"Global constraints for mining sets and sequences"

Aoga, John

Abstract

The purpose of Pattern Mining (PM) is the discovery of patterns, knowledge, and information in data (textual or structured). Major concerns in the design of PM algorithms include the efficiency (in time and memory consumption) and the flexibility of the mining algorithms. The flexibility means that one can add preferences/constraints to guide the mining process in finding relevant patterns. In recent years, the paradigm of Constraint Programming (CP) has been introduced in PM. CP is a way of solving combinatorial problem by separating its resolution into two steps: the modeling of the problem and the search for solutions. This way, CP is sufficiently flexible in the addition of new constraints. However, existing approaches towards standard and CP-based PM represent different trade-offs between these concerns (flexibility and efficiency): standard PM methods are highly efficient but lack in terms of flexibility. On the other hand, CP-based PM methods are very flexible but less efficie...

GLOBAL CONSTRAINTS FOR MINING SETS AND SEQUENCES

Aoga John Oscar Raoul

July 2019

Thesis submitted in partial fulfillment of the requirements for the degree of:

▷ *Doctor of Science of Engineering and Technology from UCL*

▷ *Docteur en Science de l'Ingénieur de l'UAC*

at

UCLouvain
Institute of Information and Communication Technologies,
Electronics and Applied Mathematics (ICTEAM)
Louvain-la-Neuve, Belgium

&

Université d'Abomey-Calavi (UAC)
Ecole Doctorale des Sciences de l'Ingénieur (ED-SDI)
Abomey-Calavi, Bénin

To my daughter, and my wife.

Abstract

The purpose of Pattern Mining (PM) is the discovery of patterns, knowledge, and information in data (textual or structured). Major concerns in the design of PM algorithms include the efficiency (in time and memory consumption) and the flexibility of the mining algorithms. The flexibility means that one can add preferences/constraints to guide the mining process in finding relevant patterns. In recent years, the paradigm of Constraint Programming (CP) has been introduced in PM. CP is a way of solving combinatorial problem by separating its resolution into two steps: the modeling of the problem and the search for solutions. This way, CP is sufficiently flexible in the addition of new constraints. However, existing approaches towards standard and CP-based PM represent different trade-offs between these concerns (flexibility and efficiency): standard PM methods are highly efficient but lack in terms of flexibility. On the other hand, CP-based PM methods are very flexible but less efficient. Furthermore, standard PM typically relies on effective data structures and algorithmic improvements, where CP-based PM relies on the power of declarative tools. In this work we aim to reconnect the two approaches to maximize both flexibility and efficiency: 1) we propose new algorithms for PM using CP, in which PM problems are modeled as global constraints; 2) we show that through these global constraints one can combine ingredients from both PM and CP to build effective data structures and efficient filtering algorithms. We experimentally demonstrate that our approaches perform well in terms of efficiency on a variety of benchmarks despite its flexibility. We show that these global constraints can be (i) combined with existing constraints in CP such as regular expression and grammar constraints, (ii) used as a building block in learning tasks such as discriminative patterns and rule-based models. All our implementation are open-source and available online (https://sites.uclouvain.be/cp4dm/).

Resume

Le but du Pattern Mining (PM) est la découverte de motifs, de connaissances et d'informations dans les données (textuelles ou structurées). L'efficacité (en temps et en mémoire) et la flexibilité des algorithmes de Data Mining sont des préoccupations majeures dans la conception des algorithmes de PM. Par flexibilité, on entend que l'on peut ajouter des préférences/contraintes pour guider le processus de fouille dans la recherche de motifs pertinents. Ces dernières années, le paradigme de la programmation par contraintes (PPC) a été introduit dans les PM. La PPC est un moyen par lequel on résout un problème combinatoire en séparant la modélisation dudit problème et la recherche de solutions. Ainsi, la PPC est assez flexible pour l'ajout de nouvelles contraintes. Cependant, les approches de PM standards existantes et celles basées sur la PPC représentent des compromis différents entre ces préoccupations (flexibilité et efficacité) : les méthodes de PM standards sont très efficaces mais manquent de flexibilité. D'autre part, celles basées sur la PPC sont très flexibles mais moins efficaces. En outre, les méthodes standards reposent généralement sur des structures de données efficaces et des améliorations algorithmiques, alors que celles basées sur la PPC repose sur la puissance déclarative des outils de modélisation. Le but de cette thèse est de reconnecter ces deux approches pour maximiser à la fois la flexibilité et l'efficacité. Pour ce faire : 1) nous proposons de nouveaux algorithmes pour le PM utilisant la PPC, dans lesquels les problèmes de PM sont modélisés comme des contraintes globales ; 2) nous montrons qu'à travers ces contraintes globales, on peut combiner les ingrédients du PM et de la PPC pour construire des structures et des algorithmes de filtrage efficaces. Nous démontrons expérimentalement que nos approches donnent des résultats compétitifs en termes d'efficacité sur une variété de base de données malgré leur flexibilité. Nous montrons que ces contraintes globales peuvent être (i) combinées aux contraintes existantes dans les solveurs PPC telles que les expressions régulières et les contraintes grammaticales, (ii) utilisées comme élément constitutif dans les tâches d'apprentissage telles que la recherche de motifs discriminants et des listes de règles.

Acknowledgements

"Gratitude is a miracle of its own recognition. It brings out a sense of appreciation and sincerity of a being."

–Auliq-Ice

Where to start when there are so many people to thank! To all of you, who directly or indirectly, have always supported me in this adventure, I say big **THANK YOU**. Be fulfilled beyond your expectations. But it would be ungrateful to not thank some people specifically.

First, I thank my promotors: Pierre Schaus and Mr Théophile Dagba; without you, this thesis would never have been what it is today. A big thank you to you Mr Dagba, for the trust you have given to me. After the sudden death of Mr Adedjouma (my former mentor), you did not hesitate to conduct my DEA at the end and to be my co-promotor for this PhD adventure despite your multiple occupations. As for you, Pierre, how can I thank you? Without your open mind, your expertise and vision, this achievement won't be what it is today. Thank you for giving me a taste for research and well-written papers and algorithms. You are contributing to the development of my country by accepting all these Beninese PhD students here. Find in this, the beginning of a long and fruitful collaboration. Why change a team that wins!

I am also very grateful to you, Tias and Siegfried. Your expertise, valuable comments, and advice contribute significantly to enhance the level of this achievement. Many thanks to you and to Pierre for all you have done mainly when the time comes to write a paper. Know that I learned a lot from you in writing excellent and beautiful papers. Here also finds its place, the jury which significantly contributed to raising the level of this manuscript. I don't forget the fruitful discussion with professors of ICTEAM, EPAC, and IFRI such as Professors Vianou, Djogbé, Assogba, Ezin, and Dagba.

My thanks also go to the Benin-Belgium cooperation and then to FRIA ("Fonds pour la Formation à la Recherche dans l'Industrie et dans l'Agriculture"), which provided substantial financial support for the accomplishment of this thesis. In particular, Marc Lobelle for all his dedication to the cooperation and for the co-tutelle agreements of which he has been a true craftsman. Thanks also to Tias, Pierre, Dagba, and Olivier for their contributions to the FRIA process.

Going to the work was just as much pleasure because of the great colleagues I had in Benin and Belgium: Helène, Ratheil, Gael, Minh, Az, Maurice, Carlos, Pat, Arsene, and Hypolite. It was a real pleasure to share the office with you. A special thanks to you Helène my office mate of every day. Speaking of colleagues, how can I forget my brothers and friend of Broloutes: François, Michael, Thanh, Raziel, and Nathan, without you, this adventure would just have no taste. BDM members (Emeline, Min, Natacha, Quentine, Cigale, Micheline,...) also find their place here. Thank you for welcoming me without prejudice and for sharing those after-foot and team-building times. I cannot forget the direction board of MIFY (Ratheil, Probus, Christian) and all the staff members. Thank you for the great work you have done. You guys, consider this as a justification for my repeated absence of board meetings.

Arrive at the turn of my family and I already seem to have exhausted all the words of thanks, but not for you! I will always find what to say. Thanks to you Dad and Mom for your vision and your unwavering and unconditional support. Thank you for always believing in your "Fiston". Francine, Nadine, Sophie, Astrid, Amael, Grace, Karol, and Sophia thank you for this family warmth that you cover me.

Honey, it's your turn, and I can only say thank you. Thank you for your love. Thank you for your warmth and your understanding. This adventure together was equally uplifting and revealing how strong we are beyond everything. Without you, this thesis would have been a vague field of loneliness. Please, simply consider that this accomplishment is also yours.

For all my friends Sandrine, Egidia, Mylo, Lionel, Parfait, Gael, Ratheil, Probus, Christian, Francois, Maurel, Thanh, Michael, Sophia, Steaven, Emery ... it's just beautiful being able to always count on you.

John, July 2019

List of Tables

Table 2.1 Three equivalent representations of itemset databases:**a)** Horizontal sparse - $\mathcal{H}$, **b)** Vertical sparse - $\mathcal{V}$, **c)** Vertical dense - $\mathcal{D}$. 20
Table 2.2 A sequence database SDB 25

Table 3.1 CPU runtime for several algorithms vs *CoverSize*.(TO≡TimeOut; * ≡CoverSize+CoverClosure; $\rho \equiv density = \frac{1}{|\mathcal{T}|\times|\mathcal{I}|}\sum_{t\in\mathcal{T},i\in\mathcal{I}}\mathcal{D}_{ti}$) 55
Table 3.2 Runtimes, in seconds, for discriminative itemset mining 58

Table 4.1 Probabilistic rule lists examples: **a)** From PRL over Door opening data (Probability=the probability that the door is opened); and From PRL**b)**, and **c)** SBRL over Mushroom data. (Probability=the probability that the mushroom is edible). 64
Table 4.2 Benchmark features 80
Table 4.3 Total code lengths for several datasets (θ is the minimum support for $\mathcal{F}$) 81

Table 5.1 A sequence database SDB_1 and list of last positions. . 97
Table 5.2 Dataset Features. Sparsity is equal to $(\frac{1}{\text{size}(SDB)} \times \sum \frac{\text{size}(s)}{\text{size}(I_{/s})})$ 104

Table 6.1 **a)** A sequence database SDB, **b)** a structure for the next position of minimum gap time N(precomputed) and **c)** the last position map. 114

Table 6.2 Seven real-life datasets features. Respectively: dataset name, number of distinct symbols, number of sequences, total number of symbols in the dataset, maximum sequence length, average sequence length, and density computed by $\frac{allsymbols(SDB)}{\text{size}(\mathcal{I})\times\text{size}(SDB)}$ (Kosarak is the sparsest dataset and Protein is the densest). 132
Table 6.3 Sequential pattern miners with supported constraints. [1] is any other user-supplied constraint that does not depend on the embeddings (not implemented but could be). **GapSeq* does not consider time but position of events, ***cSPADE* does not support minimum span constraint. 133
Table 6.4 Combination of pattern length, item inclusion/exclusion, regular expression constraints with gap constraint. 138

Table 7.1 Execution time and memory usage for several non-timed datasets and thresholds (the maximum size of episodes is set to 5). 157
Table 7.2 Execution time and memory usage for several timed datasets and thresholds (the gap and span constraints are: $gap[100, 3600]$ and $span[1, 35000]$). 160
Table 7.3 Comparison with MINEPI+ and EMMA for $\theta = 5\%$ and $W = 10$ (rows are sorted by $|s|$). 161
Table 7.4 Additional constraints on Q08379 Protein (Uniprot). . 161

List of Figures

Figure 1.1 Graph to color (Benin and its neighbouring countries). 7

Figure 1.2 Example of map coloring using CP. **a)** CP search to solve this problem and **b)** an example of solution (a colored map) 9

Figure 2.1 **a)** Transaction database with a target attribute split into **b)** positive and **c)** negative databases. 24

Figure 2.2 **a)** Projected and equivalent **b)** Pseudo-projected databases with $\theta = 3$ for SDB_1 27

Figure 3.1 Reversible Sparse Bitset techniques example. 39

Figure 3.2 Computing $min(c)$ and $max(c)$ when C is included. . 43

Figure 3.3 **a)** Plot of χ^2 ZDC function in $[0, |\mathcal{D}^+|] \times [0, |\mathcal{D}^-|]$ with $|\mathcal{D}^+| = 60$ and $|\mathcal{D}^-| = 40$; **b)** Same plot with the plane representing a threshold at $\chi^2 = 20$; **c)** Same plot with the score-axis projected in the pn-plane. Note: $p = n^+$, $n = n^-$ . . . 51

Figure 3.4 Several cases of χ^2 $ZDC(|\mathcal{D}^+|, |\mathcal{D}^-|, n^+, n^-, score)$ constraint filtering. Note: $p = n^+$, $n = n^-$ and iso-curve corresponding to $\min(score)$. 52

Figure 4.1 Itemset Database with positive/negative classes 68

Figure 4.2 Powerset lattice of $\mathcal{D}$ with equivalence classes. 68

Figure 4.3 Finding greedy and optimal solution base on the example of Figure 4.2 74

Figure 4.4 Sensitivity of PRL for several settings using mushroom and soybean datasets 82

Figure 4.5 Comparison of Area under ROC among different methods and four datasets, for all 10-folds ($\theta = 10\%$, $|I| = 1$). . . . 84
Figure 4.6 **a)** Comparison of coding length in average among *PRL* (g,c) and *SBRL* for different test datasets and **b and c)** evolution of the coding length of *data only* (top) and the AUC (bottom) for several rule lists size, for *mushroom* dataset, for all 10-folds ($\theta = 10\%$, $|I| = 2$). . . . 85

Figure 5.1 Projected databases tree obtained from SDB_1 with $\theta = 3$ (on top) and the Reversible vectors of the trailed-based data structure (on bottom). . . . 100
Figure 5.2 Prefix Projection Decreasing Counting Principle . . . 102
Figure 5.3 CPU times for PPIC, PPDC, PPmixed and Gap-Seq for several minsup (missing points indicate a timeout) . . . 106
Figure 5.4 CPU times for PPIC,PPDC,PPmixed and cSPADE for several *minsup* (missing points are due to timeout). . . . 108
Figure 5.5 Incremental impact of our solutions to the different weaknesses (yaxis is logscale for all plots) . . . 109
Figure 5.6 Handling of different additional constraints . . . 110

Figure 6.1 Embedding (Projected) databases and extension windows for patterns $\langle A\rangle$ and $\langle AD\rangle$: **a)** without time constraints **b)** with time constraints ($gap^{[3,7]}$). Note that embeddings are positions in s, not timings and these positions start from 0. . 116
Figure 6.2 Examples showing sequences **a)** $\langle ADC\rangle$ and **b)** $\langle AC\rangle$ in SDB_1 and **c)** embeddings of sequence $\langle ADC\rangle$ in sequence 3.118
Figure 6.3 Trailing-based data structure to store and restore the embeddings database: **a)** without time constraints **b)** with $gap^{[3,7]}$ constraints . . . 123
Figure 6.4 CPU times for PPIC (without time constraints) with several minsup (missing points indicate a timeout). Find more experiments in [AGS16] or Chapter 5. . . . 133
Figure 6.5 CPU times when considering minimum and maximum gap constraints for several minsup (missing points indicate a timeout) . . . 134
Figure 6.6 CPU times when considering both *gap* and *span* constraints for several minsup (missing points indicate a timeout). 135

Figure 6.7 CPU times for several maximum gap with fixed minsup over Bible, Fifa, Leviathan and PubMed datasets (missing points indicate a timeout). 136
Figure 6.8 Comparing *PPICt* without time restriction (*PPICt[0,Inf]*) with *PPIC* 137

Figure 7.1 Sequence projection (✓ indicates a match, ✗ otherwise) and its reversible vector. 145
Figure 7.2 Efficient frequency computation. 148
Figure 7.3 Data structures used for timed sequences with $gap[2, 7]$ and $span[1, 10]$. 152
Figure 7.4 Performance profiles of execution time (**top**) and Memory usage (**bottom**) for Uniprot dataset (2452 instances, where $n \in [100, 30000]$) with $\theta = 5\%$, maximum size of 5 and time limit of 600 seconds. 158
Figure 7.5 Performance profiles of execution time (**top**) and Memory usage (**bottom**) for Ubiqlog dataset (21 instances) with $gap[100, 3600]$, $span[1, 35000]$, $\theta = 5\%$, maximum size of 5 and time limit of 600 seconds. 159

List of Algorithms

Algorithm 3.1 Class `RSparseBitSet*` 38
Algorithm 3.2 Class CoverSize($[I_1, \dots I_m], \mathcal{D}, c$) 44
Algorithm 3.3 Class CoverClosure($[I_1, \dots I_m], \mathcal{D}$) 48
Algorithm 3.4 Class *ZDC*($|\mathcal{D}^+|, |\mathcal{D}^-|, n^+, n^-, score$) 53

Algorithm 4.1 *Greedy*$(\mathcal{F}, \mathcal{D})$ 75
Algorithm 4.2 *Branch-and-bound* $(\mathcal{F}, \mathcal{D})$ 77

Algorithm 5.1 PrefixProjection(SDB,P,i,θ) 95
Algorithm 5.2 ProjectAndGetFreqs(SDB, a, θ) 96
Algorithm 5.3 ProjectAndGetFreqs(SDB, a, θ, $sids$, $poss$, ϕ, φ) . 101

Algorithm 6.1 PPICt($P, SDB, \theta, M, N, W, Y$) 124
Algorithm 6.2 ProjectAndGetFreqs(i, SDB, a, M, N, W, Y) 127
Algorithm 6.3 updateSupport($b, e, sid, pos, visitedI$) 128

Algorithm 7.1 $propagate(s, \Sigma, a, P)$ 146
Algorithm 7.2 $sequenceProjection(s, \Sigma, a)$ 147
Algorithm 7.3 $sequenceProjectionTimed(s^s, s^t, \mathcal{I}, a, N, W)$ 153
Algorithm 7.4 $buildInitialTimedSeqProj(s^s, s^t, \mathcal{I}, a)$ 153
Algorithm 7.5 $buildTimedSeqProj(s^s, s^t, \mathcal{I}, a, N, W)$ 154

List of Abbreviations

CFIM Closed Frequent Itemset Mining.

CP Constraint Programming.

CPM Constraint-based Pattern Mining.

DFIM Discriminative Frequent Itemset Mining.

FEM Frequent Episode Mining.

FFIM Free Frequent Itemset Mining.

FIM Frequent Itemset Mining.

MDL Minimum Description Length.

PM Pattern Mining.

PPDC Prefix Projection Decreasing Counting.

PPIC Prefix Projection Incremental Counting.

PRL Probabilistic rule lists.

SPM Sequential Pattern Mining.

ZDC Zero Diagonal Convex.

Contents

Abstract **iii**

Résumé **v**

Acknowledgements **vii**

List of Tables **ix**

List of Figures **xi**

List of Algorithms **xv**

List of Abbreviations **xvii**

I Introductory Material **1**

1 Introduction **3**

1.1 Data Mining 3
1.2 Constraint Programming 6
1.3 Overview of the contributions and scientific achievements . . 11
1.4 Publications and scientific achievements 13
1.5 Structure of the thesis 15

2 Background **19**

2.1 Pattern Mining 19
2.1.1 Frequent Itemset Mining 20
2.1.2 Sequential Pattern Mining 24
2.2 Constraint Programming 28

2.2.1 Constraints and global constraints 29
2.2.2 State restoration in CP 29
2.3 Summary, Outlooks, Further readings 30

II Frequent itemset Mining 31

3 Frequent Itemset Mining using Constraint Programming 33
3.1 Context and motivation 34
3.2 Preliminaries 35
3.2.1 Modeling IM problem using CP 35
3.2.2 Table Constraint and Reversible Sparse Bit-Sets . . . 36
3.3 Global constraints for frequency-based itemset mining 37
3.3.1 Computing frequency: the *CoverSize* constraint 39
3.3.2 Closed itemsets: the *CoverClosure* constraint 44
3.4 Frequency-based itemset mining with *CoverSize* and *CoverClosure* 47
3.4.1 Frequent itemset mining 47
3.4.2 *Closed* frequent itemset mining 47
3.4.3 Discriminative (closed) itemset mining 47
3.5 Implementation and Practical User Guide 54
3.6 Experiments 55
3.7 Summary, Outlooks, Further readings 59

4 Frequent Itemset Mining for Compression 61
4.1 Context and motivation 62
4.2 Related Work 64
4.3 The probabilistic rule list mining problem 66
4.4 Discovering probabilistic rule lists 67
4.4.1 Coverage and probability of a rule list 67
4.4.2 Minimum Description Length encoding of rule lists . . 70
4.4.3 Coding length related to likelihood and quality of rule lists 73
4.4.4 A Greedy algorithm 75
4.4.5 Branch-and-Bound algorithm 76
4.5 Implementation and Practical User Guide 78
4.6 Experiments 79
4.6.1 Compression power of PRL 80
4.6.2 Impact of the parameters 81

4.6.3 Comparison of *PRL* with existing rule learning algorithms 83
4.6.4 Prediction power of PRL and other supervised learning approaches 83
4.7 Summary, Outlooks, Further readings 84

III Sequential Pattern Mining 87

5 Sequential Pattern Mining using Constraint Programming 89
5.1 Context and Motivation 90
5.2 Related works 91
5.3 Global constraints for projected frequency 93
5.3.1 Existing methods [NG15, KLL+15] 93
5.3.2 Improving propagation 95
5.3.3 Constraints of SPM 102
5.3.4 Time and space complexity 103
5.4 Implementation and Practical User Guide 103
5.5 Experiments 105
5.6 Summary, Outlooks, Further readings 107

6 Mining time-constrained sequential patterns with constraint programming 111
6.1 Context and Motivation 112
6.2 **Related work** 113
6.3 Preliminaries 114
6.3.1 Sequential Pattern Mining 114
6.3.2 CP-based model for SPM problem 118
6.4 Trailing-based data structure for the embedding database . . 120
6.4.1 Embedding database and extension windows 120
6.4.2 Trailing-based data structures 121
6.5 *PPICt* global constraint under time constraints 122
6.5.1 PPICt filtering algorithm and improvements 122
6.5.2 Additional constraints 126
6.5.3 Time and space complexity 129
6.6 Implementation and Practical User Guide 129
6.7 Experiments 131
6.7.1 Performances results 133
6.7.2 Handling additional Constraints 137
6.8 Summary, Outlooks, Further readings 138

7 Frequent Episode Mining using Constraint Programming 139
7.1 Context and Motivation 140
7.2 Mining Episodes in a Non Timed Sequence 141
7.2.1 Technical Background 141
7.2.2 Problem Modelling 143
7.2.3 Filtering Algorithm 144
7.3 Mining Episodes in a Timed Sequence 149
7.3.1 Technical Background 149
7.3.2 Filtering Algorithm 150
7.4 Implementation and Practical User Guide 155
7.5 Experimental Results 156
7.5.1 Memory Bound Analysis 156
7.5.2 Comparison with Decomposed Approaches 157
7.5.3 Comparison with Specialized Approaches 157
7.5.4 Handling Additional Constraints 160
7.6 Summary, Outlooks, Further readings 161

8 Conclusion 163
8.1 Summary and Main Messages 163
8.2 Discussion and perspectives 165

A Algorithms 169

Bibliography 193

Part I

Introductory Material

Chapter 1

Introduction

“The journey of a thousand miles begins with one step.”

–Lao Tzu

This thesis is at the frontier between Pattern Mining (PM) and Constraint Programming (CP). Our objective is to create new PM algorithms. Those algorithms must be flexible to discover relevant and interpretable patterns. In this chapter, we briefly introduce PM and CP concepts and present the motivation to combine those fields and our contributions.

1.1 Data Mining

The digitization of company workflows led to a significant accumulation of data. Most companies consider data as the new oil and are aiming at developing advanced analytical tools to better extract knowledge out of it. Extracting information and understanding data is the role of Data Mining (DM) and Machine Learning (ML), whose utility is now well-established.

One of the most fundamental data mining tasks is Pattern Mining, where discovered information is structured as a set, list of sets, sequence, tree, graphs, etc. This information is then called *patterns*. The interest of Pattern Mining techniques lies in their ability to find patterns (i) that are hidden in large databases, (ii) that are interpretable by humans and (iii) that are therefore useful for understanding data and decision-making [FVLK$^+$17].

A well-known problem in pattern mining is *Frequent Itemset Mining (FIM)*. This problem was introduced by the seminal paper of Agrawal [AIS93]. Given a list of itemsets $\mathcal{D}$ called an itemset database, FIM aims at finding all the item-subsets which appear in $\mathcal{D}$ more than a given user-defined threshold θ. This problem has many applications in wide-range fields such as market basket analysis [AIS93], bioinformatics [NMB+15], web mining [CMS97] and malware detection [DFL+15], to name just a few. Let us give an example of a FIM task in *Product recommendation systems* [MDLN01].

Example 1.1. Product recommendation systems are designed to suggest to customers products that are frequently purchased together by other customers. To do this, one looks for supermarket products which are purchased together by more than 75% of customers. Then, when customers buy some of the products of a set, the other products of that set are recommended to them. Assuming people frequently buy *bread*, *coffee* and *sugar* together, if a customer buys *coffee* and *sugar*, *bread* may be recommended to him/her.

The patterns discovered by the FIM algorithms are sets, and therefore information on the possible ordering relationship between items is ignored. As a result, these algorithms may fail to discover important patterns which would depend on sequential relationships. For example, in text analysis, it is often relevant to take into account the order of words in sentences [PFM16]. To face this challenge, Agrawal and Srikant [AS95] introduced the *Sequential Pattern Mining (SPM)* task. Given a sequences of itemsets, called sequence database, denoted as $\mathcal{D}$, SPM is aiming at finding all the subsequences that appear in $\mathcal{D}$ more than a given user-defined threshold θ. This problem differs from the FIM problem in that the order of the itemsets in a pattern becomes important. *Intelligent transportation system* is well-known example of application [Agg14b, GLF+18].

Example 1.2. Intelligent transportation systems aim to solve traffic congestion and designed new roads. For example, vehicle trajectories can be stored as sequences of fixed points through which they pass over time. One can determine from there the key segments in trajectories which are frequently used. This information may help in road maintenance. The less frequent segment may be suggested to vehicles to avoid traffic jams, longer kilometers, etc.

Despite valuable insights, the discovery of frequent patterns (sets or sequences) is a challenging task. Indeed, finding all frequent patterns in a database is a tough task because the search space can be extremely large. For example, with m items there are 2^m possible itemset candidates. Early research in Pattern Mining focused on how to make the approaches as effective as possible [Agg14a]. This is no longer a problem today since there are algorithms such as LCM [UKA05] and Spade [Zak01] which can enumerate all frequent patterns in a few seconds, even when the database is very large.

Example 1.3. To give an idea of this efficiency, LCM enumerates all itemsets with minimum support of 10 in less than a second on the Retail database. This database contains 88162 market baskets of anonymous customers in an anonymous Belgian supermarket with 16470 products.

Next, come pattern explosion problems. Off-the-shelf frequent pattern mining algorithms, depending on the defined threshold [1], discover a large number of patterns which are often redundant and irrelevant. They are difficult for a user to parse and are often useless. To face these problems, several solutions were investigated in the literature. One can either try to present the patterns in a condensed[2] form (such as closed [ZjH02, YHA03], free [SNK07, SR14], maximal patterns [BCG01, LC05],...) or to use measures which allow evaluating how "interesting"[3] a pattern is [VT14]. More generally, we can express some preferences as constraints (an expert can provide these constraints) to guide the algorithm towards the relevant patterns. Note that condensed patterns and interesting metrics approaches can be considered as constraints based approaches [NZ14].

In this thesis, we are interested in those Constraint-based Pattern Mining (CPM) approaches because they are more flexibles for real applications.

[1]There is a real tradeoff in the choice of θ because if the support is low "interesting" patterns can be discovered but there are often too many of them and if the support is too high only trivial patterns are discovered.

[2]The set frequent patterns is compressed by removing redundancies while allowing losing information or not.

[3]The notion of an interesting pattern is not trivial to define; it mainly relies on interestingness measures which yield non-redundant and relevant patterns..

Example 1.4. For example, for recommendation systems, we could also refine the obtained patterns by limiting the products to a given category (e.g. breakfast and baby products). For transportation systems, one may want trajectories which are frequent only in mornings or which necessarily pass through some points in the path.

Problems. A solution, to incorporate these constraints, is to enumerate all the frequent patterns first and then, in post-processing, filter out those that do not meet the defined criteria. This approach is limited because it depends on the success of the mining process. For example, on very large databases or with low minimum support, this process can take a lot of time and memory that would make difficult to complete the first step. Another approach allows pushing these constraints in the mining process directly. This approach has the potential not to suffer from the two-phase limitations and has an additional advantage. It enables digging with low supports since one can filter irrelevant patterns instantly. However, this approach also has its limitations. The number of constraints is not necessarily known in advance, so only a limited number of constraints can be taken into account. Hence, a new constraint often implies a new method. To overcome these limitations, we study in this thesis generic approaches based on the use of *Constraint Programming.*

1.2 Constraint Programming

Constraint Programming is another way of formulating and solving combinatorial problems. CP is well-positioned today as the Holy Grail of programming: the user specifies the problem and the computer solves it [Fre97]. To do this, CP separates the definition of the problem (modelling) from its resolution (search) [Lau18, RvBW06] i.e.

$$\text{CP} = \text{Model} + \text{Search}.$$

Model. Modelling consists of identifying the decision variables of the problem to be solved and the domain of each of them. One can then define constraints on these variables. A typical example is that of the colouring of a map (graph) using CP.

Example 1.5. Assume we want to assign at most three colours (green, yellow and red) to Benin and its neighbouring countries with the constraint that if these countries share a border, they cannot have the same colour (Figure 1.1). We model this in CP by first choosing the decision variables. These are the colour given to each country with the three possible colours as domains. Let BJ, BF, NE, NG and TG $\in \{G, Y, R\}$ be these variables. Then, we add as constraints that all neighbouring countries must have different colours: BJ≠BF, BJ≠NE, BJ≠NG, BJ≠TG, BF≠TG, BF≠NE and NE≠NG.

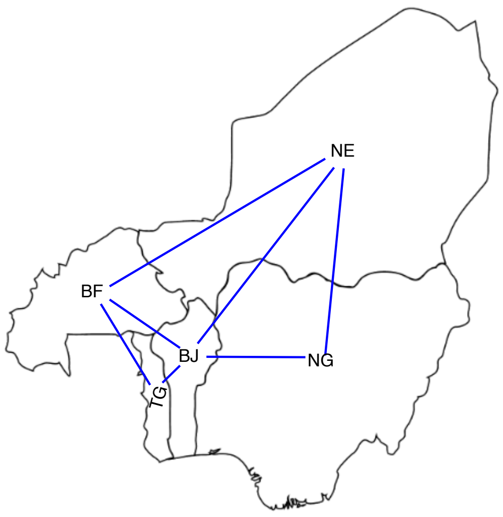


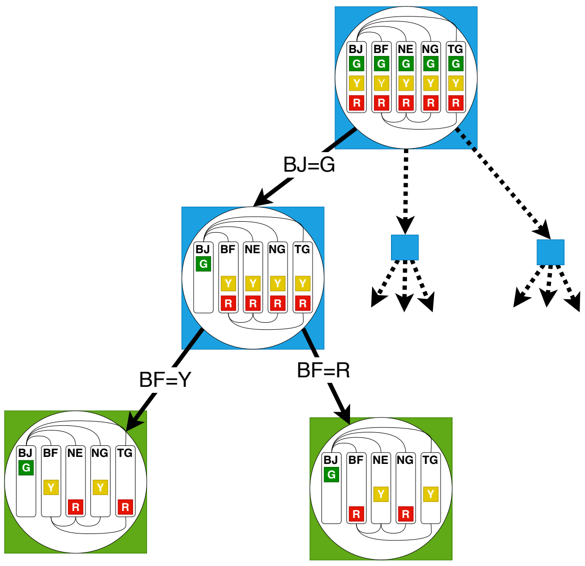
Figure 1.1. Graph to color (Benin and its neighbouring countries).

Search. The search (in CP) receives as input a problem as decision variables plus constraints (over these variables) and outputs the solutions of this problem. A solution is an assignment to all variables, of the values of their domains, that satisfies the specified constraints. This search is a simple recursive algorithm implementing a *depth-first search* with *chronological backtracking* [vB06, Lau18]. The search is done in the solution space which consists of the union of variables' domains. To reduce this space, one uses an inference procedure called *constraint propagation*. Mainly, when a decision is made on a variable (e.g. an assignment), propagation consists in relaying this information to the other variables by removing from their domain the values that have become inconsistent with the constraints plus this decision through the *fix-point* algorithm.

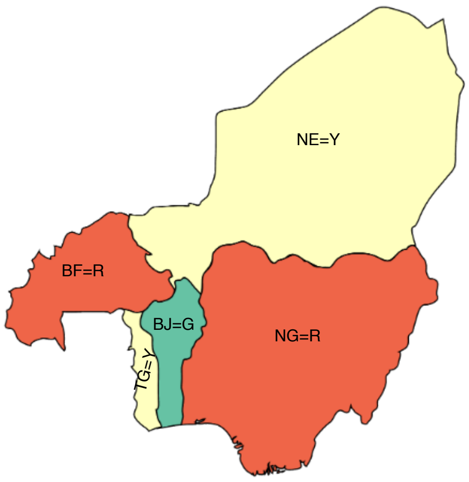
Example 1.6. Assuming our running example, as illustrated in Figure 1.2a, if through the search, Benin (BJ) is assigned to green (G) ($BJ = G$) then G will be removed from the domain of all remaining variables by propagation of the constraints BJ≠BF, BJ≠NE, BJ≠NG and BJ≠TG. After that, if we assign yellow(Y) to BF, Y will be removed from the domain of TG and NE by propagation of the constraints BF≠TG and BF≠NE. At this state, all the variables have an assignment then we have a solution. By backtracking, the search can undo the decision to assign yellow to BF and then it assigns red (R) to it. This information is propagated again, and another solution is found. We obtain the colouring of the map of the Figure 1.2b.

This way of separating modelling from search makes it easy to add other constraints. For our illustration, we can, for example, add that TG must be red (R). This new constraint impact is on the search space, which will be filtered accordingly to satisfy this new constraint. This gives to the CP framework *modularity* and *flexibility* in solving problems. It is this CP property that interests us in Pattern Mining since it can allow adding many user-defined preferences (constraints) and make the Pattern Mining method generic and flexible.

Problem. Presented in this way, CP may seem easy. However, there can be several ways of representing a problem with constraints and hence choosing the wrong model can make the problem hard to solve. Generally, creating the description of the problem which decision variables with their domains is not time-consuming. Nevertheless, it is crucial to extract the relevant constraints and properly design the filtering algorithms to get solutions efficiently. The effectiveness of filtering depends not only on its strength but also the number of decision variables and the size of their domain. Strong filtering prunes more the search space, but can consume time to execute while weak filtering will execute quickly but prunes less. It is then a tradeoff between the efficiency of the filtering (i.e. its execution time) and its effectiveness (i.e. its strength in



(a)



(b)

Figure 1.2. Example of map coloring using CP. **a)** CP search to solve this problem and **b)** an example of solution (a colored map)

reducing the search space). Therefore, a *"good" constraint* is the one that has a filtering algorithm which allows pruning more while having a low execution time [vHK06].

Several works successfully used CP to make generic and flexible Pattern Mining approaches [GNDR11, MR04, KLL$^+$15, GNDR13, DRGN08, NG10, NG15, JSS17]. While, in general, most of the existing CP-based Pattern Mining approaches are considered generic and flexible, their efficiency compared to specialized approaches is an issue. There is also a tradeoff between flexibility and efficiency. The first CP-based PM methods [GNDR11, MR04, GNDR13, DRGN08, NG15, JSS17] decomposed PM problems into several constraints (filtering algorithms) that allowed these methods to be fully flexible but in return, they lacked efficiency. To address this, methods [KLL$^+$15, NG10, NG15] using a single constraint (called *global constraint*) was proposed. These methods are more efficient than decomposed methods by sacrificing some flexibility. Despite these efforts, these approaches are not as efficient as specialised approaches, making their usages/applications limited. Also, only a few problems in Pattern Mining were tackled in CP.

⚠ In summary, to address the Pattern Mining challenges such as the combinatorial explosion, and the redundancy and irrelevancy of the found patterns, constraint based approaches were proposed. One can hence guide the mining process towards more interesting solutions by imposing preferences/constraints. Imposing constraints cannot be done in any way. Thus, constraint programming, a well-known declarative framework, is introduced to bring modularity and flexibility to Pattern Mining. However, flexibility comes at a cost; there is often a trade-off between flexibility and efficiency. So, the existing hybrid (CP-based) approaches, until now, are very flexible, sacrificing efficiency.

Putting all together, several research questions therefore arise. The main one is whether there would be a way to make flexible approaches as effective as specialised approaches. From this question emerges several others which are as the following.

Q1 Can we build new CP-based approaches that improve efficiency by combining ingredients from both Pattern Mining and Constraint Programming?

Q2 In the literature, the effectiveness of many approaches is based on the design of an appropriate data structure, can the same solution boost the effectiveness of existing CP-based methods?

Q3 Which modeling for new CP-based methods, will give a better efficiency and genericity?

Q4 How to evaluate the relevancy of the obtained patterns?

In this thesis, we investigate the possibilities of improving the effectiveness of existing approaches. To do so, we propose new filtering algorithms (global constraints) for frequent sequence and itemset mining which are generic, flexible and efficient.

1.3 Overview of the contributions and scientific achievements

The hybridisation of Pattern Mining with CP has been a great success, demonstrating the ability of PM to be flexible. However, it has become clear that the effectiveness (he execution time and memory consumption) of PM approaches can be sacrificed to allow those approaches to be flexible. Hence, the main question arises *whether there would be a way to make flexible approaches as effective as specialised approaches.* This thesis provides an affirmative answer to this question and those presented in the previous section. The **main contribution** of this thesis is new CP-based Pattern Mining approaches which either outperform [AGS16, AGS17, CAS18] or are competitive [SAG17] with specialised approaches.

In detail, the contributions of this work on "**Q1**: *Can we build new CP-based approaches that improve efficiency by combining ingredients from both Pattern Mining and Constraint Programming?*" can be summarized as follows.

In [AGS16], we showed how by combining Sequential Pattern Mining and Constraint Programming techniques, one can build flexible and more efficient approaches than existing both CP-based and specialized approaches. To do this,

- we, first, compute the **projected database efficiently using pre-computing vectors** to avoid to scan the full sequence each time;

- and, second, by taking inspiration from the *trailing* used in CP solvers to allows **fast incremental storing and restoring of the projected database**.

The main approach developed is called PPIC and has become state-of-the-art in SPM. We have proven that these are key ideas at the origin of this effectiveness by successfully applying them in the design of new SPM constraint under time restrictions over sequence database with time [AGS17], and new constraint for mining episodes in a single and long sequence [CAS18].

As an ingredient of this efficiency, the data structures developed in this thesis are examples of combinations of techniques from Pattern Mining and constraint programming. These data structures are answers to "**Q2**: *In the literature, the effectiveness of many approaches is based on the design of an appropriate data structure, can the same solution boost the effectiveness of existing CP-based methods?*".

- The **backtracking-aware data structure**, that we introduced in [AGS16], is a set of vectors to store projected databases (Pattern Mining techniques) plus a cursor on the position in the vectors and a variable that indicates the size of the current projected database. The cursor and the size variables are both "*Reversible*" (CP technique). This structure offers two significant advantages: storing and restoring incrementally projected databases and reusing memory space.
- The **Reversible sparse Bit-Set data structure** [SAG17], for its part, allows storing and performing binary operations on pattern covers efficiently. This data structure can also restore covers in backtracking. It also allows transactions to be carried out only on promising transactions, thus taking advantage of the structure of the problem.

Contributions mentioned above are integrated into global constraints. Thus, one answer to question "**Q3**: *Which modeling for new CP-based methods, will give a better efficiency and genericity?* " is that modeling plays a significant role in the effectiveness of the approaches developed.

- First, we showed that global constraints are the pieces of modeling code that make an impact. Indeed, they allow **using appropriate data structures and facilitating algorithmic improvements**.
- We also showed that by **exposing the frequency as a variable** [SAG17], it is possible to solve several problems based on this

model, namely Frequent Itemset Mining (FIM), Closed FIM, and Discriminative FIM.

We contributed on "**Q4** : *Which modeling for new CP-based methods, will give a better efficiency and genericity?*" by investigating Discriminative FIM and Probabilistic rule lists (PRL).

- We proposed a Zero Diagonal Convex (ZDC) constraint using constraint programming, which combined with the CoverSize constraint allows solving the DFIM problem. The ZDC constraint shows an example of transforming a continuous function into a discrete output function [SAG17].
- We **used the Minimum Description Length (MDL) principle to characterize small-and-good sets of probabilistic rules**, and we devised a branch-and-bound with a best-first search strategy to find better-than-greedy and optimal solutions for the proposed task [AGNS18].

Putting all together, we end up with nine realizations: PPIC, PPDC, PPMixed, PPICt, CoverSize, CoverClosure, ZDC, EpisodeSupport, and PRL. One of the problems encountered during this thesis is the unavailability of the implementation of some algorithms. We, therefore, contribute by **making available online[4] all the codes of our realizations**. We also provide additional experiments, test datasets, user guides and some demos. We keep all this up-to-date and regularly fix bugs. Most of our implementation are also directly available in the open-source *CP-Solver OscaR* [Osc12] whose implementation we are involved in.

These contributions led to publications at conferences or journals which will be detailed in the next section.

1.4 Publications and scientific achievements

The five mains publications discussed in this thesis are the following.

[4] https://sites.uclouvain.be/cp4dm/

Journal paper

[**AGS17**] John O. R. Aoga, Tias Guns, and Pierre Schaus, "*Mining time-constrained sequential patterns with constraint programming*", Constraints, **22**(2017), no. 4, 548–570.

Conference papers

[**AGS16**] John O. R. Aoga, Tias Guns, and Pierre Schaus, "*An efficient algorithm for mining frequent sequence with constraint programming*", Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part II (Paolo Frasconi, Niels Landwehr, Giuseppe Manco, and Jilles Vreeken, eds.), Lecture Notes in Computer Science, vol. 9852, Springer, 2016, pp. 315–330.

[**SAG17**] Pierre Schaus, John O. R. Aoga, and Tias Guns, "*Coversize: A global constraint for frequency-based itemset mining*", Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 September 1, 2017, Proceedings (J. Christopher Beck, ed.), Lecture Notes in Computer Science, vol. 10416, Springer, 2017, pp. 529–546.

[**CAS18**] Quentin Cappart, John O. R. Aoga, and Pierre Schaus, "*Episodesupport: A global constraint for mining frequent patterns in a long sequence of events*", Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June26-29, 2018, Proceedings, 2018, pp. 82–99.

[**AGNS18**] John O. R. Aoga, Tias Guns, Siegfried Nijssen, and Pierre Schaus, "*Finding probabilistic rule lists using the minimum description length principle*", Discovery Science - 21st International Conference,DS 2018, Limassol, Cyprus, October 29-31, 2018, Proceedings,2018, pp. 66–82.

Abstract and poster

JFPC,17 (**Abstract**) John O. R. Aoga, Tias Guns, and Pierre Schaus, "*Algorithme Efficace pour la Fouille de Séquences Fréquentes avec la*

Programmation par Contraintes". Treizièmes Journées Francophones de Programmation par Contraintes, Montreuil-sur-mer, juin, (2017).

Benelearn,16 (**Abstract+poster**) John O. R. Aoga, Tias Guns, and Pierre Schaus, "*Scalable Constraint Programming approach for Mining Frequent Sequence with gap constraints*".Benelearn, (2016).

In addition to those, two papers have been written during this thesis. They were published at the beginning of this thesis and are not discussed in this thesis because they are outside its scope.

Journal paper

[**ADF16**] John O. R. Aoga, Théophile K. Dagba, and Codjo C. Fanou, "*Integration of Yoruba language into marytts*", International Journal of Speech Technology, **19**(2016), no. 1, 151–158.

Conference papers

[**DAF16**] Théophile K. Dagba, John O. R. Aoga, and Codjo C. Fanou, "*Design of a Yoruba language speech corpus for the purposes of text-to-speech (TTS) synthesis*", Intelligent Information and Database Systems - 8th Asian Conference, ACIIDS 2016, Da Nang, Vietnam, March 14-16, 2016, Proceedings, Part I (Ngoc Thanh Nguyen,Bogdan Trawinski, Hamido Fujita, and Tzung-Pei Hong, eds.), Lecture Notes in Computer Science, vol. 9621, Springer, 2016,pp. 161–169.

1.5 Structure of the thesis

The rest of the thesis is divided into seven chapters grouped into three parts.

In the remainder of **PART I**, we give, in *Chapter 2*, the background information on Pattern Mining (i.e. Frequent Itemset Mining and Sequential Pattern Mining) and Constraint Programming.

PART II, which is dedicated to Itemset Mining problems is divided into two chapters.

Chapter 3 presents *CoverSize*, the new global constraint to solve several types of Itemset Mining problems using CP. In this chapter, we demonstrate how the techniques used in CP can be adapted to obtain flexible Itemset Mining approaches. We begin this chapter by presenting the context and motivation of this work. Then, we introduce specific notions of FIM through the technical background. Subsequently, we describe the CP model of FIM problem as a decomposition-based approach. Next, we show how this problem is close to Table constraints. Finally, we presented our approaches and experimentally evaluated their performance.

Chapter 4 introduces *PRL*, the new Pattern Mining method that uses itemsets obtained from *CoverSize* to build rule lists (set of itemsets). In this chapter, we show that by using the Minimum description length principle, one can obtain highly interpretable rule lists. We propose a greedy- and a branch-and-bound based algorithm to find this rule lists. For that, we first present related work following by the problem of finding probabilistic rule lists. After that, we describe our Minimum Description Length (MDL)-based approach in terms of formalisation and algorithms for solving it. Finally, we show experiments in which we demonstrate the quality, the accuracy and the predictive power of our method.

PART III discusses Sequential pattern mining problems. It is divided into three chapters.

Chapter 5 presents new global constraints (PPIC, PPDC, PPmixed) which can solve Sequential Pattern Mining problems. In this chapter, we show how, by combining the right ingredients from both (CP and PM) research communities in a novel way, we can design new global constraint for SPM. We begin this chapter by presenting the context, the motivation and the related work. Then, We describe how the existing CP-based methods model SPM problem and their limitations. Subsequently, we show how to improve on these methods using a *backtracking-aware trailing-based* data structure and some algorithmic improvements. Finally, we experimentally show the resulting system improves both on previous CP-based sequence miners as well as state-of-the-art specialised systems.

Chapter 6 presents new global constraints which can solve Sequential Pattern Mining problems on data with time stamps. In this chapter, we show how to extend the work in the previous chapter to handle sequence database with time. Unfortunately, some ingredients that made PPIC successful no longer apply. Thus, after presenting the context and motivation of this work, we show how the *backtracking-aware trailing-based* data structure can be adapted to support time constraints. Next, we introduced a new approach based on this. Finally, we demonstrate through experiments that the new approach maintains better performance while handling many constraints.

Chapter 7 presents global constraints capable of solving Frequent Episode Mining (FEM) problem. This problem applies to a single very long sequence. In this chapter, we show that this problem, even being similar to SPM problems, presents many challenges, particularly that of memory management. Thereafter, we showed that by using ingredients similar to those used in SPM, we could overcome this problem. Finally, we experimentally evaluated the performance of this approach.

Finally, in the concluding chapter, we present a summary of this thesis, and then we discuss some future directions.

Chapter 2

Background

"Be a lifelong student. The more you learn, the more you earn and more self confidence you will have.."

–Brian Tracy

Overview *In this chapter, we discuss Frequent Pattern Mining and Constraint Programming concepts useful for later chapters. In brief, the Frequent Pattern Mining task consists of finding relationships among items in a database, and Constraint Programming is a generic framework to solve combinatorial problems by separating the modeling of the problem from its resolution.*

Main source *This chapter is based on the background material in journal paper [AGS17] and conference papers [SAG17, CAS18].*

2.1 Pattern Mining

Pattern mining is a well-known and a widely studied field of research because of the numerous real-life applications (health, DNA analysis, errors tracking and detectors, marketing, ...) it offers and it uses in many other knowledge discovery problems such as feature engineering, classification, prediction and clustering [Agg14b]. There are tons of articles on pattern mining, especially

Table 2.1. Three equivalent representations of itemset databases:**a)** Horizontal sparse - $\mathcal{H}$, **b)** Vertical sparse - $\mathcal{V}$, **c)** Vertical dense - $\mathcal{D}$.

tid	itemset
1	{A,B,D}
2	{B,C,D}
3	{A,B,D}
4	{A,B,C,D}
5	{B,C}

(a)

item	tid-set
A	{1,3,4}
B	{1,2,3,4,5}
C	{2,4,5}
D	{1,2,3,4}

(b)

item	tid-bitvector
	[1, 2, 3, 4, 5]
A	[1 0 1 1 0]
B	[1 1 1 1 1]
C	[0 1 0 1 1]
D	[1 1 1 1 0]

(c)

on Frequent Pattern Mining. The reader interested in more details can refer to the Frequent Pattern Mining book [AH14]. In the following sections, we quickly review the relevant definitions.

