3 #maximize[covered(A,U,T) = 1, overcovered(A,U,T) = −α].
```

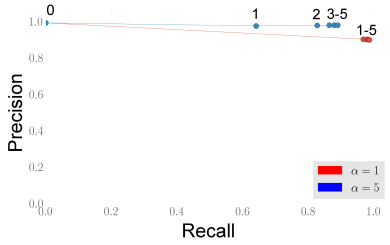
Listing A.6: Optimization criterion

A.1.3 Precision-recall curves

Figure A.1 presents the precision-recall curves corresponding to Figures 3.8 and 3.10. Figure A.1a corresponds to the panel in Figure 3.8 and Figure A.1b corresponds to the panel in Figure 3.10. Note that these precision-recall curves have been obtained by varying the number of rules.



(a) Tic-tac-toe discriminative pattern set mining



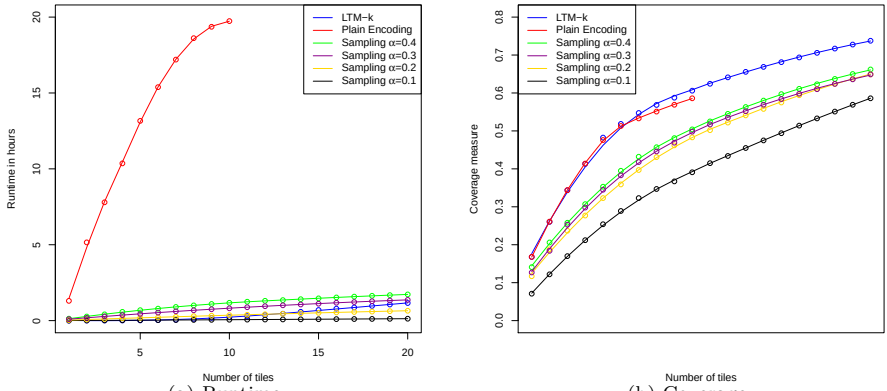
(b) Purely relational discriminative mining

Figure A.1: Precision recall curves for discriminative learning for different α, i.e., for two weights of covering negative atoms. For different number of learned rules, indicated on each point (starting from zero).

A.1.4 Additional tiling experiments with sampling

Here we present the estimation of β given corresponding values of α, as visualized in Figures A.2a and A.2b. The calculation method is described in Appendix

3.9. We computed the values for β in Table A.1 based on the data from Figure A.2b.



(a) Runtime (b) Coverage

Figure A.2: Tiling comparison (runtime, coverage) with LTM-k (Mushroom dataset)

Table A.1: An empiricial reconstruction of β, given the values of α

α	0.1	0.2	0.3	0.4
β	0.55	0.73	0.76	0.79

Bibliography

Aftrati, F., G. Das, A. Gionis, H. Mannila, T. Mielikäinen, and P. Tsaparas (2012). "Mining Chains of Relations". In: *Data Mining: Foundations and Intelligent Paradigms*. Ed. by D. E. Holmes and L. C. Jain. Vol. 24. Intelligent Systems Reference Library. Springer Berlin Heidelberg, pp. 217–246.

Aggarwal, C. C. (2015). *Data Mining: The Textbook*. Cham: Springer.

Agrawal, R., T. Imielinski, and A. Swami (1993). "Mining Association Rules between Sets of Items in Large Databases". In: *SIGMOD '93*, pp. 207–216.

Agrawal, R., H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo (1996). "Fast Discovery of Association Rules". In: *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, pp. 307–328.

Alviano, M., C. Dodaro, W. Faber, N. Leone, and F. Ricca (2013). "WASP: A Native ASP Solver Based on Constraint Learning". In: *LPNMR*, pp. 54–66.

Aoga, J. O. R., T. Guns, and P. Schaus (2016). "An Efficient Algorithm for Mining Frequent Sequence with Constraint Programming". In: *ECML PKDD*, pp. 315–330.

Arimura, H., R. Medina, and J.-M. Petit, eds. (2012). *Proceedings of the IEEE ICDM Workshop on Declarative Pattern Mining*.

Aykanat, C., A. Pinar, and Ü. V. Catalyurek (2002). "Permuting Sparse Rectangular Matrices into Block-Diagonal Form". In: *SIAM Journal on Scientific Computing* 25, pp. 1860–1879.

Bache, K. and M. Lichman (2013). *UCI Machine Learning Repository*. URL: http://archive.ics.uci.edu/ml.

Beldiceanu, N. and H. Simonis (2012). "A Model Seeker: Extracting Global Constraint Models from Positive Examples". In: *Principles and Practice of Constraint Programming*. Ed. by M. Milano. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 141–157.

Berzal, F., J. C. Cubero, F. Cuenca, and J. M. Medina (2002). "Relational Decomposition Through Partial Functional Dependencies". In: *Data Knowl. Eng.* 43.2, pp. 207–234.

Bessiere, C., F. Koriche, N. Lazaar, and B. O'Sullivan (2017a). "Constraint Acquisition". In: *Artif. Intell.* In Press.

Bessiere, C. (2006). "Chapter 3 - Constraint Propagation". In: *Handbook of Constraint Programming*. Ed. by F. Rossi, P. van Beek, and T. Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, pp. 29 –83. DOI: https://doi.org/10.1016/S1574-6526(06)80007-6. URL: http://www.sciencedirect.com/science/article/pii/S1574652606800076.

Bessiere, C., R. Coletta, F. Koriche, and B. O'Sullivan (2005a). "A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems". In: *ECML*, pp. 23–34.

Bessiere, C., R. Coletta, and F. Koriche (2005b). "A SAT-based version space algorithm for acquiring constraint satisfaction problems". In: *ECML*. Springer, pp. 23–34.

Bessiere, C., R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C. Quimper, and T. Walsh (2013). "Constraint Acquisition via Partial Queries". In: *IJCAI*, pp. 475–481.

Bessiere, C., F. Koriche, N. Lazaar, and B. O'Sullivan (2017b). "Constraint acquisition". In: *Artif. Intell.* 244, pp. 315–342.

Bird, R. and S. Curtis (2006). "FUNCTIONAL PEARLS: Finding Celebrities: A Lesson in Functional Programming". In: *J. Funct. Program.* 16.1, pp. 13–20.

Biskup, J., J. Paredaens, T. Schwentick, and J. V. den Bussche (2004). "Solving Equations in the Relational Algebra." In: *SIAM Journal on Computing* 33.5, pp. 1052–1066.

Blockeel, H., B. Bogaerts, M. Bruynooghe, B. De Cat, S. De Pooter, M. Denecker, A. Labarre, J. Ramon, and S. Verwer (2012). "Modeling Machine Learning and Data Mining Problems with FO(·)". In: *International Conference on Logic Programming – Technical Communications*, pp. 14 –25.

Bonchi, F. and C. Lucchese (2006). "On condensed representations of constrained frequent patterns". In: *Knowl. Inf. Syst.* 9.2, pp. 180–201.

Brewka, G., T. Eiter, and M. Truszczyński (2011). "Answer set programming at a glance". In: *Communications of the ACM* 54.12, pp. 92–103.

Brewka, G., J. Delgrande, J. Romero, and T. Schaub (2015). "Asprin: Customizing Answer Set Preferences Without a Headache". In: *AAAI*, pp. 1467–1474.

Broeck, E. Van den and E. Cuypers (2011). *MS Excel 2010*. Uitgeverij De Boeck.

Cabon, B., S. de Givry, L. Lobjois, T. Schiex, and J. Warners (1999). "Radio Link Frequency Assignment". In: *Constraints* 4.1, pp. 79–89.

Chang, M.-W., L.-A. Ratinov, N. Rizzolo, and D. Roth (2008). "Learning and Inference with Constraints". In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*, pp. 1513–1518.

Codd, E. F. (1970). "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6, pp. 377–387.

Colmerauer, A. and P. Roussel (1996). “History of Programming languages—II”. In: ed. by T. J. Bergin Jr. and R. G. Gibson Jr. New York, NY, USA: ACM. Chap. The Birth of Prolog, pp. 331–367.

Date, C. J. (2006). *Date on Database: Writings 2000-2006*. Berkely, CA, USA: Apress.

De Cat, B., J. Jansen, and G. Janssens (2013). “IDP3: Combining symbolic and ground reasoning for model generation”. Workshop on Grounding and Transformations for Theories with Variables, La Coruña, 15 Sept 2013.

De Cat, B., B. Bogaerts, M. Bruynooghe, and M. Denecker (2014). “Predicate Logic as a Modelling Language: The IDP System”. In: *CoRR* abs/1401.6312.

De Raedt, L. (2008a). *Logical and relational learning*. Cognitive Technologies. Springer.

– (2008b). *Logical and Relational Learning: From ILP to MRDM (Cognitive Technologies)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

– (2012). “Declarative Modeling for Machine Learning and Data Mining”. In: *The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, pp. 2–3.

– (2015). “Languages for Learning and Mining”. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. Pp. 4107–4111.

De Raedt, L. and L. Dehaspe (1997). “Clausal Discovery”. In: *Machine Learning* 26.2-3, pp. 99–146.

De Raedt, L., T. Guns, and S. Nijssen (2008). “Constraint Programming for Itemset Mining”. In: *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 204–212.

Denecker, M. and A. Kakas (2002). “Abduction in Logic Programming”. In: *Computational Logic: Logic Programming and Beyond*. Ed. by A. Kakas and F. Sadri. Vol. 2407. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 402–436.

Devlin, J., J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli (2017). “RobustFill: Neural Program Learning under Noisy I/O”. In: *CoRR* abs/1703.07469.