2.1.1 Frequent Itemset Mining

Frequent itemset mining is concerned with finding a set of *items* that frequently appears in a database of sets [AIS93]. The database is often called a *transaction database*, and each entry in the database is called a *transaction* (such as a purchase of products).

Let $\mathcal{I} = \{1, \ldots, m\}$ be a set of items and $\mathcal{T} = \{1, \ldots, n\}$ be a set of transaction identifiers. Formally, a transaction database is defined as follows.

Definition 2.1 (Transaction database (TDB))**.** *A transaction database $TDB = \{(t, T_t) \mid t \in \mathcal{T}, T_t \subseteq \mathcal{I}\}$ is a set of tuples where each tuple is composed of a transaction identifier t and the transaction T_t itself which is a subset of $\mathcal{I}$.*

FIM algorithms use several database representations that are equivalent such as

- horizontal representation: $\mathcal{H} = TDB$;
- vertical representation: $\mathcal{V} = \{(i, I_i) \mid i \in \mathcal{I}, I_i \subseteq \mathcal{T}\}$ (transposed $\mathcal{H}$) or
- dense representation: $\mathcal{D} \subseteq \mathcal{I} \times \mathcal{T}$ which is a binary matrix of size $n \times m$ with $\mathcal{D}_{ti} \in \{0, 1\}$ and $\mathcal{D}_{ti} = 1$ iff the item i is in transaction t.

Example 2.1. Assume $\mathcal{I} = \{ A, B, C, D \}$, Tables 2.1(a,b,c) show an example of a transaction database on these three equivalent forms.

In this thesis, we mainly use the dense representation. However, we overuse $\mathcal{D}$ to represent any of these databases in particular when clear from context.

Before defining the FIM problem, we first introduce the notions of *coverage* and *support* of an itemset.

Definition 2.2 (Coverage and Support/Frequency)**.** *Assume an itemset I and a database $\mathcal{D}$, The coverage is the set of transactions that contain I: $Cover_{\mathcal{D}}(I) = \{ t \in \mathcal{T} \mid \forall i \in I : \mathcal{D}_{ti} = 1 \}$. The size of the coverage is called its frequency: $Freq_{\mathcal{D}}(I) = |Cover_{\mathcal{D}}(I)|$ and its support is the relative frequency: $Supp_{\mathcal{D}}(I) = Freq_{\mathcal{D}}(I)/n$.*

Example 2.2. Assuming, the TDB of Figure 2.1a and $I = \{ A, B \}$, $Cover_{\mathcal{D}}(I) = \{ 1, 3, 4 \}$ and $Freq_{\mathcal{D}}(I) = 3$.

Definition 2.3 (Frequent Itemset Mining problem)**.** *Given a set $\mathcal{I}$ of n possible items and a transaction database of size m, the goal of the frequent itemset mining problem is to* ***enumerate all itemsets*** *$I \subseteq \mathcal{I}$ such that $Freq_{\mathcal{D}}(I) \geq \theta$ with θ a user-supplied threshold.*

Itemset I with its $Freq_{\mathcal{D}}(I) \geq \theta$ is called *frequent itemset* and the set of all frequent itemsets wrt. θ is denoted as $\mathcal{F}_\theta$. For example, the problem of FIM applied to market basket analysis, consists in finding all the products (items) that are purchased together (itemsets) by more than $\theta = 75\%$ of registered customer baskets. There exists many other applications for FIM such as web-log mining, bio-informatics, etc [Agg14a].

Computing $\mathcal{F}_\theta$ efficiently is a core aspect of itemset mining algorithms. Indeed, enumerating all itemset is a challenging task because there are 2^m possible itemsets in the search space and so it will take a lot of time to list them all even on a TDB of hundred of items and thousands of transactions. To successfully reduce the search space, FIM algorithms often rely on the *anti-monotonicity* property, also known as the *apriori* property [NLHP98].

Property 2.1 (anti-monotone constraint)**.** If a pattern p satisfies an anti-monotone constraint C then all the sub-patterns of p also satisfy C and conversely if p does not satisfy C none of its super-patterns satisfies C.

Corollary 2.1 (Coverage is a anti-monotone constraint)**.** $\forall I_1, I_2 \subseteq \mathcal{I},\ I_1 \subseteq I_2 \Rightarrow Cover_{\mathcal{D}}(I_1) \supseteq Cover_{\mathcal{D}}(I_2)$.

Corollary 2.2 (Frequency[1] is a anti-monotone constraint)**.** $\forall I_1, I_2 \subseteq \mathcal{I},\ I_1 \subseteq I_2 \Rightarrow Freq_{\mathcal{D}}(I_1) \geq Freq_{\mathcal{D}}(I_2)$.

The frequency constraint allows reducing the size of $\mathcal{F}_\theta$ but this is closely related to the choice of θ. If θ is high, the size of $\mathcal{F}_\theta$ is relatively small but the obtained itemsets are often trivial (e.g. products listed individually). Rather, if θ is low, the size of $\mathcal{F}_\theta$ can be very large with many *redundant* itemsets. An itemset will be considered redundant *if it does not provide additional information and can be found from other itemsets.*

Example 2.3. For example, if *bread* and *cheese* are **always bought together** then listing *bread* and *cheese* separately as itemset is redundant because from the frequency of *bread* and *cheese* taken together, one can directly derive the frequency of *bread* alone and *cheese* alone: $Freq_{\mathcal{D}}(\{\,\text{bread}, \text{cheese}\,\}) = Freq_{\mathcal{D}}(\{\,\text{bread}\,\}) = Freq_{\mathcal{D}}(\{\,\text{cheese}\,\})$

To prevent this redundancy, many *condensed representations* of *frequent itemsets* have been investigated. Among those *closed* and *free* representations are useful [AH14].

Closed representation. Closed Frequent Itemset Mining (CFIM) adds to FIM the additional constraint that an itemset must not have a **super-set with the same frequency**:

$$\text{I is closed} \Leftrightarrow \nexists I' \supset I : Freq_{\mathcal{D}}(I) = Freq_{\mathcal{D}}(I') \tag{2.1}$$

Free representation. Conversely, Free Frequent Itemset Mining (FFIM) adds to FIM the additional constraint that an itemset must not have a **subset with the same frequency**:

$$\text{I is free} \Leftrightarrow \nexists I' \subset I : Freq_{\mathcal{D}}(I) = Freq_{\mathcal{D}}(I') \tag{2.2}$$

[1]The Frequency function also enjoys the property of *supramodularity* [SD14] i.e. $\forall I_1, I_2 \subseteq \mathcal{I}, Freq_{\mathcal{D}}(I_1 \cup I_2) + Freq_{\mathcal{D}}(I_1 \cap I_2) \geq Freq_{\mathcal{D}}(I_1) + Freq_{\mathcal{D}}(I_2)$.

Another task in FIM is the Discriminative Frequent Itemset Mining (DFIM). In this task, there is a dataset with a Boolean target attribute which we want to explain using the other attributes. This Boolean target attribute contains two class labels (+,−). The objective is to either enumerate all frequent itemsets with high discriminating power or find the best top-k ones.

Assume transaction database with a Boolean target attribute $\mathcal{D} = \{(t, T_t, a_t) \mid t \in \mathcal{T}, T_t \subseteq \mathcal{I}, a_t \in \{+,-\}\}$ consists of triplets (t, T_t, a_t) of an transaction identifier t, a transaction itself T_t and the class label a_t to which this transaction belongs. This database can be split into a positive $\mathcal{D}^+$ and negative $\mathcal{D}^-$ itemset database, based on the target attribute value ($+$ or $-$).

Example 2.4. For instance, in market analysis, each customer's basket is associated with his or her gender, and therefore it becomes possible to perform analyses based on gender. Typically, for instance, one could study whether some products are purchased more by men than women or vice versa.

DFIM [NGDR09] answers this type of question. Formally, It is defined as follows.

Definition 2.4 (Discriminative Itemset Mining problem)**.** *Given an itemset database $\mathcal{D}$ with a target attribute and a discriminative score, DFIM aims at enumerating all the frequent itemsets which discriminate one class label in relation to the other with respect to this score; itemsets that are very frequent in one and barely frequent in the other.*

Example 2.5. Assuming the TDB in Figure 2.1, $I = \{\text{apple, kiwi}\}$ is an interesting itemset because its frequency in the positive TDB is $Cover_{\mathcal{D}^+}(I) = 3$ and $Cover_{\mathcal{D}^-}(I) = 0$ in the negative TDB i.e. $\{\text{apple, kiwi}\}$ are bought together by only men.

A range of *discriminative* measures (such as χ^2 and information gain), also called correlation measures, have been studied [MS00]. A property that we will exploit later is that these measures can be computed using just information on the frequency of the sets plus the total number of transactions of the databases. We can hence denote these measures by a function $f(|\mathcal{D}^+|, |\mathcal{D}^-|, n^+, n^-)$ where $n^+ = Freq_{\mathcal{D}^+}(I), n^- = Freq_{\mathcal{D}^-}(I)$ represents the frequency of the itemset in the two databases.

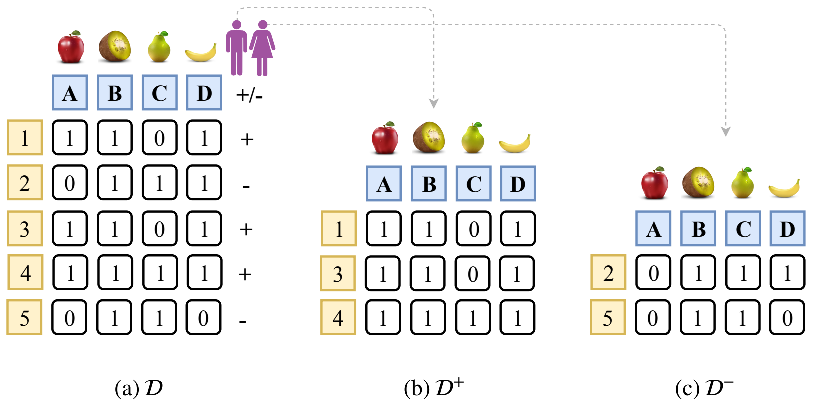


Figure 2.1. **a)** Transaction database with a target attribute split into **b)** positive and **c)** negative databases.

In an optimisation setting, one can search for *the most or least frequent itemsets* (typically under some other constraints) or find the most discriminating itemsets.

Other constraints on items and transactions have been investigated as well [NZ14]. In the literature, advanced forms of patterns such as sequences, trees and graphs have been discussed. We present the problem of mining sequences in the next section.

2.1.2 Sequential Pattern Mining

Similar to FIM, Frequent Sequence Mining is concerned with finding a set of *sequences* that appear frequently in a database of sequences [AS95]. The database is called a *sequence database* (SDB), and each entry in the database is called a *sequence* (such as a DNA sequence). In this subsection, we revisit the basic concepts for Sequential Pattern Mining (SPM). Most of these concepts can be found in [AH14, ZB03].

Let $\mathcal{I} = \{1, \ldots, m\}$ be a set of items and $\mathcal{S} = \{1, \ldots, N\}$ be a set of sequence identifiers.

Definition 2.5. ***Sequence and sequence database.*** *A sequence* $s = \langle s_1 s_2 \ldots s_n \rangle$ *over* I *is an ordered list of (potentially repeating) symbols* s_j*,* $j \in [1..n]$ *with* $\#s = n$ *the length of the sequence* s*. A set of tuples (sid,s)*

Table 2.2. A sequence database SDB

sid	sequence
sid_1	$\langle ABCBC \rangle$
sid_2	$\langle BABC \rangle$
sid_3	$\langle AB \rangle$
sid_4	$\langle BCD \rangle$

where sid is a sequence identifier and s a sequence, is called sequence database (SDB).

Example 2.6. Table 2.2 shows an example SDB over symbols $I = \{A, B, C, D\}$. For the sequence $s = \langle BABC \rangle$: $\#s = 4$ and $s_1 = B, s_2 = A, s_3 = B, s_4 = C$.

Coverage and Frequency constraints are based on the *sub-sequence* relation between a sequence and the SDB.

Definition 2.6. ***Sub-sequence ($\preceq$), Super-sequence, and Embeddings.*** *A sequence $\alpha = \langle \alpha_1 \dots \alpha_l \rangle$ is called a sub-sequence of $s = \langle s_1 s_2 \dots s_n \rangle$ and s is a super-sequence of α iff (i) $l \leq n$ and (ii) for all $i \in [1 \,..\, l]$ there exist integers j_i s.t. $1 \leq j_1 \leq \dots \leq j_l \leq n$, and $\alpha_i = s_{j_i}$. Integers j_i are called embeddings and denoted as $Emb_s(\alpha)$.*

Example 2.7. For instance $\langle BD \rangle$ is a sub-sequence of $\langle BCCD \rangle$, and inversely $\langle BCCD \rangle$ is the super-sequence of $\langle BD \rangle : \langle BD \rangle \preceq \langle BCCD \rangle$. $Emb_{\langle BCCD \rangle}(\langle BD \rangle) = \{1, 4\}$

From this relationship, we can define the notion of coverage and frequency,

Definition 2.7. ***Coverage and Frequency.*** *The coverage of sequence p in SDB, denoted by $Cover_{SDB}(p)$, is the subset of sequences in SDB that are a super-sequence of p, i.e. $Cover_{SDB}(p) = \{ (sid, s) \in SDB \mid p \preceq s \}$. The frequency of p in SDB, denoted as $Freq_{SDB}(p)$, is the number of super-sequences of p in SDB: $Freq_{SDB}(p) = |Cover_{SDB}(p)|$.*

Example 2.8. Assume that $p = \langle BC \rangle$ and $\theta = 2$, $Cover_{SDB}(p) = \{(sid_1, \langle ABCBC \rangle), (sid_2, \langle BABC \rangle), (sid_4, \langle BCD \rangle)\}$ and hence $Freq_{SDB}(p) = 3$. Hence, p is a frequent pattern for that given threshold.

The sequential pattern mining (SPM) problem, first introduced by Agrawal and Srikant [AS95], is the following:

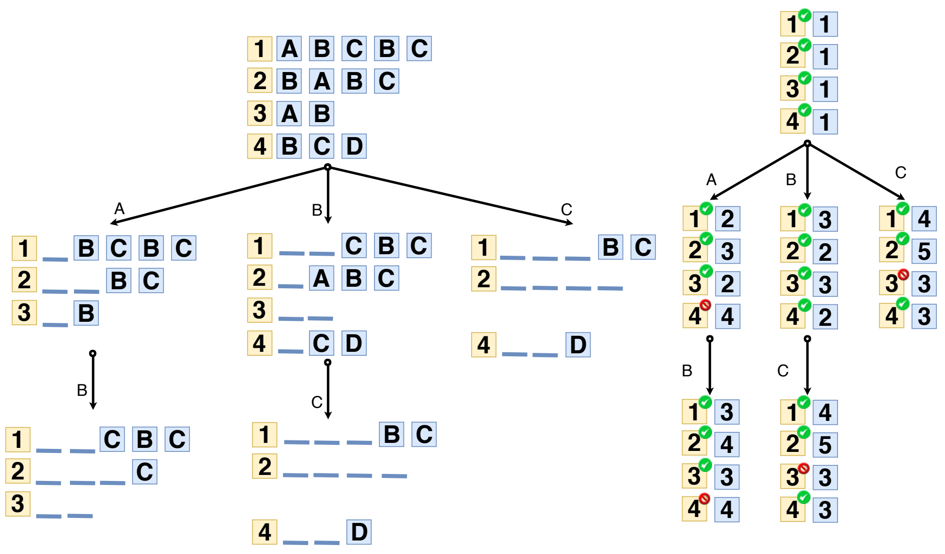
Definition 2.8. ***Sequential Pattern Mining (SPM).*** *Given a minimum support threshold θ and a sequence database SDB, the SPM problem is to find all patterns p such that $Freq_{SDB}(p) \geq \theta$.*

Sequential pattern is commonly used to mean *frequent sequence pattern* in the literature therefore in this thesis, we will use *sequential pattern* and *sequence pattern* if the frequency of the pattern does not matter.

Enumerating sequential patterns is a challenging task. The *apriori property* 2.1 is also used to reduce the search space. However, in general, a *well-developed* resolution strategy is required to enumerate all sequential pattern. In this thesis, our algorithms are based on the idea of a *prefix* and *prefix-projected* database for enumerating the frequent patterns. These concepts were first introduced in the seminal paper that presented the *PrefixSpan* algorithm [PHMA+01].

Definition 2.9. ***Prefix, prefix-projected database.*** *Let $\alpha = \langle \alpha_1 \dots \alpha_k \rangle$ be a pattern. If a sequence $\beta = \langle \beta_1 \dots \beta_n \rangle$ is a super-sequence of α: $\alpha \preceq \beta$, then the prefix of β w.r.t. α is the 'smallest prefix' of β that is still a super-sequence of α: $\langle \beta_1 \dots \beta_j \rangle$ s.t. $\alpha \preceq \langle \beta_1 \dots \beta_j \rangle$ and $\nexists j' < j : \alpha \preceq \langle \beta_1 \dots \beta_{j'} \rangle$. The sequence $\langle \beta_{j+1} \dots \beta_n \rangle = suffix_\alpha(\beta)$ is called the suffix and it represents the prefix-projection obtained by projecting the prefix away. A prefix-projected database of a pattern α, denoted by $SDB|_\alpha$, is the set of prefix-projections of all sequences in SDB that are super-sequences of α: $SDB|_\alpha = \{(sid_i, suffix_\alpha(sid_i)) \mid \alpha \preceq SDB[sid_i]\}$.*

Example 2.9. In SDB, assume $\alpha = \langle A \rangle$, then $SDB|_\alpha = \{(sid_1, \langle BCBC \rangle), (sid_2, \langle BC \rangle), (sid_3, \langle B \rangle)\}$. Figure 2.2a shows the projected databases that are obtained from SDB_1 when $\theta = 3$.



(a) (b)

Figure 2.2. **a)** Projected and equivalent **b)** Pseudo-projected databases with $\theta = 3$ for SDB_1

The *prefix-projected frequency* of the symbol $a \in \mathcal{I}$ ($freqs(a, SDB|_\alpha)$) is the number of sequences in $SDB|_\alpha$ where this symbol appears in the suffix: $freqs(a, SDB|_\alpha) = |\{(sid, s) \in SDB|_\alpha \mid a \in suffix_\alpha(s)\}|$.

⚠ We use $freqs(a)$ instead of $freqs(a, SDB|_\alpha)$ when no ambiguity is possible about the database.

Example 2.10. The prefix-projected frequencies of $SDB|_{\langle A \rangle}$ are: $freqs(A) = 0$, $freqs(B) = 3$, $freqs(C) = 2$ and $freqs(D) = 0$.

The PrefixSpan algorithm solves the SPM problem by starting from the empty pattern and extending this pattern using a depth-first search. At each step, it extends a pattern by a symbol and projects the database accordingly. The appended symbol is removed on backtrack. It hence grows the pattern incrementally, which is why it is called a pattern-growth method. A frequent pattern in the projected database is also frequent in the original database.

There are two important considerations for the efficiency of the method.

1. The first is that one does not have to consider during search any symbol that is not frequent in the prefix-projected database.
2. The second is that of *pseudo-projection*: to store the prefix-projected database during the depth-first search. It is not necessary to store (and later restore) an entire copy of the projected database. Instead, one only has to store for each sequence the pointer to the position j that marks the end of the prefix in that sequence (remember, the *prefix* of α in β is the smallest prefix $\langle \beta_1 \dots \beta_j \rangle \succeq \alpha$).

Example 2.11. Extending prefix $\langle A \rangle$ with $\langle B \rangle$ over $SDB|_{\langle A \rangle}$ gives $SDB|_{\langle AB \rangle} = \{(sid_1, \langle CBC \rangle), (sid_2, \langle C \rangle), (sid_3, \langle \rangle)\}$ and can be represented as the pseudo-projected database: $pSDB|_{\langle AB \rangle} = \{(sid_1, 3), (sid_2, 4), (sid_3, 3)\}$. Figure 2.2b shows the pseuso-projected databases that are obtained from SDB_1 when $\theta = 3$.

2.2 Constraint Programming

In this section, we briefly introduce Constraint Programming notions which will be used in the following chapters. These notions can be found in [RvBW06].

CP is a powerful declarative paradigm to solve combinatorial satisfaction and optimization problems (see, e.g., [RvBW06]). A CP problem (V, D, C) is defined by a set of variables V with their respective domains D (the values that can be assigned to a variable), and a set of constraints C on these variables. A solution of a CP problem is an assignment of the variables to a value from its domain, such that all constraints are satisfied.

At its core, CP solvers are *depth-first search* algorithms that iterate between *searching* over unassigned variables and *propagating* constraints. Propagation is the act of letting the constraints in C remove unfeasible values from the domains of its variables. This is repeated until *fixed-point*, that is, no more constraint can remove any unfeasible values. Then, a search exploration step is taken by choosing an unassigned variable and assigning it to a value from its current domain, after which propagation is triggered again.

Example 2.12. Consider two variables x, y with domains $D(x) = \{1, 2, 3\}, D(y) = \{3, 4, 5\}$. Then constraint $x + y \geq 5$ can infer during propagation that $1 \notin D(x)$ because the lowest value y can take is 3 and hence $x \geq 5 - \min(D(y)) \geq 5 - 3 \geq 2$.

2.2.1 Constraints and global constraints

Many different constraints and their propagation algorithms have been investigated in the CP community. This includes logical and arithmetic ones like the above, up to constraints for enforcing regular expressions or graph-theoretic properties. A constraint that enforces some non-trivial or application-dependent property is often called a *global constraint* [BH03]. For example, [NG15] introduced a global constraint for the pseudo-projection of a single sequence, and [KLL+16, AGS16, CAS18, AGS17] for the entire projected frequency sub-problem.

2.2.2 State restoration in CP

In any backtracking search solver, there must be some mechanism to store and restore some *state*, such that computations can be performed incrementally and intermediate values can be stored and restored. In most of the CP solvers[2] a general mechanism, called *trailing* is used for storing and restoring the state (on backtrack) [Knu15, SC06, Lau18]. Externally, the CP solvers typically expose some "reversible" objects whose values are automatically stored and restored on the trail when they change. The most important example are the domains of CP variables [dSMSSL13]. Hence, for a variable

[2]One notable exception is the Gecode copy-based solver.

the domain modifications (*assign*, *removeValue*) are automatically reversible operations. A CP solver also exposes reversible versions of primitive types such as integers and sets for use within constraint propagators. They are typically used to store incremental computations. CP solvers consist of an efficient implementation of the DFS backtracking algorithm, as well as many constraints that can be called by the fix-point algorithm. The modularity of constraint solvers stems from this ability to add any set of constraints to the fix-point algorithm.

2.3 Summary, Outlooks, Further readings

In this chapter, we have presented the concepts of frequent pattern mining and constraint programming that we use in the following chapters. A complementary and specific background will be provided, in each chapter, for the particular pattern mining problems that we have solved in this thesis.

The objective here was to present the background briefly. The interested reader can find more details in Pattern Mining book [Agg14a]. These recent surveys on Frequent itemset mining [FLV$^+$17] and Sequential pattern mining [FVLK$^+$17] are also worth looking because they provide an overview of prominent papers/methods and present possible future research directions. As for CP, the interested reader can find more details in the CP's Handbook [RvBW06] and *MiniCP* [Lau18]. *MiniCP* is a minimalist version of a CP solver that allows understanding the main concepts in CP.

Part II

Frequent itemset Mining

Chapter 3

Frequent Itemset Mining using Constraint Programming

"Intelligence is the handmaiden of flexibility and change."

–Vernor Vinge

Overview *There is always a trade-off between flexibility and efficiency in combining Frequent Itemset Mining and Constraint programming. Our objective is to maximize both flexibility and efficiency while employing a global constraint for itemset mining. Hence, this chapter is dedicated to the CoverSize constraint for itemset mining problems, a global constraint for counting and constraining the number of transactions covered by the itemset decision variables.*

Contribution *We show how advances in data structures for table constraints can benefit global constraints for itemset mining too; we propose that global constraints for itemset mining expose the frequency through a variable, and demonstrate how this allows, for example, to not only solve Frequent Itemset Mining, but also discriminative itemset mining; empirically our experiments with a generic CP solver show that this approach outperforms other CP approaches and is on par with a special-purpose CP solver, thereby decreasing the gap to the highly efficient specialized itemset miners.*

Main source *This chapter is based on our paper [SAG17].*

3.1 Context and motivation

Frequent itemset mining (FIM) is one of the well-known and most studied data mining problems [Bor12] and first introduced in [AIS93]. Guns et al. [GNDR11] showed that FIM problems could be modelled and solved using Constraint Programming (CP) with the additional benefit that new constraints can easily be integrated into the models. Since then several CP (also SAT) approaches have been proposed for other data-mining problems such as frequent sequence mining [KLL+15, NG15], dominance-based pattern mining [NDGN13] and closed FIM [JSS13, JSS17, LLL+16].

The flexibility of adding constraints when using a generic CP solver typically comes at the cost of efficiency; a well-known tradeoff. We can hence look at itemset mining papers in terms of where they are on the efficiency versus generality scale. Most works in itemset mining focus primarily on efficiency [Bor12, UKA05], while typical constraint-based mining papers hard-code a selected number of constraints based on properties like (anti-)monotonicity [NZ14]. Earlier articles on using CP for itemset mining focus mostly on generality and decompose the itemset mining constraints into many (reified) linear constraints [GNDR11] at the cost of efficiency. In line with recent works in CP for sequence mining [AGS16, AGS17, KLL+15], Lazaar et al. [LLL+16] have shown that a single global constraint for closed frequent itemset mining can outperform a decomposition approach. This comes at a significant cost for generality though, because 1) by encapsulating all but the itemset variables, only syntactic constraints on the items can be added; 2) only *closed frequent* patterns can be found and adding syntactic constraints can have unwanted side-effects [BL04].

Consequently, we aim at maximising both generality and efficiency while employing a global constraint for itemset mining. We achieve this by introducing the *CoverSize* global constraint.

⚠ For the problem of Frequent, Closed and Discriminative IM, the reader is invited to look at the Section 2.1.1 of Chapter 2.

3.2 Preliminaries

CoverSize is a global constraint which solves FIM problems using CP. Thus, in this section, we present the preliminary concepts. Namely, how to model IM problems using CP and the Reversible Sparse Bit-Sets data structure.

3.2.1 Modeling IM problem using CP

Following [DRGN08], we use an array of Boolean decision variables $\mathcal{I} = [I_1, I_2, \ldots, I_m]$ to represent an itemset $X \subset \mathcal{I}$. Each I_i is a binary variable with domain $dom(I_i) = \{0, 1\}$ and an item $i \in X \iff I_i = 1$. We say that I_i is *unbound* if there is more than one value in $dom(I_i)$. I_i is *bound* to 1 (0) means the item i is part (not part) of the itemset. Hence, one assignment to $\mathcal{I}$ corresponds to one itemset.

The *decomposition* formulation of frequent itemset mining [GNDR11] introduces an extra array of Boolean decision variables $\mathcal{T} = [T_1, T_2, \ldots, T_n]$, one for each of the n transactions. A Boolean variable T_t indicates whether the transaction with identifier t belongs to the cover of I: $\{\, (t, S) \in \mathcal{H} \mid I \subseteq S \,\}$. This is enforced with a constraint for every transaction as follows:

$$\text{Coverage constraint:} \quad \forall (t, S) \in \mathcal{H} : T_t = 0 \iff \bigvee\nolimits_{i \notin S} I_i. \tag{3.1}$$

In other words: if an item i is in the itemset I and not in the transaction (t, S) then this transaction is not covered by I and equivalently if a transaction is not covered any of the items then $i \in I$ do belong to (t, S). The size of the cover can then be constrained as follows:

$$\text{Frequency constraint:} \quad \sum\nolimits_t T_t \geq \theta. \tag{3.2}$$

This model is not domain consistent for the frequent itemset mining problem that aims to enumerate all frequent patterns for a specific θ. As suggested in [DRGN08], one can further add the redundant constraints (3.3) to achieve domain consistency for the frequent itemset problem: these constraints enforce that an item is only supported if adding it to the current itemset will not violate the frequency constraint.

$$\text{Redundant constraint:} \quad \forall i : I_i = 1 \implies \left(\sum\nolimits_{(t,S) \in \mathcal{H}, i \in S} T_t\right) \geq \theta. \tag{3.3}$$

3.2.2 Table Constraint and Reversible Sparse Bit-Sets

In contrast to most constraints in CP, what is typical about global constraints for data mining is that they must be able to handle large amounts of data. A traditional global constraint that shares this property is the table constraint, which has a rich history in CP literature [BR97, CY10, Lec11, PR14]. Its link with a global constraint for itemset mining (IM) is even stronger, as both can be seen as operating on a binary matrix; for IM the columns are *items* (Boolean variables) and for table the columns are *(variable, value) pairs.* The use of bitvectors and fast bitvector operations is common in itemset mining implementations. Indeed, it was also used for the closed FIM constraint [LLL$^+$16].

Related, a column-based bitvector representation for the table constraint was recently proposed [DHL$^+$16], and the propagator was shown to outperform all other approaches. Inspired by this relation, we show that the *reversible sparse bitset* data structure that was devised for table can also be used to implement efficient itemset mining propagators. In this section, we present this data structure.

Indeed, a table constraint enforces that an array of integer decision variables[1] $[V_1, \ldots, V_m]$ corresponds to one of the provided tuples

$$\Gamma = \{(t, \tau) | t \in \{1, \ldots, n\}\},$$

where t is the tuple identifier and each tuple $\tau = (v_1, \ldots, v_m)$ consists of m values corresponding to the m variables:

$$\text{table}([V_1, \ldots, V_m], \Gamma) \iff \exists (t, \tau) \in \Gamma : V_1 = \tau_1 \wedge \ldots \wedge V_m = \tau_m. \quad (3.4)$$

A key property to maintain is the set of tuples supported by the current domain:

$$currTable = \{(j, \tau) \in \Gamma \mid \tau_1 \in dom(V_1) \wedge \ldots \wedge \tau_m \in dom(V_m)\}.$$

In [DHL$^+$16], a reversible sparse bitset was proposed to maintain the set of tuple indices during the search. In the propagator, a dense vertical representation of Γ is used: for every variable/value combination $(V_i, v), v \in dom(V_i)$, a bitvector

$$support[V_i, v] = \{(j, \tau) \in \Gamma \mid \tau_i = v\}$$

[1]An integer decision variable is a decision variable with intergers in its domain.

is pre-computed that stores the tuple identifiers in which the pair (V_i, v) appears. The indices of $currTable$ and the consistency of each (V_i, v) is computed using bitwise operations, e.g. (V_i, v) is supported if $support[V_i, v] \cap currTable \neq \emptyset$.

We briefly recall the `RSparseBitSet` data structure [DHL$^+$16] which we will use in our propagators. The pseudo-code of this data structure is given in Algorithm 3.1 and some illustrative methods are also shown. The Reversible Sparse BitSet represents a set as a bitset (array of 64-bit Long *words*) and is *"reversible"* means that it can restore itself on backtrack. The reversibility relies on a global trail mechanism well known in the folklore of constraint programming (see [Knu15] for an introduction to trailing and time-stamping).

The originality of this structure is that it borrows the idea of reversible sparse-sets [dSMSSL13] to discard all-zero words. When a bitvector is sparse (contains many zero words), this can save unnecessary iterations and computations over those words.

The following class invariant is maintained to ignore zero words: the number of non-zero words is a reversible integer denoted `limit`; and the `limit` first entries of `index` are indexes to the non-zero words in the bitvector. All the words beyond that limit are the indexes of zero words.

For the intersect method, which is also crucial for itemset mining, one can see how this is maintained by exchanging a detected zero word with the last non-zero one before decreasing the `limit` (swapping).

Apart from skipping entire words, the bitvector representation allows using highly efficient operations over whole words such as *and* and *bitCount*.

Example 3.1. Figure 3.1 shows how the `RSparseBitSet` works for an example where we have a 16bits bitvector splits by 4bits. Especially, the intersection operation is illustrated.

3.3 Global constraints for frequency-based itemset mining

There is a close relationship between a table constraint that reasons over a binary representation of the table and itemset mining. Each variable/value pair (V_i, v) is a column and can be seen as an *item* (in the itemset mining

Algorithm 3.1: Class `RSparseBitSet`*.

```
words: array of rlong                               // reversible longs, array length = p
index: array of int                                 // array length = p
limit: rint                                         // a reversible integer
Method intersect(m: array of long)
    /* this ← this & m                                                          */
    foreach $i$ from limit downto 0 do
        o ← index[$i$]
        $w$ ← words[o] & m[o]                       // bitwise AND
        words[o] ← $w$
        if $w = 0^{64}$ then
            swap(index[$i$], index[limit])
            limit ← limit $-$ 1

Method size(): int
    /* number of bits set                                                       */
    cnt ← 0
    foreach $i$ from 0 to limit do
        o ← index[$i$]
        cnt ← cnt + bitCount(words[o])
    return cnt

Method contains(m: array of long): bool
    /* m ⊆ this                                                                 */
    foreach $i$ from 0 to limit do
        o ← index[$i$]
        if (words(o) & ~ $m$[o]) $\neq 0^{64}$ then
            return false
    return true
```

*t[0] denotes the first element of array t and 0^k denotes a sequence of k bits set to 0.

problem), and internally a dense vertical representation of the table can be used. Because each tuple in table Γ is of size n, in a binary representation of the table there will be exactly n non-zero entries per row. Further knowing that there are exactly n variables that each must be assigned one value, one can see that checking whether the set representation of $V : \{(V_i, v) \mid V_i = v \in V\}$ is a *subset* of the set representation of a tuple $\tau : \{(V_i, \tau_i) \mid \tau_i \in \tau\}$ coincides with checking whether they can be *equal* as both sets will have equal length when V is fully assigned. The cover relation of itemset mining is hence equivalent to the table support relation in this case, and the table constraint can be seen as enforcing a minimum frequency constraint with

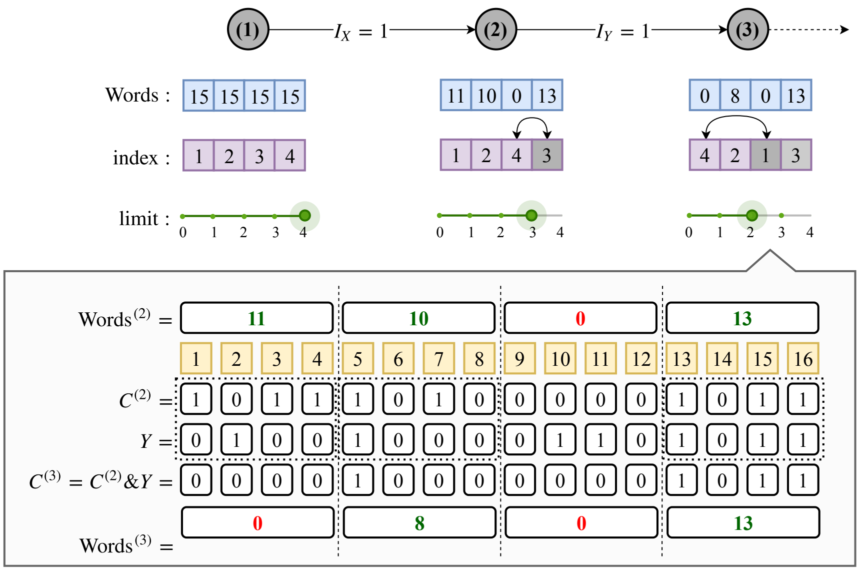


Figure 3.1. Reversible Sparse Bitset techniques example.

$\theta = 1$.

Earlier work has proposed a single global constraint for minimum frequent closed itemset mining. For efficiency reasons, we propose to use the reversible sparse bitset to maintain the set of transactions that can still be covered. For generality reasons, we aim to separate the computation of the frequency from the minimum (or maximum) frequency restriction and to separate that from enforcing the closedness property.

3.3.1 Computing frequency: the *CoverSize* constraint

Given a set of Boolean variables $\mathcal{I}$ representing the pattern (selected items), a dense vertical bitvector representation of the database $\mathcal{D}$ (see Figure 2.1c for an example), and an integer variable c, the *CoverSize* global constraint enforces the relation

$$CoverSize([I_1, \ldots, I_m], \mathcal{D}, c) \iff c = \left| \bigcap_{I_i=1} \mathcal{D}(I_i) \right| \qquad (3.5)$$

such that c represents the number of bits set in the intersection of the vertical bitvectors ($\mathcal{D}$) of the selected items. Using bitwise operations it can be formulated as

$$\mathit{CoverSize}([I_1, \ldots, I_m], \mathcal{D}, c) \iff c = \mathtt{size}(\&_{I_i=1}\mathcal{D}(I_i)) \tag{3.6}$$

Example 3.2. For example in Figure 2.1 and for itemset $\{C, D\}$, $c = |\mathcal{D}(I_C)\ \&\ \mathcal{D}(I_D))| = 2$.

Lazaar et al. [LLL+16] have argued that a global constraint is preferred over a decomposition into a constraint per transaction because the many constraints that need to be handled create overhead for the solver. This was shown earlier in [NG10], which proposed a CP-inspired dedicated solver with a global constraint for (reified) matrix-wide operations over bitvector variables.

When not exporting the cover as individual Boolean variables, we can use an internal data structure to store the cover such as `RSparseBitSet`. Note that not exposing the cover also limits the generality of the approach: no constraints can be put on the cover so that constraints such as closedness, maximality, non-frequency-based quality measures, etc either require changes to the global constraint, or a separate global constraint that recomputes the cover. However, there are some constraints that depend only on the size of the cover and hence for added flexibility we propose to hide the cover but expose the cover size.

Consistency of *CoverSize*

Theorem 3.1 is used to demonstrate that it is NP-hard to check the consistency for *CoverSize*.

Theorem 3.1. *Given a collection of sets $\{S_1, \ldots, S_m\}$, the problem of finding a subset of these such that their union is of fixed cardinality k is NP-hard.*

Proof. We build a reduction from the NP-hard exact cover by three sets (X3C) problem: given a collection $\{ C_1, C_2, \ldots, C_m \}$ of 3-element subsets built from a universe X with $|X| = 3q$ (a multiple of 3), can we find exactly q subsets of C to cover X? We reduce this X3C problem into our problem such that $S_i = C_i \cup \{a^i_1, \ldots, a^i_{|X|+1}\}\ \forall i \in \{ 1, \ldots, m \}$. All the artificial a^i_j elements added are different and not in universe X. Each set S_i has thus a cardinality of $3 + |X| + 1 = 4 + 3q$. We are looking for a collection of sets such that their union is of size $k = q(4 + 3q)$. Only q sets can be selected; even when counting just the artificial elements ($|X| + 1 = 3q + 1$ per set), more than q sets is not possible because $k - ((q + 1) \cdot (3q + 1)) < 0$. Fewer then q sets is also not possible because $k - ((q - 1) \cdot (3q + 1)) > |X|$ and hence there would need to be more than $|X|$ unique elements in universe X to achieve cardinality k. For exactly q sets, one can verify that these q sets will cover $|X|$ after the removal of the $q(|X| + 1)$ added elements: $q(4 + 3q) - q(|X| + 1) = q(4 + 3q - 3q - 1) = 3q = |X|$. □

Corollary 3.1. Given a collection of sets $\{S_1, \ldots, S_m\}$, the problem of finding a subset of these such that their intersection is of fixed cardinality k is NP-hard.

Proof. We reduce the problem of Theorem 3.1. Let $X = \bigcup_i S_i$ be the universe and $\overline{S}_i = X \setminus S_i$ the complement set of S_i w.r.t. X. There exists a subset $\Omega \subseteq \{1, \ldots, m\}$ such that $|\bigcup_{i \in \Omega} S_i| = k$ if and only if $\left|\bigcap_{i \in \Omega} \overline{S}_i\right| = |X| - k$. □

Theorem 3.2. *Determining the satisfiability for CoverSize is NP-hard.*

Proof. The problem of Corollary 3.1 is reduced to finding a feasible solution for *CoverSize*$([I_1, \ldots, I_m], \mathcal{D}, c = k)$ with $\mathcal{D}(I_i)$ the bitvector representation for set S_i. □

Despite this hardness result, we can still propagate many conditions efficiently. The hardest part is to propagate from an upper bound on c to the item variables.

***CoverSize* Propagator**

We denote by $U = \{ I_i \in \mathcal{I} \mid dom(I_i) = \{ 0, 1 \} \}$ the set of undecided items and by $P = \{ I_i \in \mathcal{I} \mid dom(I_i) = \{ 1 \} \}$ the set of included items. The filtering rules for *CoverSize* are:

1. (*Rule* 1) computes the maximum cover size (exact upper-bound) that corresponds to discarding all the undecided items:

$$\max(c) \leq \left| \bigcap_{I_j \in P} \mathcal{D}(I_j) \right| \tag{3.7}$$

2. (*Rule* 2) computes the minimum cover size (exact lower-bound) that corresponds to including all the undecided items:
$$\min(c) \geq \left| \bigcap_{I_j \in (P \cup U)} \mathcal{D}(I_j) \right| \tag{3.8}$$
3. (*Rule* 3) discards item I_i if including it would result in a cover size that is below the minimum threshold:
$$\forall I_i \in U : \left| \bigcap_{I_j \in P} \mathcal{D}(I_j) \cap \mathcal{D}(I_i) \right| < \min(c) \implies I_i = 0 \tag{3.9}$$
This rule is also implemented in [LLL+16] and can be achieved in the decomposition with a redundant constraint for each item separately.
4. (*Rule* 4) detects mandatory items. If the lower-bound is equal to the maximum allowed cover size, then if the cover size lower-bound would increase while excluding an item I_i then this item I_i is mandatory. $\forall I_i \in U$:
$$\left(\left| \bigcap_{I_j \in (P \cup U)} \mathcal{D}(I_j) \right| = \max(c) \right) \wedge \left(\left| \bigcap_{I_j \in (P \cup U)} \mathcal{D}(I_j) \right| < \left| \bigcap_{I_j \in (P \cup U) \setminus \{I_i\}} \mathcal{D}(I_j) \right| \right) \implies I_i = 1 \tag{3.10}$$

Algorithm 3.2 gives the filtering algorithm for the *CoverSize* constraint implementing the Rules 1-4. N denotes the newly bound item variables since the previous call to the `propagate` method. The algorithm is thus incremental.

The block at Line 5 updates the current cover to reflect the new items included in the itemset[2]. The second block at Line 8 filters out the items that if included would induce a cover size below the allowed threshold min(`c`). This corresponds to Rule 3.

Line 11 computes the upper-bound of the cover size according to Rule 1. The lower-bound (Line 12) is obtained by including all the unbound items according to Rule 2.

Line 13 is triggered when the smallest *possible* intersection size (*lb*) is the largest *allowed* size of the frequency variable (max(`c`)). In this case, all the

[2]This is similar to the update of *currTable* in [DHL+16] for filtering table constraints.

items that are mandatory to reach the lower-bound can be forcefully included (this is not necessarily true when $\min(\texttt{c}) = \max(\texttt{c})$ as $\min(\texttt{c})$ can be externally set, resulting in $\min(\texttt{c}) > \texttt{lb}$). An unbound item I is mandatory if it is the only item that does *not* contain a transaction that all the other unbound and included ones do; in that case $\texttt{lb}$ would increase to $\texttt{lb}' > \max(\texttt{c})$. In the algorithm, $\texttt{m} \leftarrow \texttt{cover}\ \&_{I_j \in U \setminus I_i} \mathcal{D}(I_j)$ is the cover if one would include all unbound items except I. If $\texttt{m} \not\subseteq \mathcal{D}(I_i)$ then $\texttt{m} \cap \mathcal{D}(I_i)$ would be a smaller set than m and hence I_i is mandatory to obtain the smallest cover size lb.

Example 3.3. For the example of Figure 2.1, let C be included then $\texttt{lb} = 1, \texttt{ub} = 3$ (See illustration in Figure 3.2). Let $\max(\texttt{c}) = 1$ then A is mandatory: not including it results in $m = \{2, 4\}, m \not\subseteq \{1, 3, 4\}$ because of transaction 2.

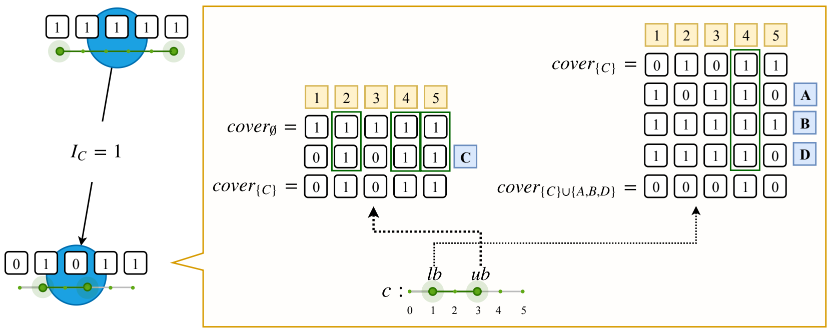


Figure 3.2. Computing $min(c)$ and $max(c)$ when C is included.

This condition is equivalent to Rule 4 but slightly more efficient to compute as it does not require to consider every non-zero words by returning true as soon as one word of $\texttt{m}$ is not included.

Time and space complexity

The time complexity for executing propagate is $\mathcal{O}(|\mathcal{I}| \times\ m/64)$ with $|\mathcal{I}|$ the number of items and $m/64$ the number of words necessary to represent the cover. In practice, the reversible sparse bitset will only iterate on the non-zero words in the cover bitvector.

Algorithm 3.2: Class CoverSize($[I_1, \dots I_m], \mathcal{D}, c$)

```
1  cover: RSparseBitSet                                        // Current cover
2  N,U                              // New bound variables, Unbound variables
3  $\mathcal{D}$                                   // $\mathcal{D}[I_i]$ = bit-set for item $I_i$
4  Method propagate()
       /* update current cover */
5      foreach variable $I_i \in$ N do
6          if $I_i = 1$ then
7              cover ← cover & $\mathcal{D}[I_i]$
       /* remove items that if included induce cover < min(c) */
8      foreach variable $I_i \in$ U do
9          if size(cover & $\mathcal{D}[I_i]$) < min(c) then
10             $I_i \leftarrow 0$
       /* cover bounds */
11     ub ← size(cover); max(c) ← min(max(c), ub)
12     lb ← size(cover $\&_{I_i \in U}$ $\mathcal{D}[I_i]$); min(c) ← max(min(c), lb)
       /* propagating maximum size */
13     if lb < ub ∧ lb = max(c) then
14         foreach variable $I_i \in$ U do
               /* include items mandatory for a cover size = lb */
15             m ← cover $\&_{I_j \in U \setminus I_i}$ $\mathcal{D}[I_j]$
16             if m $\not\subseteq \mathcal{D}[I_i]$ then
17                 $I_i \leftarrow 1$
```

The space complexity is $\mathcal{O}(|\mathcal{I}| \times \; m)$ similar to that of other approaches and due to the space needed to store the database.

Since domain consistency *CoverSize* is NP-hard we can unfortunately not clearly characterize[3] the filtering of Algorithm 3.2. Only in the case of an unconstrained max(c) (for instance for the frequent itemset problem), the filtering reaches domain consistency.

3.3.2 Closed itemsets: the *CoverClosure* constraint

The idea of mining for *closed* frequent itemsets is to reduce the set of extracted itemsets to a smaller, more interesting one. The intuitive idea is that if a

[3]As for many NP-hard global constraints like bin-packing, cumulative, circuit, etc.

frequent pattern has a cover that is exactly the same as a super pattern, then only the super pattern should be enumerated.