Eiter, T. and A. Polleres (2006). “Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications”. In: *TPLP* 6.1-2, pp. 23–60.

Eiter, T., G. Ianni, and T. Krennwallner (2009a). “Answer Set Programming: A Primer”. In: *5th International Reasoning Web Summer School (RW 2009), Brixen/Bressanone, Italy, August 30 – September 4, 2009*. Vol. 5689. LNCS. Springer.

Eiter, T., G. Brewka, M. Dao-Tran, M. Fink, G. Ianni, and T. Krennwallner (2009b). “Combining Nonmonotonic Knowledge Bases with External Sources”.

In: *Frontiers of Combining Systems, 7th International Symposium, FroCoS Italy, September 16-18*, pp. 18–42.

Elmasri, R. and S. B. Navathe (2010). *Fundamentals of Database Systems, Sixth Edition*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Enderton, H. B. (2001). *A mathematical introduction to logic*. 2nd ed. Harcourt/Academic Press.

Faber, W., G. Pfeifer, and N. Leone (2011). "Semantics and complexity of recursive aggregates in answer set programming". In: *Artificial Intelligence* 175.1. John McCarthy's Legacy, pp. 278 –298. ISSN: 0004-3702. DOI: http://dx.doi.org/10.1016/j.artint.2010.04.002.

Fan, W., F. Geerts, and L. Zheng (2012). "View determinacy for preserving selected information in data transformations." In: *Information Systems* 37.1, pp. 1–12.

Fang, J., P. Mitra, Z. Tang, and C. L. Giles (2012). "Table Header Detection and Classification". In: *AAAI*.

Feige, U. (1996). "A Threshold of Ln N for Approximating Set Cover". In: *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*. STOC '96. New York, NY, USA: ACM, pp. 314–318.

Flach, P. A. and I. Savnik (1999). "Database Dependency Discovery: A Machine Learning Approach". In: *AI Commun.* 12.3, pp. 139–160.

Flach, P. and A. Kakas (2000). In: Handbook of Defeasible Reasoning and Uncertainty Management Systems 4. Ed. by D. Gabbay and R. Kruse, pp. 1–33.

Frisch, A. M., W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel (2008). "Essence: A constraint language for specifying combinatorial problems". In: *Constraints* 13.3, pp. 268–306.

Gebser, M., R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller. "A Portfolio Solver for Answer Set Programming: Preliminary Report". In: pp. 352–357.

Gebser, M., R. Kaminski, B. Kaufmann, and T. Schaub (2012). *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

Gebser, M., B. Kaufmann, A. Neumann, and T. Schaub (2007). "*clasp* : A Conflict-Driven Answer Set Solver". In: *LPNMR*, pp. 260–265.

Gebser, M., J. Pührer, T. Schaub, and H. Tompits (2008). "A Meta-programming Technique for Debugging Answer-set Programs". In: *AAAI*. AAAI Press, pp. 448–453.

Gebser, M., B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider (2011). "Potassco: The Potsdam answer set solving collection". In: *AI Communications* 24.2, pp. 107–124.

Gebser, M., B. Kaufmann, J. Romero, R. Otero, T. Schaub, and P. Wanko (2013). "Domain-Specific Heuristics in Answer Set Programming." In: *Association*

for the Advancement of Artificial Intelligence. Ed. by M. desJardins and M. L. Littman. AAAI Press.

Gebser, M., T. Guyet, R. Quiniou, J. Romero, and T. Schaub (2016). "Knowledge-Based Sequence Mining with ASP". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*.

Geerts, F., B. Goethals, and T. Mielikäinen (2004). "Tiling Databases". In: *Discovery Science*. Springer, pp. 278–289.

Gelfond, M. and V. Lifschitz (1988). "The Stable Model Semantics for Logic Programming". In: *Proc. of ICLP/SLP*, pp. 1070–1080.

Gent, I. P., C. Jefferson, and P. Nightingale (2017). "Complexity of n-Queens Completion". In: *J. Artif. Intell. Res.* 59, pp. 815–848.

Golub, G. H. and C. F. Van Loan (1996). *Matrix Computations (3rd Ed.)* Baltimore, MD, USA: Johns Hopkins University Press.

Gopalan, P. K. and D. M. Blei (2013). "Efficient discovery of overlapping communities in massive networks". In: *Proceedings of the National Academy of Sciences* 110.36, pp. 14534–14539.

Gordon, A. D., T. Graepel, N. Rolland, C. Russo, J. Borgstrom, and J. Guiver (2013). *Tabular: A Schema-Driven Probabilistic Programming Language*. Tech. rep. MSR-TR-2013-118. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=204661.

Gulwani, S. (2011). "Automating String Processing in Spreadsheets Using Input-output Examples". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL. New York, NY, USA: ACM, pp. 317–330.

Gulwani, S., J. Hernandez-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn (2015). "Inductive programming meets the real world". In: *Commun. ACM* 58.11, pp. 90–99.

Guns, M. (2012). "Declarative Pattern Mining using Constraint Programming (Een declaratieve aanpak tot pattern mining door middel van constraint programming)". De Raedt, Luc (supervisor), Nijssen, Siegfried (cosupervisor). PhD thesis. Informatics Section, Department of Computer Science, Faculty of Engineering Science, p. 220. URL: https://lirias.kuleuven.be/handle/123456789/327778.

Guns, T., S. Nijssen, and L. De Raedt (2011). "Itemset mining: A constraint programming perspective". In: *Artif. Intell.* 175.12-13, pp. 1951–1983.

– (2013a). "k-Pattern set mining under constraints". In: *Knowledge and Data Engineering, IEEE Transactions on* 25.2, pp. 402–418.

– (2013b). "k-Pattern Set Mining under Constraints". In: *IEEE Transactions on Knowledge and Data Engineering* 25.2, pp. 402–418.

Guns, T., A. Dries, G. Tack, S. Nijssen, and L. De Raedt (2013c). "MiningZinc: A Modeling Language for Constraint-based Mining". In: *Proceedings of the*

Twenty-Third International Joint Conference on Artificial Intelligence. Beijing, China, pp. 1365–1372.

Guns, T., S. Paramonov, and B. Négrevergne (2016). "On Declarative Modeling of Structured Pattern Mining". In: *Declarative Learning Based Programming, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 13, 2016.*

Guns, T., A. Dries, S. Nijssen, G. Tack, and L. De Raedt (2017). "MiningZinc: A declarative framework for constraint-based mining". In: *Artif. Intell.* 244, pp. 6–29.

Guyet, T., Y. Moinard, and R. Quiniou (2014). "Using Answer Set Programming for pattern mining". In: *CoRR* abs/1409.7777.

Hallen, M. van der, S. Paramonov, M. Leuschel, and G. Janssens (2016). "Knowledge Representation Analysis of Graph Mining". In: *CoRR* abs/1608.08956.

Han, J., H. Cheng, D. Xin, and X. Yan (2007). "Frequent Pattern Mining: Current Status and Future Directions". In: *Data Mining and Knowledge Discovery* 14.1.

Han, J. (2005). *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1558609016.

Heath, I. J. (1971). "Unacceptable File Operations in a Relational Data Base". In: *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET '71. New York, NY, USA: ACM, pp. 19–33.

Herndon, T., M. Ash, and R. Pollin (2013). "Does high public debt consistently stifle economic growth? A Critique of Reinhart and Rogoff". In: *Cambridge Journal of Economics*. DOI: 10.1093/cje/bet075.

Hochbaum, D. S. and A. Pathria (1998). "Analysis of the Greedy Approach in Problems of Maximum K-coverage". In:

Hölldobler, S. and L. Schweizer (2014). "Answer set programming and clasp a tutorial". In: *YSIP*, p. 77.

Huhtala, Y., J. Kärkkäinen, P. Porkka, and H. Toivonen (1999). "TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies". In: *Comput. J.* 42, pp. 100–111.

Isakowitz, T., S. Schocken, and H. C. Lucas Jr. (1995). "Toward a Logical/-Physical Theory of Spreadsheet Modeling". In: *ACM Trans. Inf. Syst.* 13.1, pp. 1–37.

Jabbour, S., L. Sais, and Y. Salhi (2013). "Boolean satisfiability for sequence mining". In: *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pp. 649–658.

– (2015). "Decomposition Based SAT Encodings for Itemset Mining Problems". In: *PAKDD*, pp. 662–674.

Järvisalo, M. (2011). "Itemset Mining as a Challenge Application for Answer Set Enumeration". In: *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, pp. 304–310.

Jeon, J., X. Qiu, J. S. Foster, and A. S. Lezama (2015). "JSketch: sketching for Java". In: *ESEC/FSE*, pp. 934–937.

Jones, T. H., I.-Y. Song, and E. K. Park (1996). "Ternary Relationship Decomposition and Higher Normal Form Structures Derived from Entity Relationship Conceptual Modeling". In: *Proceedings of the 1996 ACM 24th Annual Conference on Computer Science*. CSC '96. New York, NY, USA: ACM, pp. 96–104.

Kemp, C., J. B. Tenenbaum, T. L. Griffiths, T. Yamada, and N. Ueda (2006). "Learning Systems of Concepts with an Infinite Relational Model". In: *Proceedings of the 21th National Conference on Artificial Intelligence*. AAAI Press, pp. 381–388.

Kim, M. and K. S. Candan (2011). "Approximate Tensor Decomposition Within a Tensor-relational Algebraic Framework". In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. CIKM '11. New York, NY, USA: ACM, pp. 1737–1742.