An itemset I is hence a *closed* itemset if there is no superset with the same cover (we recall that $\mathcal{H}$ represents a Horizontal Sparse database, see Table 2.1a):

$$\nexists I' \supset I : \{(t,T) \in \mathcal{H} \mid I' \subseteq T\} = \{(t,T) \in \mathcal{H} \mid I \subseteq T\}.$$

Hence, the *closure* of an itemset can be computed by verifying which items could be added to the itemset without changing the cover:

$$clo_{\mathcal{H}}(I) = I \cup \Big\{ j \notin I \mid \Big\{(t,T) \in \mathcal{H} \mid I \cup \{j\} \subseteq T\Big\} = \Big\{(t,T) \in \mathcal{H} \mid I \subseteq T\Big\}\Big\}. \tag{3.11}$$

As argued in [BL04] there are two ways of interpreting the *closed* property when combined with other constraints: 1) of all closed itemsets, keep only those that satisfy the constraints 2) take the closure such that the new itemset satisfies all constraints

This can have far-reaching consequences. Let us take the database in Figure 2.1, where item B is in all transactions and so will be in all closed itemsets. If we now add the constraint $B \notin I$, then under interpretation 1 there would not be any valid itemset, while under interpretation 2 there is D, C, AD and ACD namely all closed itemsets when ignoring B. It should be clear that interpretation 1) should not be taken by default. On the other hand, enforcing interpretation 2) requires one to reason not in terms of local constraints but over valid solutions to the CSP, for example with dominance properties [NDGN13]. Nonetheless, interpretation 1) is valid as long as all constraints have the property that adding a cover-preserving item to the set can never violate another constraint; for example, one can freely add a minimum size or maximum frequency constraint [BL04]. In case of such constraints, it must be expressed as a preference over solutions of the CSP, e.g. by adding constraints each time a solution is found [NDGN13]. We hence propose to offer the widely used unconstrained closure operator as a separate *CoverClosure* constraint.

Another argument for separating the closure constraint is that in case of discriminative itemset mining, we may want to enforce closedness on only one of the two databases or the entire database [GNDR11]. A separate constraint allows this freedom.

CoverClosure **Propagator**

Two filtering rules are enforced similar to [LLL+16]:

1. (*Rule* 5) Closure inclusion. This rule checks for each unbound item I_i if including it would result in an unchanged cover. If yes, this item should be included in the final pattern. More formally

$$\forall I_i \in U : \left(\bigcap_{I_j \in P} \mathcal{D}(I_j) \cap \mathcal{D}(I_i) \right) = \left(\bigcap_{I_j \in P} \mathcal{D}(I_j) \Longrightarrow I_i = 1 \right)$$

2. (*Rule* 6) Closure exclusion. This rule detects if extending the pattern with an item would result in a cover for which there is an already excluded item that should be added by the closure operator. Hence, including the first item would lead to an inconsistency and so it should be excluded. More formally, assuming $I_k \in \mathcal{I}, I_k = 0$ represents the excluded items:

$$\begin{aligned} &\forall I_i \in U, I_k \in \mathcal{I}, I_k = 0 \\ &\left(\Big((\bigcap_{I_j \in P} \mathcal{D}(I_j)) \cap \mathcal{D}(I_i) \Big) \subseteq \Big((\bigcap_{I_j \in P} \mathcal{D}(I_j)) \cap \mathcal{D}(I_k) \Big) \right) \Longrightarrow I_i = 0 \end{aligned} \tag{3.12}$$

Algorithm 3.3 implements the domain consistent filtering for the *CoverClosure* constraint. This constraint also uses the `RSparseBitSet` data structure to store the cover. It has a complexity of $O(|\mathcal{I}|^2 \times \ m/64)$.

A faster (but not domain consistent) filtering is obtained by replacing Rule 6 with a consistency check verifying that for each discarded item ($I_i = 0$), including it changes the cover:

$$\forall I_k \in \mathcal{I}, I_k = 0 : \left(\bigcap_{I_j \in P} \mathcal{D}(I_j) \cap \mathcal{D}(I_k) \right) = \bigcap_{I_j \in P} \mathcal{D}(I_j) \Longrightarrow fail$$

This version has a complexity of $O(|\mathcal{I}| \times \ m/64)$, similar to *CoverSize*.

3.4 Frequency-based itemset mining with *CoverSize* and *CoverClosure*

3.4.1 Frequent itemset mining

Our model for frequent itemset mining contains just one *CoverSize* and a constraint that the size of the cover is above a fixed minimum frequency θ:

$$\begin{cases} \text{enumerate } \mathit{CoverSize}([I_1,\dots,I_m],\mathcal{D},c) \\ c \geq \theta \end{cases} \tag{3.13}$$

Notice that as c is a variable, one can also add a maximum frequency constraint, or use it in branch-and-bound to search for the most frequent itemsets under constraints such as a minimum itemset cardinality:

$$\begin{cases} \text{maximize } c \\ \quad \text{s.t.} \\ \quad \mathit{CoverSize}([I_1,\dots,I_m],\mathcal{D},c) \\ \quad \sum_i I_i \geq \beta \end{cases} \tag{3.14}$$

3.4.2 *Closed* frequent itemset mining

Looking for the frequent closed itemset amounts to adding *CoverClosure*:

$$\begin{cases} \text{enumerate} & \mathit{CoverSize}([I_1,\dots,I_m],\mathcal{D},c) \\ & \mathit{CoverClosure}([I_1,\dots,I_m],\mathcal{D}) \\ c \geq \theta \end{cases} \tag{3.15}$$

As explained in Sect. 3.3.2, other constraints should only be added if they do not constrain the addition of (frequency-preserving) items.

3.4.3 Discriminative (closed) itemset mining

Given a split database $\mathcal{D}^+$ and $\mathcal{D}^-$ containing positive (+) and negative (−) transactions defined on a same set of items, the objective is to find the highest scoring itemsets (discriminating one class over another) w.r.t. a correlation (discriminative) measure[4] such as accuracy, χ^2 measure (3.16),

[4]By convention for all formula here we use $0/0 = 0$ and $0 \log 0 = 0$.

Algorithm 3.3: Class CoverClosure($[I_1, \dots I_m], \mathcal{D}$)

```
cover: RSparseBitSet                                  // Current cover
N,U                        // New bound variables, Unbound variables
$\mathcal{D}$                                 // $\mathcal{D}[I_i]$ = bit-set for item $I_i$
Method propagate()
    /* update current cover                                        */
    foreach variable $I_i \in$ N do
        if $I_i = 1$ then
            cover ← cover & $\mathcal{D}[I_i]$
    /* Rule 5                                                      */
    foreach variable $I_i \in$ U do
        if cover = (cover & $\mathcal{D}[I_i]$) then
            $I_i \leftarrow 1$
    /* Rule 6                                                      */
    foreach variable $I_i \in$ U do
        foreach variable $I_k \in \mathcal{I}$ with $I_k = 0$ do
            if (cover & $\mathcal{D}[I_i]$) ⊆ (cover & $\mathcal{D}[I_k]$) then
                $I_i \leftarrow 0$; break
```

information gain (3.17), Gini index (3.18), etc. Those itemsets are interesting as classification rules directly [BZ05, CYHP08, FZC+08], or as features (if the itemset is present or not) for another classifier [BZDRN06, DKWK05].

Some ZDC functions

Assume I, $\mathcal{D}^+$, and $\mathcal{D}^+$ be an itemset and the positive and negative transactions, the ZDC measure is a function of four parameters: $|\mathcal{D}^+|$, $|\mathcal{D}^-|$, n^+, and n^- which are respectively the size of positive ($|\mathcal{D}^+|$) and negative ($|\mathcal{D}^-|$) transactions, and the number of positive and negative transactions cover by I. The measures are the follows.

$$\begin{aligned}\chi^2(|\mathcal{D}^+|, |\mathcal{D}^-|, n^+, n^-) \;&=\; Q(|\mathcal{D}^+|, |\mathcal{D}^-|, n^+, n^-)+ \\ &\quad\; Q(|\mathcal{D}^+|, |\mathcal{D}^-|, |\mathcal{D}^+| - n^+, |\mathcal{D}^-| - n^-)\end{aligned} \tag{3.16}$$

where $Q(X,Y,x,y) = q(X,Y,x,y) + q(Y,X,y,x)$,
$q(X,Y,x,y) = \frac{\left(x - p(X,Y,x,y)\right)^2}{p(X,Y,x,y)}$
and $p(X,Y,x,y) = (x+y)\frac{X}{X+Y}$

$$\begin{aligned}\mathrm{Gain}(|\mathcal{D}^+|, |\mathcal{D}^-|, n^+, n^-) &= H(\tfrac{|\mathcal{D}^+|}{|\mathcal{D}^+|+|\mathcal{D}^-|})- \\ &\tfrac{n^+ + n^-}{|\mathcal{D}^+|+|\mathcal{D}^-|} H(\tfrac{n^+}{n^+ + n^-})- \\ &\tfrac{|\mathcal{D}^+|+|\mathcal{D}^-|-n^+-n^-}{|\mathcal{D}^+|+|\mathcal{D}^-|} H(\tfrac{|\mathcal{D}^+|-n^+}{|\mathcal{D}^+|+|\mathcal{D}^-|-n^+-n^-})\end{aligned} \tag{3.17}$$

where $H(p) = -p \log p - (1-p)\log(1-p)$ is the entropy.

$$\begin{aligned}\mathrm{Gini}(|\mathcal{D}^+|, |\mathcal{D}^-|, n^+, n^-) &= G(\tfrac{|\mathcal{D}^+|}{|\mathcal{D}^+|+|\mathcal{D}^-|})- \\ &\tfrac{n^+ + n^-}{|\mathcal{D}^+|+|\mathcal{D}^-|} G(\tfrac{n^+}{n^+ + n^-})- \\ &\tfrac{|\mathcal{D}^+|+|\mathcal{D}^-|-n^+-n^-}{|\mathcal{D}^+|+|\mathcal{D}^-|} G(\tfrac{|\mathcal{D}^+|-n^+}{|\mathcal{D}^+|+|\mathcal{D}^-|-n^+-n^-})\end{aligned} \tag{3.18}$$

where $G(p) = 1 - p^2 - (1-p)^2$.

Discriminative problem and its CP model

Using *accuracy* as a discriminative measure leads to the following problem (n^+ and n^- are CP decision variable, representing the cover size of the positive and negative transactions):

$$\begin{cases}\text{maximize } n^+ - n^- \\ \quad \text{s.t.} \\ \quad \mathit{CoverSize}([I_1, \ldots, I_m], \mathcal{D}^+, n^+) \\ \quad \mathit{CoverSize}([I_1, \ldots, I_m], \mathcal{D}^-, n^-)\end{cases} \tag{3.19}$$

Another standard discriminative measure is the χ^2 one depicted on Figure 3.3. As explained in [NGDR09] the standard discriminative functions such as χ^2 have the property that they are zero on the diagonal (relative to the possible values of p, n) and convex (denoted by ZDC). A general ZDC-based model for discriminative itemset mining is composed of

- two constraints $\mathit{CoverSize}([I_1, \ldots, I_m], \mathcal{D}^+, n^+)$ and $\mathit{CoverSize}([I_1, \ldots, I_m], \mathcal{D}^-, n^-)$ to compute the cover size on the positive and negative transactions;
- a Zero Diagonal Convex constraint $\mathit{ZDC}(|\mathcal{D}^+|, |\mathcal{D}^-|, n^+, n^-, \mathit{score})$ that links n^+, n^- and *score* using a discriminative function (such as χ^2) to maximise;

- *CoverClosure*$([I_1, \ldots, I_m], \mathcal{D}^- \cup \mathcal{D}^+)$ to obtain the closedness property. Note that posting *CoverClosure* separately on the positive (negative) can decrease n^+ (n^-) and is hence not allowed for symmetric ZDC measures.

This approach which employs a separate *ZDC* constraint that takes only the cardinalities n^+ and n^- as input, is novel and favours reusability in a different context: it is itemset-agnostic, meaning that it could also be used for example to find discriminating sequences instead of itemsets. In [NGDR09] the authors also employ a global constraint for the discriminative itemset mining problem, but one that reasons at the transaction level with one variable per transaction. The filtering they achieve is stronger than our decomposition into three constraints. They perform what they call a redundant *look-ahead* filtering[5] on each item separately.

We now describe our filtering for *ZDC*, presented in the Algorithm 3.4 and illustrated in Figure 3.4. Because of the ZDC property, the minimum and maximum is located at one of the four corners of the box $[\min(n^+), \max(n^-)] \times [\min(n^-), \max(n^-)]$ (see Figure 3.3c), and hence only these extremes need to be computed for pruning $\min(score)$. Given a minimum value for $\min(score)$, for example as enforced during branch-and-bound maximisation, the value of $\max(n^+)$ and $\max(n^-)$ can be reduced as shown in the Algorithm 3.4 at Lines 14 & 18 and illustrated on Figures 3.4c & d. For example, on Figure 3.4c, the iso-curve corresponding to $\min(score)$ is visualised. The ZDC property implies that any larger score must lay outside of the region enclosed by the iso-curves. The gray zone on Figures 3.4c & d corresponds to inconsistent combinations for n^+ and n^-, hence discovering the new minimum for n^+ requires to find v such that $ZDC(|\mathcal{D}^+|, |\mathcal{D}^-|, v, \max(n^-)) = \min(score)$. To do that, we use a *dicotomic search* described at Lines 19. Any value larger than v for n^+ would be inconsistent. The upper-cardinality of n^+ is constrained and therefore the filtering of *CoverSize*$([I_1, \ldots, I_m], \mathcal{D}^+, n^+)$ based on this upper cardinality is important to prune the search tree. A similar reasoning is used to prune $\min(n^+)$ and $\min(n^-)$ (Lines 12 & 16). As specified at the Line 7, if the box $[\min(n^+), \max(n^-)] \times [\min(n^-), \max(n^-)]$ is under the iso-curve, as illustrated in Figure 3.4a, the algorithm throws an inconsistency error. Conversely, if the box is entirely above the iso-curve (Figure 3.4b) then the domain of *score* is entirely valid. In the case where the box is

[5]A related generic technique in CP is *shaving* [Lho05].

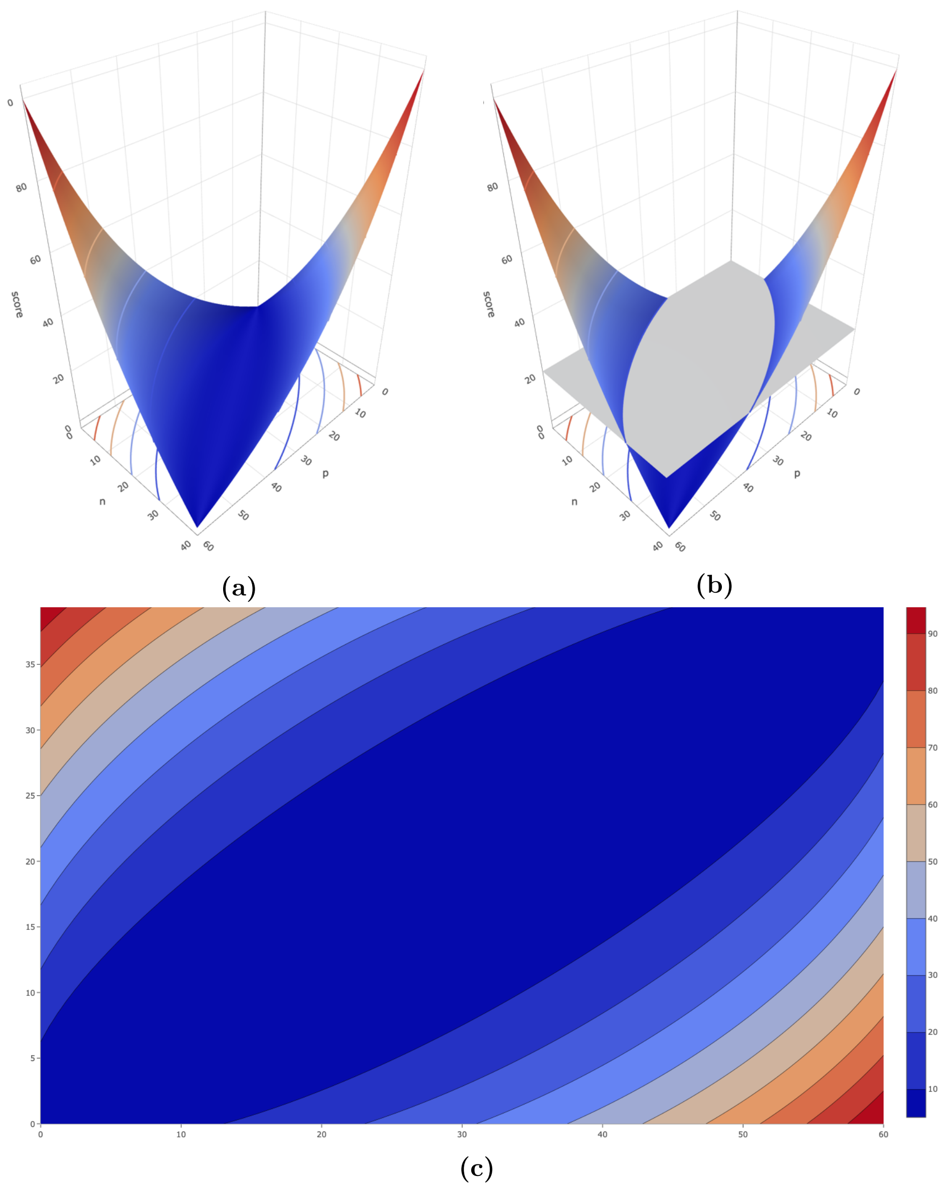


Figure 3.3. **a)** Plot of χ^2 ZDC function in $[0, |\mathcal{D}^+|] \times [0, |\mathcal{D}^-|]$ with $|\mathcal{D}^+| = 60$ and $|\mathcal{D}^-| = 40$; **b)** Same plot with the plane representing a threshold at $\chi^2 = 20$; **c)** Same plot with the score-axis projected in the pn-plane. Note: $p = n^+$, $n = n^-$

in the two zones above the iso-curve (Figure 3.4e), the lower bound of the score is updated to the minimum of $ZDC(|\mathcal{D}^+|, |\mathcal{D}^-|, \max(n^+), \min(n^-))$ and $ZDC(|\mathcal{D}^+|, |\mathcal{D}^-|, \min(n^+), \max(n^-))$ (Line 10). Therefore, one of the previous situations is retrieved.

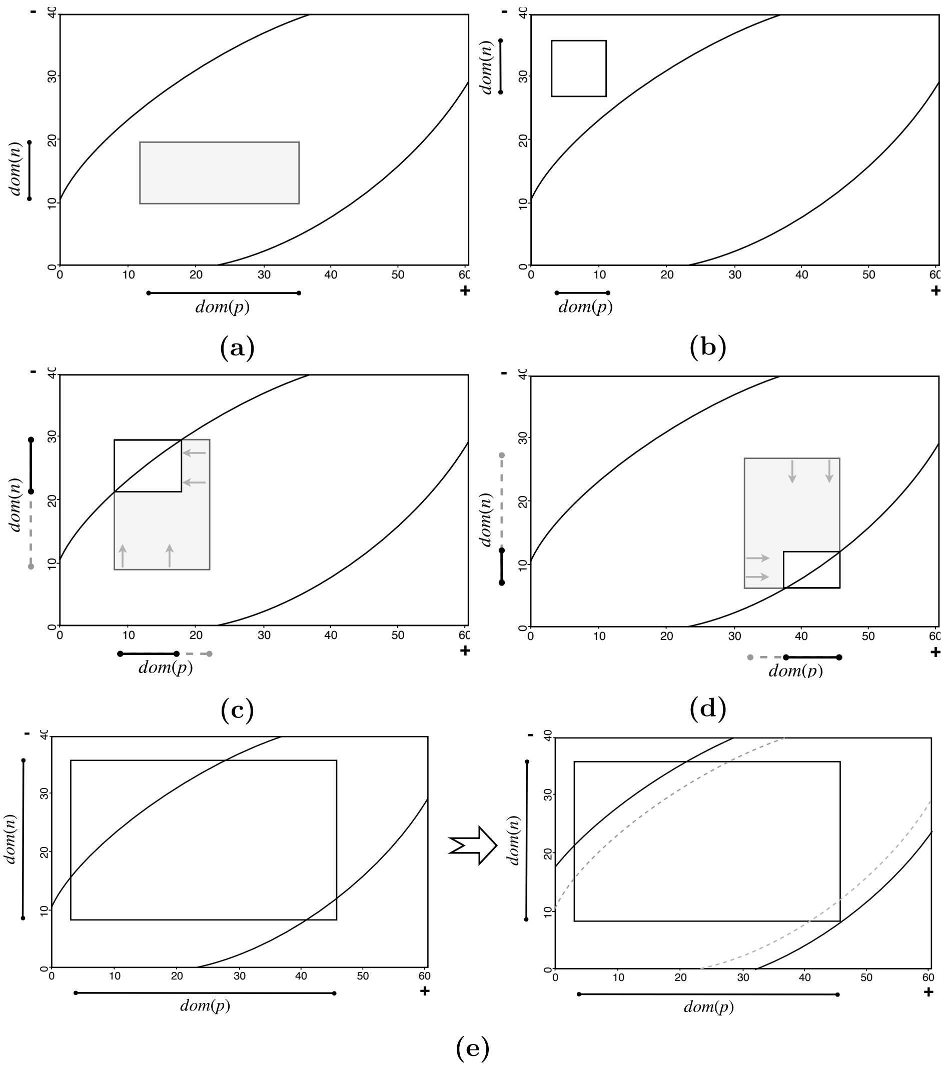


Figure 3.4. Several cases of χ^2 $ZDC(|\mathcal{D}^+|, |\mathcal{D}^-|, n^+, n^-, score)$ constraint filtering. Note: $p = n^+$, $n = n^-$ and iso-curve corresponding to $\min(score)$.

Algorithm 3.4: Class $ZDC(|\mathcal{D}^+|, |\mathcal{D}^-|, n^+, n^-, score)$

```
1  Method propagate()
2      $py \leftarrow max(n^+)$; $px \leftarrow min(n^+)$; $ny \leftarrow max(n^-)$; $nx \leftarrow min(n^-)$
3      $fpxny \leftarrow f(px, ny)$; $fpyny \leftarrow f(py, ny)$
4      $fpxnx \leftarrow f(px, nx)$; $fpynx \leftarrow f(py, nx)$
5      $fx \leftarrow min(score)$
       // when out of both opposite corner $\Longleftrightarrow$ not correct f
6      if $fpxny < fx \land fpynx < fx$ then
7          throw Inconsistency
       // ub=the max between top-left (ftl) and bottom-right (fbr) values of f
8      $max(score) = max(fpynx, fpxny)$
       // lb= min(ftl, fbr) if both ftl and fbr are over/under diagonal and valid
9      if $\left(\frac{ny}{|\mathcal{D}^-|} < \frac{px}{|\mathcal{D}^+|} \land \frac{nx}{|\mathcal{D}^-|} < \frac{py}{|\mathcal{D}^+|}\right) \lor$
       $\left(\frac{ny}{|\mathcal{D}^-|} > \frac{px}{|\mathcal{D}^+|} \land \frac{ny}{|\mathcal{D}^-|} > \frac{py}{|\mathcal{D}^+|} \land fpynx \geq fx \land fpxny \geq fx\right) \lor$
       $\Big(|dom(n^+)| = 1 \land |dom(n^-)| = 1\Big)$ then
10         $min(Score) = min(fpynx, fpxny)$
       // Pruning of $n^+$ and $n^-$
       // both bottom-(left,right) f points are out of corner $\Rightarrow$ GROW $min(n^-)$
11     if $fpxnx < fx \land fpynx < fx$ then
12         $min(n^-) \leftarrow$ dicotomicSearch$(nx, ny, px, fx)$
       // both top-(left,right) f points are out of corner $\Rightarrow$ DECREASE $max(n^-)$
13     if $fpxny < fx \land fpyny < fx$ then
14         $max(n^-) \leftarrow$ dicotomicSearch$(nx, ny, py, fx)$
       // both left-(bottom,top) f points are out of corner $\Rightarrow$ GROW $min(n^+)$
15     if $fpxnx < fx \land fpxny < fx$ then
16         $min(n^+) \leftarrow$ dicotomicSearch$(px, py, nx, fx)$
       // both right-(bottom,top) f points are out of corner $\Rightarrow$ DECREASE
       $max(n^+)$
17     if $fpyny < fx \land fpynx < fx$ then
18         $max(n^+) \leftarrow$ dicotomicSearch$(px, py, ny, fx)$
19 Method dicotomicSearch($x$, $y$, $t$, $s$)
20     $a \leftarrow x$; $b \leftarrow y$
21     if $a > b$ then
22         $b, a \leftarrow swap(a, b)$
23     while $b - a > 1$ do
24         $mid \leftarrow a + (b - a)/2$
25         if $f(t, mid) < s$ then
26             $a \leftarrow mid$
27         else
28             $b \leftarrow mid$
29     return $b$
```

3.5 Implementation and Practical User Guide

The implementation of the problem of Discriminative FIM in Scala is shown in A.1 as well the Reversible Sparse bit-set data structure A.2 and *Cover-size* A.3. The other implementations are available in the CP-Solver OscaR [Osc12] which is available online[6] in free access. In OscaR, one can combine our constraints with the existing constraints in the solver such as All-Different [Rég94], Global cardinality [QLvBG04], Grammar [QW06], etc.

For developers who are willing to modify the code directly, a lightweight version is also available[7]. One can hence add several constraints for a specific usage without understanding OscaR deeply. The installation procedure is described in the Install file in the code directory and merely consists of "importing the project" into your favourite IDE.

Users can directly download the *jar-file* which is available on our website [8]. To find the frequent itemsets with $\theta = 600$ given the database mushroom ($|\mathcal{T}| = 8124$ and $|\mathcal{I}| = 119$), we run this command

```
java -jar coversize.jar F mushroom 600
```

and here is the output:

```
/** CoverSize for FIM (OscaR Solver) v1.0
    Bugs reports : johnaoga@gmail.com , pschaus@gmail.com
    */

Start FIM on mushroom
support:600 nTrans:8124 nItems:120
nNodes: 1890618
nFails: 945310
time(ms): 3519
completed: true
timeInTrail: 278
nSols: 945310

...done fim.examples.CoverSizeRunner$
0 0 3.519 945310 0 1890618 945310 0 0
0.0 0.0
```

[6]https://bitbucket.org/oscarlib/oscar/wiki/Home
[7]https://projetsJOHN@bitbucket.org/projetsJOHN/coversize
[8]https://sites.uclouvain.be/cp4dm/fim/

Table 3.1. CPU runtime for several algorithms vs *CoverSize*.(TO≡TimeOut; * ≡CoverSize+CoverClosure; $\rho \equiv density = \frac{1}{|\mathcal{T}| \times |\mathcal{I}|} \sum_{t \in \mathcal{T}, i \in \mathcal{I}} \mathcal{D}_{ti}$)

Name $\lvert\mathcal{T}\rvert \times \lvert\mathcal{I}\rvert$ $\rho(\%)$	θ	Frequent								Closed						
		CP-based				Specialized				CP-based				Specialized		
		FIMCP	*DMCP*	*CoverSize-bitset*	*CoverSize*	*Apriori*	*Eclat*	*Nonordfp*	*LCMv3*	*FIMCP*	*ClosedPattern*	*CoverSize-DC**	*CoverSize**	*DMCP*	*Apriori-close*	*LCMv3*
retail 88162 × 16470 ($\rho = 0.06$)	80	TO	6.91	25.76	5.33	0.60	5.81	0.98	**0.21**	TO	TO	394.48	45.09	16.95	0.82	**0.26**
	60	TO	10.45	33.87	7.37	0.71	8.26	1.31	**0.24**	TO	TO	952.83	67.74	25.10	1.03	**0.31**
	40	TO	15.96	65.13	11.19	0.77	11.42	1.83	**0.27**	TO	TO	TO	125.67	41.78	1.29	**0.49**
	20	TO	26.81	132.53	19.74	1.10	17.86	2.56	**0.43**	TO	TO	TO	226.32	94.24	1.61	**0.48**
	10	TO	40.03	191.05	37.08	1.73	24.63	3.68	**0.39**	TO	TO	TO	366.83	238.71	2.48	**0.66**
online-retails 541909 × 2603 ($\rho = 0.1$)	70	TO	11.00	54.19	8.27	2.75	14.27	**0.31**	1.59	TO	TO	242.80	98.67	11.00	**1.28**	1.43
	40	TO	11.33	59.60	8.00	4.78	15.07	**0.40**	1.43	TO	TO	497.14	111.34	12.06	**1.28**	1.51
	10	TO	11.49	86.66	8.49	2.15	15.61	**0.43**	1.51	TO	TO	907.68	131.94	13.31	**1.36**	1.52
	5	TO	15.64	84.05	8.82	2.13	14.56	**0.31**	1.32	TO	TO	TO	148.29	13.72	**1.31**	1.65
	1	TO	TO	TO	TO	2.18	15.66	**0.42**	1.22	TO	TO	TO	TO	14.99	**1.44**	1.60
BMSWebView1 59602 × 497 ($\rho = 0.5$)	48	TO	1.92	1.51	0.69	0.08	0.51	0.11	**0.03**	TO	TO	3.10	2.54	48.04	0.29	**0.11**
	36	TO	17.83	7.23	1.87	0.88	0.37	0.30	**0.11**	TO	TO	3.77	3.74	512.09	1.55	**0.26**
	34	TO	63.93	23.07	8.12	7.04	0.43	0.60	**0.10**	TO	TO	3.99	4.57	746.61	10.46	**0.37**
	32	TO	TO	TO	TO	TO	0.68	60.63	**0.28**	TO	TO	5.12	8.24	TO	TO	**0.47**
	30	TO	TO	TO	TO	TO	**0.53**	TO	1.22	TO	TO	6.61	13.50	TO	TO	**0.67**
T10I4D100K 100000 × 870 ($\rho = 1.0$)	500	TO	1.07	3.58	0.79	0.48	3.32	0.80	**0.36**	TO	8.32	8.20	7.81	1.04	0.81	**0.34**
	400	TO	0.98	4.1	1.03	0.49	3.74	0.58	**0.29**	TO	13.06	9.27	8.88	1.12	0.67	**0.43**
	300	TO	1.39	4.96	1.27	0.69	4.14	0.94	**0.30**	TO	25.28	11.12	10.51	1.30	0.85	**0.41**
	200	TO	2.65	6.59	1.28	0.63	3.95	0.70	**0.39**	TO	77.20	12.95	10.98	1.61	1.07	**0.38**
	100	TO	2.35	7.27	1.78	1.02	5.08	1.01	**0.53**	TO	125.26	15.79	15.07	2.11	1.15	**0.58**
pumsb-star 49046 × 2088 ($\rho = 2.0$)	18000	302.06	1.02	1.26	0.84	6.34	0.77	0.25	**0.21**	TO	56.55	0.99	0.89	2.11	5.11	**0.22**
	16000	375.07	2.57	2.55	1.52	18.81	1.04	0.27	**0.22**	TO	120.33	0.80	0.93	4.04	15.59	**0.27**
	14000	563.21	9.14	4.72	3.36	81.93	1.34	0.31	**0.26**	TO	275.23	1.65	2.25	6.22	57.99	**0.30**
	12000	TO	33.66	20.12	10.16	285.36	1.95	0.62	**0.38**	TO	601.72	4.51	5.55	14.04	284.39	**0.44**
	10000	TO	TO	TO	TO	TO	3.45	164.76	**0.45**	TO	TO	10.37	13.97	26.96	TO	**0.54**
pumsb 49046 × 2113 ($\rho = 3.0$)	40000	237.09	2.82	2.02	1.51	1.45	0.87	**0.14**	0.15	TO	212.83	1.84	2.14	7.22	1.46	**0.15**
	35000	889.08	33.87	13.20	10.26	15.92	3.36	0.21	**0.20**	TO	TO	12.71	16.76	43.48	15.98	**0.27**
	30000	TO	220.03	124.65	63.91	356.90	10.55	0.54	**0.27**	TO	TO	62.81	82.47	121.24	370.78	**0.60**
	25000	TO	TO	TO	602.72	TO	66.46	3.56	**1.04**	TO	TO	468.99	611.62	TO	TO	**1.54**
accidents 340183 × 468 ($\rho = 7.0$)	300000	50.02	0.78	0.16	**0.05**	1.23	1.02	0.23	0.21	221.80	0.13	0.08	**0.07**	0.80	1.48	0.19
	250000	55.69	1.08	0.18	**0.17**	1.75	1.26	0.33	0.40	253.90	1.71	0.17	**0.14**	1.16	2.00	0.21
	200000	112.55	0.99	0.34	**0.33**	2.46	1.76	0.35	0.62	302.97	8.57	**0.56**	0.68	1.41	2.55	0.56
	150000	386.51	2.87	1.52	1.32	17.83	3.12	**0.52**	1.06	575.32	61.60	5.00	5.09	3.71	18.91	**0.99**
	100000	TO	18.08	10.69	7.79	116.26	7.32	**0.74**	1.73	TO	570.38	47.55	43.75	23.63	140.26	**1.47**
mushroom 8124 × 119 ($\rho = 19.0$)	600	33.06	0.85	2.14	1.92	14.61	0.30	0.05	**0.04**	8.38	0.62	0.73	0.84	0.16	10.09	**0.06**
	400	106.71	3.48	5.52	5.43	58.46	0.29	0.09	**0.04**	14.86	1.08	0.70	0.67	0.20	36.11	**0.11**
	200	449.85	16.77	23.37	20.10	133.54	0.37	0.27	**0.07**	20.12	2.35	0.75	1.56	0.35	112.82	**0.17**
	100	TO	67.09	96.13	68.46	264.69	0.65	0.95	**0.10**	27.10	4.30	1.11	2.91	0.63	284.15	**0.24**
soybeans 630 × 50 ($\rho = 32.0$)	16	1.21	0.47	1.02	1.03	0.45	0.03	0.02	**0.00**	0.31	0.20	0.36	0.35	0.08	0.71	**0.02**
	13	1.57	0.70	1.40	1.44	0.44	0.03	0.02	**0.01**	0.32	0.25	0.32	0.28	0.12	0.95	**0.02**
	10	2.54	0.75	1.69	1.44	0.71	0.04	0.03	**0.02**	0.34	0.29	0.40	0.29	0.11	1.20	**0.02**
	7	3.54	1.46	2.35	2.07	0.84	0.05	0.03	**0.01**	0.47	0.33	0.23	0.22	0.14	1.61	**0.03**
	4	7.05	3.86	4.02	3.87	1.69	0.05	0.07	**0.02**	0.42	0.42	0.25	0.20	0.18	2.98	**0.03**
chess 3196 × 75 ($\rho = 49.0$)	2000	3.71	0.30	1.59	1.39	3.14	0.11	0.01	**0.01**	1.85	1.71	1.11	1.02	0.42	3.48	**0.09**
	1500	46.05	3.05	4.81	3.88	77.85	0.39	0.11	**0.07**	14.06	15.53	4.73	3.42	1.88	69.56	**0.59**
	1000	577.29	35.16	52.60	44.94	849.49	2.15	0.68	**0.37**	101.68	96.65	28.11	22.76	14.96	885.14	**4.90**
	500	TO	959.16	TO	TO	TO	19.06	12.35	**2.96**	882.16	900.45	304.74	282.50	144.04	TO	**51.66**
	250	TO	TO	TO	TO	TO	72.69	129.05	**14.42**	TO	TO	TO	TO	580.64	TO	**211.99**

3.6 Experiments

In this section, we report the experimental results on frequent, closed as well as discriminative itemset mining. A concrete question drives each experiment. All experiments were run in the JVM with maximum memory set to 8GB on PCs with Intel Core i5 64bits processor (2.7GHz) and 8GB of RAM running

Linux Mint 17.3. Execution time is limited to 1000 seconds.

Datasets and mining algorithms. We use data from the FIMI[9] repository and from the CP4IM[10] website. The properties of the datasets are presented in Table 3.1 (first column) and Table 3.2a (these latter are labelled positive/negative datasets). We compare with the following methods:

- Frequent Itemset Mining: *FIMCP* [GNDR11] using the Gecode solver [Gec06], *DMCP* [NG10] a custom CP bitvector solver, and four dedicated algorithms namely Borgelt's *Apriori* and *Eclat* implementations [Bor03], *Nonordfp* [Rác04] and *LCMv3*[11] [UKA05].
- Closed Frequent Itemset Mining: *FIMCP*, *DMCP*, Borgelt's Apriori and LCMv3 again, as well as *ClosedPattern* [LLL$^+$16] using the or-tools solver [Goo15].
- Discriminative Itemset Mining: *CIMCP* [GNDR11] based on Gecode and the specialised algorithm *corrmine* [NGDR09].

We denote our approach by *CoverSize* and it uses the OscaR solver [Osc12].

Q1: What is the impact of using a reversible *"sparse"-bitset* over a reversible non-sparse one? In Table 3.1 *CoverSize-bitset* is the same implementation as *CoverSize* but using a reversible bitset implementation that does not check for zero words. The results on *Frequent* in Table 3.1 convincingly show that using the sparse data structure is always better and sometimes an order of magnitude faster, especially on large and sparse datasets.

For closed, we can also compare *Coversize-DC** to *ClosedPattern* [LLL$^+$16] which uses the same filtering rules but in a single global constraint and with a different solver and non-reversible non-sparse bitsets [LLL$^+$16]. We only have the binaries, and although different solvers will perform differently, *or-tools* has won *MiniZinc* challenge [SBF10] gold medals and so the remarkable difference in runtime with our method provides strong evidence that the reversible sparse bitset is a well suited and very scalable data structure for itemset propagators.

[9] http://fimi.ua.ac.be/data/

[10] https://dtai.cs.kuleuven.be/CP4IM/datasets/

[11] http://research.nii.ac.jp/~uno/codes.htm (v3 is fastest of all versions in our experiments)

Q2: Is domain consistency interesting for closed frequent itemset mining? In [LLL$^+$16] the authors concluded that using the domain consistent version of Rule 6 dominates the simpler non-domain consistent one because of the resulting reduction in number of explored nodes. We reran the same experiment, *CoverSize-DC** and *CoverSize** in Table 3.1, and the conclusion changes when using reversible sparse bitsets: while on pumsb and pumsb-star the runtime increases when using the simpler non-DC version, on the other datasets it is similar or faster to use the simpler one. For the largest and sparsest datasets retail and online-retails, the difference is even up to an order of magnitude.

Q3: How does *CoverSize* compare with existing approaches? The *CoverSize* approach clearly outperforms the decomposition-based *FIMCP*. For frequent, *CoverSize* is on par (sometimes somewhat better or worse) with *DMCP*, the dedicated CP solver which uses bitvector variables. By profiling the execution we observed that for the instances where *DMCP* was faster (such as *mushroom*) only 1% of the time was spent in *CoverSize*. The remaining time is devoted to the solver (propagation management, search, trailing), which a dedicated solver like *DMCP* has less overhead in. Hence, here we show that similar performance can be achieved with a generic solver, through the use of global constraints with carefully designed data structures.

For closed, our approach outperforms *FIMCP*, and also *ClosedPattern* as discussed in Q2. The differences between *CoverSize* and *DMCP* become more varied and pronounced, for example for the sparsest retail and online-retails dataset in favor of DMCP, and for BMSWebView1 in favor of *CoverSize*.

Specialised algorithms. There remains a significant gap between CP-based methods and specialised methods though, and especially the highly praised LCMv3 algorithm lives up to its reputation. It should be pointed out that these algorithms do not allow any constraints and for example a version of LCM (LCMv5) that allow some constraints is also remarkably slower. For denser datasets, our method does typically outperform Apriori.

Q4: What is the difference in performance for discriminative itemset mining with *CoverSize*? The state-of-the-art for this problem is the generic CP-based CIMCP method and the specialized *corrmine* method which implement the same bounds [NGDR09]. Table 3.2b shows a comparison using Information Gain as the ZDC measure, which is the one implemented

Table 3.2. Runtimes, in seconds, for discriminative itemset mining

a) Dataset features.				*b) Discriminative*		
Name	Dense	Trans	Item	CIMCP	CoverSize	corrmine
anneal	0.45	812	93	**0.167**	0.24	0.014
australian-cr	0.41	653	125	**0.166**	0.195	0.012
breast-wisc	0.5	683	120	**0.193**	0.345	0.037
diabetes	0.5	768	112	**1.564**	1.769	0.28
german-cr	0.34	1000	112	**1.521**	1.659	0.09
heart-clevel	0.47	296	95	**0.175**	0.221	0.055
hypothyroid	0.49	3247	88	0.592	**0.118**	0.016
ionosphere	0.5	351	445	1.047	**0.336**	0.23
kr-vs-kp	0.49	3196	73	0.698	**0.145**	0.01
letter	0.5	20000	224	54.255	**4.547**	0.367
mushroom	0.18	8124	119	15.979	**0.069**	0.025
pendigits	0.5	7494	216	2.939	**1.196**	0.138
primary-t	0.48	336	31	**0.02**	0.058	0.003
segment	0.5	2310	235	1.154	**0.15**	0.052
soybean	0.32	630	50	**0.046**	0.048	0.003
splice-1	0.21	3190	287	22.341	**0.113**	0.025
vehicle	0.5	846	252	0.551	**0.55**	0.094
yeast	0.49	1484	89	3.386	**1.366**	0.818
	Average. when found			5.933	0.729	0.126

in *corrmine*. Despite the stronger filtering of CIMCP, *CoverSize* outperforms *CIMCP* for the most challenging instances showing the importance of globals with a suitable data structure. *Corrmine* is superior though specialized to this specific problem.

3.7 Summary, Outlooks, Further readings

We showed that compared to the *ClosedPattern* approach [LLL+16] of using a global constraint for frequent closed itemset mining, both generality and efficiency can be significantly improved. Generality can be improved by a separation of concerns in terms of global constraints. We propose to use one global constraint that exposes the frequency through a decision variable which can then be used in other constraints (e.g. frequency constraints, objective functions or discrimination scores). Another global constraint can be used to enforce the closure property, though care has to be taken when combining it with other constraints.

Efficiency-wise we showed the connection with a well-known constraint that also has to handle a lot of data: the table constraint. Using the *Reversible Sparse bitset* data structure that was recently proposed [DHL+16] allows our global constraints to scale to even larger and sparser datasets while still in a generic CP solver. This is relevant not just for frequency-based itemset mining, but also for other existing as well as novel data mining problems in CP, and perhaps beyond.

Chapter 4

Frequent Itemset Mining for Compression

"The greatest artist is the simplifier."

–Donald M. Murray

Overview *One of the growing data mining task is the search for the most relevant subset of frequent patterns. This set is called pattern-set. Several measures are then used to judge its relevancy. In this chapter, we present a new method that searches for a small rule list (pattern-set) where each rule captures the probability of the Boolean target attribute being true. This rule list is able to compress data well.*

Contribution *Our contribution is a new method to discover a small rule list in labeled data which is built on a novel combination of two main building blocks: (i) the use of the Minimum Description Length (MDL) principle to characterize good-and-small sets of probabilistic rules, (ii) the use of branch-and-bound with a best-first search strategy to find better-than-greedy and optimal solutions for the proposed task. We experimentally show the effectiveness of our approach, by providing a comparison with other supervised rule learning algorithms on real-life datasets.*

Main source *This chapter is entirely based on our paper [AGNS18].*

4.1 Context and motivation

Rule learning in supervised data is a well-established problem in data mining and machine learning. Compared to many other methods, a clear benefit of rule-based methods is that the rule format is more comfortable to interpret and hence is useful in knowledge discovery. Well-known examples of rule learning are

Rule-based classification, in which the aim is to find a set of rules that predicts the class of examples well;

Subgroup discovery, in which the aim is to find a set of rules that describes subgroups of examples in the data; in these subgroups, the distribution of the target attribute is different from the overall population.

The main difference between subgroup discovery and rule-based classification is that rule-based classification aims to find a set of rules that can be applied on *any* example to obtain a prediction for that example. Subgroup discovery aims to characterise subgroups of examples, but not necessarily all examples.

Similar to rule-based classification, in this work we are also interested in finding a set of rules that describe a target attribute entirely and in an interpretable manner. However, we make a specific assumption that is not common in rule-based classification: we assume that the class attribute has a skewed distribution, and that exact prediction is certainly not possible. The following example illustrates a problem that has these characteristics.

Example 4.1. Assume that we characterise every minute in a year in terms of the following attributes: the part of the day the minute belongs to (morning, afternoon), the day the minute belongs to (Sunday, Monday, ...), the month the minute belongs to (January, ...) and the minute of the day (1, 2, ..., 24×60); furthermore, over a year we use a sensor to monitor when an individual opens a specific door in his house. Can we use rules to characterise when this individual opens her door?

In this example, the event of “opening a door” is expected to be a rare event; if we use a classification algorithm on the above dataset, we will notice that the class attribute is very unbalanced. Most classification algorithms will either prefer always to predict the default label (the door is closed) or will construct many very specific rules to cover the small number of examples that are the exception. The reason for this is that many rule-based classifiers find *lists* of rules; a rule that makes an error in its prediction, cannot be corrected by a later rule. Hence, most classification rule learning algorithms favour rules with lower recall but high precision.

In this work, we propose a new algorithm for finding rule lists, designed to work well in this specific setting. It identifies simple probabilistic rule lists, such as in Tables 4.1a & b; in contrast to other rule learners that will prefer default or very specific rules (in Table 4.1c).

Hence, the rule mining setting studied in this work can be characterised by these properties:

- it learns rules with probabilities in the head; these probabilities represent the class distribution for the examples covered by the rule, and should not be understood as class prediction;
- the list of rules is intended to characterise the class distribution over the entire data, in contrast to subgroup discovery;
- it favours smaller rule lists to ease interpretation.

Finding lists of rules that satisfy these requirements is not a straightforward task. To address these challenges, this work proposes the following contributions.

1. We propose a new optimisation criterion based on the *Minimum Description Length* (MDL) principle [Grü07]; this criterion aims to find small rule lists, yet characterizing the target distribution well.
2. We propose a new search algorithm based on branch-and-bound search; this search algorithm aims to find the global optimum for the proposed optimisation criterion under given constraints.

The approach that we take in this work is a *pattern set mining* approach. We first use itemset mining algorithms to find a candidate set of itemsets. From this set, we select a subset that describes the target attribute well. From the pattern set mining perspective, we propose a new supervised optimisation criterion for selecting a set of *free* patterns and a new search algorithm for finding a set of patterns that optimises the criterion.

Table 4.1. Probabilistic rule lists examples: **a)** From PRL over Door opening data (Probability=the probability that the door is opened); and From PRL**b)**, and **c)** SBRL over Mushroom data. (Probability=the probability that the mushroom is edible).

(a) PRL (Our approach) output

	rule list	Probability
IF	Wednesday ***and*** Morning	0.879
ELSE IF	Holidays ***and*** Thursday	0.011
ELSE IF	Thursday ***and*** Afternoon	0.987
ELSE IF	Sunday	0.001
ELSE	(default rule)	0.101

(b) PRL (Our approach) output

	rule list	Probability
IF	Gill-spacing is closed ***and*** No odor	0.95
ELSE IF	Gill-spacing is closed ***and*** Stalk-shape is tapering	0.0
ELSE IF	Stalk-color-above-ring is white ***and*** Gill-size is broad	1.0
ELSE IF	Gill-spacing is closed	0.0
ELSE	(default rule)	0.56

(c) SBRL (other rule learner) output

	rule list	Probability
IF	no bruises ***and*** odor is not-in-(none,foul)	0.00112
ELSE IF	odor is foul ***and*** gill-attachment is free	0.0007
ELSE IF	gill-size is broad ***and*** there is one ring	0.999
ELSE IF	stalk-root is unknown ***and*** stalk-surface-above-ring is smooth	0.996
ELSE IF	stalk-root is unknown ***and*** there is one ring	0.0385
ELSE IF	bruises is foul ***and*** veil-color is white	0.995
ELSE IF	stalk-shape is tapering ***and*** there is one ring	0.986
ELSE IF	habitat is paths	0.958
ELSE	(default rule)	0.001

4.2 Related Work

This work builds on a number of areas in the literature.
Rule-based classification. There is a large literature on rule-based classification; a good overview of these algorithms, including classic algorithms such as CN2 and RIPPER, can be found in a textbook by Fürnkranz et al. [FGL14]. Two types of rule-based classifiers can be distinguished: classifiers based on rule sets and on rule lists. In set-based classifiers, all rules that match an example are used to obtain a prediction for that example. In list-based

classifiers, the first matching rule is used; we build on this class of methods.

Covering algorithms are the most popular type of rule learning algorithm. These algorithms iteratively search for a rule to add to a rule set or list. Most often, in each iteration, a greedy search algorithm is used, which constructs a rule by iteratively adding the condition that improves the quality of the rule the most.

The main challenge faced by pure covering algorithms is that later rules cannot correct errors made by earlier rules in a rule list. Such algorithms hence need to favour precision over recall to obtain accurate classifiers. As a result rule lists may become unnecessarily long. One way to solve this is using *pruning*: the rule set is reduced in a post-processing step.

Pattern-based classification. Compared to traditional rule learning algorithms, pattern-based classifiers use pattern mining algorithms, such as frequent itemset mining algorithms, to identify candidate rules [ZN14]. These frequent itemsets are post-processed to construct rule sets or rule lists. Most of these post-processing approaches use heuristic search algorithms, although the use of exact search has also been studied [GNDR13].