Knobbe, A. J. and E. K. Y. Ho (2006). "Pattern Teams". In: *Principles and Practice of Knowledge Discovery in Databases*, pp. 577–584.

Koehler, H. (2007). "Domination Normal Form: Decomposing Relational Database Schemas". In: *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*. ACSC '07. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., pp. 79–85.

Kok, S. and P. Domingos (2007). "Statistical predicate invention". In: *Proceedings of The 24th International Conference on Machine Learning*, pp. 433–440.

Kolb, S., S. Paramonov, T. Guns, and L. De Raedt (2017). "Learning constraints in spreadsheets and tabular data". In: *Machine Learning* 106.9, pp. 1441–1468.

Kowalski, R. (1979). "Algorithm = Logic + Control". In: *Commun. ACM* 22.7, pp. 424–436. DOI: 10.1145/359131.359136.

Lallouet, A., M. Lopez, L. Martin, and C. Vrain (2010). "On learning constraint problems". In: *ICTAI* 1.section II, pp. 45–52.

Langley, P. "Data-Driven Discovery of Physical Laws". In: *Cognitive Science* 5.1, pp. 31–54. DOI: 10.1111/j.1551-6708.1981.tb00869.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1551-6708.1981.tb00869.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1551-6708.1981.tb00869.x.

Law, M., A. Russo, and K. Broda (2014). "Inductive Learning of Answer Set Programs". In: *Logics in Artificial Intelligence*. Ed. by E. Fermé and J. Leite. Cham: Springer International Publishing, pp. 311–325.

Law, M., A. Russo, and K. Broda (2015). *The ILASP system for learning Answer Set Programs*. https://www.doc.ic.ac.uk/~ml1909/ILASP.

– (2016). "Iterative Learning of Answer Set Programs from Context Dependent Examples". In: *TPLP* 16.5-6, pp. 834–848. DOI: 10.1017/S1471068416000351. URL: https://doi.org/10.1017/S1471068416000351.

Le, V. and S. Gulwani (2014). "FlashExtract: a framework for data extraction by examples". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, p. 55.

Leone, N., G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello (2002). "The DLV System for Knowledge Representation and Reasoning". In: *ACM Transactions on Computational Logic* 7, pp. 499–562.

Lezama, A. S. (2009). "The Sketching Approach to Program Synthesis". In: *APLAS*, pp. 4–13.

– (2013). "Program sketching". In: *STTT* 15, pp. 475–495.

Li, T. (2005). "A General Model for Clustering Binary Data". In: *ACM SIGKDD*. New York, NY, USA: ACM, pp. 188–197.

Li, T., M. D. Vos, J. Padget, K. Satoh, and T. Balke (2015). "Debugging ASP using ILP". In: *Technical Communications ICLP Cork, Ireland*.

Lifschitz, V. (2008). "What is Answer Set Programming?" In: AAAI.

Liu, B., W. Hsu, and Y. Ma (1998). "Integrating Classification and Association Rule Mining". In: *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 80–86.

Lu, H., J. Vaidya, and V. Atluri (2008). "Optimal Boolean Matrix Decomposition: Application to Role Engineering". In: *IEEE 24th ICDE*, pp. 297–306.

Mannila, H. and K. Räihä (1994). "Algorithms for Inferring Functional Dependencies from Relations". In: *DKE* 12.1, pp. 83–99.

Mannila, H. and H. Toivonen (1997). "Levelwise Search and Borders of Theories in Knowledge Discovery". In: *Data Min. Knowl. Discov.* 1.3, pp. 241–258.

Mansinghka, V. K., R. Tibbetts, J. Baxter, P. Shafto, and B. Eaves. "BayesDB: A probabilistic programming system for querying the probable implications of data". In: *CoRR* abs/1512.05006.

McCarthy, J. (1999). *Elaboration Tolerance*.

Métivier, J.-P., P. Boizumault, B. Crémilleux, M. Khiari, and S. Loudni (2012). "A constraint language for declarative pattern discovery". In: *Proceedings of the ACM Symposium on Applied Computing*. Ed. by S. Ossowski and P. Lecca, pp. 119–125.

Métivier, J., S. Loudni, and T. Charnois (2013). "A Constraint Programming Approach for Mining Sequential Patterns in a Sequence Database". In: *CoRR* abs/1311.6907.

Miettinen, P., T. Mielikäinen, A. Gionis, G. Das, and H. Mannila (2008). "The Discrete Basis Problem". In: *IEEE Transactions on Knowledge and Data Engineering* 20.10, pp. 1348–1362. ISSN: 1041-4347.

Miettinen, P. (2008). "On the positive-negative partial set cover problem". In: *Inform. Process. Lett.* 108.4, pp. 219–221.
– (2009). *Matrix decomposition methods for data mining: computational complexity and algorithms*. Department of Computer Science, series of publications A, report A-2009-4. University of Helsinki 2009 (Ph.D. thesis, monograph).
– (2012). "Dynamic Boolean Matrix Factorizations." In: *Proceedings of International Conference on Data Mining*. Ed. by M. J. Zaki, A. Siebes, J. X. Yu, B. Goethals, G. I. Webb, and X. Wu. IEEE Computer Society, pp. 519–528. ISBN: 978-1-4673-4649-8.
– (2015). "Generalized Matrix Factorizations as a Unifying Framework for Pattern Set Mining: Complexity Beyond Blocks". In: *ECMLPKDD '15*, pp. 36–52.
Miyata, Y., T. Furuhashi, and Y. Uchikawa (1995). "A study on fuzzy abductive inference". In: *Proceedings of 1995 IEEE International Conference on Fuzzy Systems*. Vol. 1. Citeseer, pp. 337–342.
Muggleton, S. and L. De Raedt (1994b). "Inductive Logic Programming: Theory and Methods". In: *Journal of Logic Programming* 19.20, pp. 629–679.
– (1994a). "Inductive Logic Programming: Theory and Methods". In: *J. Log. Program.* 19/20, pp. 629–679.
Muggleton, S. H., D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad (2014). "Meta-interpretive learning: application to grammatical inference". In: *Machine Learning* 94.1, pp. 25–49.
Muggleton, S. H., D. Lin, and A. Tamaddoni-Nezhad (2015a). "Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited". In: *Machine Learning* 100.1, pp. 49–73.
– (2015b). "Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited". In: *Machine Learning* 100.1.
Négrevergne, B. and T. Guns (2015). "Constraint-Based Sequence Mining Using Constraint Programming". In: *CPAIOR*, pp. 288–305.
Négrevergne, B., A. Dries, T. Guns, and S. Nijssen (2013). "Dominance Programming for Itemset Mining". In: *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*, pp. 557–566.
Negrevergne, B., A. Dries, T. Guns, and S. Nijssen (2013). "Dominance Programming for Itemset Mining". In: *Proceedings of IEEE International Conference on Data Mining*.
Nethercote, N., P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack (2007). "Principles and Practice of Constraint Programming". In: ed. by C. Bessière. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. MiniZinc: Towards a Standard CP Modelling Language, pp. 529–543.

Neumann, S. and P. Miettinen (2017). "Reductions for Frequency-Based Data Mining Problems". In: *CoRR* abs/1709.00900.

Niemeyer, G. https://labix.org/python-constraint.

Oetsch, J., J. Pührer, and H. Tompits (2011). "Stepping through an Answer-Set Program". In: *LPNMR*. Ed. by J. P. Delgrande and W. Faber. Springer Berlin Heidelberg, pp. 134–147.

Osherson, D., J. Stern, O. Wilkie, M. Stob, and E. Smith (1991). "Default probability". In: *Cognitive Science* 15.2, pp. 251–269.

Özgür, A. (2015). *CSPLib Problem 110: Peaceably Co-existing Armies of Queens*. Ed. by C. Jefferson, I. Miguel, B. Hnich, T. Walsh, and I. P. Gent. http://www.csplib.org/Problems/prob110.

Paatero, P. and U. Tapper (1994). "Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values". In: *Environmetrics* 5.2, pp. 111–126.

Paramonov, S., M. van Leeuwen, M. Denecker, and L. De Raedt (2015a). "An exercise in declarative modeling for relational query mining". In: *International Conference on Inductive Logic Programming, ILP, Kyoto, 20-22 August 2015*. Springer.

– (2015b). "An Exercise in Declarative Modeling for Relational Query Mining". In: *ILP*, pp. 166–182.

Paramonov, S., S. Daria, and M. Pauli (2017a). "Hybrid ASP-based approach to pattern mining". In: *Rules and Reasoning - International Joint Conference, RuleML+RR 2017, RuleML+RR 2017: International Joint Conference on Rules and Reasoning, July 12-15, London, UK*. Springer Lecture Notes in Computer Science (LNCS). New York: Springer, pp. 199–214. DOI: 10.1007/978-3-319-61252-2_14. URL: https://lirias.kuleuven.be/handle/123456789/582160.

Paramonov, S., S. Kolb, T. Guns, and L. De Raedt (2017b). "TaCLe: Learning Constraints in Tabular Data". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. CIKM '17. Singapore, Singapore: ACM, pp. 2511–2514. ISBN: 978-1-4503-4918-5. DOI: 10.1145/3132847.3133193.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 0-934613-73-7.

Pei, J. and J. Han (2000). "Can we push more constraints into frequent pattern mining?" In: *ACM SIGKDD, Boston, MA, USA*, pp. 350–354.

Pei, J., J. Han, and R. Mao (2000). "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets". In: *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, Dallas, TX, USA*, pp. 21–30.