Pattern set mining. From a pattern mining perspective, selecting a small set of patterns from a larger set of patterns can be seen as a pattern set mining problem [ZN14]. In contrast to unsupervised methods, supervised methods aim to find a balance between pattern sets that are non-redundant and that are accurate. One popular approach for evaluating the quality of a pattern set is based on the Minimum Description Length principle, as pioneered in the unsupervised setting by the *KRIMP* algorithm [VvLS11]. Exact methods for pattern set mining were studied by Guns et al. [GNDR13], among others, but these studies did not consider scoring functions based on MDL or did not exploit freeness, as we do.

Subgroup discovery. Strongly related to both pattern mining and rule-based classification is subgroup discovery. Subgroup discovery differs from classification in that it does not aim to build a predictive model; rather, subgroup discovery algorithms are intended to return small and interpretable sets of *local* patterns; subgroups are not necessarily ordered in a specific manner. For this reason, traditional subgroup discovery algorithms were modifications of covering-based rule-learning algorithms to allow for overlap between patterns explicitly [LKFT04].

Bayesian rule lists. Most related to this work is recent work by Yang et al. [YRS17] on probabilistic rule lists. This work also finds ordered lists of probabilistic rules. Contrary to our work, however, the aim of the work of Yang et al. is to identify accurate classifiers, and not to identify as small and interpretable representations of the class distribution as possible. Furthermore, Yang et al. use a sampling-based algorithm to identify good sets of patterns. We propose an alternative, exact algorithm in this work.

4.3 The probabilistic rule list mining problem

This work is motivated by the creation of a probabilistic rule list that summarizes labeled data well. In order to be easily interpretable, the rule list and the individual rules should be concise.

We assume a set of discrete attributes describing the data. These attributes can be represented as a set of Boolean properties using a one-hot encoding. These properties are referred to as items in the following, in line with the itemset mining literature.

More formally, let $\mathcal{I} = \{ 1, \cdots, m \}$ represent a set of m possible items and let $\mathcal{F} \subseteq 2^{\mathcal{I}}$ be a set of itemsets built on those items. A probabilistic rule list (PRL) built on $\mathcal{F}$ is a sequence of rules of the form $\mathcal{R} = \Big\langle (I^{(1)}, p^{(1)}), (I^{(2)}, p^{(2)}), \cdots, (I^{(k)}, p^{(k)}) \Big\rangle$ with $p^{(i)}$ being a probability and $I^{(i)} \in \mathcal{F}, \forall i = 1, \ldots, k-1$ and $I^{(k)} = \emptyset$. This latter is the default rule. The sequence of itemsets in the rule list can be expressed as membership to the regular language: $\langle I^{(1)}, \ldots, I^{(k)} \rangle \in \mathcal{L}(\mathcal{F}^* \cdot \emptyset)$ with $\mathcal{F}^*$ the *Kleene* operator on $\mathcal{F}$. Table 4.1 shows three example rule lists (generated from different data).

The rule list has a sequential interpretation, in that the set of data instances that match the first rule $I^{(1)}$ are assumed to have a positive label with probability $p^{(1)}$. The other data instances, those that do not match $I^{(1)}$, but do match $I^{(2)}$ have a probability of $p^{(2)}$ to be positive, etc. The final empty set $I^{(k)} = \emptyset$ hence captures all instances not matched by the other rules.

We now formalise the problem of creating the probabilistic rule list based on $\mathcal{F}$ and a dataset $\mathcal{D}$.

Definition 4.1. *As input we receive a set of itemsets $\mathcal{F}$ that can be used to compose the rule list, and a database $\mathcal{D}$ of instances, with for each a Boolean target attribute: $\mathcal{D} = \{ (t, I_t, a_t) \mid t \in \mathcal{T}, I_t \subseteq \mathcal{I}, a_t \in \{ +, - \} \}$, where the set $\mathcal{T}$ contains the instance or transaction identifiers $\mathcal{T} = \{ 1, \ldots, n \}$. The* *database can be split into a positive $\mathcal{D}^+$ and negative $\mathcal{D}^-$ database, based on the target attribute value ($+$ or $-$).*

The problem of finding a probabilistic rule list is formalised as:

$$\text{argmin}_{\mathcal{R}}\, \text{score}(\mathcal{R}, \mathcal{F}, \mathcal{D}) \tag{4.1}$$

where $\mathcal{R} = \left\langle (I^{(1)}, p^{(1)}), (I^{(2)}, p^{(2)}), \cdots, (I^{(k)}, p^{(k)}) \right\rangle$ is a probabilistic rule list such that $\langle I^{(1)}, \ldots, I^{(k)} \rangle \in \mathcal{L}(\mathcal{F}^* \cdot \emptyset)$ and score is an optimisation criterion. Various optimisation criteria can be defined, including criteria inspired by classification rule learning, subgroup discovery and pattern set mining. Our aim in this work is to develop an optimisation criterion that explicitly favours smaller rule lists that describe the entire target distribution well. For this purpose, we will use the Minimum Description Length principle[1], discussed in the next section.

4.4 Discovering probabilistic rule lists

4.4.1 Coverage and probability of a rule list

To evaluate the quality of a rule set on a given dataset, we will use some concepts taken from the itemset mining literature [AIS93]. In particular, we use the notions of Coverage and Frequency defined in the background chapter (Section 2.1.1 of Chapter 2) that we recall here.

Definition 4.2 (Coverage and support of an itemset)**.** *The set of transactions in a database $\mathcal{D}$ containing an itemset I is called the cover: $Cover_{\mathcal{D}}(I) = \{\, (t, I_t, a_t) \in \mathcal{D} \mid I \subseteq I_t \,\}$. The size of the cover is called the support $Freq_{\mathcal{D}}(I)$ $= |Cover_{\mathcal{D}}(I)|$.*

Example 4.2. An example itemset database is given in Figure 4.1. $I = \{\, A, C \,\}$ is an example itemset; $\varphi(\mathcal{D}, I)$ contains transaction identifiers $\{\, 1, 2, 5 \,\}$, so $\psi(\mathcal{D}, I) = 3$. The set of frequent itemsets with support at

[1]If the criterion is the entropy over the distribution, then the principle of maximum entropy; which states that the probability distribution which best represents the current state of knowledge is the one with largest entropy; can be used. This principle does not fit with our objectives. However, it has been proven in [Fed86] that the *maximum entropy principle* is a special case of the Minimum Description Length principle.

Itemset Database

$\mathcal{D}$	A	B	C	E	
1	1	1	1	1	$+$
2	1	1	1	1	$-$
3	1	1	0	1	$+$
4	0	1	1	1	$-$
5	1	0	1	0	$-$

Figure 4.1. Itemset Database with positive/negative classes

least 4 is $\{\emptyset, \{A\}, \{B\}, \{C\}, \{E\}, \{B, E\}\}$ (Figure 4.2).

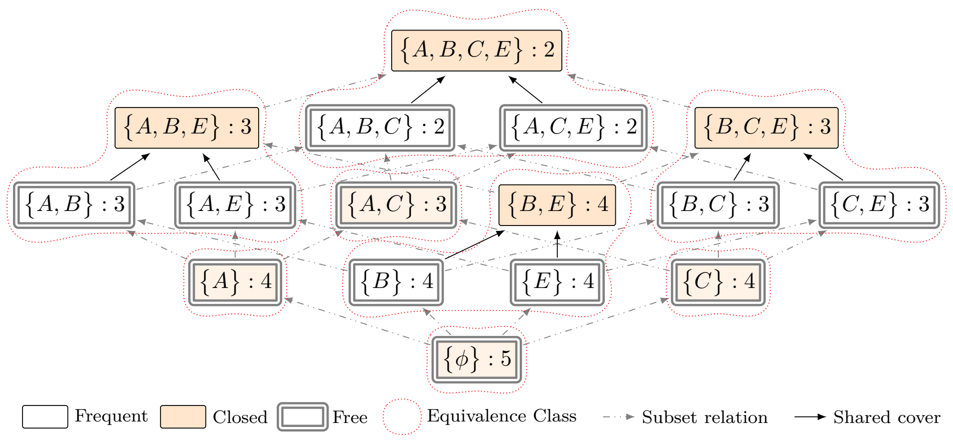


Figure 4.2. Powerset lattice of $\mathcal{D}$ with equivalence classes.

In the remainder of this chapter, for the sake of simplicity we denote $Cover_{\mathcal{D}}(I)$ as $\varphi(\mathcal{D}, I)$ and as $\varphi(I)$ when no ambiguity regarding $\mathcal{D}$ is possible. Similarly, we will use $\varphi^{+}(I)$ to denote $\varphi(\mathcal{D}^{+}, I) = Cover_{\mathcal{D}^{+}}(I)$ where $\mathcal{D}^{+} = \{(t, I_t, a_t) \in \mathcal{D} \mid a_t = +\}$ and likewise for $\varphi^{-}(I)$ with $a_t = -$.

We are interested in finding a list of rules. Each itemset in the list has a cover that is defined as follows.

Definition 4.3 (Coverage of an itemset in a sequence)**.** *Assume the sequence of itemsets $\langle I^{(1)}, \ldots, I^{(k)} \rangle$, the coverage of an itemset $I^{(j)}$ over $\mathcal{D}$ is its cover in the database of transactions not covered by the previous itemsets $I^{(1)}$, $I^{(2)}, \ldots, I^{(j-1)}$:*

$$\Phi(\mathcal{D}, \langle I^{(1)}, \ldots, I^{(k)} \rangle, j) = \varphi\Big(\mathcal{D} \setminus (\varphi(I^{(1)}) \cup \varphi(I^{(2)}) \cup \cdots \cup \varphi(I^{(j-1)})), I^{(j)}\Big) \tag{4.2}$$

with $\Phi(\mathcal{D}, \langle I^{(1)}, \ldots, I^{(k)} \rangle, 1) = \varphi(\mathcal{D}, I^{(1)})$.

Note that in a rule list $\mathcal{R}$, the last itemset is always $I^{(k)} = \emptyset$, which is the *default rule* or final *else-case*. This *empty set* inherently covers all instances not covered by any of the $k-1$ previous rules since $\varphi(\mathcal{D}, \emptyset) = \{\, (t, I_t, a_t) \in \mathcal{D} \mid \emptyset \subseteq I_t \,\} = \mathcal{D}$ for any $\mathcal{D}$.

Given a rule list $\mathcal{R} = \Big\langle (I^{(1)}, p^{(1)}), (I^{(2)}, p^{(2)}), \cdots, (I^{(k)}, p^{(k)}) \Big\rangle$ we will denote by $\Phi(\mathcal{D}, \mathcal{R}, j)$ the cover of the jth itemset in the rule list's sequence of itemsets. If no ambiguity is possible we simply write Φ_j. Similarly $\Phi_j^{+} = \Phi(\mathcal{D}^{+}, \mathcal{R}, j)$ and $\Phi_j^{-} = \Phi(\mathcal{D}^{-}, \mathcal{R}, j)$.

When creating a rule list $\mathcal{R}$ from a dataset $\mathcal{D}$ given $\mathcal{F}$, we define the probability $p^{(j)}$ of a rule $I^{(j)}$ as $p^{(j)} = P(a_t = + | (t, I_t, a_t) \in \Phi(\mathcal{D}, \mathcal{R}, j)) = \frac{|\Phi_j^{+}|}{|\Phi_j^{+}| + |\Phi_j^{-}|}$.

Example 4.3. Assume the running example database (Figure 4.1) and a rule list with corresponding sequence of itemsets $\Big\langle \{\, A, B, C \,\}, \{\, C \,\}, \emptyset \Big\rangle$. The coverage of $I^{(2)} = \{\, C \,\}$ over $\mathcal{D}$ is $\Phi_2 = \{\, 4, 5 \,\}$, instead of $\{\, 1, 2, 4, 5 \,\}$, as the transactions 1 and 2 were already covered by $I^{(1)} = \{\, A, B, C \,\}$.

Its probability is hence $p^{(2)} = \frac{|\Phi_2^+|}{|\Phi_2^+|+|\Phi_2^-|} = \frac{0}{0+2} = 0$, which indicates that no positive transaction was observed with the condition of the rule, after observing the previous rules.

At this stage, an open question is how to evaluate the quality of a probabilistic rule list $\mathcal{R}$. In this work, we propose to evaluate how well the rule list allows to *compress* the values for the class attribute observed in a training dataset. For this, we will use the Minimum Description Length (MDL) principle.

4.4.2 Minimum Description Length encoding of rule lists

The Minimum Description Length (MDL) principle [Grü07,Ris78] is a general method for *inductive inference*, based on the idea that '*the more we can compress the data, the more there are regularities in it and the more we learn from it*' [Grü07]. MDL allows making a trade-off between the complexity of rules and their ability to capture the distribution of the class attribute. To do this, we use a two-part code that minimizes the *number of bits* needed to encode the data with a model, as well as the number of bits to encode the model itself. As stated earlier, the focus in this work is on a code that favors simplicity.

Let $\mathcal{M} = M_1, M_2, \dots$ be a list of model candidates. In two-part MDL, the best model $M \in \mathcal{M}$ to capture information in a given database $\mathcal{D}$ is the one which minimizes the code length $L(M) = L_{model}(M) + L_{data}(\mathcal{D}|M)$, where $L_{model}(M)$ is the length, in bits, of the description of the model itself and $L_{data}(\mathcal{D}|M)$ the length of the data, in bits, when it is encoded with this model.

In our case, models correspond to rule lists of the form $\mathcal{R} = \Big\langle (I^{(1)}, p^{(1)}), (I^{(2)}, p^{(2)}), \cdots, (I^{(k)}, p^{(k)}) \Big\rangle$ with $I^{(j)} \in \mathcal{F}, \forall j \in 1, \dots, k-1$, $I^{(k)} = \emptyset$ and $p^{(j)} = \frac{|\Phi_j^+|}{|\Phi_j^+|+|\Phi_j^-|}$. We thus need to define an encoding with $L_{model}(\mathcal{R})$, an encoding of the rule list, and $L_{data}(\boldsymbol{\mathcal{D}}|\mathcal{R})$ such that $L_{data}(\mathcal{D}|\mathcal{R})$ can be interpreted as the *coding length* of the distribution of $+/-$'s in $\mathcal{D}$ when it is encoded with $\mathcal{R}$. The best rule list ($\mathcal{R}^*$) is then the one that minimizes the total length $L(\mathcal{R})$:

$$\mathcal{R}^* = \text{argmin}_{\mathcal{R} \in \mathcal{L}(\mathcal{F}^* \cdot \emptyset)} L_{data}(\mathcal{D}|\mathcal{R}) + L_{model}(\mathcal{R}), \tag{4.3}$$

where we identified $\mathcal{R}$ by its sequence of itemsets to ease notation; each itemset has a probability $p^{(j)}$ as defined earlier.

We now first discuss how we encode $\mathcal{R}$ when $k \leq 2$ (i.e. $\mathcal{R} = \Big\langle(\emptyset, p^{(1)}\Big\rangle$ or $\mathcal{R} = \Big\langle(I^{(1)}, p^{(1)}), (\emptyset, p^{(2)})\Big\rangle$) and then generalize to the case $k > 2$.

Case $k = 2$: To understand the computation of the coding length of $\mathcal{R}$, we first show how we can encode a target attribute if we have an itemset I and then a *default rule*. Given a rule $(I^{(1)}, p^{(1)})$, we assume that the positive and negative labels in $\varphi(\mathcal{D}, I^{(1)})$ follow a *Bernoulli distribution*, with a probability $p^{(1)}$ for the class label. The probability mass of the observed labels according to I is hence (omitting $\mathcal{D}$ from the notation):

$$P_r\Big(a_t = + \mid \varphi(I)\Big) = \prod_{(c, I_c, a_c) \in \varphi(I)} \Big(p^{(1)}\Big)^{a_c=+}\Big(1 - p^{(1)}\Big)^{a_c=-}, \tag{4.4}$$

which can be simply rewritten:

$$P_r\Big(a_t = + \mid \varphi(I)\Big) = \Big(p^{(1)}\Big)^{|\varphi^+(I)|}\Big(1 - p^{(1)}\Big)^{|\varphi^-(I)|}. \tag{4.5}$$

We estimate $p^{(1)}$ by $\frac{|\varphi^+(I)|}{|\varphi^+(I)| + |\varphi^-(I)|}$.

Definition 4.4 (Local Coding length of data)**.** *Using Shannon's Noiseless Channel Coding Theorem [BRY98, CT06, Grü07] the number of bits needed to encode the class labels of $\mathcal{D}$ using I is at least the logarithm*[2] *of the probability mass of the class labels in $\mathcal{D}$ given I:*

$$L_{local\,data}(\mathcal{D}|I) = -\log P_r(a_t = + \mid \varphi(\mathcal{D}, I)).$$

Using (4.5) *we can hence encode each label (positive/negative) at a cost of*

$$L_{local\,data}(\mathcal{D}|I) = \mathcal{Q}\Big(|\varphi^+(I)|, |\varphi^-(I)|\Big) + \mathcal{Q}\Big(|\varphi^-(I)|, |\varphi^+(I)|\Big), \tag{4.6}$$

with $\mathcal{Q}\Big(a, b\Big) = -a \log \frac{a}{a+b}$.

We will use this bound, which can be approximated closely using arithmetic coding, as the coding length for the class labels. Based on the above definition and assuming a rule list is $\mathcal{R} = \Big\langle(I^{(1)}, p^{(1)}), (\emptyset, p^{(2)})\Big\rangle$, the coding length of Φ is the sum of local data coding lengths [Grü07]

$$L_{data}(\mathcal{D}|\mathcal{R}) = L_{local\,data}(\mathcal{D}|I^{(1)}) + L_{local\,data}(\mathcal{D} \setminus \varphi(I^{(1)})|\emptyset). \tag{4.7}$$

[2] All logarithms are to base 2 and by convention, we use $0 \log 0 = 0$.

Example 4.4. Assume the rule list is $\mathcal{R} = \Big\langle (\{A, B, C\}, 0.50), (\emptyset, 0.33) \Big\rangle$ and that our database $\mathcal{D}$ (Figure 4.1) is duplicated 256 times. $L_{local\,data}(\mathcal{D}|\{A, B, C\}) = -256\log 0.5 - 256\log(1-0.5) = 512bits$ and $L_{local\,data}(\mathcal{D}\backslash\varphi(I^1)|\emptyset) = -256\log 0.33 - 512\log(1-0.33) = 705bits$; then $L_{data}(\mathcal{D}|\mathcal{R}) = 1217bits$.

When we encode the class label using this model, we do not only need to encode the data, but also the model itself.

Definition 4.5 (Length of the model)**.** *Assume a rule list* $\mathcal{R} = \Big\langle (I^{(1)}, p^{(1)}), (\emptyset, p^{(2)}) \Big\rangle$, *$m$ the number of items and n the number of transactions, we represent $(I^{(1)}, p^{(1)})$ as a string "$m_1\ I_1^{(1)}\ \ldots\ I_{m_1}^{(1)}\ n_1^+$" where, $m_1 = |I^{(1)}|$ is the number of items in $I^{(1)}$, followed by the identifiers of each item in $I^{(1)}$ and finally the number of positive labels in $\mathcal{D}$: $n_1^+ = |\varphi^+(I^{(1)})|$. The length, in bits, to encode this string is:*

$$L_{local\,model}(I^{(1)}) = \underbrace{\log m}_{|I^{(1)}|} + \underbrace{|I^{(1)}|\log m}_{I_1^{(1)}\ \ldots\ I_{|I^{(1)}|}^{(1)}} + \underbrace{\log n}_{n_1^+}, \tag{4.8}$$

where $\log m$ bits are required to represent m_1, as $m_1 \leq m = |I|$, and also $\log m$ bits for each item identifier plus $\log n$ bits to encode n_1^+. Coding n_1^- is unnecessary as it can be retrieved from the data using the itemset: $n_1^- = |\varphi(\mathcal{D}, I^(1))| - n_1^+$. From there, assuming that the itemset database $\mathcal{D}$ and the set of items $\mathcal{I}$ are known, one can easily retrieve the coverage of $I^{(1)}$ and then compute the probability $p^{(1)}$ using the number of positive labels n_1^+. The coding length of the model $\mathcal{R}$ is

$$L_{model}(\mathcal{R}) = L_{local\,model}(I^{(1)}) + L_{local\,model}(\emptyset).$$

Example 4.5. We continue on Example 4.4. To encode the model, the string "3 A B C 256" is encoded: $L_{local\,model}(\{A, B, C\}) = \log 4 + 3\log 4 + \log 1280 = 19bits$ similarly $L_{local\,model}(\emptyset) = \log 4 + 0\log 4 + \log 1280 = 13bits^a$ then $L_{model}(\mathcal{R}) = 32bits$. Together with $L_{data}(\mathcal{D}|\mathcal{R}) = 1217bits$ computed in Example 4.4, the total coding length of $\mathcal{R}$ is $L(\mathcal{R}) = 1217 + 32 = 1249 bits$.

[a]Note that by convention the size of the default rule is $m_2 = 0$.

Case $k > 2$: Assuming now a rule list $\mathcal{R} = \Big\langle (I^{(1)}, p^{(1)}), (I^{(2)}, p^{(2)}), \cdots, (I^{(k)}, p^{(k)}) \Big\rangle$ with $k > 2$. For $k > 1$ we need to modify the definition of $L_{local\,data}$ such that it does not consider parts of the data covered by a previous itemset in the sequence. Hence,

$$L_{local\,data}(\mathcal{D}|I^{(j)}) = \mathcal{Q}\Big(|\Phi_j^+|, |\Phi_j^-|\Big) + \mathcal{Q}\Big(|\Phi_j^-|, |\Phi_j^+|\Big) \tag{4.9}$$

and the total coding length is the summation of local lengths:

$$L_{data}(\mathcal{D}|\mathcal{R}) = \sum_{j=1}^{k} L_{local\,data}(\mathcal{D}|I^{(j)}); \tag{4.10}$$

the coding length of the model is:

$$L_{model}(\mathcal{R}) = \log n + \sum_{j=1}^{k-1} \Big(\log m + m_j \log m + \log n\Big) \tag{4.11}$$

To encode the size of $\mathcal{R}$ itself, we need $\log n$ bits. Because all rule list include the *default rule*, we omit these $\log m + \log n$ bits.

Example 4.6. Figure 4.3 shows example rule lists with coding lengths.

4.4.3 Coding length related to likelihood and quality of rule lists

The coding length of the class labels given a model $\mathcal{R}$ is the number of bits needed to encode the class labels with $\mathcal{R}$. As a consequence of our choice to use Shannon's theorem, this coding length corresponds to the *(-log) likelihood* of the class labels according to the model. In other words, if we would minimize the coding length of the data only, we would maximize the likelihood of the data under the model. However, as stated earlier, in this work our aim is also to find small and interpretable rule lists. We choose our code such that a relatively large weight is given to the complexity of the model.

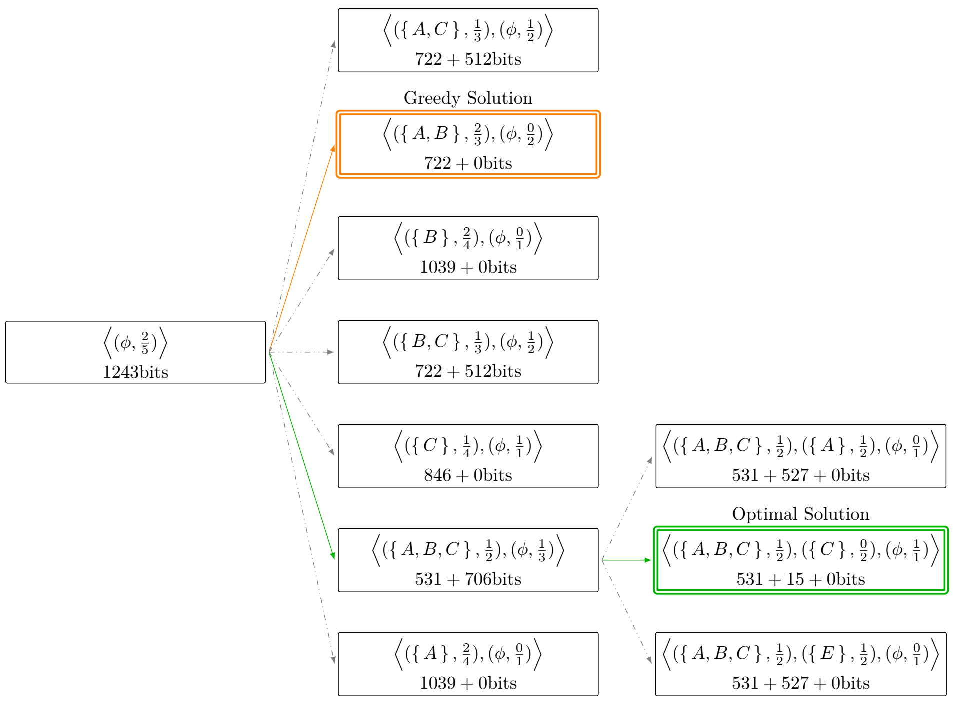

$\left\langle(\phi, \frac{2}{5})\right\rangle$
1243bits
$\left\langle(\{A, C\}, \frac{1}{3}), (\phi, \frac{1}{2})\right\rangle$
$722 + 512$bits
Greedy Solution
$\left\langle(\{A, B\}, \frac{2}{3}), (\phi, \frac{0}{2})\right\rangle$
$722 + 0$bits
$\left\langle(\{B\}, \frac{2}{4}), (\phi, \frac{0}{1})\right\rangle$
$1039 + 0$bits
$\left\langle(\{B, C\}, \frac{1}{3}), (\phi, \frac{1}{2})\right\rangle$
$722 + 512$bits
$\left\langle(\{C\}, \frac{1}{4}), (\phi, \frac{1}{1})\right\rangle$
$846 + 0$bits
$\left\langle(\{A, B, C\}, \frac{1}{2}), (\phi, \frac{1}{3})\right\rangle$
$531 + 706$bits
$\left\langle(\{A\}, \frac{2}{4}), (\phi, \frac{0}{1})\right\rangle$
$1039 + 0$bits
$\left\langle(\{A, B, C\}, \frac{1}{2}), (\{A\}, \frac{1}{2}), (\phi, \frac{0}{1})\right\rangle$
$531 + 527 + 0$bits
Optimal Solution
$\left\langle(\{A, B, C\}, \frac{1}{2}), (\{C\}, \frac{0}{2}), (\phi, \frac{1}{1})\right\rangle$
$531 + 15 + 0$bits
$\left\langle(\{A, B, C\}, \frac{1}{2}), (\{E\}, \frac{1}{2}), (\phi, \frac{0}{1})\right\rangle$
$531 + 527 + 0$bits

Figure 4.3. Finding greedy and optimal solution base on the example of Figure 4.2

Example 4.7. Assuming the database of Example 4.4, the size of the original data is $5 \times 256 = 1280$. Encoding this data with $\mathcal{R}_1 = \left\langle(\{A, B, C\}, 0.50), (\emptyset, 0.33)\right\rangle$ we obtained $L_{data}(\mathcal{D}|\mathcal{R}_1) = 1217$bits, $L_{model}(\mathcal{R}_1) = 32$bits and in total $L(\mathcal{R}_1) = 1249 bits$. Instead, when we encode this data with $\mathcal{R}_2 = \left\langle(\emptyset, 0.40)\right\rangle$ we obtain $L_{data}(\mathcal{D}|\mathcal{R}_2) = 1243$bits, $L_{model}(\mathcal{R}_2) = 6$bits and in total $L(\mathcal{R}_2) = 1249 bits$.

Looking at likelihoods only, one can see that $\mathcal{R}_1$ is a better model for representing this data, as it captures more information than $\mathcal{R}_2$. However, in total, it is not preferable over $\mathcal{R}_2$, since it is more complex to encode. The model coding length penalizes the likelihood and ensures a simple model is preferred.

For our example, the only way to improve $\mathcal{R}_1$ is to add (if possible) a new rule that reduces the error made by $\mathcal{R}_1$ by assuming that the part not covered by $\{A, B, C\}$ is for the *default rule*. Thus, by adding the itemset $\{C\}$ to $\mathcal{R}_1$, which covers all 0s still present, we obtain the best model $\mathcal{R} = \left\langle (\{A, B, C\}, \frac{1}{2}), (\{C\}, \frac{0}{2}), (\phi, \frac{1}{1}) \right\rangle$ with $L(\mathcal{R}) = 546 bits$ since the *default rule* now covers only remaining 1s.

4.4.4 A Greedy algorithm

The probabilistic rule list that minimizes the MDL score (4.3) can be constructed greedily, extending the list by one rule at each step. Greedy algorithms are known to be efficient and approximate optimal solutions well in other rule learning tasks.

Algorithm 4.1 shows a greedy algorithm that starts with the empty rule list $\mathcal{R}$, and then iteratively finds within a given set of patterns the rule that minimises the coding length. The local best rule is obtained by considering at each iteration the sub-problem of finding the optimal rule list with $k \leq 2$ on the remaining data. This corresponds to finding the itemset $I^{(1)}$ such that the *coding length* is smallest (Line 3). Once the local best rule is selected the rule list is updated in Line 6 and in Line 7; its coverage is removed from $\mathcal{D}$. The process is then run again until $\mathcal{D}$ is empty or the *default rule* is selected.

Algorithm 4.1: *Greedy*$(\mathcal{F}, \mathcal{D})$

```
1 $\mathcal{R} \leftarrow \langle\rangle$
2 do
3     $I^* \leftarrow \text{argmin}_{I \in \mathcal{F}^*} L\left(\left\langle (I, p^{(1)}), (\emptyset, p^{(2)}) \right\rangle\right)$
4     if $L\left(\left\langle (I, p^{(1)}), (\emptyset, p^{(2)}) \right\rangle\right) \geq L\left(\left\langle (\emptyset, p^{(1)}) \right\rangle\right)$ then
5         $I^* \leftarrow \emptyset$
6     $\mathcal{R} \leftarrow \mathcal{R} \cup (I^*, p^{(1)})$        ▷ Add this rule to the rule list
7     $\mathcal{D} \leftarrow \mathcal{D} \setminus \varphi(I^*)$
8 while $I^* \neq \emptyset$
9 return $\mathcal{R}$
```

Example 4.8. Assuming our running example, at the first iteration of the greedy algorithm, the minimum code-length $L(\langle \{A, B\}, \emptyset \rangle) = 722 bits$ and then it is the greedy solution (See Figure 4.3).

The greedy algorithm may be sub-optimal. For instance it fails to discover the $L(\langle \{ A, B, C \}, \{ C \}, \emptyset \rangle) = 546 bits$ on our example.

4.4.5 Branch-and-Bound algorithm

For finding solutions that are better than the greedy solution, we propose a best-first branch-and-bound algorithm that can prune away candidates based on a lower-bound on the MDL value. Each node in the search tree is a partial rule list, consisting of a sequence of rules *without* the *default rule*. The children of each node correspond to appending one additional rule from $\mathcal{F}$ to the partial rule list.

Algorithm 4.2 shows the pseudo-code of this branch-and-bound expansion search. For clarity, we omit the probabilities in the rule list representation. The algorithm receives as input a list of rule candidates $\mathcal{F}$ and database $\mathcal{D}$. A priority queue is used to store the set of rule lists not yet expanded, ordered by the code-length obtained when extending the partial rule with the *default rule* (best-first strategy). The initial best rule is the *default rule* (Line 2) and the empty rule list is added as initial search node. As long as the queue is not empty, the priority queue is dequeued and the returned partial rule list is expanded (Line 6). Each new partial rule list is evaluated as if it was completed with the *default rule* ($\emptyset$) and checked whether it is better than the current best rule list (Lines 7,8).

Before adding the new partial rule list to the queue, a lower-bound on the code length is computed, that is, an optimistic estimate of the code length achievable (see next section). Only if the lower-bound is better than the current best value is the rule list added to the queue (Lines 9,10). If not, this part of the search tree is effectively pruned.

Lower-bound on a partial rule list A good lower-bound is difficult to compute since there is an exponential number of rules that can be added to the list. Because the rule list itself is already evaluated in the algorithm, we are seeking a lower-bound on any expansion of the rule list. The coding length is determined by $L(\mathcal{R}) = L_{model}(\mathcal{R}) + L_{data}(\mathcal{D}|\mathcal{R})$ according to (4.10) and (4.11).

The most optimistic expansion is hence achieved with the smallest possible expansion of the rule list yielding the most significant reduction of the coding length for the data. In the best case, this is a rule of length one ($|I^{(j+1)}| = 1$) that perfectly separates the positives from the negatives. In this case, the

Algorithm 4.2: *Branch-and-bound* $(\mathcal{F}, \mathcal{D})$

```
1  $PQ$ : PriorityQueue ▷ Partial rule lists ordered by code-length when adding default
     rule
2  $best\mathcal{R} \leftarrow \langle\emptyset\rangle,\ best \leftarrow L(best\mathcal{R})$
3  $PQ$.enqueue-with-priority$\Big(\langle\rangle, L\big(\langle\emptyset\rangle\big)\Big)$
4  while $\mathcal{R} \leftarrow PQ.dequeue()$ do
5      for each $I \in \mathcal{F} \setminus \mathcal{R}$ do
6          $\mathcal{R}' \leftarrow \langle\mathcal{R}, I\rangle$
7          if $L\big(\langle\mathcal{R}', \emptyset\rangle\big) < best$ then
8              $best\mathcal{R} = \langle\mathcal{R}', \emptyset\rangle,\ best \leftarrow L(best\mathcal{R})$
9          if lower-bound$(\mathcal{R}') < best$ then
10             $PQ$.enqueue-with-priority$\Big(\mathcal{R}', L\big(\langle\mathcal{R}', \emptyset\rangle\big)\Big)$
11 return $best\mathcal{R}$
```

additional code length of the rule list corresponds to a rule of length one:

$$L_{local\,model}(I^{j+1}) = \log m + 1 \log m + \log n$$

and the addition to the code length of the data is:

$$L_{local\,data}(\mathcal{D}|I^{j+1}) = \mathcal{Q}\Big(|\Phi^+_{j+1}|, 0\Big) + \mathcal{Q}\Big(0, |\Phi^-_{j+1}|\Big) = 0$$

with the data coding length of the *default rule* also being 0.

While such a rule expansion may not exist, the resulting value is a valid lower-bound on the code length achievable by any expansion of the partial rule list. This is because any expansion has to be greater than or equal in size to 1, and any expansion will achieve at best a data compression of 0.

Implementation details *Choice of $\mathcal{F}$.* The complexity of Algorithm 4.2 is $\mathcal{O}(|\mathcal{F}|^d)$ where d is the depth in the best-first search tree. The efficiency of the algorithm strongly depends on $|\mathcal{F}|$ since in the worst case the number of nodes is in $\mathcal{O}(|\mathcal{F}|^{|\mathcal{F}|})$.

To control the size of $\mathcal{F}$ one can consider all frequent itemsets with a given minimum frequency threshold. Because we are interested in a small coding length, we propose to further restrict the set of patterns to the set of *frequent free itemsets* [SNK07]. Known also as generators, a free itemset is the smallest itemset (in size) that does not contain a subset with the same cover: if I is free, $\nexists J \subset I$ s.t. $\varphi(I) = \varphi(J)$. There may be multiple free itemsets with the same cover and for our purposes just a single one of them is sufficient. In Figure 4.2, all the itemsets in a double bordered rectangle are free.

Set representation as bitvectors. Each candidate itemset in $\mathcal{F}$ is represented by the tuple (set of items, set of covered transactions). Operations on sets such as union, intersection and count being at the core of our implementation, they must be implemented very effectively. For this, we represent each set by bitvectors, and all the cover computation are bitwise operations on bitvectors. A rule list is represented by an array of *itemset indices* into $\mathcal{F}$. From the index, one can identify the itemset and its coverage. During the search process at each iteration, a new itemset I is added to the partial rule list (Line 6 of Algorithm 4.2). This operation involves updating the cover of the rule list computed using (4.2) which depends on all the transactions already covered. To do this effectively, we keep the transactions already covered in a single bitvector $T^{(j)}_{covered} = \varphi(I^{(1)}) \cup \varphi(I^{(2)}) \cup \cdots \cup \varphi(I^{(j)})$. The coverage after the addition of a new itemset $I^{(j+1)}$ is then

$$\Phi(\mathcal{D}, \mathcal{R} \cup I^{(j+1)}, j+1) = \neg T^{(j)}_{covered} \cap \varphi(I^{(j+1)}). \tag{4.12}$$

4.5 Implementation and Practical User Guide

The implementation of *PRL* is in Scala and is available online[3] in free access.

For developers, the installation procedure is described in the Install file in the code directory and merely consists of "importing the project" into your favourite IDE.

Users can directly download the *jar-file* which is available on our website [4]. For example, given the train- and the test- sets (from mushroom dataset), we can find the best rule-list, using the branch-and-bound algorithm and limiting the size of each rule to 10, by running the following command.

```
java -jar prl.jar F train.txt -p test.txt -i 10
```

and here is the result:

[3]https://projetsJOHN@bitbucket.org/projetsJOHN/mdlrulesets

[4]https://sites.uclouvain.be/cp4dm/prl/

```
>mushroom.1.train.txt      7310    112     3796    1462    10
112     1145    1947
Best Rule List (k=4) =
<
({gill_spacing_c, stalk_surface_above_ring_s, odor_n},2058/2126,
488bits),
({gill_spacing_c, stalk_shape_t},0/1806, 47bits),
({stalk_color_below_ring_w, gill_size_b},1593/1593, 47bits),
({gill_spacing_c, ring_number_o},0/1509, 47bits),
({}, 145/276, 276bits)
> : score=905bits (629.0)
Time(s) : 715.290714503
```

4.6 Experiments

Our objective being to find a small-and-good rule list, we evaluate our approach from three perspectives: (i) the quality of obtained solutions: how expressive and concise are the rule lists; what is the log-likelihood of the data given the lists; (ii) the accuracy and sensitivity of our method under various parameters, evaluated using the area under ROC curves (AUC), (iii) the predictive power of our method, using AUC as well.

Note that we add a comparison with other classification methods to properly position our work; our aim is not to build a classification model that is more accurate on commonly used datasets.

Datasets. We use nine annotated datasets publicly available from the *CP4IM*[5] and UCI[6] repositories. We also used the *door* dataset as described in the introduction (Example 4.1). Furthermore, we used the *Gallup* dataset [ERP18], from a project with the same name on migratory intentions. This data set is not publicly available, but can be purchased. Our objective here is to understand the migratory intentions between two countries by considering the socio-parameters of education, health, security and age. All these datasets have been preprocessed and their characteristics are given in Table 4.2.

Algorithms. We compare with popular tree-based classification methods such as Random Forests (*RF*) and decision trees (*CART*) from the *scikit-learn library*, as well as the rule-learning methods *JRIP* (Weka version of

[5]https://dtai.cs.kuleuven.be/CP4IM/datasets/

[6]http://archive.ics.uci.edu/ml/datasets.html

Table 4.2. Benchmark features

name	anneal	car	australian-cr.	heart-cl.	krvskp	mushroom
$\|\mathcal{D}\|$	812	1728	653	296	3196	8124
$\|\mathcal{I}\|$	89	21	124	95	73	112
$\frac{\|\mathcal{D}^+\|}{\|\mathcal{D}\|}$	0.77	0.7	0.55	0.54	0.52	0.52

name	primary-tu.	dermatology	gallup	door	soybean
$\|\mathcal{D}\|$	336	366	15734	3216	630
$\|\mathcal{I}\|$	31	133	41	11	50
$\frac{\|\mathcal{D}^+\|}{\|\mathcal{D}\|}$	0.24	0.2	0.19	0.16	0.15

RIPPER) and *SBRL* [YRS17] available in R CRAN (see Sect. 2). We run *SBRL* with the default setting (number of iterations set to 30.000, number of chains 10 and a lambda parameter of 10).

Protocols. All experiments were run in the JVM with maximum memory set to 8GB on PCs with Intel Core i5 64bits processor (2.7GHz) and 16GB of RAM running MAC OS 10.13.3. Our approach is called *PRL* (for probabilistic rule lists) and is implemented in Scala. The candidate itemsets $\mathcal{F}$ are the frequent free itemsets. *PRL* name can be followed by g for greedy or c for complete branch-and-bound. Evaluation of AUC is done using *stratified 10-fold cross-validation*. For the reproducibility of results, all our implementations are open source and available online[7].

4.6.1 Compression power of PRL

Table 4.3 gives the total code length obtained for the greedy *PRLg* and the complete branch-and-bound *PRLc* approaches. As can be observed, the *compression ratio* (total code length/size of the datasets) is substantial. For instance, it is 10% for the dermatology dataset. For 8/11 instances *PRLc* discovers a probabilistic rule list compressing better than the one obtained with *PRLg*. The gain obtained with *PRLc* is sometimes substantial, for instance on the `krvskp` and `mushroom` data sets.

[7]https://projetsJOHN@bitbucket.org/projetsJOHN/mdlrulesets

Table 4.3. Total code lengths for several datasets (θ is the minimum support for $\mathcal{F}$)

name	anneal	car	australian-cr.	heart-cl.	krvskp	mushroom
θ	20	5	20	20	5	20
$\lvert\mathcal{F}\rvert$	1361	22	2495	2024	65	1145
PRLg	587	710	386	262	2594	1978
PRLc	**532**	**628**	**380**	**249**	**845**	**967**

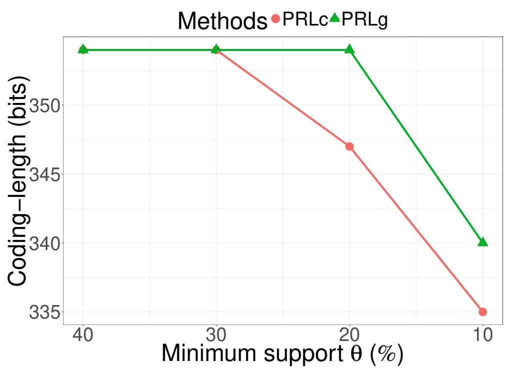
name	primary-tu.	dermatology	gallup	door	soybean
θ	20	20	10	1	1
$\lvert\mathcal{F}\rvert$	214	763	15	35	49
PRLg	249	39	10327	1876	356
PRLc	249	39	**10163**	1876	**314**

4.6.2 Impact of the parameters

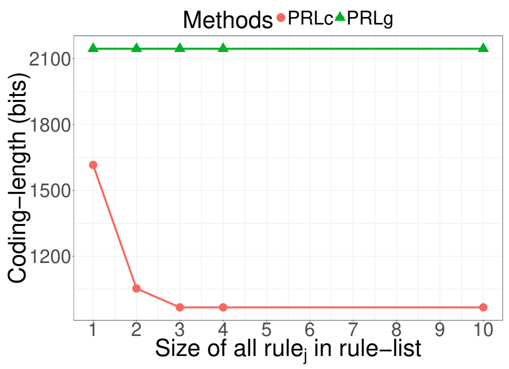
The set of possible itemsets $\mathcal{F}$ to create the rule list is composed of the frequent free itemsets generated with a minimum support threshold θ. Figure 4.4a reports the compression ratio for decreasing values of θ. As expected the compression ratio becomes smaller whenever θ decreases. The reason is that the set $\mathcal{F}$ is growing monotonically, allowing more flexibility to discover a probabilistic rule list that compresses well.

Both the greedy and the complete branch-and-bound algorithms can easily limit the size of the probabilistic rule list they produce. This is done by stopping the expansion of the list beyond a given size limit k. Figure 4.4b reports the compression ratio for increasing values of k. As expected the compression ratio becomes smaller whenever k increases for PRLc and stabilises at some point when the limit k becomes larger than the length of the optimal rule list. Surprisingly this is not necessarily the case for the greedy approach that is not able to take advantage of longer rule lists on this benchmark.

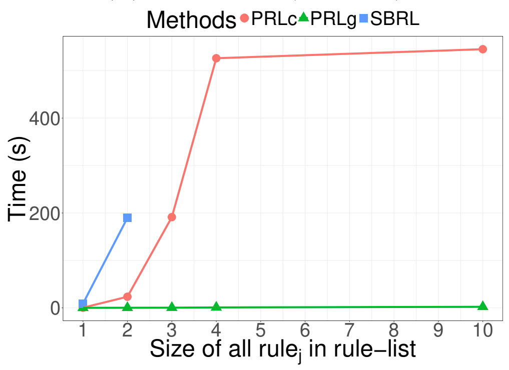
Regarding the execution time according to the size of the rules, as shown in Figure 4.4c, with a time limit of 10 minutes, we can see that the greedy approach is more scalable. PRLc and SBRL execution time evolve exponentially, PRLc being faster than SBRL though. Note that as soon as the optimal solution is found, in the case of PRLc, the execution time does not increase so much anymore. The reason is that most of the branches are cut-off by the branch-and-bound tree exploration beyond that depth limit.



(a) Soybean



(b) Mushroom ($\theta = 20\%$)



(c) Mushroom ($\theta = 20\%$)

Figure 4.4. Sensitivity of PRL for several settings using mushroom and soybean datasets

4.6.3 Comparison of *PRL* with existing rule learning algorithms

We compare the rule list produced by our approaches (PRLg and PRLc) and by SBRL [YRS17]. Figure 4.6a gives the code length for the model and for the data (class labels) for various datasets for the different approaches. Note that the code length for the data corresponds to the *log-likelihood* of the class labels under the rule list. From the rule lists obtained using the training set, the probability (to be positive) of each transaction in the test set is predicted and the coding lengths are computed using the (4.10) and (4.11). The reported values are averaged over 10 folds. The model coding length represents the size of the encoding of the initial rule list.

One can see that the PRL approaches are competitive with SBRL. On Figure 4.6a, it often obtains the smallest data coding length except for the *mushroom* dataset. The reason is that the test set of *mushroom* is classified perfectly by SBRL. The rule lists produced are arguably shorter with PRLg and PRLc than with SBRL.

The *mushroom* dataset is investigated further in Figures 4.6b and 4.6c. The data coding length and the area under the ROC curve are computed for increasing prefixes of the lists. As we can see, at equal prefix size ($k < 5$) our approach obtains *better likelihood* and is *more accurate* than SBRL. Then beyond $k \geq 5$ SBRL continues to improve on accuracy while PRLg and PRLc stagnate. The lists indeed have reached their optimal length at $k = 5$. This evolution is a clear illustration of the difference between the type of rule lists produced by SBRL and our approach. While SBRL lists are more focused on classification, MDL-based lists are a trade-off between the data-coding length (classification) and the complexity of lists (model code length).

4.6.4 Prediction power of PRL and other supervised learning approaches

Although our approach is not designed to generate the best rule list for classification, we evaluate its prediction power in the light of well-known classification methods: *CART*, *RF*, *SBRL* and *JRIP* using 10-fold cross-validation and default settings. For *PRL* the classification is done by associating with each transaction the probability that its label is positive. This probability is that of the first rule of the rule list (obtaining from the training set) that matches with this transaction. The results are shown in Figure 4.5.

In general, the AUC of our methods are greater than 0.6 and the best

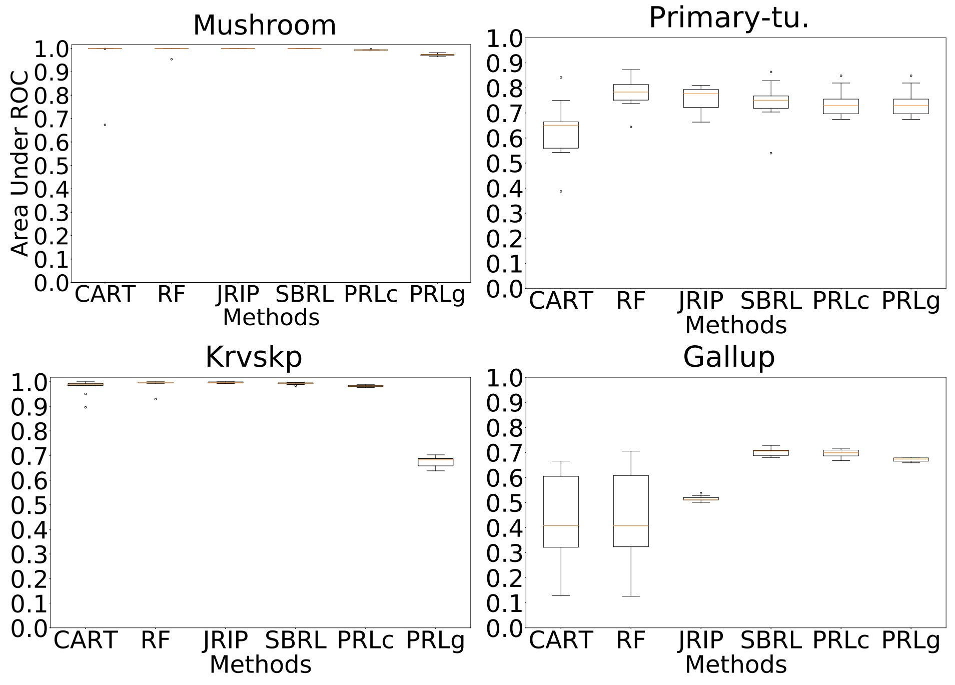


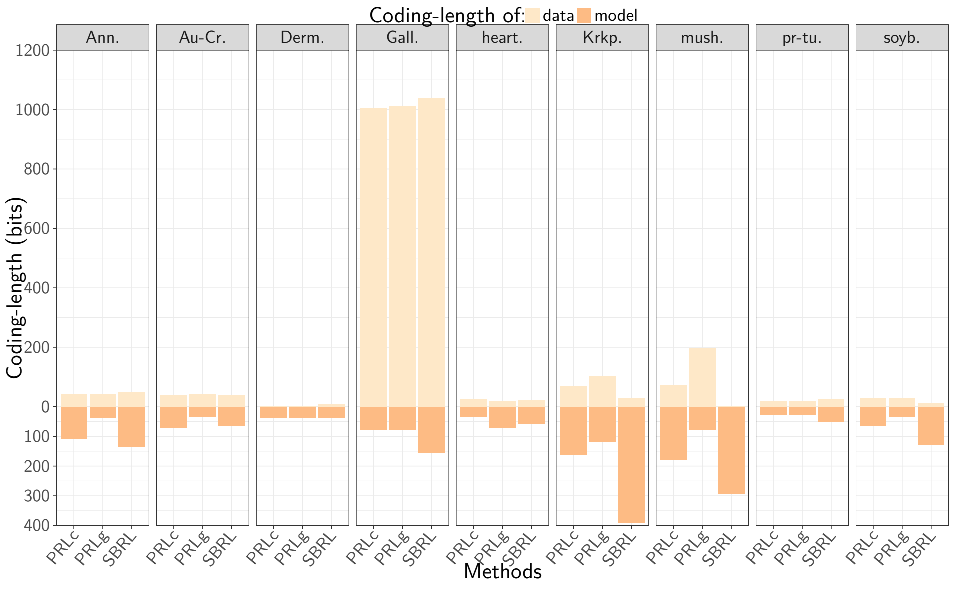
Figure 4.5. Comparison of Area under ROC among different methods and four datasets, for all 10-folds ($\theta = 10\%$, $|I| = 1$).

solution always has a greater or equal accuracy compared to the greedy approach. The difference becomes significant on databases like *Krvskp* where the difference in compression ratio is also high (Figure 4.3).

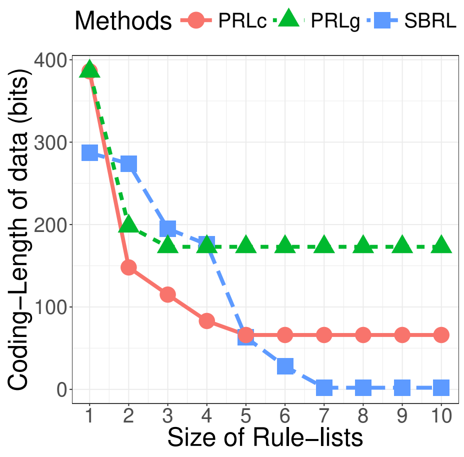
State-of-the-art methods are often more accurate, except in unbalanced datasets (Gallup, primary-tu.) where our approaches are very competitive. One can see that rule-based methods do better on very unbalanced databases like Gallup.

4.7 Summary, Outlooks, Further readings

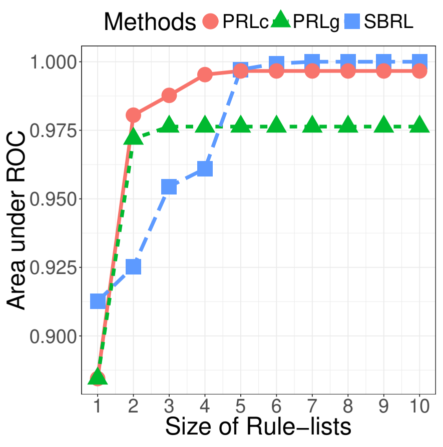
This work proposed a supervised rule discovery task focused at finding probabilistic rule lists that can concisely summarize a boolean target attribute, rather than accurately classify it. Our method is in particular applicable when the target attribute corresponds to rare events. Our approach is based on two ingredients, namely, the Minimum Description Length (MDL) principle,



(a)



(b)



(c)

Figure 4.6. **a)** Comparison of coding length in average among *PRL* (g,c) and *SBRL* for different test datasets and **b and c)** evolution of the coding length of *data only* (top) and the AUC (bottom) for several rule lists size, for *mushroom* dataset, for all 10-folds ($\theta = 10\%$, $|I| = 2$).

and a branch-and-bound search strategy. We have experimentally shown that obtained rule lists are compact and expressive. Future work will investigate the support of multivariate target attributes (> 2 classes) and new types of patterns, such as sequences.

Part III

Sequential Pattern Mining

Chapter 5

Sequential Pattern Mining using Constraint Programming

"The greatest challenge to any thinker is stating the problem in a way that will allow a solution."

–Bertrand Russell

Overview *The main advantage of Constraint Programming (CP) approaches for pattern mining is their modularity, which includes the ability to add new constraints (regular expressions, length restrictions, etc). However, there is always a trade-off between flexibility and efficiency. In this chapter, our objective is to build global constraints for the sequential pattern mining (SPM) problem by maximizing both efficiency and flexibility. We will investigate SPM problem both on sequence database with or without time consideration.*

Contribution *The main contribution of this work is to show how by combining SPM and CP techniques, one can build flexible and more efficient approaches than existing both CP-based and specialized approaches. This was made possible by, first, computing efficiently the projected database using pre-computing the positions at which a symbol can become unsupported by a sequence, thereby avoiding to scan the full sequence each time; and second by taking inspiration from the trailing used in*

CP solvers to devise a backtracking-aware data structure that allows fast incremental storing and restoring of the projected database.

Main source *This chapter is mainly based on our paper [AGS16].*

5.1 Context and Motivation

Sequence mining is a widely studied problem concerned with discovering subsequences in a dataset of given sequences, where each (sub) sequence is an ordered list of symbols. It has applications ranging from web usage mining, text mining, biological sequence analysis and human mobility mining [ME10]. We focus on the problem of finding patterns in sequences of individual symbols, which is the most commonly used setting in those applications.

In recent years, constraint programming (CP) has been proposed as a general framework for pattern mining [GNDR11, CJSS12, NG15, KLL$^+$16]. The main benefit of CP-based approaches over dedicated algorithms is that it is *modular*. In a CP framework, a problem is expressed as a set of constraints that the solutions must satisfy. Each such constraint can be seen as a *module*, and can range from being as simple as ensuring that a subsequence does not contain a certain symbol at a certain position, up to computing the frequency of a pattern in a database. This modularity allows for flexibility, in that certain constraints such as symbol restrictions, length, regular expressions etc can easily be added and removed to existing problems. Another advantage is that improving the efficiency of one constraint will improve the efficiency of all problems involving this constraint.

However, this increased flexibility can come at a cost. Negrevergne et al. [NG15] have shown that a fine-grained modular approach to sequence mining can support any type of constraints, including gap and span constraints and any quality function beyond frequency, but that this is not competitive with state-of-the-art specialized methods. On the other hand, they showed that by using a global constraint (a module) that computes the pseudo-projection of the sequences in the database similar to PrefixSpan [PHMA$^+$01], this overhead can be reduced. Kemmar et al. [KLL$^+$16, KLL$^+$15] propose to use a single global constraint for pseudo-projection as well as frequency counting over all sequences. This approach is much more efficient than the one of [NG15] that uses many reified constraints. These CP-based methods obtain reasonable performance, especially for mining under regular expressions. While they improve scalability compared to each other, they are not on par with some of the best specialized systems such as Zaki's cSpade [Zak00].

In this work, we show for the first time that a generic CP system with a custom global constraint can outperform existing specialised systems including Zaki's.

Our global constraint improves on earlier global constraints for sequence mining by combining ideas from both pattern mining and constraint programming as follows:

- first, we improve the efficiency of computing the projected database and the projected frequency using last-position lists, similar to the LAPIN algorithm [YK05] but within a PrefixSpan approach.
- Second, we take into account not just the efficiency of computing the projected database, but also that of storing and restoring it during depth-first search. For this we use the *trailing* mechanism from CP solvers to avoid unnecessary copying of the pseudo-projection data structure. Such an approach is in fact applicable to any depth-first algorithm in pattern mining and beyond.

By combining the right ingredients from both research communities in a novel way, we end up with an elegant algorithm for the projected frequency computation. When added as a module to a generic CP solver, the resulting system improves both on previous CP-based sequence miners as well as state-of-the-art specialized systems. Furthermore, we show that by improving this one module, these improvements directly translate to other problems using this module, such as regular-expression based sequence mining.

5.2 Related works

We review specialized methods as well as CP-based approaches. A more thorough review of algorithmic developments is given in [ME10].

Specialized methods. Introduced by Srikant and Agrawal [AS95], GSP was the first approach to extract sequential patterns from a sequential database. Many works have improved on this apriori-based method, typically employing depth-first search. A seminal work is that of PrefixSpan [PHMA$^+$01].

A prefix in this context is a sequential pattern that can only be extended by appending symbols to it. Given a prefix, one can compute the *projected database* of all suffixes of the sequences that have the prefix as a subsequence. This projected database can then be used to compute the frequency of the prefix and of all its 1-extensions (projected frequency). A main innovation in PrefixSpan is the use of a *pseudo-projected* database: instead of copying the entire (projected) database, one only has to maintain pointers to the position in each sequence where the prefix matched.

Alternative methods such as SPADE [Zak00] and SPAM [AFGY02] use a vertical representation of the database, having for each symbol a list of sequence identifiers and positions at which that symbol appears.

Yang et al. [YWK07] have shown that algorithms with either data representation can be improved by precomputing the last position of each symbol in a sequence. This can avoid scanning the projected database, as often the reason for scanning is to know whether a symbol still appears in the projected sequence.

The standard sequence mining settings have been extended in a number of directions, including user-defined constraints on length or on the gap or span of a sequence such as in the cSPADE algorithm [Zak00], closed patterns [YHA03] and algorithms that can handle regular expression constraints on the patterns such as SMA [TBG08]. These constraints are typically hard-coded in the algorithms.

CP-based approaches for SPM. CP-based approaches for sequence mining are gaining interest in the CP community. Early work has focused on fixed-length sequences with wildcards [CJSS12]. More generally, [NG15] proposed two approaches: a full decomposition of the problem in terms of constraints and an approach using a global constraint to construct the pseudo-projected database similar to PrefixSpan. It uses one such constraint for each sequence. Kemmar et al [KLL$^+$15] propose to gather all these constraints into a unique global constraint to reduce the overhead of the multiple constraints. They further showed how the constraint could be modified to take a maximal gap constraint into account [KLL$^+$16].

The reader who is not familiar with frequent sequence mining problems and Constraint Programming is referred to Chapter 2, Sections 2.1.2 and 2.2. The notions of *prefix-projected database* in SPM and the mechanism of storing and restoring states through "Reversibles" will be particularly useful.

5.3 Global constraints for projected frequency

We first introduce the basic CP model of frequent sequence mining introduced in [NG15] and extended in [KLL+15]. Then, we present how we improve the computation of the pseudo-projection, followed by the projected frequency counting and pruning.

5.3.1 Existing methods [NG15, KLL+15]

A constraint model consists of variables, domains and constraints. The CP model will be such that a single solution corresponds to a frequent sequence, meaning that all sequences can be extracted by enumerating all solutions.

Notation: We use the `dom` and `size` functions to designate respectively the domain of a decision variable in CP and the size of an object. We also use, the $array[i]$ to denote the element at the index i in this $array$.

Variables and domains. Let $\mathcal{I} = \{1, \dots, m\}$ be a set of items identifiers and L be an upper bound on the pattern length, e.g. the length of the longest sequence in the sequence database. The variables used to represent the unknown pattern P is modelled as an array of L integer variables $P = [P_1, P_2, \dots, P_L]$. Each variable has an initial domain $dom(P_i) = \{0\} \cup \mathcal{I}$, corresponding to all possible symbol identifiers and augmented with an additional identifier 0. The symbol with identifier 0 represents ϵ, the empty symbol. It will be used to denote the end of the sequence in P, using a trailing suffix of such 0's.

SPM constraints. Let us now introduce SPM constraints over variables P.

Definition 5.1. *A CP model over P represents the frequent sequence mining problem with threshold θ, iff the following three conditions are satisfied by every valid assignment to P:*

1. $P_1 \neq 0$
2. $\forall i \in \{\, 2, \cdots, L-1 \,\} : P_i = 0 \Rightarrow P_{i+1} = 0$
3. $\begin{cases} \text{size}\Big(\{(sid, s) \in SDB \mid \langle P_1 \ldots P_j \rangle \preceq s\}\Big) \geq \theta \\ s.t.\ j = \max(\{i \in \{\, 1, \cdots, L \,\} \,|P_i \neq 0\}) \end{cases}$

The first requirement states that the sequence may not start with the empty symbol, e.g. no empty sequence. The second requirement enforces that the pattern is in a canonical form such that after the empty symbol, all other symbols are the empty symbol too. Hence, a sequence of length $l < L$ is represented by l non-zero symbols, followed by $L - l$ zero symbols. The last requirement states that the frequency of the non-zero part of the pattern must be above the threshold θ.

Prefix projection global constraint

Initial work [NG15] proposed to decompose these three conditions into separate constraints, including a dedicated global constraint for the inclusion relation $\langle P_1 \ldots P_j \rangle \preceq s$ for each sequence separately. It used the pseudo-projection technique of PrefixSpan for this, with the projected frequency enforced on each symbol in separate constraints.

Kemmar et al. [KLL+15] extended this idea by encapsulating the filtering of all three conditions into one single (global) constraint called `PrefixProjection`. It also uses the pseudo-projection idea of PrefixSpan, but over the entire database. The propagation algorithm for this constraint, as executed when the next unassigned variable P_i is assigned during search, is given in Algorithm 5.1.

An initial assumption is that the database SDB does not contain any infrequent symbols, which is a simple preprocessing step. The code is divided in three parts: (i) if P_i is assigned to 0 the remaining P_k with $k > i$ is assigned to 0; else (ii) from the second position onwards (remember that the first position can take any symbol and be guaranteed to be frequent as every symbol is known to be frequent), the projected database and the projected frequency of each symbol is computed; and (iii) all symbols that have a projected frequency below the threshold are removed from the domain of the subsequent pattern variables.

Algorithm 5.1: PrefixProjection(SDB,P,i,θ)

```
▷ Input: SDB is a sequence database;
         $P$ is the list of decision variables;
         $i$ is the position in $P$ representing the current bound variable;
         $\theta$ is the support threshold.
▷ Pre: variables $\langle P_1, \ldots, P_i \rangle_{i \in [1..l]}$ are bound;
         $P_i$ is the new assigned variable since previous call.

if $P_i = 0$ then
    foreach $j \in \{ i+1, \cdots, L \}$ do $P_j.assign(0)$
else if $i \geq 2$ then
    projFreqs ← ProjectAndGetFreqs($SDB, P_i, \theta$)
    foreach $j \in \{ i+1, \cdots, L \}$ do
        foreach $a \in D(P_j)$ do
            if $a \neq 0 \wedge projFreqs[a] < \theta$ then $P_j.removeValue(a)$
```

The algorithm for computing the (pseudo) projected database and the projected frequencies of the symbols is given in Algorithm 5.2. It operates as follows with a the new symbol appended to the prefix of assigned variables since previous call. The first loop at Line 8 attempts to discover for each sequence s in the projected database if it can be a sub-sequence of the extended prefix. If yes, this sequence is added to the next projected database at Line 14. The second loop at Line 17 computes the frequency of each symbol occurring in the projected database but counting it at most once per sequence.

5.3.2 Improving propagation

Although being the state-of-art approach for solving SPM with CP, the filtering algorithm of Kemmar et al [KLL$^+$16] presents room for improvement. We identify four weaknesses and propose solutions to them.

Weakness 1 (Pruning the search tree). Databases with long sequences will have a large upper-bound L. For such databases, removing infrequent symbols from all remaining pattern variables P in the loop defined at Line 12 of Algorithm 5.1 can take time. This is not only the case for doing the action, but also for restoring the domains on backtracking. On the other hand, only the next pattern variable P_{i+1} will be considered during search, and in most cases a pattern will never actually be of length L, so all subsequent domain

Algorithm 5.2: ProjectAndGetFreqs(SDB, a, θ)

```
▷ Input: SDB is a sequence database;
         $a$ is the current projected symbol;
         $\theta$ is the support threshold.
▷ Output: $freq$ list of projected frequencies
▷ Internal State: $startv$, $esize$, $embs$, $\phi$, $\varphi$, $\psi$, $\gamma_s(P_{:\psi})$.

$PSDB_i \leftarrow \emptyset$
foreach (sid,start) $\in PSDB_{i-1}$ do
    $s \leftarrow$SDB[sid]    $pos \leftarrow$ start
    while $pos <$ size($s$) $\wedge$ $a \neq s[pos]$ do
    $pos = pos + 1$

    if $pos <$ size($s$) then
        $PSDB_i = PSDB_i \cup \{(sid, pos)\}$
projFreqs[$b$]$\leftarrow 0$ $\forall b \in \{1, \cdots, m\}$
if size($PSDB_i$) $\geq \theta$ then
    foreach (sid,start) $\in PSDB_i$ do
        $s \leftarrow$ SDB[sid]    existsSymbol[b] = false $\forall b \in \{1, \cdots, m\}$
        foreach $i \in \{start, \cdots, $size$(s)\}$ do
            if $\neg$ existsSymbol[s[i]] then
                projFreqs[s[i]] $\leftarrow$ projFreqs[s[i]]+1
                existsSymbol[s[i]] $\leftarrow$ true
return projFreqs
```

changes are unnecessary. This weakness is a peculiarity of using a fixed-length array P to represent a variable-length sequence. Mining algorithms typically have a variable length representation of the pattern, and hence only look one position ahead. In our propagator we only remove values from the domain of P_{i+1}.

Weakness 2 (Computing frequencies). When computing the projected frequencies of the symbols, one has to scan each sequence from its current pseudo-projection pointer *start* till the end of the sequence. This can be time-consuming in case of many repetitions of only a few symbols for example. Thanks to the *lastPosList* defined next, it is possible to visit only the last position of each symbol occurring after *start*. This idea was first introduced in [YWK07] and exploited in the LAPIN family of algorithms.

Definition 5.2. ***(Last position list).*** *For a current sequence s,*

sid	sequence	*lastPosList*	*lastPosMap*
sid_1	$\langle ABCBC \rangle$	[(C,5),(B,4),(A,1)]	{A→1, B→4, C→5,D→0}
sid_2	$\langle BABC \rangle$	[(C,4),(B,3),(A,2)]	{A→2, B→3, C→4,D→0}
sid_3	$\langle AB \rangle$	[(B,2),(A,1)]	{A→1, B→2, C→0,D→0}
sid_4	$\langle BCD \rangle$	[(D,3),(C,2),(B,1)]	{A→0, B→1, C→2,D→3}

Table 5.1. A sequence database SDB_1 and list of last positions.
1) SDB, 2) lastPosList, 3) lastPosMap

lastPosList is a sequence of pairs $(symbol, pos)$ *giving for each symbol that occurs in* s *its last position:* $pos = max\{p \leq \text{size}(s) \mid s[p] = symbol\}$*. The sequence is of length* m*, the number of distinct symbols in* s*. This sequence is decreasing according to positions:* $lastPosList[i].pos > lastPosList[i+1].pos$ $\forall i \in \{1, \cdots, m-1\}$.

Example 5.1. Table 5.1 shows the *lastPosList* sequences for SDB_1. We consider the sequence with sid_1 and a prefix $\langle A \rangle$. The computation of the frequencies starts at position 2, remaining suffix is $\langle BCBC \rangle$. Instead of visiting all the 4 positions of this suffix, only the last two can be visited thanks to the information contained in $lastPosList[sid_1]$. Indeed according to $lastPosList[sid_1][1]$ the maximum last position is 5 (corresponding to the last C). Then according to $lastPosList[sid_1][2]$ the second maximum last position is 4 (corresponding to the last position of symbol B). The third maximum last position is 1 for symbol A. Since this position is smaller than 2 (our initial start), we can stop.

Weakness 3 (finding a new item position). Related to weakness 2, Line 11 in Algorithm 5.2 finds the new position (pos_s) of a in $SDB[sid]$. This code is executed even if the new symbol no longer appears in that sequence. Currently, the code has to loop over the entire sequence until it reaches the end before discovering this.

Assume that the current position in the sequence s is already larger than the position of the last occurrence of a. Then we immediately know this sequence cannot be part of the projected database. To verify this in $O(1)$ time, we use a *lastPosMap* as follows:

Definition 5.3. ***(Last position map of symbols).*** *For a given sequence s with id sid, $lastPosMap[sid]$ is a map such that $lastPosMap[sid][i]$ is the last position of symbol i in the sequence s. In case the symbol i is not present: $lastPosMap[sid][i] = 0$ (positions are assumed to start at index 1).*

Example 5.2. Table 5.1 shows the *lastPosMap* arrays next to SDB_1. For instance for sid_2 the last position of symbol C is 4. Assume we want to build the projected database of A $SDB_{|\langle A\rangle}$. We don't visit sid_4 since its last position is 0, which means that A does not exist in this sequence. In Figure 5.1 one can see all the sequences (with the symbol ⊘) that won't be visited through the tree for enumeration of all the solutions for $\theta = 3$.

Weakness 4 (storing projected databases). Algorithm 5.2 creates a new set $PSDB_i$ to represent the projected database. This projected database is computed many times during the search, namely at least once in each node of the search tree (more if there are other constraints in the fixPoint set). This is a source of inefficiency for garbage collected languages such as Java but also for C since it induces many "slow" system calls such as free and malloc leading to fragmentation of the memory. We propose to store and restore the pseudo-projected databases with reversible vectors making use of CP trailing techniques. The idea is to use one and the same array throughout the search in the propagator, and only maintain the relevant start/stop position during search. Each call to propagate will read from the previous start to stop position, and write after the previous stop position plus store the new start/stop position. The projected databases are thus *stacked* in the array along a branch of the search tree. We implement the pseudo-projected database with two reversible vectors: *sids* and *poss* respectively for the sequence ids and the current position in the corresponding sequences. The position ϕ is the start entry (in *sids* and *poss*) of the current projected database, and φ is the size of the projected database. We thus have the current projected database contained in sub-arrays $sids[\phi, \ldots, \phi + \varphi - 1]$ and $poss[\phi, \ldots, \phi + \varphi - 1]$. In order to make the projected database reversible, ϕ and φ are reversible integers. That is on backtrack to an ancestor node those integers retrieve their previous value and entries of *sids* and *poss* starting from ϕ can be reused.

Example 5.3. Figure 5.1 is an example using SDB_1. Initially all the sequences are present $\varphi = 4$ and position is initialized $\phi = 0$. The $\langle A \rangle$-projected database contains sequence $1, 2, 3$ at positions $1, 2, 1$ with $\phi = 4$ and $\varphi = 3$. To build the $\langle B \rangle$-projected database since we have finished with the AB-branch, we can reuse the reversible vectors from position 4 i.e. the old values will be overwritten.

Prefix Projection Incremental Counting propagator (PPIC).

Putting all the solutions to the identified weaknesses together, we list the code of the main function of our propagator's in Algorithm 5.3.

The main loop at Line 10 iterates over the previous *(parent)* projected database. In case the sequence at index i in the projected database contains the new symbol at a subsequent position larger or equal to $start$, the matching position is searched and added to the new projected database (at index j of reversible vectors *sids* and *poss*) at Line 17. Then the contribution of the sequence to the projected frequencies is computed in the loop at Line 19. Only the entries in the `lastPosList` with position larger than current *pos* are considered (recall that his list is decreasing according to positions). Finally, Line 23 updates the reversible integers ϕ and φ to reflect the newly computed projected database. Based on these projected frequencies a filtering similar to the one of Algorithm 5.1 is achieved except that only the domain of the next variable $D(P_{i+1})$ is filtered according to the solution to Weakness 1.

Prefix Projection Decreasing Counting propagator (PPDC).

The key idea of this approach is not to count the projected frequencies from scratch, but rather to *decrement* them. More specifically, when scanning the position of the current symbol at Line 15, if *pos* happens to be the last position of a symbol (`pos==lastPosMap[sid][s[pos]]`) then `projFreqs[s[pos]]` is decremented. This requires `projFreqs` to be an array of reversible integers. With this strategy the loop at Line 19 disappears, but in case the current sequence is not added to the projected database, the frequencies of all its last symbols occurring after *pos* must also be decremented. This can be done by adding an **`else`** block to the **`if`** defined at

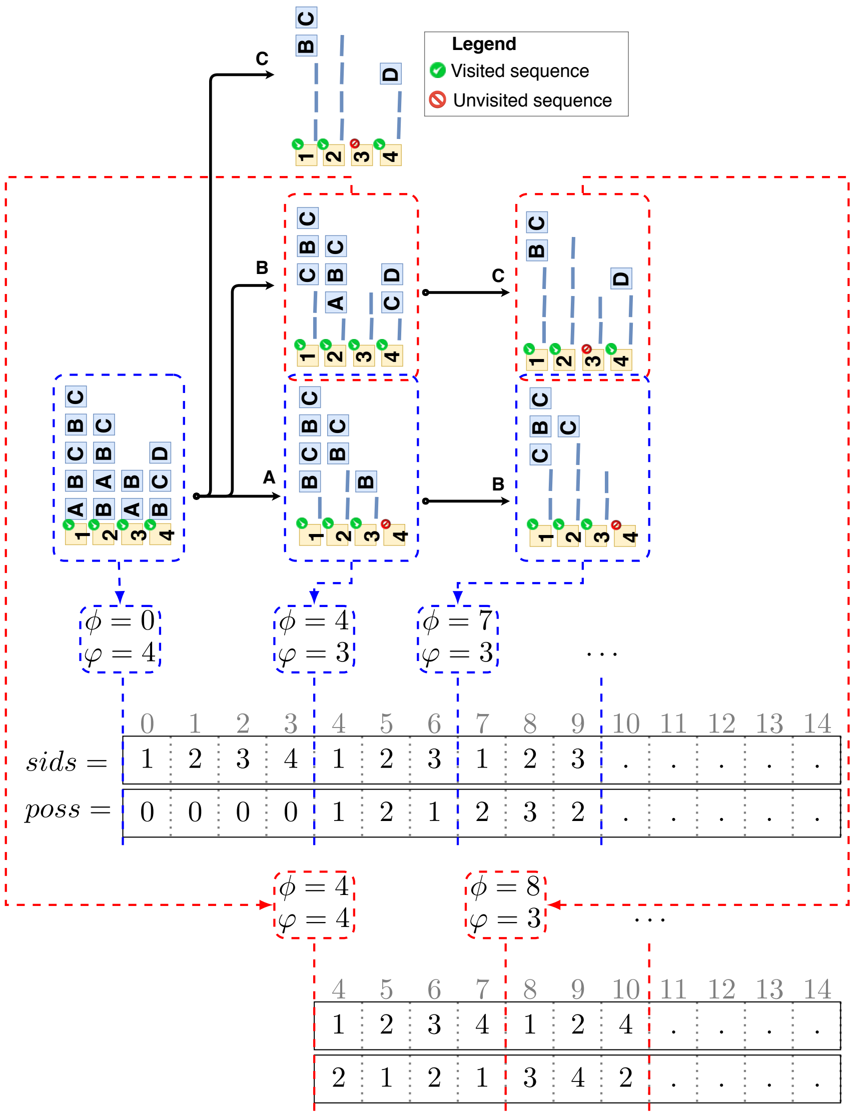


Figure 5.1. Projected databases tree obtained from SDB_1 with $\theta = 3$ (on top) and the Reversible vectors of the trailed-based data structure (on bottom).

Line 12 that will iterate over the `lastPosList` and decrement the symbol frequencies.

Algorithm 5.3: ProjectAndGetFreqs(SDB, a, θ, *sids*, *poss*, ϕ, φ)

```
▷ Input: SDB is a sequence database;
        $a$ is the current projected symbol;
        $\theta$ is the support threshold;
        $sids$ and $poss$ are the reversible vectors.
▷ Output: $freq$ list of projected frequencies
▷ Internal State: $startv$, $esize$, $embs$, $\phi$, $\varphi$, $\psi$, $\gamma_s(P_{:\psi})$.

projFreqs[b]$\leftarrow 0\ \forall b \in \{1, \cdots, m\}$
$i \leftarrow \phi$    $j \leftarrow \phi + \varphi$    $sup \leftarrow 0$
while $i < \phi + \varphi$ do
    $sid \leftarrow sids[i]$    $pos \leftarrow poss[i]$    $s \leftarrow$ SDB[$sid$]
    if $lastPosMap[sid][a] - 1 \geq start$ then
        ▷ find the next position of a in s
        while $pos <$ size($s$) and $a \neq s[pos]$ do
            $pos \leftarrow pos + 1$
        ▷ update projected database
        $sids[j] \leftarrow sid$    $poss[j] \leftarrow pos + 1$    $j \leftarrow j + 1$    $sup \leftarrow sup + 1$
        ▷ recompute projected frequencies
        foreach $(symbol, pos_x)$ in $lastPosList[sid]$ do
            if $pos_x \leq pos$ then break
            projFreqs[$symbol$] $\leftarrow$ projFreqs[$symbol$] + 1
    $i \leftarrow i + 1$
$\phi \leftarrow \phi + \varphi$    $\varphi \leftarrow sup$
return projFreqs
```

Example 5.4. Assume SDB_1. The initial projected frequency array is `projFreqs= [A:3,B:4,C:3,D:1]`. Consider now the $\langle A \rangle$-projected database illustrated on Figure 5.2. The projected frequency array becomes `projFreqs=[A:0,B:3,C:2,D:0]`. The entry at `A` is decremented three times as *pos* moved beyond its *lastPos* for each of the sequences sid_1, sid_2 and sid_3. The entry `B` at position 1 in sid_2 won't be decremented because its *lastpos* is at position 3. Since sid_4 is removed from the projected database, the frequency of all its last symbols occurring after *pos* is also decremented, that is for entries B, C and D.

PP-mixed. Both PPIC and PPDC approaches can be of interest depending on the number of removed sequences in the projected database. If the number of sequences removed is large, then PPIC is preferable. On the other

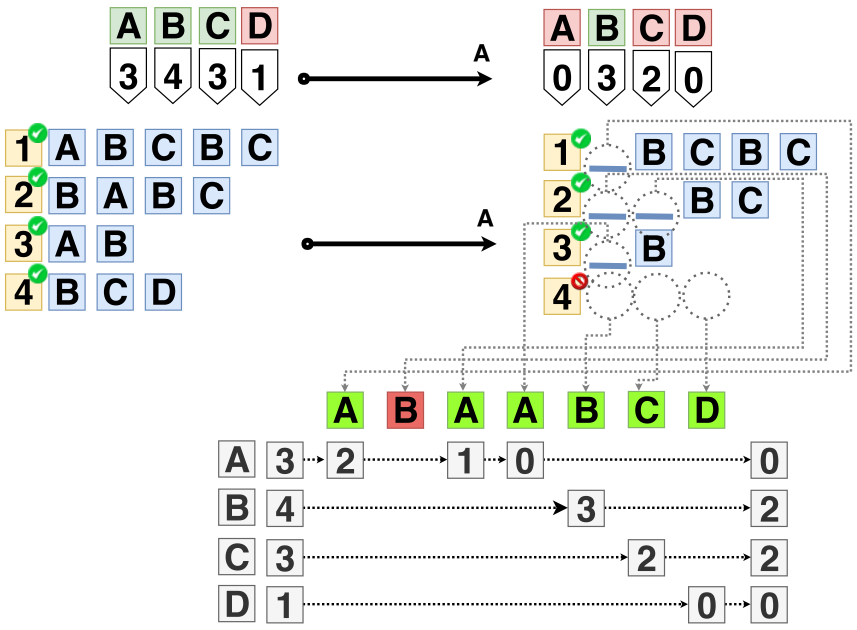


Figure 5.2. Prefix Projection Decreasing Counting Principle

hand, if only a few sequences are removed then PPDC can be more interesting. Inspired from the *reset* idea of [PR14] the PP-mixed approach dynamically chooses the best strategy: if $projFreqs_{SDB}(a) < \text{size}(PSDB_i)/2$ (i.e., more than half of sequences will be removed) then PPIC is used otherwise PPDC.

5.3.3 Constraints of SPM

We implemented common constraints such as minimum and maximum pattern size, symbol inclusion/exclusion, and regular expression constraints. Time constraints (maxgap, mingap, maxspan, etc) are outside the scope of this work: they change the definition of what a valid prefix is, and hence require changing the propagator (as in [KLL$^+$16]). We discuss this in the following chapter.

5.3.4 Time and space complexity

Let us denote by $l = \text{size}(SDB)$ the number of sequences, $L = \text{size}(P)$ the length of the longest sequence, $m = \text{size}(\mathcal{I})$ the size of item alphabet. In the worst case, the time and the space complexity of our propagator is in $\mathcal{O}(l \times (L + m))$.

Space complexity. Storing the database itself costs $\mathcal{O}(l \times L)$. The same amount of memory is necessary to store the data structure (at most l embeddings by search node entry and at most a pattern with size L will be found). The pre-computed lists cost $\mathcal{O}(l \times (L + m))$. Hence the space complexity of our algorithms is $\mathcal{O}(l \times L + l \times (L + m)) = \mathcal{O}(l \times (L + m))$. $\square$

Time complexity. *PPIC* needs $\mathcal{O}(l \times (L + m) + m) = \mathcal{O}(l \times (L + m))$ time to be complete since building the projected database costs $\mathcal{O}(l \times (L + m))$ (Line 6 of the Algorithm 5.1) and the pruning costs $\mathcal{O}(m)$. $\square$

5.4 Implementation and Practical User Guide

All our software, datasets and results are available online as open source in order to make this research reproducible (http://sites.uclouvain.be/cp4dm/).

The implementation of *PPIC*, *PPDC* and *PPmixed* is available in the CP-Solver OscaR [Osc12] which is available online[1] in free access. We also provide the implementation of *PPIC* in Section A.4. In OscaR, one can combine our constraints with the existing constraints in the solver such as All-Different [Rég94], Global cardinality [QLvBG04], Grammar [QW06], etc.

For developers who are willing to modify the code directly, a lightweight version is also available[2]. One can hence add several constraints for a specific usage without understanding OscaR deeply. The installation procedure is described in the Install file in the code directory and merely consists of "importing the project" into your favourite IDE.

[1] https://bitbucket.org/oscarlib/oscar/wiki/Home

[2] https://bitbucket.org/pschaus/cp4d

Users can directly download the *jar-file* which is available on our website [3]. On this website, there is a user guide. The general format of the command to run *PPIC* is:

```
java -jar ppic.jar [options] <SDB File> <Lmin > < Lmax>
```

with several options available, including:

- $-f$: for the frequency constraint (i.e. θ),
- $-i$: for items inclusion/exclusion constraints,
- $-e$: for the regular expression constraint,
- etc.

For example, to find the frequent sequences with size between 2 and 3 given a database (named test.txt) and the minimum support $\theta = 2$ (-f 2), we run the following command:

```
java -jar ppic.jar test.txt 2 3 -f 2
```

Here is the result:

```
Input                            Output
(each line is a sequence)        (<sub-sequence> : <support>)
                                 < 1 2  > : 3
1 2 3 2 3                        < 1 2 3  > : 2
2 1 2 3                          < 1 3  > : 2
1 2                              < 2 2  > : 2
2 3 4                            < 2 2 3  > : 2
                                 < 2 3  > : 3
```

Table 5.2. Dataset Features. Sparsity is equal to $(\frac{1}{\text{size}(SDB)} \times \sum \frac{\text{size}(s)}{\text{size}(I_{/s})})$

SDB	size(SDB)	size($\mathcal{I}$)	avg.size(s)	avg.size($I_{/s}$)	max.size(s)	sparsity	description
BIBLE	36369	13905	21.64	17.85	100	1.2	text
FIFA	20450	2990	36.24	34.74	100	1.2	web click stream
Kosarak	69999	21144	7.98	7.98	796	1.0	web click stream
Leviathan	5834	9025	33.81	26.34	100	1.3	text
PubMed	17237	19931	29.56	24.82	198	1.2	bio-medical text
data200k	200000	26	50.25	18.25	86	2.8	synthetic data
protein	103120	25	482.25	19.93	600	24.2	protein sequences

[3] https://sites.uclouvain.be/cp4dm/spm/

5.5 Experiments

In this section, we report our experimental results on the performance of our approaches with six real-life datasets[4] and one synthetic (data200k [TBG08]) with various characteristics shown in Table 5.2. Sparsity, representing the average of the number of symbols that appear in each sequence, is a good indicator of how sparse or dense a dataset is.

Our work is run under JVM with maximum memory set to 8GB. We used a machine with a 2.7Hz Intel core i5 processor and 8GB of RAM with Linux 3.19.0-32-generic 64 bits distribution Mint 17.3. Execution time limit is set to 3600 seconds (1 hour). Our proposals are compared, first, with CPSM[5] [NG15] and Gap-Seq[6] [KLL$^+$16], the recently CP-based approaches including Gap constraint and the previous version of Gap-Seq, PP[7] [KLL$^+$15] without Gap but with regular expression constraint. Second, we made comparison with cSpade[8] [Zak00], PrefixSpan [PHMA$^+$01][9] and SPMF[10].

PPIC vs PPDC vs PPmixed. The CPU time of PPIC, PPDC and PPmixed models are shown in Figure 5.3. PPIC is more efficient than PPDC in 80% of datasets. This is essentially because in many cases at the beginning of mining, there are many unsupported sequences for which the symbol counters must be decremented (compared to not having to increase the counters in PPIC). For instance with BIBLE SDB and *minsup* = 10% PPDC need to see 21,979,585 symbols to be complete while only 15,916,652 is needed for PPIC. Unsurprisingly, PPmixed is between these approaches.

Our proposals vs Gap-Seq (CP method). Figure 5.3 confirms CPSM is outperformed by Gap-Seq which itself improves PP (without gap). We can clearly notice our approaches outperform Gap-Seq (and hence PP) in all cases. In the case of FIFA SDB, Gap-Seq reach time limit when *minsup* $\leq$ 9%. PPIC is very effective in large and dense datasets regarding of CPU-times.

Comparison with specialized algorithms. Our third experience is the comparison with specialized algorithms. As we can see in the Figure 5.4, we perform better on 84% of the datasets. However, cSpade is still the most efficient for Kosarak. Kosarak doesn't contain any symbol repetition in its sequences. So it is a

[4]http://www.philippe-fournier-viger.com/spmf/
[5]https://dtai.cs.kuleuven.be/CP4IM/cpsm/
[6]https://sites.google.com/site/cp4spm/
[7]https://sites.google.com/site/prefixprojection4cp/
[8]http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software
[9]http://illimine.cs.uiuc.edu/software/
[10]http://www.philippe-fournier-viger.com/spmf/index.php?link=download.php

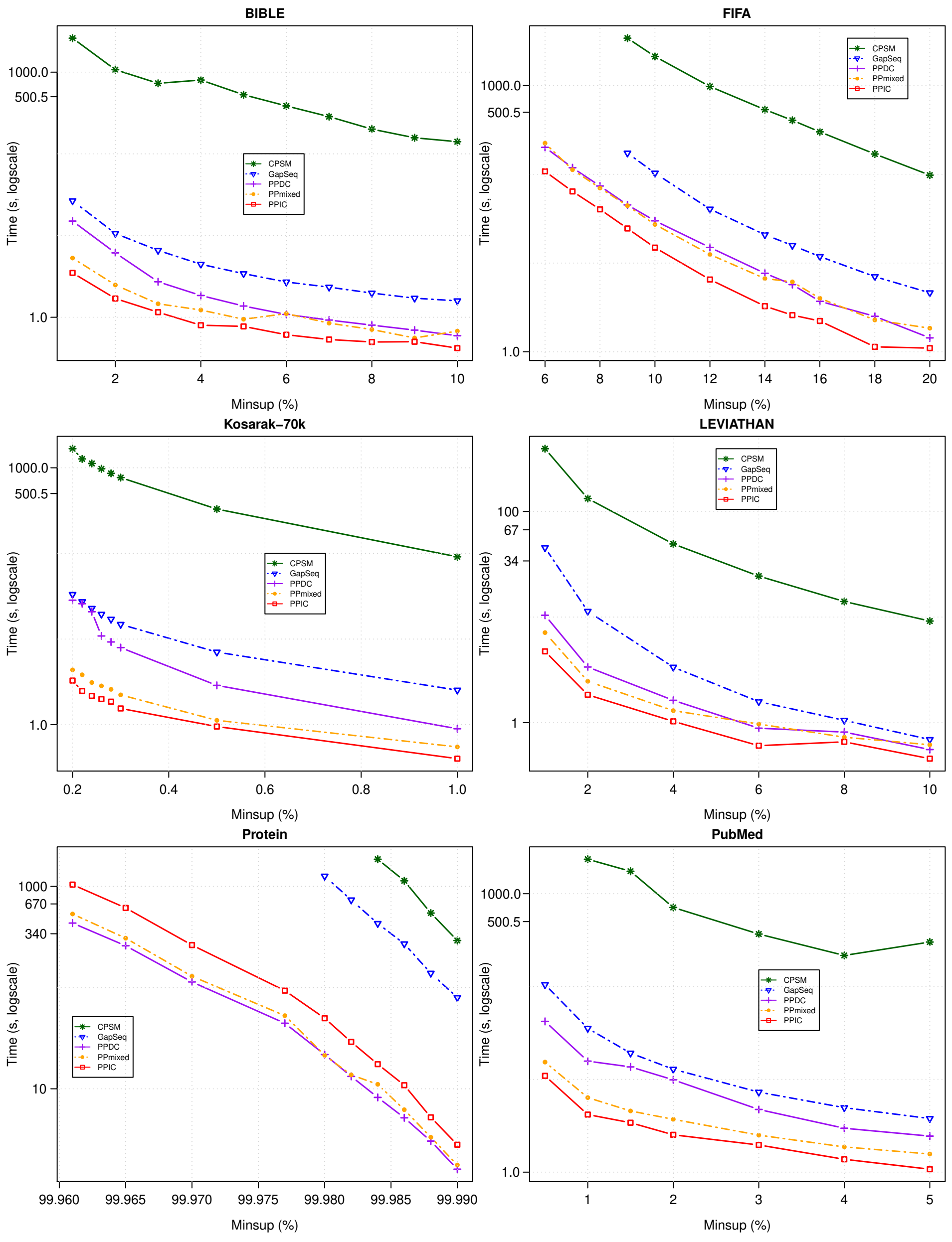


Figure 5.3. CPU times for PPIC, PPDC, PPmixed and Gap-Seq for several minsup (missing points indicate a timeout)

bad case for prefix-projection-based algorithms which need to scan all the positions. On the contrary, with protein dataset (the sparse one) cSpade requires much more CPU time. The SPMF implementation of SPAM, PrefixSpan and LAPIN appears to be consistently slower than cSpade but there is no clear domination among these.

Impact of the improvements. Figure 5.5 shows the incremental impact of our proposed solutions to the weaknesses defined in Section 5.3.2, starting from reversible vectors (fix of weakness 4) up to all our proposed modifications. Fix 1 has limited effect, while adding fix 3 is data dependent but adding fix2 always improves further.

Handling different additional constraints. In order to illustrate the modularity of our approach we compare with a number of user-defined constraints that can be added as additional modules without changing the main propagator (Figure 5.6). (a) We compared PPIC and PP (unfortunately the Gap-Seq tool does not support a regular expression command-line argument) under various size constraints on the protein dataset with $minsup = 99.984$. (b,c) We also selected data200k adding a regular expression constraint $RE10 = A * B(B|C)D * EF * (G|H)I*$ and $RE14 = A * (Q|BS * (B|C))D * E(I|S) * (F|H)G * R$ [TBG08]. The last experiment reported on Figure 5.6d consists in combining size and symbols constraints on the protein dataset: only sequential patterns that contain VALINE and GLYCINE twice and ASPARATE and SERINE once are valid. PPIC under constraints still dominates PP.

5.6 Summary, Outlooks, Further readings

This work improved the existing CP-based sequential pattern mining approaches [NG15, KLL$^+$16] up to the point that it also outperforms specialized mining systems in terms of efficiency. To do so, we combined and adapted some ideas from both the sequence mining literature and the constraint programming literature; correspondingly last-position information [YK05] and reversible data-structures for storing and restoring state during backtracking search. We introduced the PrefixProjection-Inc (PPIC) global constraint and two variants proposing different strategies to compute the projected frequencies: from scratch, by decreasing the counters, or a mix of both. These can be plugged in as modules in a CP solver. These constraints are implemented in Scala and made available in the generic OscaR solver. Furthermore, the approach is compatible with some constraints including size and regular expression constraints. There are other constraints which change the subsequence relation and which would hence require hardcoding changes in the propagator (gap [KLL$^+$16], span, etc). We think many of our improvements can be applied to such settings as well.

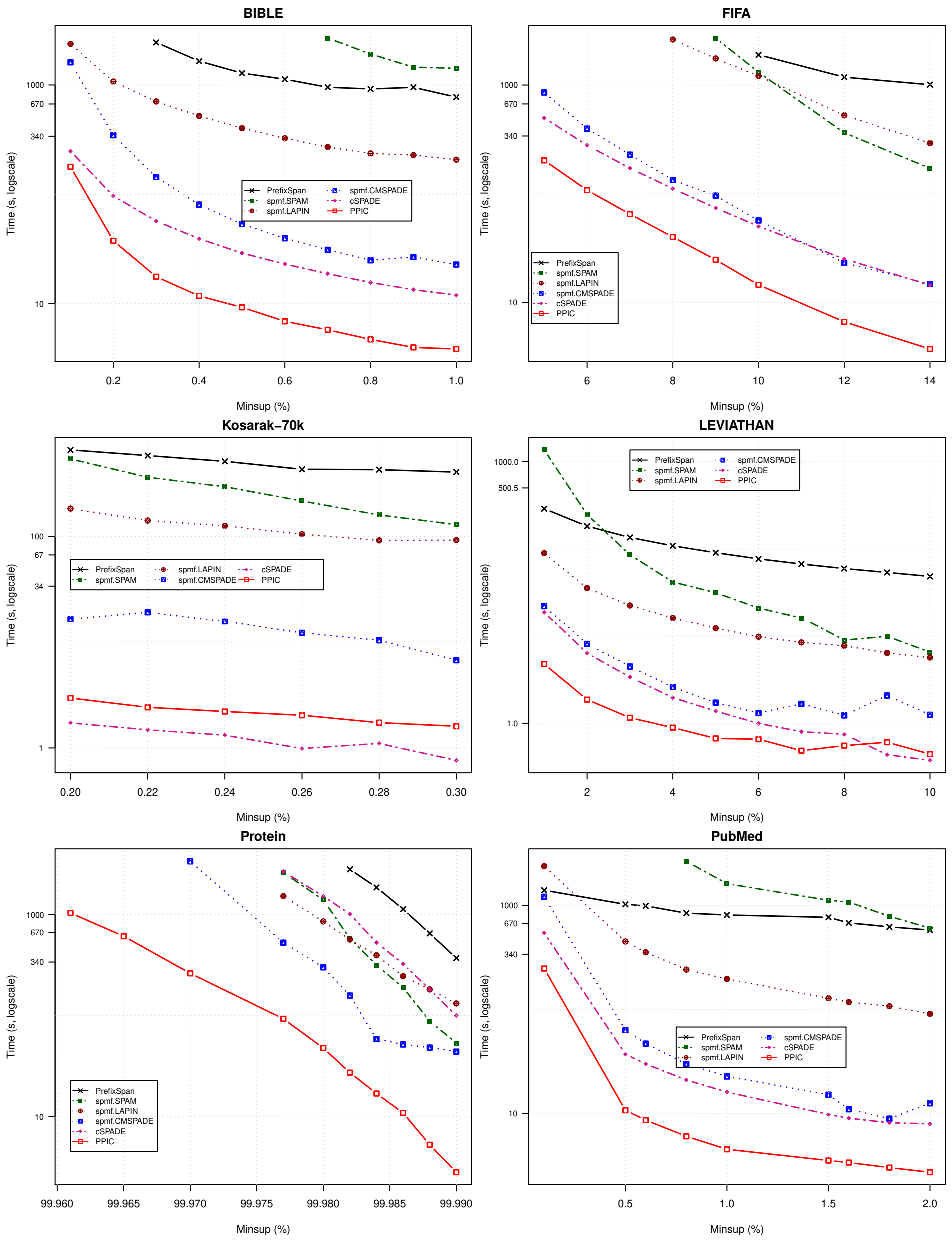


Figure 5.4. CPU times for PPIC,PPDC,PPmixed and cSPADE for several *minsup* (missing points are due to timeout).