Pensa, R. G. and J.-F. Boulicaut (2005). "Towards fault-tolerant formal concept analysis". In: *AI*IA '05*, pp. 212–223.

Plotkin, G. D. (1971). "A further note on inductive generalization". In: volume 6, pages 101–124.

Quinlan, J. R. (1986). "Induction of Decision Trees". In: *MACH. LEARN* 1, pp. 81–106.

Ray, O. (2009). "Nonmonotonic abductive inductive learning". In: *Journal of Applied Logic* 7.3. Special Issue: Abduction and Induction in Artificial Intelligence, pp. 329 –340.

Reinhart, C. M. and K. S. Rogoff (2010). *Growth in a Time of Debt*. Working Paper 15639. National Bureau of Economic Research. DOI: 10.3386/w15639. URL: http://www.nber.org/papers/w15639.

Reiter, R. (1980). "Equality and Domain Closure in First-Order Databases". In: *J. ACM* 27.2, pp. 235–249.

– (1984). "Towards a Logical Reconstruction of Relational Database Theory". In: *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. Ed. by M. L. Brodie, J. Mylopoulos, and J. W. Schmidt. New York, NY: Springer New York, pp. 191–238.

Rojas, W. U., P. Boizumault, S. Loudni, B. Crémilleux, and A. Lepailleur (2014). "Mining (Soft-) Skypatterns Using Dynamic CSP". In: *CPAIOR*, pp. 71–87.

Rossi, F., P. van Beek, and T. Walsh, eds. (2006). *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier.

Savnik, I. and P. A. Flach (2000). "Discovery of multivalued dependencies from relations". In: *Intell. Data Anal.* 4.3-4, pp. 195–211.

Schulte, C. and P. J. Stuckey (2008). "Efficient Constraint Propagation Engines". In: *ACM Trans. Program. Lang. Syst.* 31.1, 2:1–2:43. ISSN: 0164-0925. DOI: 10.1145/1452044.1452046. URL: http://doi.acm.org/10.1145/1452044.1452046.

Singh, A. P. and G. J. Gordon (2008). "Relational Learning via Collective Matrix Factorization". In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 650–658.

Singh, R., R. Singh, Z. Xu, R. Krosnick, and A. Solar-Lezama (2014). "Modular Synthesis of Sketches Using Models". In: *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, pp. 395–414.

Todorovski, L. (2010). "Equation Discovery". In: *Encyclopedia of Machine Learning*. Ed. by C. Sammut and G. I. Webb. Boston, MA: Springer US, pp. 327–330. ISBN: 978-0-387-30164-8.

Tyukin, A., S. Kramer, and J. Wicker (2014). "BMaD – A Boolean Matrix Decomposition Framework". In: *ECML PKDD '14*. Ed. by T. Calders, F. Esposito, E. Hüllermeier, and R. Meo, pp. 481–484.

Van den Broeck, G. and A. Darwiche (2013). "On the Complexity and Approximation of Binary Evidence in Lifted Inference". In: *The Neural Information Processing Systems*, pp. 2868–2876.

Van Hentenryck, P., L. Perron, and J.-F. Puget (2000). "Search and Strategies in OPL". In: *ACM Trans. Comput. Logic* 1.2, pp. 285–320. ISSN: 1529-3785. DOI: 10.1145/359496.359529. URL: http://doi.acm.org/10.1145/359496.359529.

Vojtás, P. (1999). "Fuzzy logic abduction". In: *Proceedings of the EUSFLAT-ESTYLF Joint Conference, Palma de Mallorca, Spain, September 22-25, 1999*, pp. 319–322.

Yan, X. and J. Han (2002). "gSpan: Graph-Based Substructure Pattern Mining". In: ICDM '02.

Yan, X., J. Han, and R. Afshar (2003). "CloSpan: Mining Closed Sequential Patterns in Large Datasets". In: *In SDM*, pp. 166–177.

Zaki, M. J., S. Parthasarathy, M. Ogihara, and W. Li (1997). *New Algorithms for Fast Discovery of Association Rules*. Tech. rep. Rochester, NY, USA.

List of publications

Articles in internationally reviewed journals

- Paramonov, Sergey; Stepanova, Daria; Miettinen, Pauli. Hybrid ASP-based Approach to Pattern Mining, Theory and Practice of Logic Programming, 2018
- Paramonov, Sergey; van Leeuwen, Matthijs; De Raedt, Luc. Relational data factorization, Machine Learning, volume 106, issue 12, pages 1867-1904, 2017
- Kolb, Samuel (*); Paramonov, Sergey (*); Guns, Tias; De Raedt, Luc. Learning constraints in spreadsheets and tabular data, TaCLe, Machine Learning, volume 106, issue (9-10), pages 1441-1468, 2017

(*) – equal contribution.

Papers at international conferences and symposia, published in full in proceedings (7 references)

- Paramonov, Sergey; Bessiere, Christian; Dries, Anton; De Raedt, Luc. Sketched Answer Set Programming, 30th International Conference on Tools with Artificial Intelligence, IEEE, 2018, Volos, Greece.