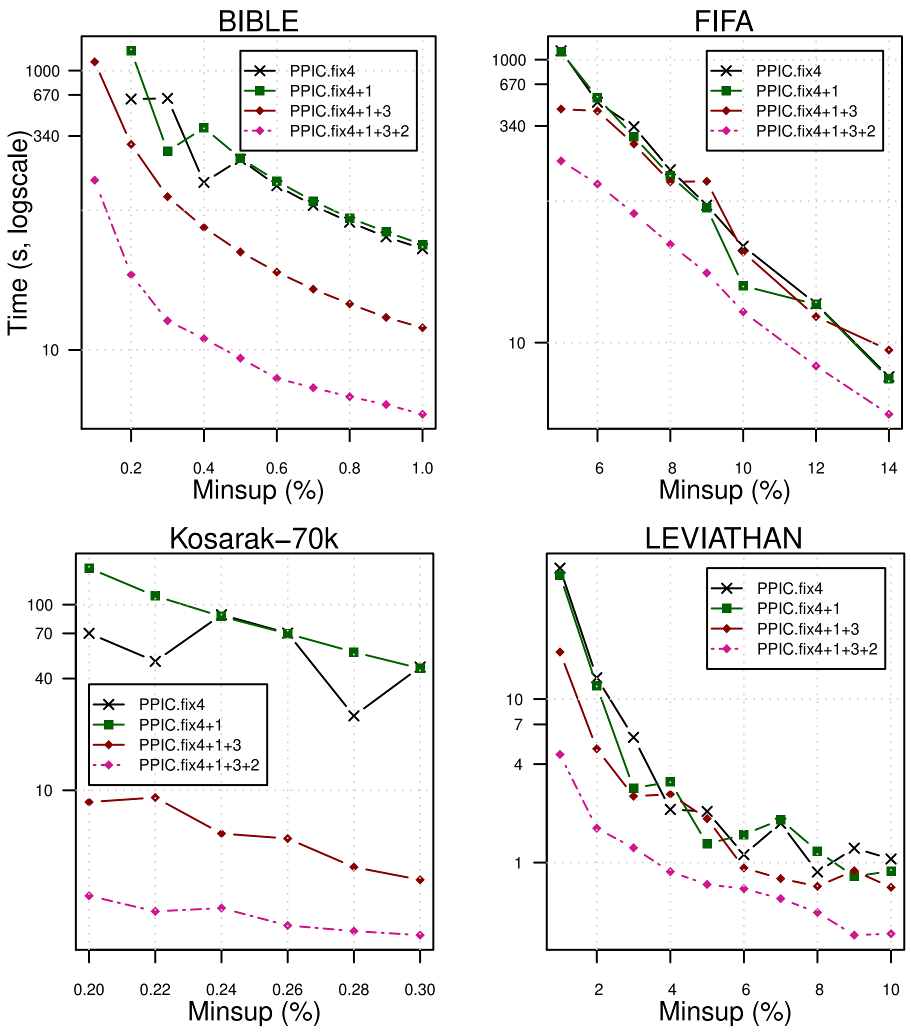


Figure 5.5. Incremental impact of our solutions to the different weaknesses (yaxis is logscale for all plots)

Our work shows that generic CP solvers can indeed be used as framework to build scalable mining algorithms, not just for generic yet less scalable systems as was done for itemset mining [GNDR11]. Furthermore, advanced data-structures for backtracking search, such as trailing and reversible vectors, can also be used in non-CP algorithms. This appears to be an understudied aspect of backtracking algorithms in pattern mining and data mining in general. We believe there is much

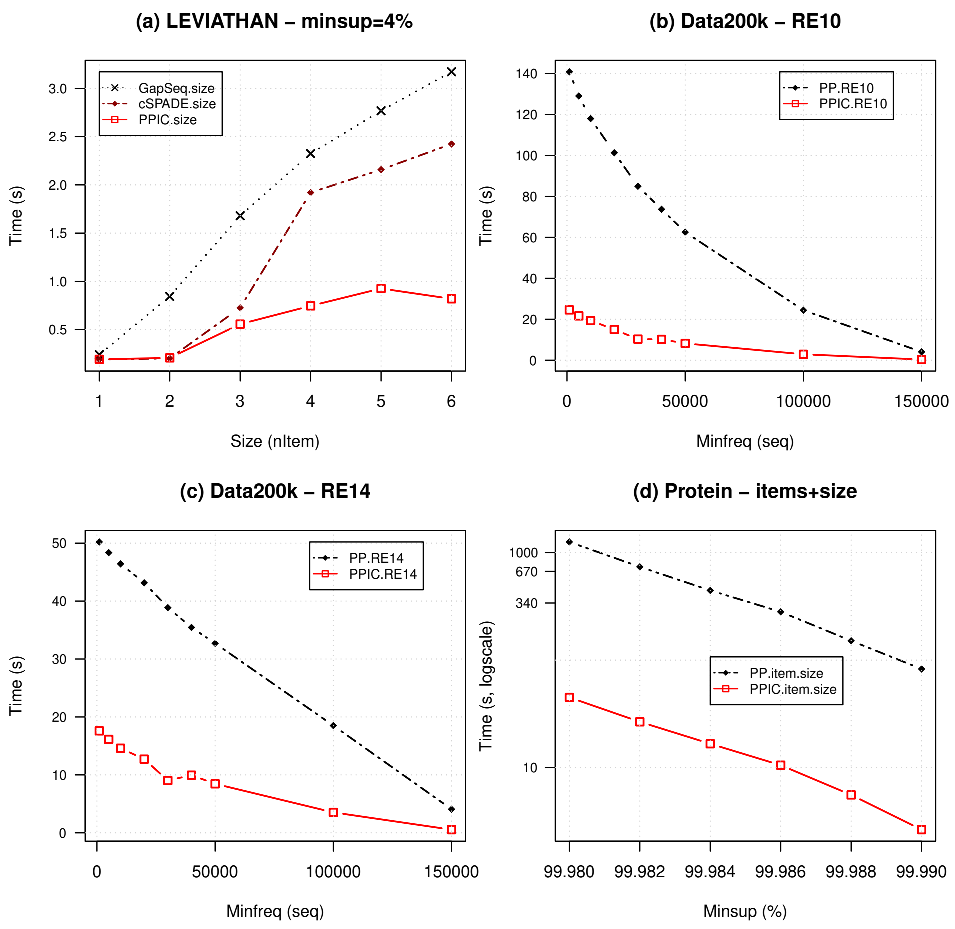


Figure 5.6. Handling of different additional constraints

more potential for combinations of techniques from data mining and CP.

CHAPTER 6

Mining time-constrained sequential patterns with constraint programming

"It is upon the old string that we weave the new rope."

–African Wisdom

Overview *In the previous chapter, we showed that by combining techniques from Constraint Programming and Sequential Pattern Mining, we could build new flexible and efficient approaches which outperform both CP-based and Specialized approaches becoming the state-of-the-art of standard sequential pattern mining. Commonly, the sequence dataset contains time information. However, this information is often ignored because it is a challenging task to take it into account. In this chapter, we introduce a new constraint, called PPICt, which adapted ideas from PPIC, to address Sequential Pattern Mining with timed dataset.*

Contribution *The main contribution in this work, is to improve on [KLL^{+}16] and [AGS16] by modifying the global frequency constraint (PPIC) to capture the most common time-related constraints: explicitly timed events, minimum/maximum gap (i.e. time between two matching events in a sequence), and minimum/maximum span (i.e. time between the first and last matching event). To maintain scalability, we also ensure that we don't needlessly scan the sequences in the database during the search.*

Main source *This chapter is mainly based on our paper [AGS17].*

6.1 Context and Motivation

Sequential pattern mining (SPM) is an important research domain within data mining and widely used in applications such as weblog mining, disease diagnoses mining, event sequence mining, etc [AH14]. The problem of SPM is to find frequent sequence patterns (also called sequential patterns) in a database of sequences, i.e. an ordered list of events which together occur in the data more than a given number of times. This task is a great challenge since the search space is extremely large; $\mathcal{O}(m^n)$ solutions are available for patterns with length at most n and for sequences with an average number of m events.

In practice, finding all sequential patterns is typically not enough, as often an overwhelming number of patterns is returned. Hence, there is a need to guide the search towards patterns of interest to the practitioner. This calls for techniques which can incorporate preferences or restrictions on the length and content of the patterns (constraints). In many applications, the time elapsed between events is also important to take into account.

Assume for instance a database containing sequences of web pages visited by users on a given website. One could be interested in access patterns within a single browsing session, for example with no more than 20 minutes between two pages. Also in biological sequence mining the position and distance of the symbols in the sequence matter. A constraint on the maximum time between any two consecutive symbols in the pattern is called a *gap* constraint, while a constraint on the time from the first to the last event is called a *span* constraint.

In this work, we assume all sequences have explicit timestamps and the goal is to support gap and span constraints as well as constraints on frequency and syntax of patterns.

In this work, we wish to improve on [KLL$^+$16] and [AGS16] by modifying the global frequency constraint to capture the most common time-related constraints: explicitly timed events, minimum/maximum gap, and minimum/maximum span. To maintain scalability, we must ensure that we do not needlessly scan the sequences in the database during the search. Our contributions can be summarized as follows:

1. we adapt the backtracking-aware data structure introduced in [AGS16] to store all possible occurrences of the pattern in a sequence, including the first matching symbol to support *span* constraints;
2. we avoid scanning a sequence for a symbol beyond the (precomputed) last occurrence of that symbol in the sequence;
3. we introduce the concept of *extension window* of an embedding and avoid to scan overlapping windows multiple times;
4. we avoid scanning for the start of an extension window, which is specific to the minimum gap constraint, by precomputing these in advance; and finally

5. we experimentally show that using this global constraint we outperform other sequence mining algorithms in all but a few cases. Furthermore, we show that in a CP framework this global constraint can be combined with a number of other independent constraints: item inclusion/exclusion constraints, pattern length constraints or string constraints [HFPZ13] such as regular expression [Pes04] and grammar [KS10, QW06] constraints.

6.2 Related work

The problem of sequential pattern mining, first introduced by Agrawal et al. [AS95], is widely studied [AS95, HPYM04, PHMA+01, SA96, YHA03, Zak98, Zak00] with many applications as well [HAM14, HM14]. These works can be categorized into *1)* apriori-based (horizontal/vertical formatting) [AFGY02, SA96, Zak00] and *2)* projection-based [PHMA+01] methods. In general, users only need a small subset of the found patterns. Hence, some works have focused on the addition of user-defined constraints such as inclusion/exclusion items, pattern length (minimum/maximum), super-pattern, aggregate function (sum, average, maximum, minimum and standard deviation), regular expression and span/gap. They are discussed in more detail in [PHW07].

GSP [SA96] was the first approach including gap and span constraints. This method is not very efficient since it requires to generate all candidate patterns and to scan the dataset several times. Some approaches added the constraints in a post-processing step [PZOD99]. In the *cSPADE* algorithm [Zak00], the constraints are directly integrated into the sequential pattern search process. It efficiently takes into account constraints such as length and width restrictions on the pattern, item constraints, minimum and maximum gaps between events, as well as a maximum span. Unlike *cSPADE*, *GenPrefixSpan* [AO03] is an extension of the depth-first *PrefixSpan* [PHMA+01] algorithm to allow gap constraints. Time constraints on the sequences (instead of events) have also been investigated in [DG15]. Special classes of SPM problem or constraints was also tackled: the closed/maximal SPM [FVWT13, LW08, LL04, WHL07, YHA03] the multi-dimensional SPM [PHP+01], the episodes events [MTV97], etc. However, all the above-mentioned approaches lack flexibility at the algorithmic level, since adding a new constraint often involves changing the whole algorithm and may hinder scalability. For instance, methods that can efficiently take regular expression constraints into account together with time constraints are rare in specialized methods. An exception is *PG* [PHW07], which starts from the observation that many constraints are prefix-monotone. This property is weaker than standard (anti)monotonicity when used in pruning, but still valid. PG's pruning principles are specific to prefix-monotonicity however, which does not allow it to fully exploit regular expression constraints for example.

As an alternative, the use of Constraint Programming (CP) has been investigated [AGS16, CJSS12, GNDR13, KLL+17, KLL+15, KLL+16, MBC+11, NG15].

a) Sequence database (SDB)		b) *nextPosGap*							c) *lastPosMap*				
sid	sequence	1	2	3	4	5	6	7	A	B	C	D	E
sid_1	$\langle(A,2)(B,5)(D,6)(C,10)(B,11)\rangle$	2	4	4	6	6			1	5	4	3	0
sid_2	$\langle(B,1)(A,2)(A,9)(D,12)(C,15)(A,18)(B,24)\rangle$	3	3	4	5	6	7	8	6	7	5	4	0
sid_3	$\langle(A,2)(B,4)(D,6)(D,8)(B,10)(E,12)(C,14)\rangle$	3	4	5	6	7	8	8	1	5	7	4	6
sid_4	$\langle(A,1)(C,2)(C,3)(B,4)\rangle$	4	5	5	5				1	4	3	0	0

Table 6.1. a) A sequence database SDB, **b)** a structure for the next position of minimum gap time N(precomputed) and **c)** the last position map.

Kemmar et al. [KLL+15] have subsequently shown that this approach can be made more scalable by grouping all low-level constraints involving the frequency computation into one global constraint. Moreover, they investigated the top-k sequential pattern mining problem [KLL+17]. More recently, we have shown that combining this approach with algorithmic techniques from both the CP community and the data mining community can result in a global constraint that outperforms generic as well as specialized methods [AGS16].

While the above CP methods can handle constraints on the pattern syntax, gap and span constraints are only supported by the much less efficient approach of [NG15]. The reason is that the timing information is hidden in the global *frequency* constraint. Hence, Kemmar et al. [KLL+16] extended their global constraint for the gap constraint specifically.

6.3 Preliminaries

In this section, we revisit the notions around Sequential pattern mining through an example, then we consider the problem of Sequential pattern mining with time constraints such as the gap and span. Finally, we will discuss how these problems are modelled in CP.

6.3.1 Sequential Pattern Mining

Assume $\mathcal{I} = \{1, \ldots, m\}$ is a list of possible symbols. Table 6.1a represents an example sequence database (SDB) with timestamps. The database is a set of tuples (sid, s) where sid is the sequence identifier and $s = \langle(s_1, t_1)(s_2, t_2) \ldots (s_n, t_n)\rangle$ is a sequence; an ordered list of symbols/events (s_k) occurred at time t_k, where $t_1 \leq t_2 \leq \ldots \leq t_n$. We use s_i^s, respectively s_i^t, to represent just the list of symbols, respectively timestamps, of sequence i.

⚠ In the rest of the chapter, we assume a sequence database has timestamps, and when the exact timing is not important we will write $\langle ABC \rangle$ to mean $\langle (A,1)(B,2)(C,3) \rangle$.

Example 6.1. Assume $s = \langle (A,2)(B,4)(D,6)(D,8)(B,10)(E,12)(C,14) \rangle$ is a sequence, $s^s = \{A, B, D, D, B, E, C\}$, $s^t = \{2, 4, 6, 8, 10, 12, 14\}$ and its length size$(s) = 7$.

SPM without time consideration

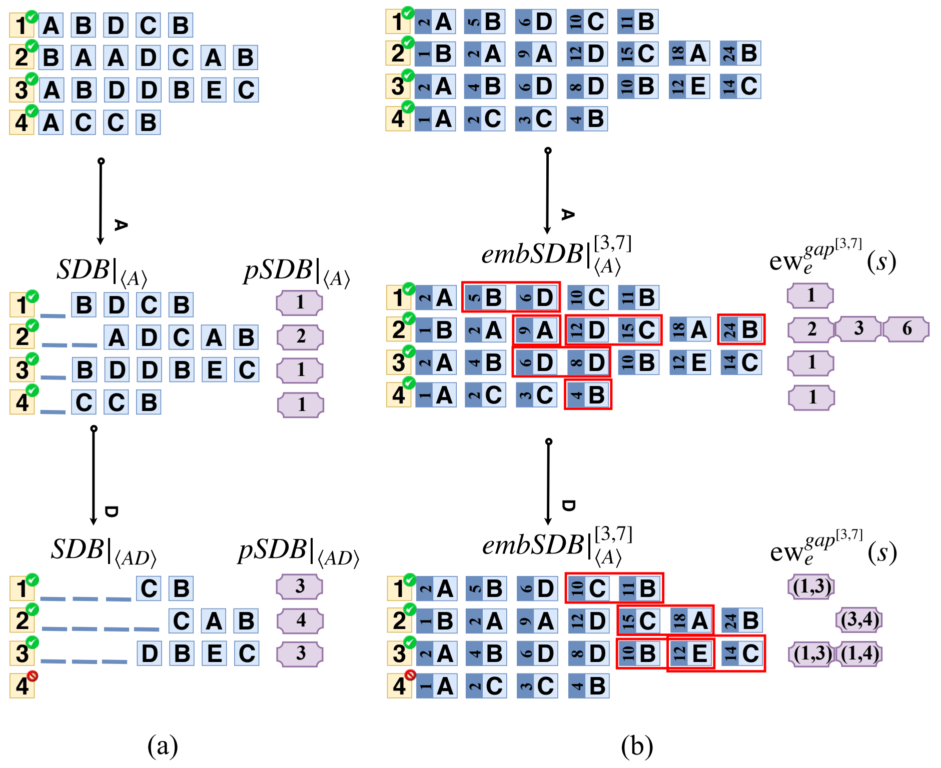
Let's first consider the database without timestamps. Then, the problem of the SPM [AS95] is to enumerate all the frequent sequences with the coverage and frequency constraints (this is what we discussed in Chapter 5).

⚠ The reader can refer to the background Chapter 2 for definitions and specific details. Here we will only give a concrete example.

Example 6.2. Assuming the sequence database in Table 6.1a without timestamps, sequence $\alpha = \langle ADC \rangle$ is a subsequence of s, denoted as $\alpha \preceq s$, with embedding $(1, 3, 7)$. Another valid embedding would be $(1, 4, 7)$. Note that for this standard subsequence relation, timing is not important. Sequence α is a subsequence of sequences $1, 2$ and 3, hence $Cover_{SDB}(\langle ADC \rangle) = \{sid_1, sid_2, sid_3\}$ and its frequency is $Freq_{SDB}(\langle ADC \rangle) = 3$.

There exist multiple algorithms for the SPM problem. The *PrefixSpan* algorithm [PHMA$^+$01] is among the most famous ones and relies on the idea of the prefix-projected database. Our approaches are built on this concept.

Example 6.3. Consider our running example in Table 6.1a, where we omit timing information. Assume $\alpha = \langle A \rangle$, then $SDB|_\alpha = \{(sid_1, \langle BDCB \rangle), (sid_2, \langle ADCAB \rangle), (sid_3, \langle BDDBEC \rangle), (sid_4, \langle CCB \rangle)\}$. The prefix-projected frequencies of $SDB|_{\langle A \rangle}$ are: $freqs(A) = 1$, $freqs(B) = 4$, $freqs(C) = 4$, $freqs(D) = 3$, $freqs(E) = 1$. Extending prefix $\langle A \rangle$ with $\langle D \rangle$ over $SDB|_{\langle A \rangle}$ gives $SDB|_{\langle AD \rangle} = \{(sid_1, \langle CB \rangle), (sid_2, \langle CAB \rangle), (sid_3, \langle DBEC \rangle)\}$ and can be represented as the pseudo-projected database: $pSDB|_{\langle A \rangle} = \{(sid_1, 4), (sid_2, 5), (sid_3, 4)\}$ (details in Figure 6.1a).



***(a)** Without time constraints: Projected and Pseudo-projected databases

sid	$pSDB\|_{\langle A\rangle}$	$SDB\|_{\langle A\rangle}$'s of $pSDB\|_{\langle A\rangle}$	$pSDB\|_{\langle AD\rangle}$	$SDB\|_{\langle AD\rangle}$'s of $pSDB\|_{\langle AD\rangle}$
sid_1	1	$\langle BDCB\rangle$	3	$\langle CB\rangle$
sid_2	2	$\langle ADCAB\rangle$	4	$\langle CAB\rangle$
sid_3	1	$\langle BDDBEC\rangle$	2	$\langle DDBEC\rangle$
sid_4	1	$\langle CCB\rangle$		

****(b)** Considering time constraints: Embedding database and extension windows

sid	$embSDB\|^{[3,7]}_{\langle A\rangle}$	$\mathrm{ew}_e^{gap^{[3,7]}}(s)$'s of $embSDB\|^{[3,7]}_{\langle A\rangle}$	$embSDB\|^{[3,7]}_{\langle AD\rangle}$	$\mathrm{ew}_e^{gap^{[3,7]}}(s)$'s of $embSDB\|^{[3,7]}_{\langle AD\rangle}$
sid_1	(1)	$\langle(B,5)(D,6)\rangle$	(1,3)	$\langle(C,10)(B,11)\rangle$
sid_2	(2),(3),(6)	$\langle(A,9)\rangle$,$\langle(D,12)(C,15)\rangle$,$\langle(B,24)\rangle$	(3,4)	$\langle(C,15)(A,18)\rangle$
sid_3	(1)	$\langle(D,6)(D,8)\rangle$	(1,3),(1,4)	$\langle(B,10)(E,12)\rangle$,$\langle(E,12)(C,14)\rangle$
sid_4	(1)	$\langle(B,4)\rangle$		

Figure 6.1. Embedding (Projected) databases and extension windows for patterns $\langle A\rangle$ and $\langle AD\rangle$: **a)** without time constraints **b)** with time constraints ($gap^{[3,7]}$). Note that embeddings are positions in s, not timings and these positions start from 0.

SPM with time consideration

We now look at the subsequence relation under a $gap^{[M,N]}$ constraints, with M the minimum and N the maximum gap between two subsequent events, and under a $span^{[W,Y]}$ constraints with W the minimum and Y the maximum span between the first and last event. This requires changing the subsequence definition in Definition 2.6.

Definition 6.1. ***Subsequence relation under gap ($\preceq^{gap^{[M,N]}}$).*** *A sequence $\alpha = \langle \alpha_1 \alpha_2 \ldots \alpha_k \rangle$ is a subsequence of $s = \langle (s_1, t_1)(s_2, t_2) \ldots (s_n, t_n) \rangle$ under $gap^{[M,N]}$ constraint ($\alpha \preceq^{gap^{[M,N]}} s$) iff* (i) *$k \leq n$;* (ii) *there exists a list of integers $(e_1, \ldots, e_k)$, an embedding, with $1 \leq e_1 \ldots \leq e_k \leq n$ such that $s^s[e_i] = \alpha_i$; and* (iii) *the time between two consecutive events t_{e_i-1} and t_{e_i} must be between M and N for all $i \in [2\,..\,k]$, $M \leq t_{e_i} - t_{e_{i-1}} \leq N$. An embedding $(e_1, \ldots, e_k)$ for $\alpha \preceq^{gap^{[M,N]}} s$ is called a $gap^{[M,N]}$-embedding.*

We can similarly define $\preceq^{span^{[W,Y]}}$ where condition *(iii)* becomes: the time between the first event t_{e_1} and the last t_{e_k} must be between W and Y i.e. $W \leq t_{e_k} - t_{e_1} \leq Y$. We can similarly define the *gap + span* subsequence relation which contains both conditions.

Example 6.4. Given sid_1 with $s_1^t = \{2, 5, 6, 10, 11\}$ and sid_2 with $s_2^t = \{1, 2, 9, 12, 15, 18, 24\}$ in Table 6.1a. Then, $\langle ADB \rangle \preceq^{gap^{[3,7]}} SDB[sid_1]$ with embedding
$(e_1, e_2, e_3) = (1, 3, 5)$ because $\Big\{ 3 \leq (s_1^t[e_2] - s_1^t[e_1] = 4) \leq 7$ and $3 \leq (s_1^t[e_3] - s_1^t[e_2] = 5) \leq 7 \Big\}$. Note the difference between the positions e_i of the embedding and its time $s_1^t[e_i]$. As another example, $\langle ADB \rangle \npreceq^{gap^{[3,7]}} SDB[sid_2]$ because $3 \leq (s_2^t[e_3] - s_2^t[e_2] = 12) \nleq 7$. Similarly, this embedding and hence the sequence respects a $span^{[8,10]}$ constraint in sid_1: $8 \leq s_1^t[e_3] - s_1^t[e_1] = 9 \leq 10$.

The definition of $Cover_{SDB}(\alpha)$, $Freq_{SDB}(\alpha)$ and the SPM problem can be easily adapted to use the gap/span subsequence relation instead of the original relation $\preceq$.

Example 6.5. Assume $\alpha = \langle ADC \rangle$, $\theta = 3$ and $gap^{[3,7]}$, $Cover_{SDB}^{gap^{[3,7]}}(\alpha) = \{sid_1, sid_2, sid_3\}$ and hence $Freq_{SDB}^{gap^{[3,7]}}(\alpha) = 3$. Thus, α is a gap-constrained sequential pattern for the given threshold.

The problem of SPM under time constraints is as follows:

Definition 6.2. ***SPM under time constraints problem.*** *Find all subsequences (α) in sequence database (SDB) such that $Freq_{SDB}^{gap^{[M,N]},span^{[W,Y]}}(\alpha) \geq \theta$ where θ is the given support threshold and M, N, W, Y are the minimum and maximum gap and span.*

The search space to find the sequential patterns can become intractably large. Hence, to reduce this space, several algorithms rely on the *anti-monotonicity property* 2.1. However, only the frequency with minimum *gap* (M) and maximum *span* (Y) are anti-monotone constraints but the frequency with maximum *gap* (N) and the minimum *span* (W) constraints violate this property.

Example 6.6. Assuming our running example, as illustrated in Figure 6.2a and 6.2b, $\langle ADC \rangle$ is frequent under $gap^{[3,7]}$ with $\theta = 3$ but $\langle AC \rangle$ is not frequent ($Freq_{SDB}^{gap^{[3,7]}}(\langle AC \rangle) = 1 < 3$).

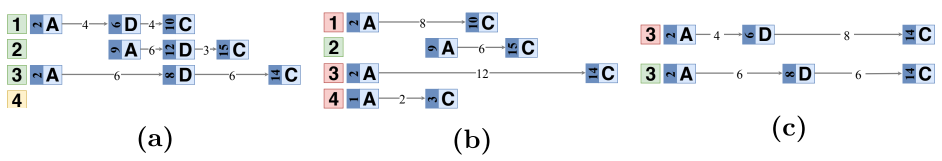


Figure 6.2. Examples showing sequences **a)** $\langle ADC \rangle$ and **b)** $\langle AC \rangle$ in SDB_1 and **c)** embeddings of sequence $\langle ADC \rangle$ in sequence 3.

Fortunately, the maximum gap constraint is *prefix anti-monotone* i.e. every prefix of p satisfies the maximum *gap* constraint if p satisfies it [PHW07]. We use this property to filter embeddings which helps us discover if a prefix is infrequent. The minimum *span* (W) does not satisfy this property, hence we use it only to verify embeddings of fully assigned patterns (post-processing step).

6.3.2 CP-based model for SPM problem

A constraint satisfaction problem [RvBW06] is defined as a triplet (V, D, C) where V is a set of decision variables with their domains D (possible values of V). C is a set of constraints, each constraint is defined over V and restricts the possible values that these variables can take. Solving the problem of SPM using Constraint Programming (CP) consists of defining the model (V, D, C).

We present *CP model* of sequential pattern mining introduced in [NG15] and the *GapSeq* [KLL$^+$16] and *PPIC (Prefix Projection Incremental Counting)* [AGS16] global constraints.

Definition 6.3. ***Variables and Domains for SPM [NG15].*** *Let l be the length of the longest sequence in SDB ($l = max(\{\text{size}(s) \mid s \in SDB\})$); $P = [P_1, P_2, \ldots, P_l]$ is an array of variables, representing a pattern, where each P_i represents the i^{th} symbol in the pattern. The domain D_i of P_i is the set of symbols $\mathcal{I}$ plus the empty symbol ϵ: $D_i(P_i) = \{\epsilon\} \cup \mathcal{I}$.*

Example 6.7. For instance for the dataset in Table 6.1a, $l = 7$, $P = [P_1, \ldots, P_7]$ and for all $i \in [1\,..\,l]$, $D_i = \{\epsilon, A, B, C, D, E\}$. $\langle A, D, C\rangle$ corresponds to $P = [A, D, C, \epsilon, \epsilon, \epsilon, \epsilon]$.

Definition 6.4. ***Filtering rules.*** *Assume $\forall i \leq l$, $p = \langle p_1, \ldots, p_i\rangle$ is the assigned values of variables $\{P_1, \ldots, P_i\}$, a CP model over P represents the SPM problem given a threshold θ, gap$^{[M,N]}$ and span$^{[W,Y]}$ iff the following conditions are satisfied by every valid assignment to P:*

1. *$P_1 \neq \epsilon$ (to avoid an empty pattern);*
2. *$\forall i \in \{2, \cdots, l-1\} : P_i = \epsilon \Rightarrow P_{i+1} = \epsilon$ (to allow pattern with length $< l$);*
3. *Frequency constraint: $Freq_{SDB}(p) \geq \theta$;*
4. *Frequency under gap$^{[M,N]}$ constraint: $Freq_{SDB}^{gap^{[M,N]}}(p) \geq \theta$;*
5. *Frequency under span$^{[W,Y]}$ constraint: $Freq_{SDB}^{span^{[W,Y]}}(p) \geq \theta$.*

PPIC **[AGS16] global constraint.** $PPIC(P, SDB, \theta)$ is a global constraint for the SPM problem without gap/span, built on the prefix-projection principle, which enforces filtering rules 1, 2 and 3 in a single propagator. It improved on the state-of-the-art with four elements: *(a)* a backtracking-aware data structure inspired by *trailing-based CP technique*, *(b)* efficient support counting by precomputing the last positions of each symbol, *(c)* not scanning sequences whose prefix can not contain the symbol (precomputed) and *(d)* removing the infrequent symbols of the projected database only from the next domain D_{i+1}.

GapSeq **[KLL$^+$16] global constraint.** $GapSeq(P, SDB, \theta, M, N)$ is a global constraint for SPM problem under *gap*$^{[M,N]}$ which enforces filtering rules 1, 2, 3 and 4 in a single propagator with the limitation that gap constraints are expressed in terms of position distances i.e. the gap are measured according to the number of events hence time does not matter.

Our global constraint. PPICt$(P, SDB, \theta, M, N, W, Y)$ is a global constraint for SPM problem under *gap*$^{[M,N]}$ and *span*$^{[M,N]}$ over time-stamped databases which enforces filtering rules 1 to 5 in a single propagator. Hence, gap/span constraints are expressed in terms of time, but it can be used for position distances as well.

6.4 Trailing-based data structure for the embedding database

In this section, we introduce the notions of *embedding database* and *extension windows* which reconsider the concept of projected database to incorporate time constraints. Then, we present a trailing-based data structure to store this database.

6.4.1 Embedding database and extension windows

In fact, when having a gap constraint and using prefix-projection (see Definition 2.9), the assumption that a pattern can be extended with the symbol appearing after the *smallest* matching prefix does not hold anymore. That is, given a sequence, if the first embedding of the prefix cannot be extended because the gap is too small or large, there could exist another embedding that can be extended.

Example 6.8. Assume the pattern $\alpha = \langle ADC \rangle$ and $gap^{[3,7]}$. There are two embeddings of α in sid_3: $(1, 3, 7)$ and $(1, 4, 7)$. The first embedding is not a $gap^{[3,7]}$-embedding since $3 \leq (t_7 - t_3 = 8) \nleq 7)$ while the second one is (Figure 6.2c illustrates this).

Hence, it is not sufficient to store just the (suffix of the) smallest embedding as is done in the pseudo-projected database. Instead, we can store all embeddings. One can draw the parallel of this notion with the notion the pseudo-projected database $pSDB|_\alpha$ but instead of only storing the first embedding, we store all available embeddings:

Definition 6.5. ***Embedding database*** **(*$embSDB|_\alpha$*).** *Assume a sequence $s = \langle (s_1, t_1)(s_2, t_2) \dots (s_n, t_n) \rangle$ and a subsequence $\alpha = \langle \alpha_1 \alpha_2 \dots \alpha_k \rangle$ with $k \leq n$. The set of all embeddings of α in s is $emb_\alpha(s) = \{(e_1, \dots, e_k) | 1 \leq e_1 \leq e_k \leq n$ such that $s^s[e_i] = \alpha_i\}$. The embedding database of α is now defined as $embSDB|_\alpha = \{\, (sid, emb_\alpha(s)) \,|(sid, s) \in SDB\}$.*

The embedding database under $gap^{[M,N]}$ of a sequence s is the set of transactions identifiers together with all embeddings of s that satisfy the $gap^{[M,N]}$ constraint; denoted as $embSDB|_\alpha^{[M,N]}$. Similarly for the embedding database under $span^{[W,Y]}$ and the combination of gap and span.

GapSeq [KLL+16] stores for each embedding the position after the last embedding, called *right pattern extensions*, but this is not sufficient to support a *span* constraint. Our method will store the start and stop position of each embedding, which is sufficient for span, gap and the combination of the two.

Given a span and/or gap constraint, an embedding can only be extended with events whose timing satisfies the span/gap constraints. We call this subsequence of events the extension window of an embedding:

Definition 6.6. ***Extension Window (*ew*).*** *Assume a given sequence $s = \langle(s_1, t_1)(s_2, t_2)\dots(s_n, t_n)\rangle$, a subsequence $\alpha = \langle\alpha_1\alpha_2\dots\alpha_k\rangle$ and a gap$^{[M,N]}$ constraint. Let $e = (e_1, e_2, \dots, e_k)$ be any valid gap$^{[M,N]}$-embedding of α in s. The extension window of this embedding, denoted $\mathrm{ew}_e^{gap^{[M,N]}}(s)$, is the subsequence $\langle(s_u, t_u)(s_{u+1}, t_{u+1})\dots(s_{v-1}, t_{v-1})(s_v, t_v)\rangle$ such that $(t_{e_k} + M \le t_u) \wedge (t_v \le t_{e_k} + N) \wedge (\nexists t'_u \in s^t, t_{e_k} + M \le t'_u < t_u) \wedge (\nexists t'_v \in s^t, t_v < t'_v \le t_{e_k} + N\}$. The start and the end position of this extension window are respectively u and v.*

Example 6.9. Assume *gap*$^{[3,7]}$ and $\alpha = \langle A\rangle$. For *sid*$_3$ in Table 6.1a, there is one *gap*$^{[3,7]}$-embedding: (1) with extension window $\langle(D, 6)(D, 8)\rangle$; hence, if $\langle A\rangle$ is extended with any symbol other than D it will no longer be covered. For $\alpha = \langle A, D\rangle$ there are now two possible embeddings: $(1, 3)$ and $(1, 4)$. Their extension windows are: $\langle(B, 10)(E, 12)\rangle$,$\langle(E, 12)(C, 14)\rangle$. Figure 6.1b shows the embeddings and extension windows for these two patterns for all sequences in the SDB of Table 6.1a. A comparison can be made with the same versions without time restrictions presented in Figure 6.1a.

6.4.2 Trailing-based data structures

Trailing as a mechanism to restore the state. CP-Solvers implementing the depth-first search backtracking algorithms need an efficient state restoration system [Knu15]. The interested reader can refer to Section 2.2.2 of the background chapter or the papers [Knu15,Lau18] for a comprehensive introduction. Our trailing-based data structure also uses this mechanism.

Trailing the embedding database. We introduce a trailing-based data structure to efficiently store and restore the embedding database. The key idea is to store the embedding database in 'backtracking aware' vectors. This data structure briefly described here, is the one presented in the (previous) chapter 5.

This idea was introduced in *PPIC* [AGS16] allowing to drastically speeding up the search for sequential patterns without time constraints. See an illustration of this data structure based on the projected database examples of the Figure 6.1a in the Figure 6.3a. Two reversible integers store respectively the start position in the vector (ϕ) and the number of entries (φ) in the embedding database. When branching, data is appended at the $\phi + \varphi$ position and ϕ and φ are updated. When backtracking, only the start position and number of elements need to be restored/trailed; the vector can stay unchanged in memory, with the parts after $\phi + \varphi$ overwritten later. In [AGS16], only the sequence ids (*sids* vector) and the start position of the suffixes (*embs* vector) must be stored[1]. This is not sufficient to handle time constraints information.

Trailing-based embedding database. We use reversible vectors to store the start and the end positions of all the possible time-constrained embeddings for every sequence. The vectors: *sids*, emb_{size} and *embs* respectively represent the sequence ids, the number of embeddings and the start/end positions of the embeddings. These start and end positions are sufficient to verify the span and gap constraints during pattern extension.

Example 6.10. Figure 6.3b depicts an example of this data structure: for pattern $\langle A\rangle$, the data of $embSDB|^{[3,7]}_{\langle A\rangle}$ is stored between indices $\phi = 1$ and $\phi + \varphi - 1 = 4$. This pattern $\langle A\rangle$ is then further extended by the symbol D and $embSDB|^{[3,7]}_{\langle AD\rangle}$ is stacked next to it, between indices $\phi = 5$ and $\phi + \varphi - 1 = 7$. This pattern $\langle AD\rangle$ is then further extended by the symbol C and $embSDB|^{[3,7]}_{\langle ADC\rangle}$ is stacked between indices $\phi = 8$ and $\phi + \varphi - 1 = 10$. The $gap^{[3,7]}$-embedding of $\langle ADC\rangle$ in sid_2 is $(3, 4, 5)$ but we only store the start (3) and end (5) as $(3, 5)$ we can compute the valid extension window based on gap and span constraint.

6.5 *PPICt* global constraint under time constraints

This section presents *PPICt* (Prefix Projection Incremental Counting with time restrictions), our filtering algorithm for finding sequential patterns under gap and span constraints. This algorithm support sequences with timestamps (as presented in Table 6.1a). Constraints such as regular expression, item inclusion/exclusion, pattern length and all other constraints that do not depend on the embeddings can be added to the model. Constraints that can change what a valid embedding is would need to be added to the propagator, as we do for the time-based gap and span constraints.

6.5.1 PPICt filtering algorithm and improvements

The PPICt$(P, SDB, \theta, M, N, W, Y)$ global constraint is given in Algorithm 6.1. The filtering procedure is triggered whenever a pattern is extended by a new symbol

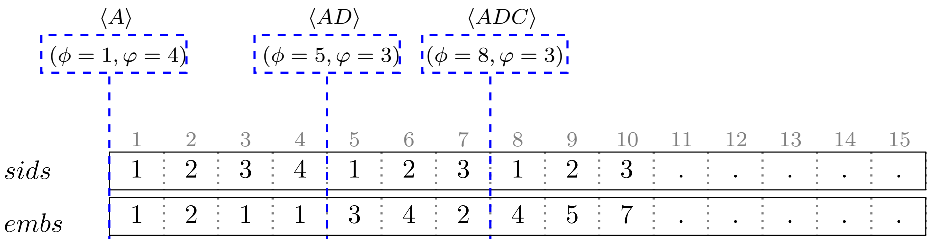
[1] We discussed this in the Chapter 5.

(P_{i+1}). If that symbol is ϵ then by Rule 2 all $P_j, j > i$ should also be ϵ. The pattern is hence fully assigned and so we can filter with the minimum span constraint which is not prefix-monotone. If the pattern is still frequent then this is a new solution.

If P_{i+1} was assigned a *non-*ϵ value, then the procedure ProjectAndGetFreqs counts for each symbol what the size of the projected database would be if the new pattern is extended with that symbol. The computed result, denoted $freqs$ in the pseudo-code, is used to prune the domain of the next pattern variable P_{i+1} by removing infrequent symbols; this is valid because the constraints filtered in ProjectAndGetFreqs are *prefix anti-monotone*.

The main difference with *PPIC* [AGS16] is that all embeddings must be stored instead of just the prefix (see Section 6.4.1). Embeddings can only be extended by symbols appearing in its *extension windows*. The projected frequency counting

(a) Trailing-based data structure for SPM problem without time constraints - in *PPIC*



(b) Trailing-based data structure for SPM problem with gap/span constraints - in *PPICt*

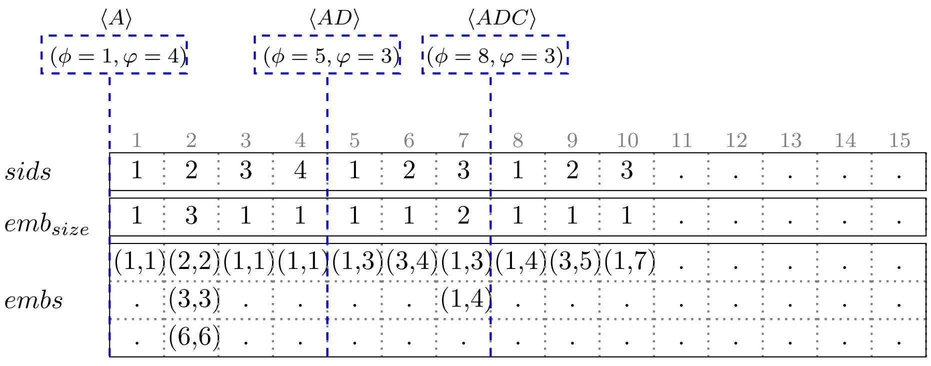


Figure 6.3. Trailing-based data structure to store and restore the embeddings database: **a)** without time constraints **b)** with $gap^{[3,7]}$ constraints

Algorithm 6.1: PPICt($P, SDB, \theta, M, N, W, Y$)

```
▷ Input: $P$ is the list of decision variables;
         $SDB$ is the sequence database ;
         $\theta$ is the support threshold;
         $M$ and $N$ are the gap min and max bound;
         $W$ and $Y$ are the span min and max bound.
▷ Pre: variables $\langle P_1, \ldots, P_i \rangle_{i \in [1..l]}$ are bound
         $P_i$ is the new assigned variable since previous call (let's $P_i = a$).

if $a = \varepsilon$ then
    for $j \in \{ i+1, \cdots, l \}$ do
        $P_j.assign(\varepsilon)$                                    ▷ Filtering rule.2
    Remove all embeddings that do not satisfy minimum span $W$ and fail if the
      pattern should no longer be frequent
else
    $freqs$ = ProjectAndGetFreqs($i, SDB, a, M, N, W, Y$)     ▷ Detailed in Alg. 6.2.
    foreach $b \in D(P_{i+1})$ do
        if $b \neq 0 \wedge freq[b] < \theta$ then
            $P_{i+1}.remove(b)$
```

should only count symbols appearing in an extension window. Indeed, a symbol not appearing in an extension window of any embedding of the sequence would not be a valid support for extending the current pattern as it would not satisfy the time constraints.

This leads to the following key ingredients of the ProjectAndGetFreqs function: 1) as presented in Section 6.4.2, we adapt the backtracking-aware data structure introduced in [AGS16] to store all possible occurrences of the pattern in a sequence, including the starting symbol to support *span* constraints; 2) we avoid scanning a sequence for a symbol beyond the (precomputed) last occurrence of that symbol in the sequence; 3) we introduce the concept of *extension window* of an embedding and avoid to scan overlapping windows multiple times; 4) we avoid searching for the position of the start of an extension window, which depends on the minimum gap time, by precomputing these position in advance. These ingredients are detailed next.

Ingredient 1. Avoid scanning all sequences

We reuse the $lastPosMap$ precomputed structure of $PPIC$ to avoid scanning a sequence if the last position of that sequence is before the start of the extension window. For a symbol a, the $lastPosMap[a]$ is the last position of this symbol in the sequence: $lastPosMap[a] = \max\{p \leq \text{size}(s) : s[p] = a\}$.

Example 6.11. Assuming the $lastPosMap$ precomputed structure provided in Table 6.1c and the symbol A, $lastPosMap[A]$ is $\{1, 6, 1, 1\}$. Hence, when searching for A, we must stop at the first position for the sequences sid_1, sid_3 and sid_4 but for the sequence sid_2 we stop at position 6.

However, we cannot use the same structure for support counting (which also need to search symbols over sequences) as *PPIC* did, since this assumes that all symbols up to the end of the sequence must be counted, while we should only count symbols in the extension windows. This can have a significant impact if the sequences contain many duplicates symbols as shown in [AGS16]. In our case, this problem is minimised since extension windows are often smaller.

Ingredient 2. Avoid scanning more than once the events occurring in overlapping extension windows

The extension windows of a sequence can overlap. For instance in Figure 6.1b with $\alpha = \langle AD \rangle$, in sid_3, $\langle (E, 12) \rangle$ is present in both extension windows. Then when computing the $freqs$ vector, some positions could be revisited several times. This source of inefficiency can be avoided by keeping track of the current largest position visited so far in any extension window. This position is denoted *pos* in Algorithms 6.2 and 6.3. When the next extension window for the current sequence is considered by updateSupport in Algorithm 6.3, all symbols before *pos* have already been counted, so only positions after *pos* should be visited and afterwards *pos* is updated.

Ingredient 3. Avoid scanning the sequences when given a minimum gap

Given the current pattern $\alpha = \langle \alpha_1 \alpha_2 \dots \alpha_k \rangle$, a sequence s and a valid $gap^{[M,N]}span^{[W,Y]}$-embedding $e = (e_1, e_2, \dots, e_k)$, all the symbols in the extension window $\text{ew}_e^{gap^{[M,N]}span^{[W,Y]}}(s)$ must be visited for updating the frequency counters. While it is easy to compute the start *time* of the extension window using the minimum gap M: $t_{e_k} + M$; finding the first position u in the sequence such that $t_u \geq t_{e_k} + M$ requires scanning the sequence starting from e_k. To avoid this, we propose to precompute, for the given minimum gap, the position of the beginning of the extension window from any possible position. This can be done with one linear scan over each sequence when the propagator is initialised; and the precomputed positions are stored in a structure called *nextPosGap*.

Definition 6.7. ***Building the* nextPosGap *structure.*** *Assume $s = \langle (s_1, t_1)(s_2, t_2) \dots (s_n, t_n) \rangle$ is a sequence. Given $k \in [1 \,..\, n]$ a position in s and M a minimum gap, the nextPosGap$[s][k]$ is the position of the* smallest time *satisfying the minimum gap: nextPosGap$[s][k] = i$ such that $(i > k) \wedge (t_i \geq t_k + M) \wedge (\nexists i' < i : t_{i'} \geq t_k + M)$.*

Example 6.12. Assume $s = \langle (A,2)(B,5),(D,6),(C,10)(B,11) \rangle$, $k = 2$ and $M = 3$ $nextPosGap[s][k] = 4$ because $t_4 = 10$ is the smallest time such that $t_4 \geq 5 + 3 = 8$. Table 6.1b shows the *nextPosGap* of SDB (the values $nextPosGap[s][k] >$ size$(s) + 1$ means the minimum gap is not available for that position).

Putting it all together

The core of the algorithm is in the ProjectAndGetFreqs procedure (presented in Algorithm 6.2) that gathers all the ingredients. We distinguish two cases. Assuming $i \in [1 \mathrel{..} l]$, if $i == 1$ it means that the pattern was previously empty and is now composed of one unique symbol. If $(i > 1)$ the pattern is composed of at least two symbols which means that the gap/span must be considered.

In the first case $(i == 1)$, all sequences of SDB are considered. For every sequence, all the positions having the symbol a are stored as an embedding. As the embedding is a singleton, there is not need to consider the gap/span constraints at this point.

The call to updateSupport (Algorithm 6.3) will update the $freqs$ vector by visiting each symbol present in the extension window of the current embedding (position of symbol a). Variable *pos* is used to avoid incrementing the frequency of a symbol twice in the same sequence.

In the second case $(i > 1)$, the main loop at line 21 iterates over the previous *(parent)* projected database stored between ϕ and $\phi + \varphi - 1$ and builds the new one starting at index $\phi + \varphi$. For each embedding of a sequence, s (line 24), the maximum of the time window is computed. We search all positions only in extension window ensuring to have time greater than minimum gap time and lower than the maximum gap and span times computed based on the first and the last element of the embedding (line 26). The updateSupport is also called to update the $freqs$ vector for every extended embedding created.

Finally, lines 19 and 35 update *sids* and emb_{size} in order to ensure the consistency of the data structure. Then, line 36 updates the reversible integers ϕ and φ to reflect the newly computed projected database.

6.5.2 Additional constraints

The advantage of CP based sequence mining is its capacity to accept additional constraints. The global constraint approach *PPICt* is less flexible than the decomposition approach of [NG15] as it does not expose the embeddings. Nevertheless many useful syntax constraints [HFPZ13] can be added on the sequence pattern variables:

Algorithm 6.2: ProjectAndGetFreqs(i, SDB, a, M, N, W, Y)

```
▷ Input: $i$ is the position in $P$ representing the current bound variable;
         $SDB$ is the sequence database ;
         $M$ and $N$ are the gap min and max bound;
         $W$ and $Y$ are the span min and max bound.
▷ Output: $freq$ list of projected frequencies.
▷ Internal State: $\phi$, $\varphi$, $sids$, $emb_{size}$, $embs$.

$\phi' \leftarrow \phi + \varphi$   $\varphi' \leftarrow 0$   $freqs[b] \leftarrow 0\ \forall b \in \mathcal{I}$
if $i = 1$ then ▷ first assigned symbol, scan for symbol
    for $sid = 1$ to size($SDB$) do                          ▷ for every sequence in SDB
        $seq \leftarrow$ SDB[$sid$]   $nEmb \leftarrow 0$   $pos \leftarrow 0$
        $visitedI[b] \leftarrow false\, \forall b \in \mathcal{I}$
        for $j \leftarrow 0$ to $lastPosMap[sid][a]$ do                ▷ find each symbol a
            if $seq[j] = a$ then                                  ▷ new match
                $embs[\phi' + \varphi'][nEmb] \leftarrow (j, j)$   $nEmb \leftarrow nEmb + 1$
                $pos \leftarrow$ updateSupport($j, j, sid, pos, visitedI$)   ▷ See Alg. 6.3.
                if $pos \geq$ size($seq$) then break             ▷ window ends with seq
        if $nEmb > 0$ then                                   ▷ store sequence meta-data
            $sids[\phi' + \varphi'] \leftarrow sid$   $emb_{size}[\phi' + \varphi'] \leftarrow nEmb$   $\varphi' \leftarrow \varphi' + 1$
else ▷ non-empty prefix
    for $c \leftarrow \phi$ to $\phi + \varphi - 1$ do                ▷ for all seq in projected database
        $sid \leftarrow sids[c]$   $seq \leftarrow$ SDB[$sid$]   $nEmb \leftarrow 0$   $pos \leftarrow 0$
        $visitedI[b] \leftarrow false\, \forall b \in \mathcal{I}$
        for $k \leftarrow 1$ to $emb_{size}[c]$ do                    ▷ for each prefix embedding
            $(b, e) \leftarrow embs[c][k]$               ▷ begin and end position of embedding
            $maxT \leftarrow min(seq^t[sid][b] + Y, seq^t[sid][e] + N)$  ▷ max time window
            $j \leftarrow nextPosGap[sid][e]$                ▷ precomputed position of minT
            while $j < lastPosMap[sid][a] \wedge seq^t[sid][j] \leq maxT$ do
                if $seq[j] = a$ then                              ▷ new embedding
                    $embs[\phi' + \varphi'][nEmb] \leftarrow (b, j)$   $nEmb \leftarrow nEmb + 1$
                    $pos \leftarrow$ updateSupport($b, j, sid, pos, visitedI$)  ▷ See Alg. 6.3.
                    if $pos \geq$ size($seq$) then break          ▷ window ends with seq
                $j \leftarrow j + 1$
        if $nEmb > 0$ then                                   ▷ store sequence meta-data
            $sids[\phi' + \varphi'] \leftarrow sid$   $emb_{size}[\phi' + \varphi'] \leftarrow nEmb$   $\varphi' \leftarrow \varphi' + 1$
$\phi \leftarrow \phi'$   $\varphi \leftarrow \varphi'$
return $freqs$
```

Algorithm 6.3: updateSupport($b, e, sid, pos, visitedI$)

```
▷ Input: $sid$ is the sequence id;
         $b$ and $e$ are the beginning and end of the time window in $sid$;
         $pos$ is the current position in $sid$;
         $visitedI$ is the list of the status of items.
▷ Output: $k$ is the new position in $sid$ to consider.

$s \leftarrow$SDB[$sid$]    $k \leftarrow max(nextPosGap[sid][e], pos)$
$maxT \leftarrow min(s^t[sid][e] + N, s^t[sid][b] + Y)$
while $k <$ size($s$) $\wedge s^t[sid][k] \leq maxT$ do
    if $\neg$ $visitedI[s[k]]$ then
        $freqs[s[k]] \leftarrow freqs[s[k]] + 1$    visitedI[$s[k]$]$\leftarrow true$
    $k \leftarrow k + 1$
return $k$
```

$P = [P_1, P_2, \ldots, P_l]$. Note that these constraints are directly[2] posted as independent constraints in the solver constraint store and run together with the mining process. We can indeed post any such constraint, for example:

Pattern length constraints. One can impose a minimum and a maximum over the length of the pattern. These constraints are easy to handle considering all patterns are terminated by the empty symbol (ϵ). Hence, the minimum pattern length ($Lmin$) is defined as $\forall i \in [1 \,..\, Lmin]$ and $Lmin < l$, $P_i \neq \epsilon$. The Maximum pattern length ($Lmax$) is obtained by limiting the length of P to $Lmax$: $P = [P_1, P_2, \ldots, P_{Lmax}]$ with $Lmax < l$.

Symbol inclusion/exclusion. The number of occurrences of symbols in the sequence pattern can be modeled with *Among* [BC94] and global cardinality (GCC) [Rég96] constraints largely available in CP-Solvers. Considering $v \geq 0$ a given number of occurrences of symbol $i \in \mathcal{I}$, the $Among(P, i, v)$ constraint will only allow a number v times i ($v = 0$ prohibits i). To handle several symbols inclusion/exclusion, one can use several *Among* constraints or $GCC(P, \mathcal{V})$ constraint with $\mathcal{V}$ a collection of pairs (v, i). GCC also offers the possibility of defining range of values of occurrences.

Regular expression and grammar. The *Regular* global constraint [Pes04] can be used to enforce that P satisfies a given regular expression. Most of CP-based approaches [AGS16,KLL$^+$16,KLL$^+$15] hence easily support regular expression constraints. Considering e a user-supplied regular expression and $\mathcal{A}$ its deterministic finite automaton, the $Regular(P, \mathcal{A})$ constraint only allows patterns matching $\mathcal{A}$. Since a regular expression is a formal language, the grammar constraint can also be used. However, grammar constraint could also use to define context-free languages [KS10] or general languages [QW06].

[2]It is neither post-processing nor hard-coded

6.5.3 Time and space complexity

Let us denote by $L = \text{size}(SDB)$ the number of sequences, l the length of the longest sequence, $m = \text{size}(\mathcal{I})$ the size of item alphabet. In the worst case, the time and space complexities of our propagator is in $\mathcal{O}(L \times l^2)$.

Space complexity. Our data structure needs $\mathcal{O}(l)$ memory entries to store the embeddings for one sequence of length one. The projected database for such a sequence thus requires $\mathcal{O}(L \times l)$. During search, the search will branch over each sequence variable in turn which corresponds to extending the pattern by one symbol at a time. Each time a symbol is added, an extra layer of embeddings is stored in our trailing-based data structure (Figure 6.3). Given that each pattern has a length of at most l, the space complexity of our data structure is hence $\mathcal{O}(l \times L \times l) = \mathcal{O}(L \times l^2)$. $\square$

Time complexity. PPICt needs $\mathcal{O}(l + (L \times l^2) + m) = \mathcal{O}(L \times l^2 + m)$ time to be complete since the lines 10-11 of the Algorithm 6.1 cost $\mathcal{O}(l)$, the ProjectAndGetFreqs method $\mathcal{O}(L \times l^2)$ and line 15 costs $\mathcal{O}(m)$. The complexity of the loops 13 and 28 including the updateSupport in Algorithm 6.2 is $\mathcal{O}(l)$ since we avoid scanning overlapping symbols. Hence, the complexity of the ProjectAndGetFreqs method is $\mathcal{O}(L \times l + L \times l \times l) = \mathcal{O}(L \times l^2)$. $\square$

6.6 Implementation and Practical User Guide

All our software, datasets and results are available online as open source in order to make this research reproducible (http://sites.uclouvain.be/cp4dm/).

The implementation of *PPICt* is available in the CP-Solver OscaR [Osc12] which is available online[3] in free access. In OscaR, one can combine our constraints with the existing constraints in the solver such as All-Different [Rég94], Global cardinality [QLvBG04], Grammar [QW06], etc.

For developers who are willing to modify the code directly, a lightweight version is also available[4]. One can hence add several constraints for a specific usage without understanding OscaR deeply. The installation procedure is described in the Install file in the code directory and merely consists of "importing the project" into your favourite IDE.

[3] https://bitbucket.org/oscarlib/oscar/wiki/Home
[4] https://bitbucket.org/projetsJOHN/cp4dt

Users can directly download the *jar-file* which is available on our website[5]. On this website, there is a user guide. The general format of the general command is

```
java -jar ppict.jar [options] <SDB File> <Lmin > < Lmax>
```

with several options available, including:

- $-t$: to indicate that the database is provided with timestamps.
- $-f$: for the frequency constraint (i.e. θ),
- $-i$: for items inclusion/exclusion constraints,
- $-e/-r$: for the regular expression constraint,
- $-m$: for the minimum gap constraint,
- $-n$: for the maximum gap constraint,
- $-w$: for the maximum span constraint,
- $-y$: for the maximum span constraint,
- etc.

Given a sequence database with timestamps (named test.txt), here are some examples of command:

```
[Cmd 1]
java -jar ppict.jar test.txt 1 0 -f 2 -t -m 3 -n 7
[Cmd 2]
java -jar ppict.jar test.txt 1 0 -f 2 -t -m 3 -n 7 -y 3
[Cmd 3]
java -jar ppict.jar test.txt 2 0 -f 2 -t -m 3 -n 7 -r "A+B*" A=20,B=30
```

The test-file input and the outputs of those commands are:

[5] https://sites.uclouvain.be/cp4dm/spm/ppict/

```
                  Input                                    Output
<sid> <timestamps> <size_of_itemset> <itemset>             <sup> : <patern>
1 2 1 10                                                   [output 1]
1 5 1 20                                                   4 : < 10  >
1 6 1 40                                                   3 : < 10 20  >
1 10 1 30                                                  3 : < 10 40  >
1 11 1 20                                                  3 : < 10 40 30  >
2 1 1 20                                                   4 : < 20  >
2 2 1 10                                                   2 : < 20 20  >
2 9 1 10                                                   2 : < 20 30  >
2 12 1 40                                                  3 : < 40  >
2 15 1 30                                                  2 : < 40 20  >
2 18 1 10                                                  3 : < 40 30  >
2 24 1 20                                                  4 : < 30  >
3 2 1 10
3 4 1 20                                                   [output 2]
3 6 1 40                                                   4 : < 10  >
3 8 1 40                                                   2 : < 10 20  >
3 10 1 20                                                  4 : < 20  >
3 12 1 50                                                  3 : < 40  >
3 14 1 30                                                  4 : < 30  >
4 1 1 10
4 2 1 30                                                   [output 3]
4 3 1 30                                                   2 : < 20 20  >
4 4 1 20                                                   2 : < 20 30  >
```

6.7 Experiments

This section reports the experiments we made to evaluate our approach in comparison with other CP-based and specialized methods. More specially, we answer the following questions: *Q1.* What is the performance of the state-of-the-art for sequential pattern mining without time constraints? *Q2.* What is the difference in performance of *PPICt* for sequential patterns mining with time restrictions? *Q3.* What is the effect of a standalone constraint in the mining process? *Q4.* What is the impact of the computation of the additional embeddings in *PPICt*?

Before answering these questions, we present in Table 6.2 the features of the seven real-life datasets that we use, as well as the experimental protocol and the alternative sequential pattern miners used for the comparisons. Note that the data and framework are available online and open source[6].

[6] http://sites.uclouvain.be/cp4dm/spm/ppict/

SDB	size($\mathcal{I}$)	size(SDB)	$allsymbols(SDB)$	max size(s)	$avg.$ size(s)	density
BIBLE	13905	36369	787066	100	21.64	0.0016
FIFA	2990	20450	741092	100	36.24	0.0121
Kosarak	21144	69999	558373	796	7.98	0.0004
LEVIATHAN	9025	5834	197251	100	33.81	0.0037
MSNBC	17	31790	423776	100	13.33	0.7841
PubMed	19931	17237	509440	198	29.56	0.0015
protein	25	103120	49729890	600	482.25	19.2901

Table 6.2. Seven real-life datasets features. Respectively: dataset name, number of distinct symbols, number of sequences, total number of symbols in the dataset, maximum sequence length, average sequence length, and density computed by $\frac{allsymbols(SDB)}{\text{size}(\mathcal{I})\times\text{size}(SDB)}$ (Kosarak is the sparsest dataset and Protein is the densest).

Experimental protocol

PPICt is implemented in the Scala language in the CP-Solver OscaR [Osc12]. All experiments are run in the JVM with maximum memory set to 8GB. All the experiments are conducted using a 2.7GHz Intel Core i5 64 bit processor and 8GB of RAM with Linux 3.19.0-32-generic from Mint 17.3. We set the execution time limit to 3600 seconds (1 hour). We also restrict the output of all software to only the mining statistics and do not print the patterns found. $Minsup$ denotes the minimum support θ.

Alternative sequential patterns miners

We make comparisons with *GapSeq*[7] [KLL+16], a CP approach that outperforms other CP-based methods supporting gap constraints; *cSPADE*[8] [Zak00] a highly scalable specialized sequence miner that supports gap and span constraints. Its search is not based on pattern extension as *GapSeq* and *PPICt* are, but on repeated (temporal) joins of embeddings. We also provide a comparison to *PPIC*[9] [AGS16] without gap constraints, *PPIC* has shown to outperform both specialized and generic miners for standard frequent sequence mining. Table 6.3 shows the supported constraints for these miners.

[7] https://sites.google.com/site/cp4spm/
[8] http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software
[9] http://sites.uclouvain.be/cp4dm/spm/

Methods	Frequency	Gap	Span	Regular	Among	Length	other constraints[1]
PPICt	x	x	x	x	x	x	x
GapSeq	x	x*			x	x	x
cSPADE	x	x	x**			x	

Table 6.3. Sequential pattern miners with supported constraints. [1] is any other user-supplied constraint that does not depend on the embeddings (not implemented but could be). * *GapSeq* does not consider time but position of events, ** *cSPADE* does not support minimum span constraint.

6.7.1 Performances results

Q1: *GapSeq* vs *PPIC* vs *cSPADE* for SPM without time restriction

As shown in [AGS16] and illustrated in Figure 6.4, *PPIC* clearly outperforms both CP-based and specialized approaches for many datasets (with several different features) except for the sparsest dataset Kosarak-70k where it is competitive with *cSPADE*. The protein dataset is large and dense[10] with many patterns even at high support. For such a challenging dataset, *PPIC* is at least one hundred times faster.

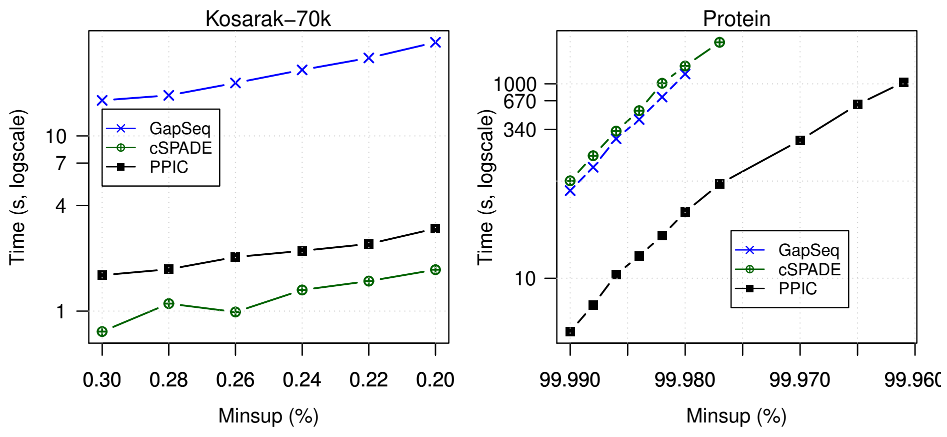


Figure 6.4. CPU times for PPIC (without time constraints) with several minsup (missing points indicate a timeout). Find more experiments in [AGS16] or Chapter 5.

[10] A dataset is dense if its density, computed by $allsymbols(SDB)/\big(\text{size}(\mathcal{I})\times\text{size}(SDB)\big)$, is closed to 1.

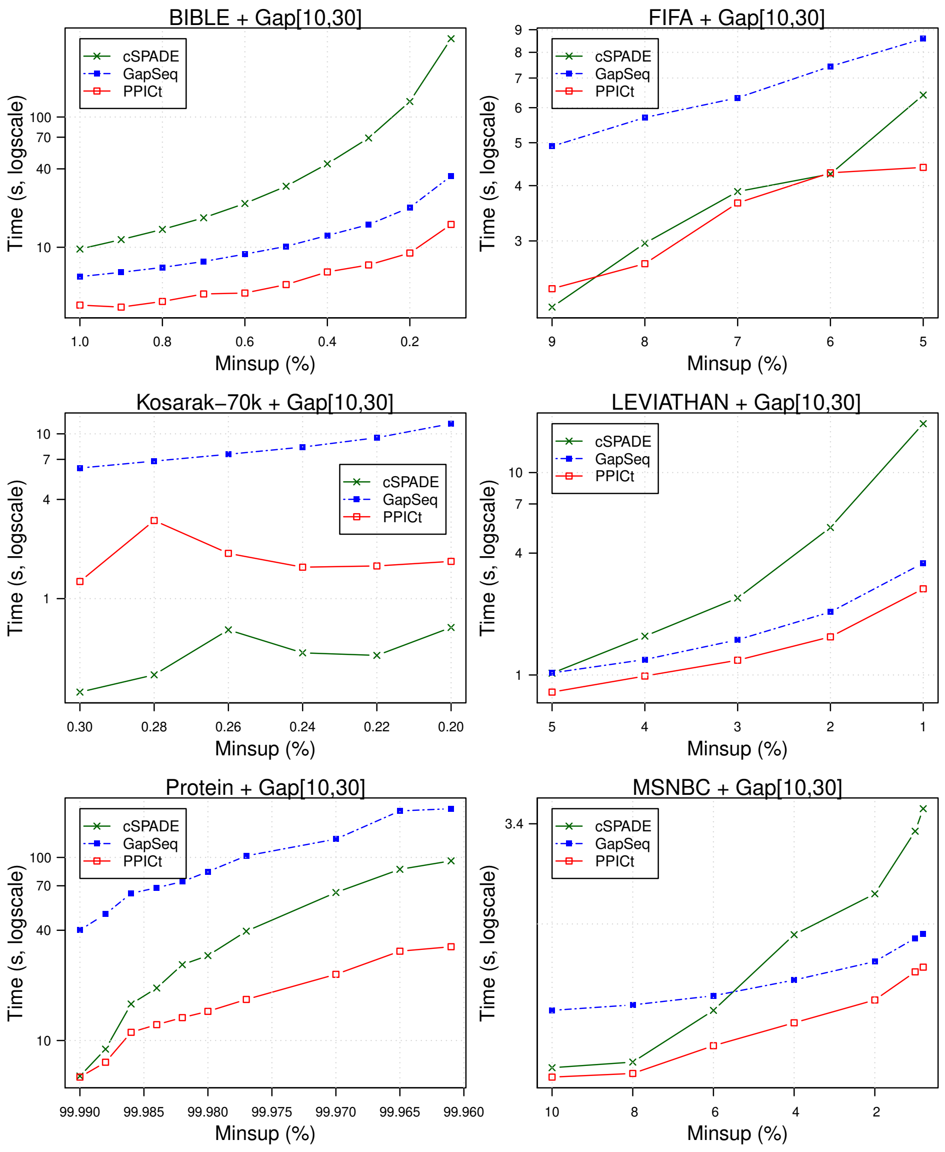


Figure 6.5. CPU times when considering minimum and maximum gap constraints for several minsup (missing points indicate a timeout)

Q2: Time performance for *PPICt* under *gap* and *span* constraints

We first compare *PPICt* with *GapSeq* and *PPIC* for gap constraints. Then, we combine gap and span constraints.

Figure 6.5 shows the CPU time for the sequence mining task under minimum and maximum gap for several θ ($Minsup$) values over six datasets. *PPICt* dominates both CP-based and specialized methods. Except for the Kosarak dataset, PPICt is often faster, and increasingly so for low-frequency thresholds. Upon inspecting the output of the Kosarak dataset we see that several frequent patterns have the same size and cover the same set of sequences. The temporal join approach used by *cSPADE* is very fast in this case. This was also the case for *PPIC* in the non-time constrained case.

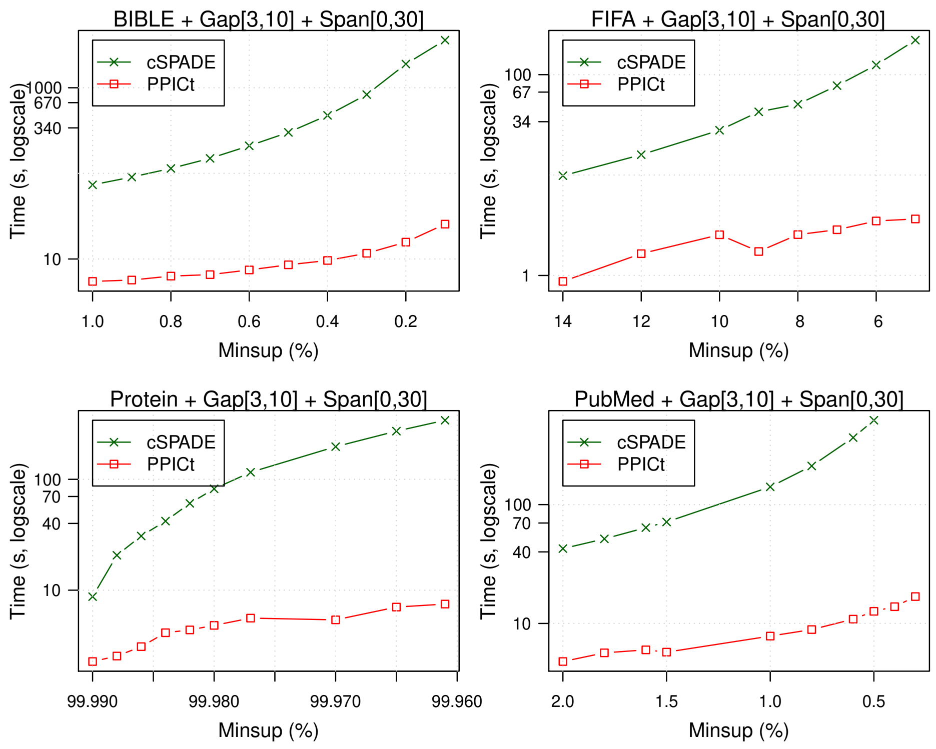


Figure 6.6. CPU times when considering both *gap* and *span* constraints for several minsup (missing points indicate a timeout).

We also combine the *gap* and *span* constraints, which is not supported by GapSeq. The results are presented in Figure 6.6. Our approach outperforms *cSPADE* by a wide margin in this case. These results show that *PPICt* is still efficient when combining time constraints.

Q3: Effect of maximum gap constraint

We now look at the sensitivity of the methods to the threshold of the maximum gap constraint. We fix the frequency threshold to a low value that makes mining without further constraints challenging and increase the maximum gap constraint from 1 to 9. As can be seen in Figure 6.7, the runtime of *cSPADE* increases much more quickly with increasing maximum gap. For *GapSeq* it depends on the dataset, but *PPICt*'s performance is more stable and increases more moderately compared to the other methods.

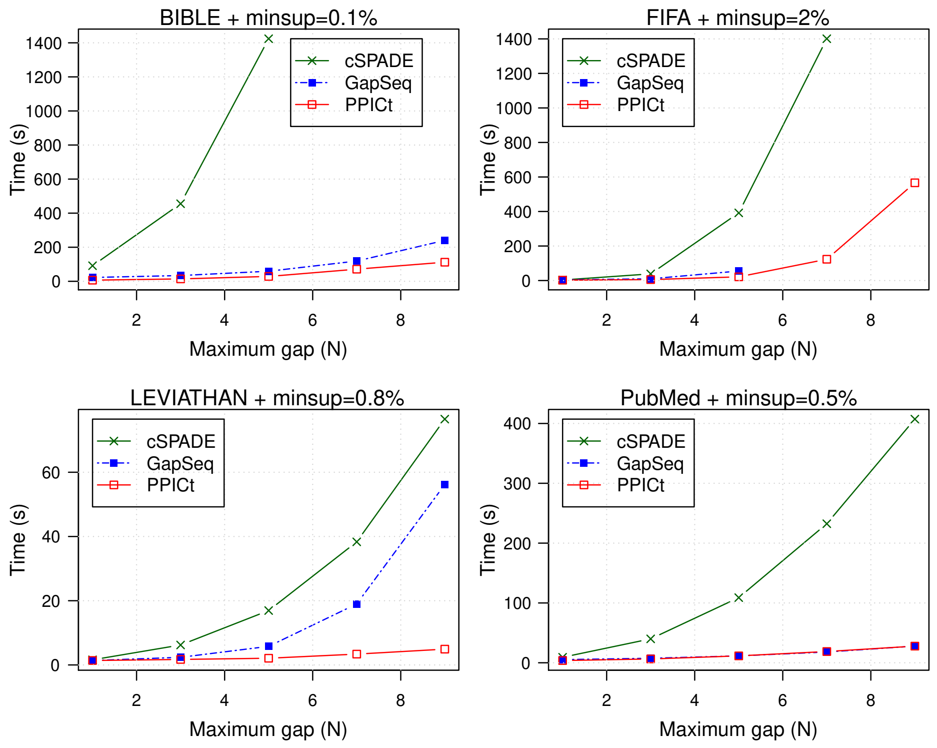


Figure 6.7. CPU times for several maximum gap with fixed minsup over Bible, Fifa, Leviathan and PubMed datasets (missing points indicate a timeout).

Q4: Experiments over databases without time restrictions

To answer *Q4.*, we use *PPICt* to find only the sequential patterns without any time considerations. That is *PPICt* where minimum gap/span is 0 and maximum gap/span is the infinity, denoted by *PPICt[0,Inf]*. Hence, we compare *PPIC* with *PPICt[0,Inf]*. The results are reported in Figure 6.8. We can notice that *PPIC* is always faster. This is possible since such *PPIC* improvements could not be used under time restrictions. Moreover, to preserve the structure of datasets the reduction of datasets by preprocessing is forbidden.

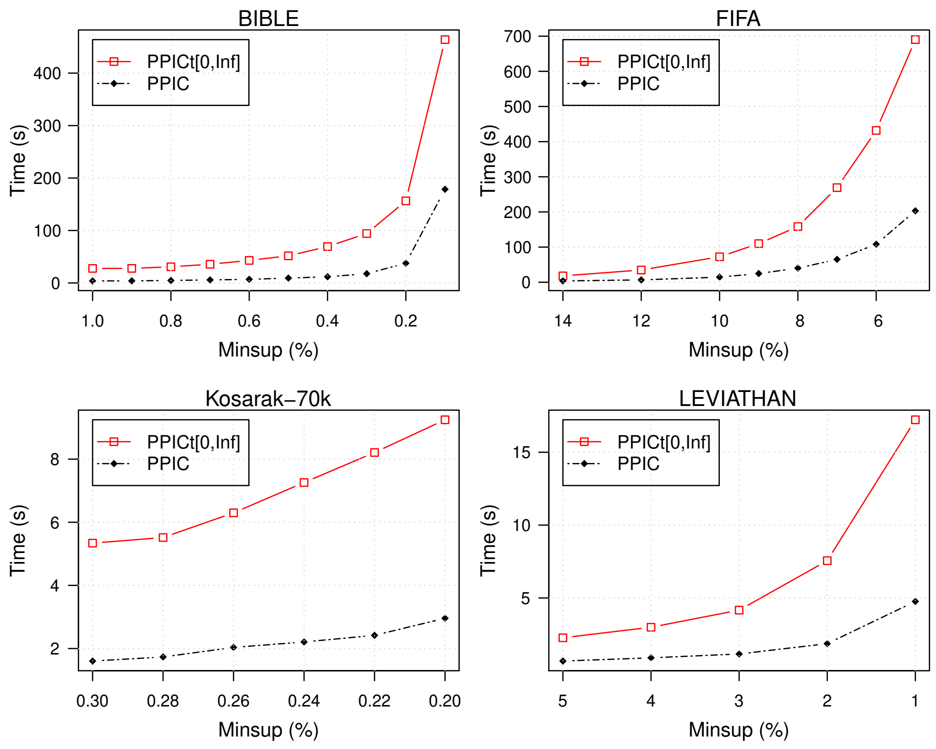


Figure 6.8. Comparing *PPICt* without time restriction (*PPICt[0,Inf]*) with *PPIC*

6.7.2 Handling additional Constraints

To demonstrate the ability to accommodate additional constraints we experiment the combination of *PPICt* with some other syntax constraints. The result is shown in Table 6.4; the constraint parameters were artificially constructed in an interactive setting. We can observe that the addition of the constraints reduces the number of solutions and the computation time. A *generate-and-filter* approach using a specialized algorithm would not be able to benefit from stronger filtering.

	Gap		+Pattern Length		+Among		+Regular	
	nSols	time(s)	nSols	time(s)	nSols	time(s)	nSols	time(s)
BIBLE*	32307	46.181	1542	45.622	171	43.390	8	0.191
PubMed**	13086	22.632	1304	21.600	235	19.889	3	0.091

* $\theta = 0.1\% \wedge Gap[10, 30] \wedge (Lmin = Lmax = 5) \wedge$ the number of A equal to 1 $\wedge$ E is forbidden $\wedge$ Regular(`A+(B{2,}|C*|D+)B*C*D*`) where ($A = 11829$, $B = 2$, $C = 8212$, $D = 6556$, $E = 5590$)

** $\theta = 0.3\% \wedge Gap[10, 30] \wedge (Lmin = Lmax = 4) \wedge$ the number of A and B equal to 1 $\wedge$ Regular(`B+A*C*A*`) where ($A = 3335$, $B = 12155$, $C = 16599$)

Table 6.4. Combination of pattern length, item inclusion/exclusion, regular expression constraints with gap constraint.

6.8 Summary, Outlooks, Further readings

We introduced *PPICt*, a global constraint to solve sequential pattern mining problem under time constraints. It integrates *gap* and *span* constraints for databases with or without timestamps. Our approach often outperforms *cSPADE*, the state-of-the-art specialized method and always outperforms *GapSeq*, the state-of-the-art CP based approach allowing to handle time constraints. This was made possible thanks to the backtracking-aware data structure to store embeddings of pattern based on trailing techniques. Also, algorithmic ingredients help to improve further: the precomputed next position of minimum gap, the avoidance of scanning all dataset and the avoidance of the overlapping between extension windows when computing the frequencies of symbols. Moreover, we report experimental results over several real-life datasets which demonstrate that our proposal is mostly competitive with or outperforms both specialized and CP-based methods. Additional constraints such as regular expression, item inclusion/exclusion, pattern length constraints are also available to increase the flexibility of users and practitioners.

Chapter 7

Frequent Episode Mining using Constraint Programming

"A teacher is one who makes himself progressively unnecessary.."

–Thomas Carruthers

Overview *The number of applications generating sequential data is exploding. This work studies the discovering of frequent patterns in a large sequence of events, possibly time-stamped. This problem is known as the Frequent Episode Mining (FEM). Similarly to the mining problems recently tackled by Constraint Programming (CP) in Chapters 5 and 6, FEM would also benefit from the flexibility offered by CP to accommodate easily additional constraints on the patterns. These advantages do not offer a guarantee of efficiency. Therefore, we introduce two global constraints for solving FEM problems with or without time consideration.*

Contribution *The contribution of this work is a flexible and efficient approach for solving the frequent episode mining problem, which use an implicit decomposition having a $\mathcal{O}(n)$ spatial complexity. This is impressive because spatial complexity is often a main issue for existing FEM methods. To achieve this, we adapt the backtracking-aware data structure and we are also able to take some algorithmic advantages in the filtering algorithms using the property that the (implicit) database is composed of sorted suffixes from a same sequence.*

Main source *This chapter is mainly based on our paper [CAS18].*

7.1 Context and Motivation

The trend in data science is to automate the data-analysis as much as possible. Examples are the *Automating machine learning* project [HKV19], or the commercial products www.automaticstatistician.com and www.datarobot.com. The Auto-Weka [KTH$^+$17] and Auto-sklearn [FKE$^+$15] modules can automate the selection of a machine learning algorithm and its parameters for solving standard classification or regression tasks. Most of these automated tools target tabular datasets, but not yet sequences and time-series data. Data-mining problems on sequences and time series remain challenging [YW06] but are nevertheless of particular interest [DLM$^+$98, SYCC$^+$15]. We believe that CP, because of the flexibility it offers, may play a role in the portfolio of techniques available for automating data-science on sequential data. As an illustration of this flexibility, Negrevergne and Guns [NG15] identified some constraints that could be stated on the patterns to discover in a database of sequences: length, exclusion/inclusion on symbols, membership to a regular language [Pes04], etc. The idea of using CP for data-mining is not new. It was already used for item-set mining [GDT$^+$13, GNDR11, NG10, SAG17], for SPM [AGS16, AGS17, KLL$^+$16, NG15] or even for mobility profile mining [KNGO15].

In this work we address the Frequent Episode Mining (FEM), first introduced with the apriori-like method WINEPI [MTV95] and improved on MINEPI [MT96], with a CP approach. Contrarily to the traditional SPM, FEM aims at discovering frequent patterns in a single but very long sequence of symbols possibly time-stamped. Assume for instance a non time-stamped sequence $\langle a, b, a, c, b, a, c\rangle$ and we are looking for patterns of length three occurring at least two times. Such a subsequence is $\langle a, b, c\rangle$ that occurs exactly two times. A first occurrence is $\langle \mathbf{a}, \mathbf{b}, a, \mathbf{c}, b, a, c\rangle$ and a second one is $\langle a, b, \mathbf{a}, c, \mathbf{b}, a, \mathbf{c}\rangle$. The attentive reader may wonder why $\langle \mathbf{a}, \mathbf{b}, a, c, b, a, \mathbf{c}\rangle$ is not counted. The reason is that the *head/total frequency* measure [ITN04] avoids duplicate counting by restricting a counting position to the first one. This measure has some interesting properties such as the well-known anti-monotonicity which states that if a sequence is frequent all its subsequences are frequent too and reversely. This property makes it possible to design faster data-mining algorithm. Indeed, based on these properties, Huang and Chang [HC08] proposed two algorithms, MINEPI+ and EMMA. While the first one is only a small adaptation of MINEPI [MT96], the second uses memory anchors in order to accelerate the mining task with the price of a greater memory consumption. Variants of this problem such as closed episodes [TC10, ZLC10], episode or itemset in tream [LSU07, CDG07], and set of sequences [CGR09,] can be considered. When considering time-stamped sequences such as $\langle (a, 1), (b, 3), (a, 5), (c, 6), (b, 7), (a, 8), (c, 14)\rangle$, one may also want to impose time constraints on the time difference between any two matched symbols or between the first and last matched ones. Such constraints, called *gap* and *span* were also introduced for the SPM [AGS17] with CP.

The problem of discovering frequent pattern in a very long sequence can be reduced do the standard SPM problem [AMS+96]. The reduction consists in creating a database of sequences composed of all the suffixes of the long sequence. For our example, the sequence database would be: $\langle a,b,a,c,b,a,c\rangle$, $\langle b,a,c,b,a,c\rangle$, $\langle a,c,b,a,c\rangle$, $\langle c,b,a,c\rangle$, $\langle b,a,c\rangle$, $\langle a,c\rangle$, $\langle c\rangle$. A small adaptation of existing algorithms is required though to match any sequence of the database on its first position in accordance with the *head/total frequency* measure. This reduction has one main drawback. The spatial complexity is $\mathcal{O}(n^2)$ with n the length of the sequence. Such a complexity will quickly exceed the available memory for sequence lengths as small as a few thousands.

We introduce flexible and efficient approaches (two global constraints) for solving the frequent episode mining problem, which use an implicit decomposition having a $\mathcal{O}(n)$ spatial complexity.

Our global constraints are inspired by the state-of-the-art approaches [AGS16, AGS17, KLL+16] but keeping the reduction into a suffix database implicit instead of explicit. We propose two versions: with and without considering *gap* and *span* constraints. We are also able to take some algorithmic advantages in the filtering algorithms using the property that the (implicit) database is composed of sorted suffixes from a same sequence. To the best of our knowledge, this work is the first CP-based approach proposed for solving efficiently this family of problems with the benefit that several other constraints such as Regular expression and Grammar can be added.

7.2 Mining Episodes in a Non Timed Sequence

7.2.1 Technical Background

Let $\mathcal{I} = \{1,\dots,m\}$ be an alphabet representing a set of possible symbols. We define a non timed sequence $s = \big\langle s_1,\dots,s_n\big\rangle$ over $\mathcal{I}$ as an ordered list of symbols such that $\forall i \in [1\mathinner{..}n], s_i \in \mathcal{I}$. Let us consider the following definitions based on the formalization of Aoga et al. [AGS16] and Huang and Chang [HC08].

Definition 7.1 (Subsequence relation, Embedding)**.** *$\alpha = \langle\alpha_1,\dots,\alpha_k\rangle$ is a subsequence of $s = \langle s_1,\dots,s_n\rangle$, denoted by $a \preceq s$, if $k \leq n$ and if there exists a list of indexes $(e_1,\dots,e_k)$ with $1 \leq e_1 \leq \dots \leq e_k \leq n$ such that $s_{e_i} = \alpha_i$. Such a list is referred as an embedding of s. Sequence s is also referred as a super-sequence of α.*

Example 7.1. $\langle a,b,c\rangle$ is a subsequence of the sequence $s = \langle a,b,a,c,b,a,c\rangle$ with embeddings $(1,2,4)$ or $(1,2,7)$ or $(3,5,7)$.

Definition 7.2 (Episode-embedding)**.** *Let us consider $\alpha = \langle \alpha_1, \ldots, \alpha_k \rangle \preceq s$. Embedding $(e_1, \ldots, e_k)$ is an episode-embedding if it is an embedding of s and if all the other embeddings $(e_1, e'_2, \ldots, e'_k)$ are such that $(e_2, \ldots, e_k) \preceq_L (e'_2, \ldots, e'_k)$ where $\preceq_L$ represents a lexicographic ordering.*

Example 7.2. $\langle a, b, c \rangle$ is a subsequence of s with $(1, 2, 4)$ and $(3, 5, 7)$ as episode-embeddings. Besides, $(1, 2, 7)$ is not an episode-embedding because $(2, 7)$ is lexicographically greater than $(2, 4)$.

Definition 7.3 (Support)**.** *The support $Sup_s(\alpha)$ of a subsequence α in a sequence s is the number of episode-embeddings of α in s.*

Example 7.3. For s of Example 7.1, we have $Sup_s(\langle a, b, c \rangle) = 2$.

Frequent Episode Mining (FEM) problem can then be formalised. Let us underline that this definition is related to the *total frequency* measure introduced by Iwanuma et al. [ITN04]. The goal is to count up occurrences without duplication. To do so, we use the concept of prefix-projection introduced in PrefixSpan [HPMA$^+$01] and used thereafter by Kemmar et al. [KLL$^+$16] and Aoga et al. [AGS16] for SPM.

Definition 7.4 (Frequent Episode Mining (FEM))**.** *Given a set of symbols $\mathcal{I}$, a sequence s over $\mathcal{I}$ and a threshold θ, the goal is to find all the subsequences α in s such that $Sup_s(\alpha) \geq \theta$. These subsequences are called episodes.*

Let now recall the notion of *Prefix-Projection*.

Definition 7.5 (Prefix, Projection, Suffix)**.** *Let $\alpha = \langle \alpha_1, \ldots, \alpha_k \rangle$ and $s = \langle s_1, \ldots, s_n \rangle$ be two sequences. If $\alpha \preceq s$, then the prefix of s w.r.t. α is the smallest prefix of s that remains a super-sequence of α. Formally, it is the sequence $\langle s_1, \ldots, s_j \rangle$ such that $\alpha \preceq \langle s_1, \ldots, s_j \rangle$ and such that there exists no $j' < j$ where $\alpha \preceq \langle s_1, \ldots, s_{j'} \rangle$. The sequence $\langle s_{j+1}, \ldots, s_n \rangle$ is then called the suffix of s w.r.t. α, or the α-projection, and is denoted by $s|_\alpha$. If α is not a subsequence of s, the α-projection is empty.*

Example 7.4. Given sequence s of Example 7.1 and $\alpha = \langle b \rangle$, sequence $\langle a, b \rangle$ is a prefix of s w.r.t. α and $\langle a, c, b, a, c \rangle$ is a suffix ($s|_\alpha = \langle a, c, b, a, c \rangle$).

Definition 7.6 (Initial Projection)**.** *An initial projection of a sequence $s = \langle s_1, \ldots, s_n \rangle$ w.r.t. a symbol x, denoted by $s|^{\mathcal{I}}_x$, is the list of all the suffixes $s' = \langle s_i, \ldots, s_n \rangle$ such that $s_{i-1} = x$ for all $i \in (1 \,..\, n]$.*

Example 7.5. For s and a symbol a, we have $s|_a^{\mathcal{I}} = \big[\langle b, a, c, b, a, c\rangle,\ \langle c, b, a, c\rangle,\ \langle c\rangle\big]$.

Definition 7.7 (Internal Projection)**.** *Given a list of sequences Ω, an internal projection of Ω w.r.t. pattern α, denoted by $\Omega|_\alpha$, is the list of the α-projection of all sequences in Ω. All the empty sequences are removed from $\Omega|_\alpha$.*

Example 7.6. For $\alpha = \langle b\rangle$ and $\Omega = \big[\langle b, a, c, b, a, c\rangle,\ \langle c, b, a, c\rangle,\ \langle c\rangle\big]$, we obtain $\Omega|_\alpha = \big[\langle a, c, b, a, c\rangle,\ \langle a, c\rangle\big]$.

Definition 7.8 (Projected Frequency)**.** *Given the list of sequences Ω, and a projection $s|_\alpha$ for each sequence $s \in \Omega$, the projected frequency of a symbol is the number of α-projected sequences where the symbol appears.*

Example 7.7. Given the internal projection $\Omega|_\alpha$ of Example 7.6, the projected frequencies are $\mathit{freq}(a) = 2$, $\mathit{freq}(b) = 1$ and $\mathit{freq}(c) = 2$.

In practice, the initial projections and internal projections can be efficiently stored as a list of pointers in the original sequence s. In our example ($s = \langle a, b, a, c, b, a, c\rangle$), we have $s|_a^{\mathcal{I}} = [2, 4, 7]$ and starting from $\Omega = s|_a^{\mathcal{I}}$ we can represent $\Omega|_{\langle b\rangle} = \big[3, 6\big]$. This representation introduced in *PrefixSpan* [HPMA+01] is called the *pseudo projection* representation (find more detail in Chapter 2).

Since the search follows a depth-first-search strategy, the pseudo projections can be stacked on a same vector allowing to reuse allocated entries on backtrack. This memory management is known as a trailing in CP and was introduced for SPM by Aoga et al. [AGS16, AGS17].

7.2.2 Problem Modelling

Our first contribution is a global constraint, `episodeSupport`, dedicated to find frequent patterns (or *episodes* [MTV95]) in a sequence without considering time. Let $s = \langle s_1, \ldots, s_n\rangle$ be a sequence of n symbols over $\mathcal{I}$, the set of distinct symbols appearing in s, and θ, the minimum support threshold desired.

Decision Variables Let $P = \langle P_1, \ldots, P_n\rangle$ be a sequence of variables representing a pattern. The domain of each variable is defined as $P_i = \mathcal{I} \cup \{\varepsilon\}$ for all $i \in [1\,..\,n]$. It indicates that each variable can take any symbol appearing in s as value in addition to ε, which is defined as the empty symbol. An assignment of P_i to ε means that P_i

has matched no symbol. It is used to model patterns having a length lower than n. A solution is an assignation of each variable in P.

EpisodeSupport Constraint The `episodeSupport`(P, s, θ) constraint enforces the three following constraints:

1. $P_1 \neq \varepsilon$
2. $P_i = \varepsilon \rightarrow P_{i+1} = \varepsilon, \forall i \in \big[2\,..\,n\big)$ and
3. $Sup_s(P) \geq \theta$.

The first constraint states that a pattern cannot begin with the empty symbol. It indicates that a valid pattern must contain at least one symbol. The second constraint ensures that ε can only appear at the end of the pattern. It is used in order to prevent same patterns with ε in different positions to be part of the same solution (such as $\langle a, b, \varepsilon\rangle$ and $\langle a, \varepsilon, b\rangle$). Finally, the last constraint states that a pattern must occur at least θ times in the sequence. The goal is then to find an assignment of each P_i satisfying the three constraints. The `episodeSupport` constraint filters from the domains of variables P the infrequent symbols in s at each step in order to find an assignment representing a frequent pattern according to θ. All the inconsistent values of the next uninstantiated variables in the pattern are then removed. Assuming the pattern variables are labeled in static order from left to right, the search is failure free when only this constraint must hold (i.e. all the leaf nodes are solution). Besides, `episodeSupport` is domain consistent: the remaining values in the domain of each variable are part of a solution because all of them have, at least, one support. Additional constraints can also be integrated to the model in order to define properties that the patterns must satisfy. For instance, we can enforce patterns to have at most k symbols or to follow a regular expression.

7.2.3 Filtering Algorithm

Preprocessing The index of the last position of each symbol in s is stored into a map (*lastPos*). For instance, $s = \langle a, b, a, c, \mathbf{b}, \mathbf{a}, \mathbf{c}\rangle$ gives $\big\{(c \rightarrow 7), (a \rightarrow 6), (b \rightarrow 5)\big\}$. The map can be iterated in a decreasing order by the last positions.

Sequence Projection and Pseudo Projection The key idea is to successively compute a projection from the previous one each time a variable has been assigned. The assignment of the first variable of the pattern (P_1) involves an *initial projection*. It splits s into a list of subsequences such that each one begins with the projected symbol. The assignment of the other variables (P_2 to P_n) implies an *internal projection*. This behavior is illustrated in the upper part of Figure 7.1 for an arbitrary example. The steps leading to pattern $\langle a, b, c, c\rangle$ are detailed. Three subsequences are obtained after an *initial projection* of symbol a $\big((0) \rightarrow (1)\big)$. While there are non-empty sequences, *internal projections* are successively performed $\big((1) \rightarrow (4)\big)$ and the pattern (P) is incrementally built.

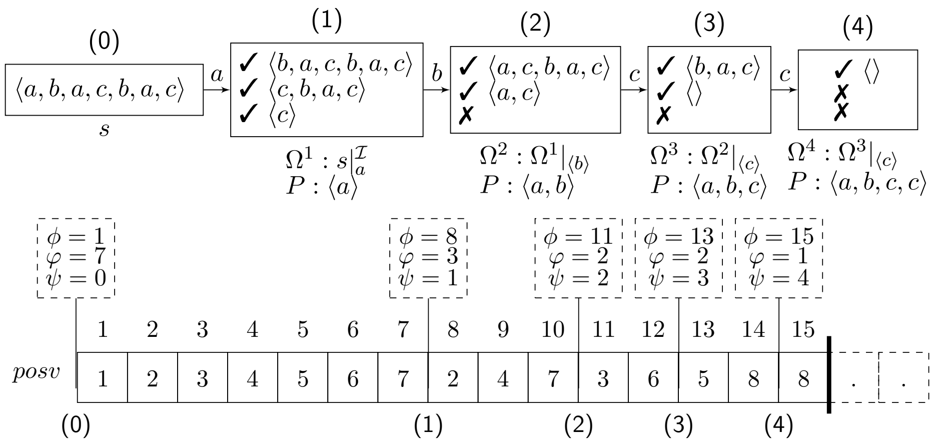

(0)
$\langle a,b,a,c,b,a,c\rangle$
s
a
(1)
✓ $\langle b,a,c,b,a,c\rangle$
✓ $\langle c,b,a,c\rangle$
✓ $\langle c\rangle$
$\Omega^1 : s|_a^{\mathcal{I}}$
$P : \langle a\rangle$
b
(2)
✓ $\langle a,c,b,a,c\rangle$
✓ $\langle a,c\rangle$
✗
$\Omega^2 : \Omega^1|_{\langle b\rangle}$
$P : \langle a,b\rangle$
c
(3)
✓ $\langle b,a,c\rangle$
✓ $\langle\rangle$
✗
$\Omega^3 : \Omega^2|_{\langle c\rangle}$
$P : \langle a,b,c\rangle$
c
(4)
✓ $\langle\rangle$
✗
✗
$\Omega^4 : \Omega^3|_{\langle c\rangle}$
$P : \langle a,b,c,c\rangle$
$\phi = 1$ $\varphi = 7$ $\psi = 0$
$\phi = 8$ $\varphi = 3$ $\psi = 1$
$\phi = 11$ $\varphi = 2$ $\psi = 2$
$\phi = 13$ $\varphi = 2$ $\psi = 3$
$\phi = 15$ $\varphi = 1$ $\psi = 4$
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
$posv$
1 2 3 4 5 6 7 2 4 7 3 6 5 8 8 . .
(0) (1) (2) (3) (4)

Figure 7.1. Sequence projection (✓ indicates a match, ✗ otherwise) and its reversible vector.