- Paramonov, Sergey; Daria, Stepanova; Pauli, Miettinen. Hybrid ASP-based approach to pattern mining, Rules and Reasoning - International Joint Conference, RuleML+RR 2017, Springer Lecture Notes in Computer Science (LNCS), pages 199-214, Springer, 2017, RuleML+RR 2017: International Joint Conference on Rules and Reasoning, July 12-15, London, UK
- Paramonov, Sergey(*); Kolb, Samuel(*); Guns, Tias; De Raedt, Luc. TaCLe: Learning constraints in tabular data, Proceedings of the 2017 ACM

on Conference on Information and Knowledge Management, CIKM 2017, pages 2511-2514, Sheridan Communications, Conference on Information and Knowledge Management (CIKM), Singapore, 6-10 Nov 2017

- Nutt, Werner; Paramonov, Sergey; Ognjen, Savkovic. Implementing query completeness reasoning, International Conference on Information and Knowledge Management, CIKM, pages 733-742, ACM, Information and Knowledge Management Conference, Melbourne, 19-23 October 2015

- Paramonov, Sergey; van Leeuwen, Matthijs; Denecker, Marc; De Raedt, Luc. An exercise in declarative modeling for relational query mining, Lecture Notes in Computer Science, volume 9575, pages 166-182, Springer, Inductive Logic Programming, Kyoto, 20-22 August 2015

- Werner, Nutt; Paramonov, Sergey; Ognjen, Savkovic. An ASP approach to query completeness reasoning, Theory and Practice of Logic Programming, volume 13, issue 4-5-Online-Supplement, pages 1-10, Cambridge University Press, ICLP, Istanbul, 24-29 August 2013

- Savković, Ognjen; Paramita, Mirza; Paramonov, Sergey; Nutt, Werner. MAGIK: Managing completeness of data, Proceedings of the 21st ACM International Conference on Information and Knowledge Management, CIKM '12, pages 2725-2727, ACM, CIKM, Maui, Hawaii, USA, 29 Oct - 2 Nov 2012

Article in book, internationally recognised scientific publisher

- Dries, Anton; Guns, Tias; Nijssen, Siegfried; Babaki, Behrouz; Le Van, Thanh; Negrevergne, Benjamin; Paramonov, Sergey; De Raedt, Luc. Modeling in MiningZinc, Bessiere, Christian; De Raedt, Luc; Kotthoff, Lars; Nijssen, Siegfried; O'Sullivan, Barry; Pedreschi, Dino (eds.), Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach, Lecture Notes in Computer Science, volume 10101, pages 257-281, Springer, 2016

Papers at international and national workshops, proceedings (5 references)

- Paramonov, Sergey; Tao, Chen; Guns, Tias. Generic mining of condensed pattern representations under constraints, Young Scientist's Second International Workshop on Trends in Information Processing (YSIP2), pages 168-177, CEUR Workshop Proceedings, Young Scientist's International Workshop on Trends in Information Processing, Dombai, Russia, 16-20 May 2017

- Guns, Tias; Paramonov, Sergey; Negrevergne, Benjamin. On declarative modeling of structured pattern mining, AAAI Workshop on Declarative Learning Based Programming, Phoenix, Arizona USA, 13 February 2016
- van der Hallen, Matthias; Paramonov, Sergey; Leuschel, Michael; Janssens, Gerda. Knowledge representation analysis of graph mining, Bogaerts, Bart; Harrison, Amelia (eds.), Proceedings of ASPOCP 2016, pages 55-76, Workshop on Answer Set Programming and Other Computing Paradigms, New York, 16 October 2016
- De Raedt, Luc; Paramonov, Sergey; van Leeuwen, Matthijs. Relational decomposition using Answer Set Programming, Online Preprints 23rd International Conference on Inductive Logic Programming, International Conference on Inductive Logic Programming, Rio de Janeiro, 28-30 August 2013
- De Raedt, Luc; Paramonov, Sergey; van Leeuwen, Matthijs. Relational decomposition using Answer Set Programming, Preprints First International Workshop on Learning and Nonmonotonic Reasoning (LNMR 2013), pages 5-15, Workshop on Learning and Nonmonotonic Reasoning, La Coruna, 15 September 2013

Meeting abstracts, presented at international conferences and symposia

- Kolb, Samuel; Paramonov, Sergey; Guns, Tias; De Raedt, Luc. Learning constraints in spreadsheets and tabular data, Inductive Logic Programming Conference, Orléans, France, 4-6 September 2017

Internal reports

- Paramonov, Sergey. Query Completeness – A Logic Programming Approach, Technical Report, 96 pages, KRDB, Bolzano, Italy, April 2013



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A
B-3001 Leuven
sergey.paramonov@kuleuven.be