In practice, only pointers to the position in each sequence where the prefix has matched are stored. It is the mechanism of *pseudo projection*. As Aoga et al. [AGS16], we implement it with a reversible vector (*posv*) and a trail-based structure (lower part of Figure 7.1). The idea is to use the same vector during all the search inside the propagator, and only to maintain relevant start and stop positions. At each propagator call, three steps are performed. First, the last recorded start and stop positions are taken. Secondly, the propagator records the new information in the vector after the previous stop position. Finally, the new positions are updated in order to retrieve the information added. The reversible vector is then built incrementally. For each projection, the corresponding start index (ϕ) in the vector as well as the number of sequences inside the projection (φ) are stored. In other words, information related to a projection are located between indexes $i \in [\phi, \phi + \varphi)$. Besides, the index of the variable P_i that has been assigned (ψ) is also recorded after each projection step. Before the first assignation ψ is equal to zero. The three variables are implemented as reversible integers. Initially, all the indexes are present in the vector, but all along the pseudo-projections, only the non-empty sequences are considered.

Propagation The goal is to compute a projection each time a variable has been assigned to a symbol a. Assignments of variables are done successively from the first variable to the last one. The propagator is then called after each assignment. It is shown in Algorithm 7.1. Initialisation of reversible structures is done when $\psi = 0$ (lines 8-9). If the last assigned variable (P_ψ) has been bound to ε, the algorithm enforces all the next variables to be also bound to ε (lines 10-12). The pattern is then completed and the propagation is finished. Otherwise, after each variable

Algorithm 7.1: *propagate*(s, Σ, a, P)

```
▷ Input: $s$ is the initial long sequence of size $n$.
        $\Sigma$ is the set of symbols and $a$ is a symbol.
        If $\psi > 0$ then $\langle P_1, \ldots, P_\psi \rangle \in P$ are bound and $P_\psi$ is assigned to $a$.
        $\theta$ is the support threshold.
        $posv$, $\phi$, $\varphi$ and $\psi$ are the reversible structures as defined before.
▷ Internal State: $posv$, $\phi$, $\varphi$ and $\psi$.

if $\psi = 0$ then
    $\phi := 1$    $\varphi := n$    $\psi := 1$    $posv[i] := i$                    $i \in [1..n]$
if $P_\psi = \varepsilon$ then
    for $j \in [\psi + 1, n]$ do
        $P_j.assign(\varepsilon)$
else
    $freq := sequenceProjection(s, \Sigma, a)$                ▷ Detailed in Alg. 7.2.
    foreach $b \in Domain(P_{\psi+1})$ do
        if $b \neq \varepsilon \wedge freq[b] < \theta$ then
            $P_{\psi+1}.removeValue(b)$
```

binding, the projected sequence and the *projected frequencies* are computed (line 14). Finally, all the infrequent symbols are removed from the domain of $P_{\psi+1}$ (lines 15-17). *Projected frequency* of each symbol in the domain of $P_{\psi+1}$ (except ε) is compared to the threshold and removed if it is infrequent.

Sequence Projection Let us now present how sequences are projected (Algorithm 7.2). First, the *projected frequency* of each symbol for the current pseudo projection is set to 0 (*freq* on line 9). The main loop (lines 10-24) iterates over the previous projection thanks to the reversible integers ϕ and φ. At each iteration a value in *posv* is considered. The next condition (line 13) is used to distinguish the *initial* from an *internal projection.* If a is the first projected symbol and if it does not match with the first symbol of the sequence, then the sequence is not included in the projection. Otherwise, an *internal projection* is applied.

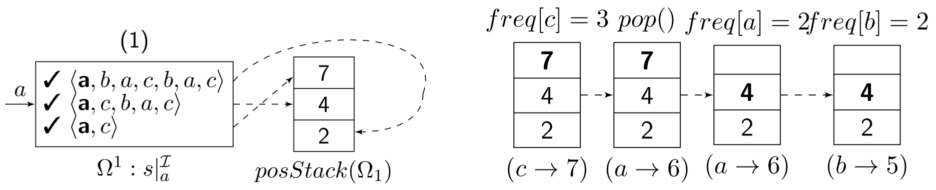
The next expression (lines 14-15) is an optimization we introduced, called *early projection stopping.* This optimization is based on one invariant of our structure: it stores suffixes of s with a decreasing order by their size. Each suffix in a projection is then strictly included in all the previous ones. When a no-match has been detected in a sequence, all the next ones can be directly discarded without being checked. It stops the internal projection as soon as possible. The *early projection stopping* gains importance when the number of sequences is large. Then, if a is not present in the current considered sequence, the loop can be stopped and unnecessary computation is avoided.

Algorithm 7.2: $sequenceProjection(s, \Sigma, a)$

```
1  ▷ Input: $s$ is the initial long sequence.
2        $\Sigma$ is the set of symbols.
3        $a$ is the current projected symbol ($a = P_\psi$).
4        $posv$, $\phi$, $\varphi$ and $\psi$ are reversible structures as defined before.
5        $posv[i]$ with $i \in [\phi, \phi + \varphi)$ is initialized.
6  ▷ Output: $freq$ list of projected frequencies
7  ▷ Internal State: $posv$, $\phi$, $\varphi$ and $\psi$.
8
9  $j \leftarrow \varphi$   $sup \leftarrow 0$   $prevPos \leftarrow -1$   $freq[b] \leftarrow 0\ \forall b \in \Sigma$
10 $posStack \leftarrow Stack()$
11 for $i \in [\phi, \phi + \varphi - 1]$ do
12     $pos \leftarrow posv[i]$
13     if $\psi > 1 \vee a = s[pos]$ then
14         if $pos > lastPos[a]$ then
15             break                                        ▷ Early projection stopping
16         else
17             if $prevPos < pos$ then
18                 while $a \neq s[pos]$ do
19                     $pos \leftarrow pos + 1$
20                 $prevPos \leftarrow pos$                     ▷ Position caching
21             else
22                 $pos \leftarrow prevPos$
23             $posStack.push(pos + 1)$
24             $posv[j] \leftarrow pos + 1$   $j \leftarrow j + 1$   $sup \leftarrow sup + 1$
25 foreach $(x, pos_x)$ in $lastPos$ do
26     while $posStack.notEmpty \wedge posStack.top > pos_x$ do
27         $posStack.pop$
28     $freq[x] \leftarrow posStack.size$                      ▷ Projected frequency
29     if $posStack.isEmpty$ then break
30 $\phi \leftarrow \phi + \varphi$   $\varphi \leftarrow sup$   $\psi \leftarrow \psi + 1$
31 return $freq$
```

At this step, we are sure that a appears at least once in the current sequence in the projection. Lines 17 to 22 make the search for the match, either by *position caching*, or by iteration on the positions. *Position caching* is a second optimization we introduced. Once a match has been detected in a sequence, the position of the match is recorded. Thanks to the aforementioned invariant, we are sure that the match cannot occur before this position (because we only have one sequence). If this position is greater than the start position of the sequence (in *posv*), a match is directly detected. The reversible vectors are then updated (line 24). Variable *sup* is used to store the size of the new projection.

The last loop (lines 25-29) updates the *projected frequency* of each symbol. The *projected frequency* of a symbol in a projection corresponds to the number of sequences of the projection beginning at an index lower than the index of the last position of the symbol. This idea was introduced in LAPIN [YWK07] and exploited by Aoga et al. [AGS16]. It can be implemented efficiently thanks to the invariant and *lastPos* map. It is illustrated in Figure 7.2 (upper part) with $lastPos = \{(c \to 7), (a \to 6), (b \to 5)\}$. The position just after each match is pushed in a LIFO structure, *posStack*. The last matched position is located on the top of the stack.



(1)
a
✓ $\langle \mathbf{a}, b, a, c, b, a, c\rangle$
✓ $\langle \mathbf{a}, c, b, a, c\rangle$
✓ $\langle \mathbf{a}, c\rangle$
$\Omega^1 : s|_a^{\mathcal{I}}$
7
4
2
$posStack(\Omega_1)$
$freq[c] = 3$
$pop()$
$freq[a] = 2$
$freq[b] = 2$
$(c \to 7)$
$(a \to 6)$
$(a \to 6)$
$(b \to 5)$

Figure 7.2. Efficient frequency computation.

Once the stack is obtained, the idea is to successively compare in a decreasing order the last position of each symbol with the top of the stack. Illustration of this behavior is presented in Figure 7.2 (lower part). If the last position of a symbol is greater than the top of the stack, it indicates that the symbol occurs at least once in the current sequence and consequently in all the previous ones in the stack. The *projected frequency* of this symbol corresponds then to the remaining size of the stack and the next symbol in *lastPos* can be processed. Otherwise, we are sure that the symbol has no occurrence in the current sequence. Its position is popped and the comparison is done with the new top. The resulting *projected frequencies* are $c = 3$, $a = 2$ and $b = 2$. This mechanism has a time complexity of $\mathcal{O}(n + |\mathcal{I}|)$. For comparison, *projected frequencies* are computed in $\mathcal{O}(n \times |\mathcal{I}|)$ by Aoga et al. [AGS16] (each subsequence is scanned at each projection). Finally, the reversible integers are updated and the *projected frequency* map is returned.

Time and Spatial Complexity Main loop of Algorithm 7.2 (lines 11-24) is computed in $\mathcal{O}(n^2)$ and the *projected frequencies* (lines 25-29) in $\mathcal{O}(n + |\mathcal{I}|) = \mathcal{O}(n)$ (because the number of different symbols is bounded by the sequence size). In Algorithm 7.1, lines 8-9 cost $\mathcal{O}(n)$ and the domain pruning (lines 15-17) is performed in $\mathcal{O}(|\mathcal{I}|)$. It gives $\mathcal{O}(n+(n^2+n)+|\mathcal{I}|) = \mathcal{O}(n^2)$. For the spatial complexity, we have $\mathcal{O}(n + n \times d) = \mathcal{O}(n \times d)$ with d the maximum depth of the search tree, which is the maximum size of the reversible vector. For comparison, an explicit decomposition of the problem gives $\mathcal{O}(n^2 + n \times d) = \mathcal{O}(n^2)$.

7.3 Mining Episodes in a Timed Sequence

7.3.1 Technical Background

So far, `episodeSupport` can only deal with *sequences of symbols* where time is not considered. In practice, sequences can also be time-stamped. Such sequences are most often referred as *sequences of events* instead of *sequences of symbols* and new constraints can then be expressed. For instance, we can be interested in finding episodes such that the elapsed time between two events does not exceed one hour. We define a sequence of events $s = \big\langle (s_1, t_1), \dots, (s_n, t_n) \big\rangle$ over $\mathcal{I}$ as an ordered list of events (s_i) occurring at time t_i such that for all $i \in \{1, .., n\}$ we have $s_i \in \mathcal{I}$ and $t_1 \leq t_2 \leq \dots \leq t_n$. The list containing only the events is denoted by s^s and the list of timestamps by s^t. All the principles defined in the previous sections are reused. In addition, we are now able to enforce time restrictions. Two of them are often used in practice: *gap* and *span*. The former (*gap*) restricts the time between two consecutive events while the latter (*span*) restricts the time between the first and the last event. Considering such restrictions cannot be done only by imposing additional constraints in the model [AGS17]. It requires to adapt the subsequence relations (Definition 7.9) and to design a dedicated propagator. We need to define the *extension window*. The extension window of an embedding contains only events whose timing satisfies *gap* constraint.

Definition 7.9 (Subsequence under gap/span). *$\alpha = \langle \alpha_1, \dots, \alpha_x \rangle$ is a subsequence of $s = \big\langle (s_1, t_1), \dots, (s_n, t_n) \big\rangle$ under $gap[M, N]$, denoted by $\alpha \preceq^{gap[M,N]} s$, if and only if s^s is a subsequence of embedding $(e_1, \dots, e_k)$ according to Definition 7.1, and if $\forall i \in [2..k]$ we have $M \leq t_{e_i} - t_{e_{i-1}} \leq N$. The embedding $(e_1, \dots, e_k)$ under $\preceq^{gap[M,N]}$ relation is called a $gap[M, N]$-embedding. $(e_1, \dots, e_k)$ is an episode-embedding of α according to Definition. 7.2 where $\preceq^{gap[M,N]}$ is considered for the subsequence relation. The support of α, denoted by $Sup_s^{gap^{[M,N]}}(e)$, is the number of $gap[M, N]$-embeddings of α in s. Similarly, α is a subsequence of s under $span[W, Y]$, denoted by $\alpha \preceq^{span[W,Y]} s$, if and only if s^s is a subsequence of embedding $(e_1, \dots, e_k)$ according to Definition 7.1, and if $W \leq t_{e_k} - t_{e_1} \leq Y$. Relation $\preceq^{span[W,Y]}$ and $Sup_s^{span^{[W,Y]}}(e)$ are also defined similarly.*

Example 7.8. Let us consider $s = \big\langle (a, 2), (b, 4), (a, 5), (c, 7), (b, 8), (a, 9), (c, 12) \big\rangle$. $\langle a, b, c \rangle$ is a subsequence of s under $gap[1, 3]$ with embedding $(1, 2, 4)$. $(3, 5, 7)$ is not a $gap[1, 3]$-embedding because $t_{e_3} - t_{e_2} = 12 - 8 > 3$. In addition, $\langle a, b, c \rangle$ is a subsequence of s under $span[6, 10]$ with embedding $(3, 5, 7)$. $(1, 2, 4)$ is not valid because $t_{e_3} - t_{e_1} = 7 - 2 < 6$.

Definition 7.10 (Extension window)**.** *Let $e = (e_1, e_2, \dots, e_k)$ be any $gap[M, N]$-embedding of a subsequence α in a sequence s. The extension window of this embedding, denoted $\mathrm{ew}_e^{gap[M,N]}(s)$, is the subsequence $\big\langle (s_u, t_u), \dots, (s_v, t_v) \big\rangle$ such that $(t_{e_k} + M \leq t_u) \wedge (t_v \leq t_{e_k} + N) \wedge (t_{u-1} < t_{e_k} + M) \wedge (t_{v+1} > t_{e_k} + N)$. Each embedding has a unique extension window, which can be empty.*

Example 7.9. Let $(3, 4)$ be a $gap[2, 6]$-embedding of $\langle a, c\rangle$ in sequence s (Example 7.8). We have $\mathrm{ew}_e^{gap[2,6]}(s) = \big\langle (a, 9), (c, 12) \big\rangle$.

The goal is to find the all patterns having a support, possibly under *gap* and *span*, greater than the threshold. Let $P = \langle P_1, \dots, P_n\rangle$ be a sequence of variables representing a pattern. the timed version of `episodeSupport`(P,s,θ,M,N,W,Y) enforces the four following constraints:

1. $P_1 \neq \varepsilon$,
2. $P_i = \varepsilon \rightarrow P_{i+1} = \varepsilon$, $\forall i \in \big[1, n\big)$,
3. $Sup_s^{gap^{[M,N]}}(P) \geq \theta$ and
4. $Sup_s^{span^{[W,Y]}}(P) \geq \theta$.

7.3.2 Filtering Algorithm

Precomputed Structures The three structures are shown in Figure 7.3a. First, the *lastPos* map is adapted from the previous section in order to store the last position of each event that can be matched while satisfying the maximum span (Y). The last position of each event inside each range $[t, t + Y]$ is recorded, where t is the timestamp of the event. Maximum span is then implicitly handled by this structure, which is not done by Aoga et al. [AGS17]. Besides, for each position i in s, the index of the first (u) and the last (v) positions after i such that $t_u \geq t_i + M$ and $t_v \geq t_i + N$ are stored into a map (*nextPosGap*), where M and N are the minimum and maximum gap. The *nextPosGap* is used after each projection in order to directly access the next extension window. Finally, for each position i in s, the number of times that each event has occurred inside the range $[1, i]$ in s is stored (*freqMap*). It is used in order to efficiently compute the *projected frequency* of each symbol during a projection. We can be sure that an event a appears at least once in a window of range $[u, v]$ if the occurrence of a at the end of the window is strictly greater than the occurence of a just before the window ($freqMap[v][a] > freqMap[u - 1][a]$). It has not been used by Aoga et al. [AGS17].

Storing Several Embeddings When a *gap* constraint is considered, the anti-monotonicity property does not hold anymore [AGS17]. The main consequence is that all the possible embeddings must be considered, and not only the first one. The projection mechanism (described in Figure 7.1) has then to be adapted. This is illustrated in Figure 7.3b. For instance, two embeddings are considered for the projection from (1) to (2) of the first sequence. It is required to record all of the corresponding extension windows in order to miss no supporting event. To do so, a reversible vector (*startv*) recording the start index in s for each sequence is used (Figure 7.3c). Besides, other reversible vectors are added: *esize*, which represents the number of embeddings related at each projected sequence and *embs*, which records the start index of the different embeddings. It is a simplified adaptation of the structure proposed by Aoga et al. [AGS17].

Minimum Span The minimum span is not anti-monotonic. Therefore, we do not consider it during the projection but *a posteriori*: it is only checked when a complete pattern is obtained and not before. It requires slight modifications in the *propagate* method (Algorithm 7.1). A variable $\gamma_s(P)$ representing the number of supports satisfying the minimum span constraint for P is recorded and computed during the projection. Once the projections are completely done for this episode (after the line 12), $\gamma_s(P)$ is compared with the support threshold and an *inconsistency* is raised if it is below the threshold.

Sequence Projection The projection mechanism is presented in Algorithm 7.3. Initially, the *projected frequencies* of each event is set to 0 (line 10). When $\psi = 1$, an initial projection is performed (lines 11-12). In the *buildInitialTimedSeqProj* function (Algorithm 7.4), we are looking for events that match with a (line 12). Once a match is detected, reversible vectors are updated (line 13). *Projected frequencies* are computed using *nextPosGap* and *freqMap* structures (lines 14-19). First, the window where the events must be considered is computed. Secondly, the *projected frequency* of each event appearing in the window is incremented. When $\psi > 1$, we have an internal projection (lines 13-14), which is performed in *buildTimedSeqProj* function of Algorithm 7.5. Each embedding is successively considered (line 15). For each one, the sequence is iterated from the first next position satisfying the minimum gap to the last one satisfying the maximum gap, or to the last symbol of the sequence (line 17). Once a match has been detected, the number of possible embeddings is incremented and its position is recorded (line 19). If the embedding of the current pattern satisfies the minimum span constraint, γ_s is incremented (lines 20-21). It is used in the *propagate* method as explained before. Then, *projected frequencies* are computed as in the initial projection (lines 22-28).

Time and Spatial Complexity Let us assume k is the maximum length of time window (often $k \ll n$) and d the maximum depth of the tree search ($d \leq k$). **Initial projection** (lines 10-19) in Algorithm 7.4 is computed in $\mathcal{O}(n \times |\mathcal{I}|)$: the sequence is completely processed and frequencies are computed at each match. **Internal projection** (lines 11-31) in Algorithm 7.5 is computed in $\mathcal{O}(n \times |\mathcal{I}| \times k^2)$. It gives $\mathcal{O}(n \times |\mathcal{I}| + n \times |\mathcal{I}| \times k^2) = \mathcal{O}(n \times |\mathcal{I}| \times k^2)$. For the spatial complexity, vectors have a maximum length of $k \times d$ and there are at most k embeddings, which gives

	1	2	3	4	5	6	7
a	1	1	2	2	2	3	3
b	0	1	1	1	2	2	3
c	0	0	0	1	1	1	1

freqMap

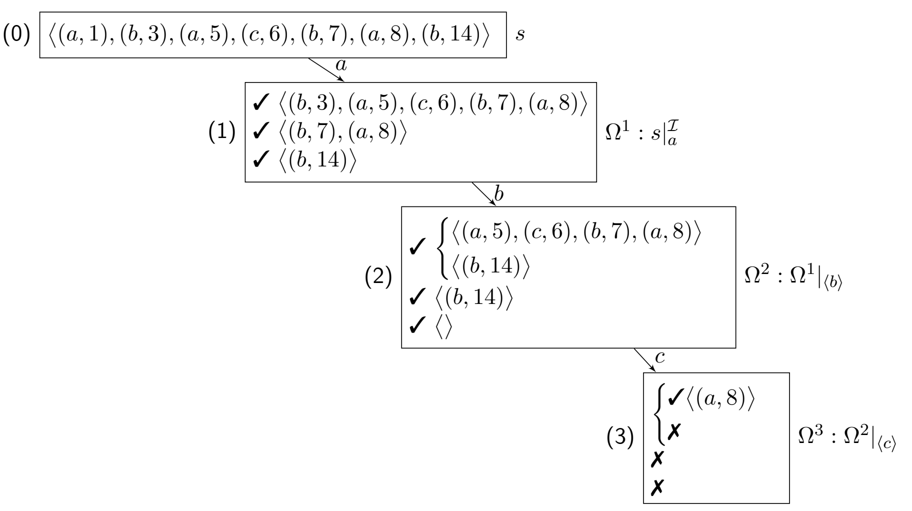
	1	2	3	4	5	6	7
a	6	6	6	6	6	6	.
b	5	5	7	7	7	7	7
c	4	4	4	4	.	.	.

lastPosMap for $span[Y = 10]$

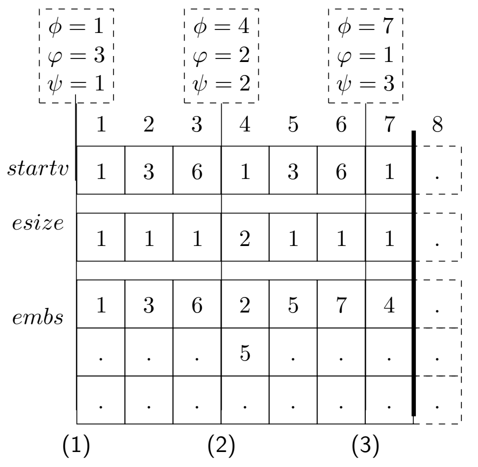
	1	2	3	4	5	6	7
1	2	3	5	6	7	7	.
2	6	6	6	6	7	.	.

nextPosGap for $gap[2, 7]$

(a) Precomputed structures.


(0) $\langle(a,1),(b,3),(a,5),(c,6),(b,7),(a,8),(b,14)\rangle$ s
a
(1) ✓ $\langle(b,3),(a,5),(c,6),(b,7),(a,8)\rangle$
✓ $\langle(b,7),(a,8)\rangle$
✓ $\langle(b,14)\rangle$
$\Omega^1 : s|^{\mathcal{I}}_a$
b
(2) ✓ $\begin{cases}\langle(a,5),(c,6),(b,7),(a,8)\rangle\\ \langle(b,14)\rangle\end{cases}$
✓ $\langle(b,14)\rangle$
✓ $\langle\rangle$
$\Omega^2 : \Omega^1|_{\langle b\rangle}$
c
(3) $\begin{cases}✓\langle(a,8)\rangle\\ ✗\end{cases}$
✗
✗
$\Omega^3 : \Omega^2|_{\langle c\rangle}$

(b) Sequence projection.



(c) Reversible vectors.

Figure 7.3. Data structures used for timed sequences with $gap[2, 7]$ and $span[1, 10]$.

Algorithm 7.3: $sequenceProjectionTimed(s^s, s^t, \mathcal{I}, a, N, W)$

```
▷ Input: $s^s$ and $s^t$ are the event/timestamp list of the initial long sequence.
        $\mathcal{I}$ is the set of symbols;
        $a$ is the current projected symbol ($a = P_\psi$);
        $startv[i]$, $esize[i]$ and $embs[i]$ with $i \in [\phi, \phi + \varphi)$ are initialized;
        $\gamma_s(P_{:\psi}) = 0$ with $P_{:\psi}$ the episode represented by $\langle P_1, \ldots, P_\psi \rangle$;
        $N$ and $W$ are the gap max bound and of the span min bound.
▷ Output: $freq$ list of projected frequencies
▷ Internal State: $startv$, $esize$, $embs$, $\phi$, $\varphi$, $\psi$, $\gamma_s(P_{:\psi})$.

$freq[b] \leftarrow 0 \quad \forall b \in \mathcal{I}$
if $\psi = 1$ then
    $(j,\ freq) \leftarrow buildInitialTimedSeqProj(s^s, s^t, \mathcal{I}, a)$    ▷ Detailed in Alg. 7.4.
else
    $(j,\ freq) \leftarrow buildTimedSeqProj(s^s, s^t, \mathcal{I}, a, N, W)$    ▷ Detailed in Alg. 7.5.
$\phi \leftarrow \phi + \varphi \quad \varphi \leftarrow j - \phi$
return $freq$
```

Algorithm 7.4: $buildInitialTimedSeqProj(s^s, s^t, \mathcal{I}, a)$

```
▷ Input: $s^s$ and $s^t$ are the event/timestamp list of the initial long sequence.
        $\mathcal{I}$ is the set of symbols;
        $a$ is the current projected symbol ($a = P_\psi$);
        $startv[i]$, $esize[i]$ and $embs[i]$ with $i \in [\phi, \phi + \varphi)$ are initialized;
        $\gamma_s(P_{:\psi}) = 0$ with $P_{:\psi}$ the episode represented by $\langle P_1, \ldots, P_\psi \rangle$.
▷ Output: $j$ position in vectors and $freq$ list of projected frequencies.
▷ Internal State: $startv$, $esize$, $embs$, $\phi$, $\varphi$, $\psi$, $\gamma_s(P_{:\psi})$.

$freq[b] \leftarrow 0 \quad \forall b \in \mathcal{I}$
$j \leftarrow 1$
for $pos \in \big[1, |s^s|\big]$ do
    if $s^s[pos] = a$ then
        $startv[j] \leftarrow pos \quad esize[j] \leftarrow 1 \quad embs[j][1] \leftarrow pos \quad j \leftarrow j + 1$
        $(u, v) \leftarrow nextPosGap[pos]$    ▷ Precomputed structure
        if $u \leq |s^s|$ then
            for $b \in Domain(P_{\psi+1})$ do
                $l \leftarrow \mathbf{min}\big(v, |s^s|\big)$
                if $freqMap[l][b] > freqMap[u-1][b]$ then
                    $freq[b] \leftarrow freq[b] + 1$    ▷ Projected frequency
return $(j\ ,\ freq)$
```

Algorithm 7.5: *buildTimedSeqProj*$(s^s, s^t, \mathcal{I}, a, N, W)$

```
▷ Input: $s^s$ and $s^t$ are the event/timestamp list of the initial long sequence;
        $\mathcal{I}$ is the set of symbols;
        $a$ is the current projected symbol ($a = P_\psi$);
        $startv[i]$, $esize[i]$ and $embs[i]$ with $i \in [\phi, \phi + \varphi)$ are initialized;
        $\gamma_s(P_{:\psi}) = 0$ with $P_{:\psi}$ the episode represented by $\langle P_1, \dots, P_\psi \rangle$;
        $N$ and $W$ are the gap max bound and of the span min bound.
▷ Output: $j$ position in vectors and $freq$ list of projected frequencies.
▷ Internal State: $startv$, $esize$, $embs$, $\phi$, $\varphi$, $\psi$, $\gamma_s(P_{:\psi})$.

$freq[b] \leftarrow 0 \quad \forall b \in \mathcal{I}$
$j \leftarrow \phi + \varphi$
for $i \in [\phi, \phi + \varphi - 1]$ do
    $id \leftarrow startv[i] \quad nEmb \leftarrow 0 \quad k \leftarrow 1 \quad v \leftarrow -1 \quad isIncremented \leftarrow$ false
    $isVisited[b] \leftarrow$ false $\quad \forall b \in \mathcal{I}$
    while $v < |s^s| \wedge k \leq esize[i]$ do
        $e \leftarrow embs[i][k] \quad (pos, \_) \leftarrow nextPosGap[e]$        ▷ 2nd element unused
        while $v < |s^s| \wedge pos \leq lastPosMap[id][a] \wedge s^t[pos] \leq s^t[e] + N$ do
            if $s^s[pos] = a$ then
                $nEmb \leftarrow nEmb + 1 \quad embs[j][nEmb] \leftarrow pos$
                if not $isIncremented \wedge s^t[pos] - s^t[id] \geq W$ then
                    $isIncremented \leftarrow$ true $\quad \gamma_s(P_{:\psi}) \leftarrow \gamma_s(P_{:\psi}) + 1$
                $(u, v) \leftarrow nextPosGap[pos]$        ▷ Precomputed structure
                if $u \leq |s^s|$ then
                    for $b \in Domain(P_{\psi+1})$ do
                        $l \leftarrow$ min$\big(v, |s^s|\big)$
                        if $\big(freqMap[l][b] > freqMap[u-1][b]\big) \wedge$ not
                         $isVisited\big[b\big]$ then
                            $isVisited\big[b\big] \leftarrow$ true
                            $freq[b] \leftarrow freq[b] + 1$        ▷ Projected frequency
            $pos \leftarrow pos + 1$
        $k \leftarrow k + 1$
    if $nEmb > 0$ then $startv[j] \leftarrow id \quad esize[j] \leftarrow nEmb \quad j \leftarrow j + 1$
return $(j\ , freq)$
```

$\mathcal{O}(d \times k^2)$.

7.4 Implementation and Practical User Guide

For the reproducibility of results, the implementation of both constraints is open source and available online (http://sites.uclouvain.be/cp4dm/).

The implementation of *EpisodeSupport* is available in the CP-Solver OscaR [Osc12] which is available online[1] in free access. In OscaR, one can combine our constraints with the existing constraints in the solver such as All-Different [Rég94], Global cardinality [QLvBG04], Grammar [QW06], etc.

For developers who are willing to modify the code directly, a lightweight version is also available[2]. One can hence add several constraints for a specific usage without understanding OscaR deeply. The installation procedure is described in the Install file in the code directory and merely consists of "importing the project" into your favourite IDE.

Users can directly download the *jar-file* which is available on our website [3]. On this website, there is a user guide. The general format of the command to run *EpisodeSupport* is:

```
java -jar episodesupport.jar [options] <SDB File> <Lmin > < Lmax>
```

with several options available, including:

- $-f$: for the frequency constraint (i.e. θ),
- $-i$: for items inclusion/exclusion constraints,
- $-t$: to indicate that the database is provided with timestamps,
- $-m$: for the minimum gap constraint,
- $-n$: for the maximum gap constraint,
- $-w$: for the maximum span constraint,
- $-y$: for the maximum span constraint,
- etc.

For example, to find the frequent episodes given a database (named test.txt) and the minimum support $\theta = 2$ (-f 2), we run the following command:

```
java -jar episodesupport.jar test.txt -f 2
```

Here is the result:

[1]https://bitbucket.org/oscarlib/oscar/wiki/Home
[2]https://bitbucket.org/projetsJOHN/episodesupport
[3]https://sites.uclouvain.be/cp4dm/fem/

Input (each line is a sequence)	Output (<sub-sequence> : <support>)
	`< 1 2  > : 3`
`1 2 3 2 3`	`< 1 2 3  > : 2`
`2 1 2 3`	`< 1 3  > : 2`
`1 2`	`< 2 2  > : 2`
`2 3 4`	`< 2 2 3  > : 2`
	`< 2 3  > : 3`

7.5 Experimental Results

This section evaluates the performance of `episodeSupport` on different datasets with and without time consideration. Experiments have been realised on a computer with a 2.7 GHz Intel Core i5 64 bits processor and with a RAM of 8 Go using a 64-Bit HotSpot(TM) JVM 1.8 running on Linux Mint 17.3. Execution time is limited to 1800 seconds unless otherwise stated. The algorithms have been implemented in Scala with OscaR solver [Tea12] and memory assessment has been performed with *java Runtime* classes. For the reproducibility of results, the implementation of both constraints is open source and available online.[4] One synthetic and three real-data sets are considered: proteins from Uniprot database [C$^+$08], UCI Unix dataset [Lic13] and UbiqLog [RMD$^+$15, RTWT13].

Our approach is compared with the existing methods. We identified two ways to mine frequent patterns in a sequence. On the one hand, we can resort to a specialized algorithm. To the best of our knowledge, MINEPI+ and EMMA [HC08] are the state-of-the-art methods for that. On the other hand, we can explicitly split the sequence into a database and then reduce the problem into an SPM problem. Once done, CP-based methods can be used [AGS16, AGS17, KLL$^+$16, NG15]. Our comparisons are based on the approach of Aoga et al. [AGS16, AGS17] that turns out to be the most efficient. We refer to it as the Decomposed Frequent Episode Mining (DFEM) approach, or DFEMt when time is considered.

7.5.1 Memory Bound Analysis

We applied DFEM and `episodeSupport` on synthetic sequences of different sizes with 100 distinct symbols uniformly distributed in order to define what are the largest sequences that can be processed. We observed that with decomposed approaches, sequences greater than 30000 symbols cannot be processed when memory is limited to 8GB. With `episodeSupport` memory is not a bottleneck.

[4]https://bitbucket.org/projetsJOHN/episodesupport (also available in [Osc12])

7.5.2 Comparison with Decomposed Approaches

Experiments and results with Uniprot and UbiqLog datasets are shown in Figure 7.4 & 7.5 and Table 7.1 & 7.2. The figures shows the performance profiles [DM02] and the tables present results for different settings for both the memory consumption and the computation time.

We can observe that `episodeSupport` outperforms both decomposed approaches in terms of execution time and memory consumption for most of the instances. Both gains become more important when the sequence is large. Besides, decomposed approaches cannot process the largest sequences regarding the time limitation imposed.

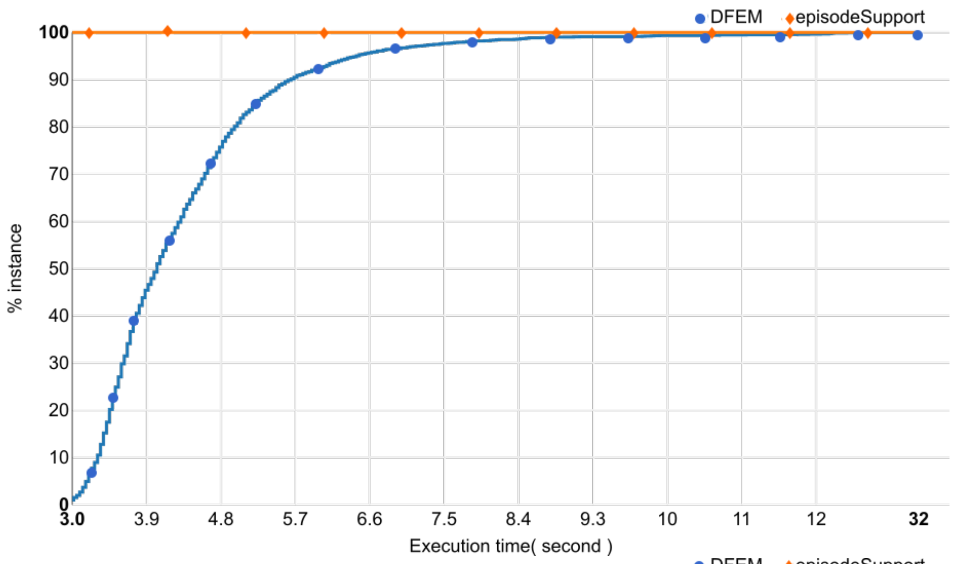
Table 7.1. Execution time and memory usage for several non-timed datasets and thresholds (the maximum size of episodes is set to 5).

Name |s| + |Σ|	θ	nSol	Memory (Mb) DFEM	Memory (Mb) episode Support	Execution time (s) DFEM	Execution time (s) episode Support
Q08379 1002 + 20	100	437048	45	**34**	9.08	**2.45**
	90	533395	46	**38**	8.68	**1.97**
	70	645834	35	**16**	9.16	**1.67**
	50	10481	1226	**505**	118.56	**57.48**
Q54CU4 11103 + 20	1110	0	1969	**31**	0.11	**0.02**
	999	157003	2057	**33**	113.32	**3.49**
	777	1178939	1980	**33**	657.68	**14.90**
	555	1515789	1849	**31**	1414.83	**18.41**
Q9I7U4 18141 + 20	1814	336842	6898	**38**	946.54	**20.90**
	100	38061	3301	**1179**	745.41	**488.32**
	1269	705640	6819	**21**	1674.80	**22.63**
	907	1515791	6819	**21**	timeout	**52.79**

7.5.3 Comparison with Specialized Approaches

Experiments on Unix dataset with a threshold of 5% and a maximum span of 10 are provided in [HC08][5]. Comparisons of these specialized approaches with ours are presented in Table 7.3. It shows that `episodeSupport` seems competitive with MINEPI+ and EMMA. For the largest sequences (USER8 and USER6), `episodeSupport` is the most efficient. For some instances (USER5 and USER7) that are quickly solved, the cost of

[5]Results provided in [HC08] are directly used since the implementation is not available.



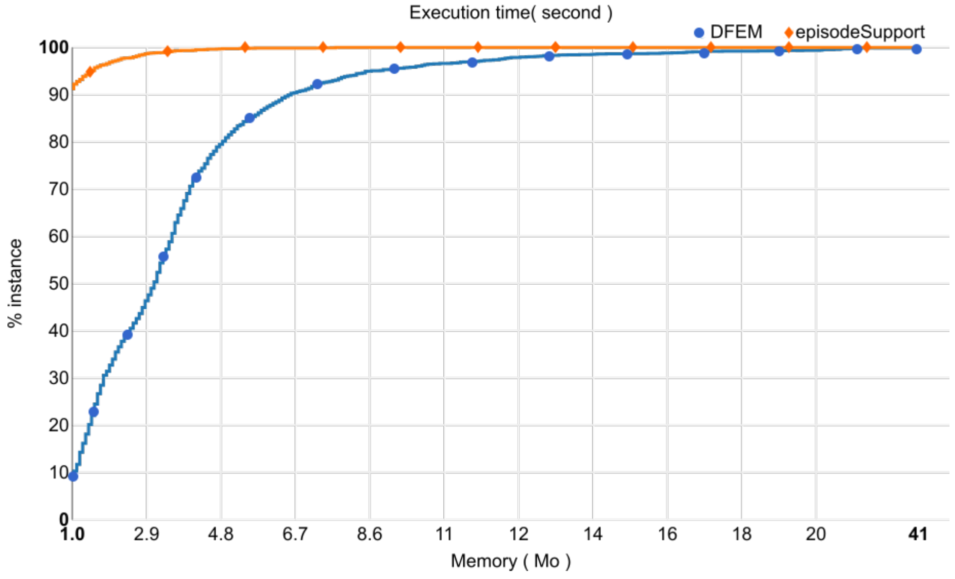


Figure 7.4. Performance profiles of execution time (**top**) and Memory usage (**bottom**) for Uniprot dataset (2452 instances, where $n \in [100, 30000]$) with $\theta = 5\%$, maximum size of 5 and time limit of 600 seconds.

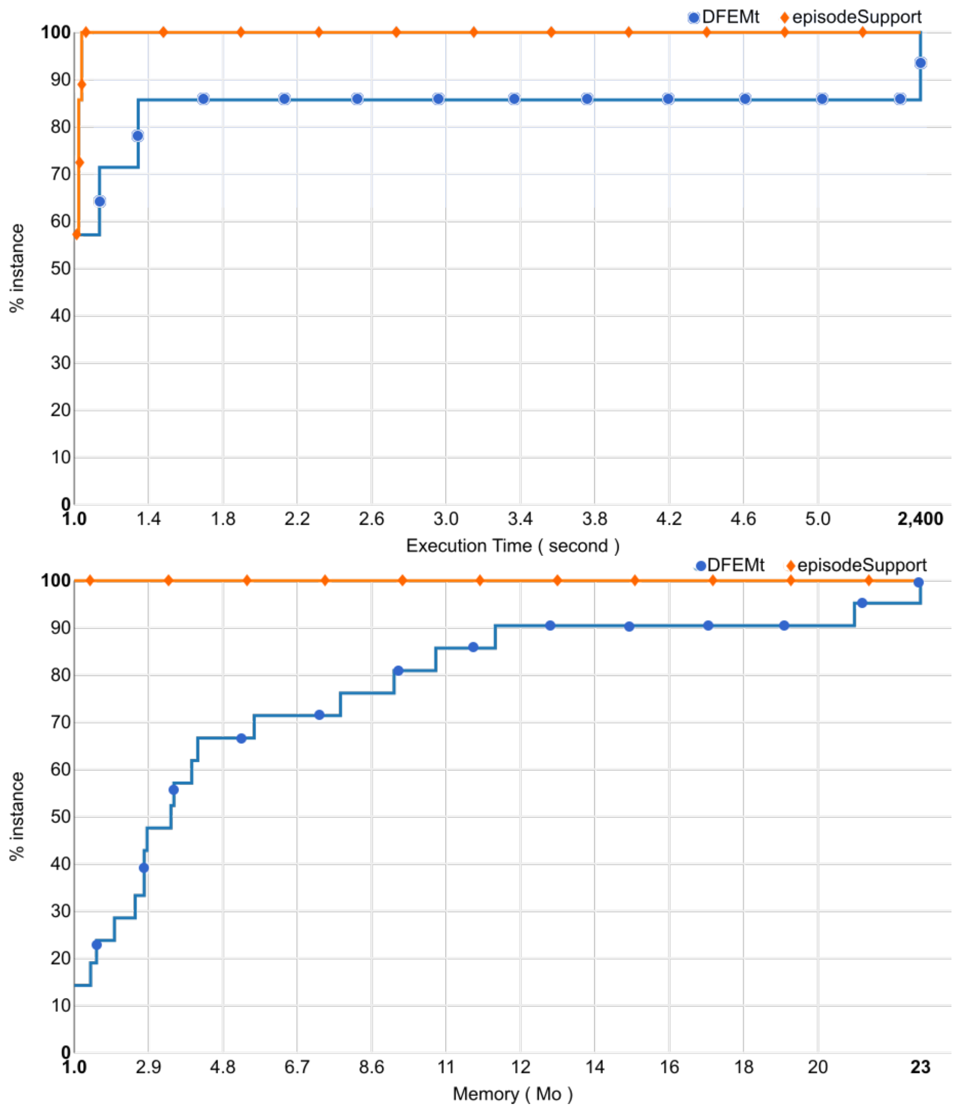


Figure 7.5. Performance profiles of execution time (**top**) and Memory usage (**bottom**) for Ubiqlog dataset (21 instances) with $gap[100, 3600]$, $span[1, 35000]$, $\theta = 5\%$, maximum size of 5 and time limit of 600 seconds.

Table 7.2. Execution time and memory usage for several timed datasets and thresholds (the gap and span constraints are: *gap*[100, 3600] and *span*[1, 35000]).

Name $\lvert s\rvert + \lvert\Sigma\rvert$	θ	nSol	Memory (Mb) DFEMt	Memory (Mb) episode Support	Execution time (s) DFEMt	Execution time (s) episode Support
10Mcomplete 1072 + 30	300	10	107	**35**	0.24	**0.20**
	100	3113	999	**469**	67.83	**29.53**
	50	1128537	67	**43**	13.29	**2.59**
	20	51108	1110	**599**	204.98	**80.93**
9Mcomplete 1128 + 45	10	6724	244	**139**	**1.00**	1.35
	8	11626	318	**173**	1.35	**0.71**
	6	19340	339	**142**	1.42	**0.86**
	4	23225	349	**138**	1.48	**1.05**
8Mcomplete 3305 + 44	300	3734	1888	**506**	197.16	**143.94**
	1632	505263	6426	**40**	1146.22	**24.20**
	50	123133	3594	**1496**	1489.18	**740.97**
	20	516478	3859	**1309**	timeout	**1163.34**

initializing the data structures with our approach is higher than the gain obtained. In general, the gain becomes more important when sequences are larger or harder to solve. Finally, given that the implementation of MINEPI+ and EMMA is missing, it is difficult to perform a fair comparison of the approaches.

7.5.4 Handling Additional Constraints

Additional constraints can be considered in order to define properties that the patterns must satisfy. No modification of `episodeSupport` is required. Results of experiments are presented in Table 7.4. The goal was to find frequent episodes ($\theta \geq 20$) having a maximum length of 6, containing at least three `Q` (`atLeast` constraint) but no `D` (`exclusion`), and satisfying the regex `M(A|T).`*`F` (`regular`). Two episodes (`MTQQQF` and `MAQQQF`) have been discovered. As observed, the additional constraints reduce the execution time as CP takes advantage of the stronger filtering to reduce the search space. This reduction would not be observed with a *generate and filter* approach.

Table 7.3. Comparison with MINEPI+ and EMMA for $\theta = 5\%$ and $W = 10$ (rows are sorted by $|s|$).

Databases Features				Execution Time (s)		
name	$\lvert s \rvert$	$\lvert\Sigma\rvert$	nSol	MINEPI+	EMMA	episodeSupport
USER3	16866	273	46	13.2	0.4	**0.173**
USER7	17329	449	25	0.6	**0.2**	0.563
USER2	18738	310	38	43.3	0.6	**0.259**
USER1	19881	288	57	93.7	1.2	**0.232**
USER5	34821	563	37	4.8	**0.3**	0.724
USER4	37817	479	48	165.3	1.3	**0.636**
USER8	54042	706	40	1362.3	9.8	**2.214**
USER6	64152	609	68	2853.3	14.6	**2.178**

Table 7.4. Additional constraints on Q08379 Protein (Uniprot).

`Only episodeSupport`		+ `exclusion`		+ `atLeast`		+ `regular`	
nSol	time(s)	nSol	time(s)	nSol	time(s)	nSol	time(s)
46,221,933	83.2	33,388,768	62.6	104,536	0.642	2	0.002

7.6 Summary, Outlooks, Further readings

There is a growing interest for solving data-mining challenges with CP. In addition to the flexibility it brings, recent works have shown that it can provide similar performances, or even better, than specialized algorithms [AGS16,AGS17]. So far CP has not been considered yet for mining frequent episodes. We introduced two global constraints (`episodeSupport`) for solving this problem with or without time-stamps. It relies on techniques used for SPM such as *pattern growth*, *pseudo projections* and *reversible vectors* but also on new ideas specific to this problem for improving the efficiency of the filtering algorithms (*early projection stopping*, *position caching* and *efficient frequency computation*). Experimental results have shown that our approach provides better performances in terms of execution time and memory consumption than state-of-the-art methods, with the additional benefits that it can accommodate additional constraints.

CHAPTER 8

Conclusion

"In literature and in life we ultimately pursue, not conclusions, but beginnings.."

–Sam Tanenhaus

The key message of this thesis is that the hybridization of techniques from several fields can lead to efficient and flexible tools. With this flexibility, one can plug humans (experts) into the loop and they can express some preferences (as Constraints) to guide the mining process. This opens up other fields of application.

8.1 Summary and Main Messages

The main motivation for this work is to *maximise the flexibility and efficiency* of CP-based approaches.

At the beginning of this thesis, in late of 2015, the use of Constraint Programming (CP) in Pattern Mining (PM) allowed developing general and flexible approaches [GNDR11, DRGN08, JSS13, MBC+11, NG15, KLL+15]. Frequent Itemset and Sequential Pattern Mining problems were addressed using CP. Several constraints were proposed, ranging from decomposition-based approaches [GNDR11, DRGN08, JSS13, MBC+11, NG15] to global constraints [NG15, KLL+15] in CP. However, the general observation was that there was always a *tradeoff between flexibility and efficiency.* The contribution of the flexibility degrades efficiency. Flexible approaches (CP-based ones) were not as (or more) efficient as specialized approaches. The question then arose: whether there was a way to find the best tradeoff between flexibility and efficiency having flexible approaches that consume less time and memory?

Through this thesis, we answer *YES* to this question. To do so, we showed one could combine the right techniques and ideas from both CP and PM communities to build new flexible and efficient methods for DM using CP.

Here are the main take away messages of this thesis, which are answers to our research questions:

Q1 Can we build new CP-based approaches that improve efficiency by combining ingredients from both Pattern Mining and Constraint Programming?

Q2 In the literature, the effectiveness of many approaches is based on the design of an appropriate data structure, can the same solution boost the effectiveness of existing CP-based methods?

Q3 Which modeling for new CP-based methods, will give a better efficiency and genericity?

Q4 How to evaluate the relevancy of the obtained patterns?

Message 1: *The combination of several ingredients can lead to new and successful approaches.* In this thesis, we showed that by combining (or adapting) properly techniques from CP and PM communities and some algorithmic knowledge, one could achieve very efficient and flexible CP-based approach for PM. In Chapter 5, thanks to the combination of:

(i) "reversible" structures and the trailing mechanism from CP [Lau18] and

(ii) the principle of (pseudo-) projected database [PHMA$^+$01] and pre-computed structure of last item positions [YK05],

we designed *PPIC* (also *PPDC* and *PPmissed*) which became the *state-of-the-art of both CP-based and specialized methods.* This result is impressive because, to the best of our knowledge, none of CP-based approaches had ever achieved this before and SPADE [Zak01] stayed the state-of-the-art for more than 15 years.

Message 2: *The design of proper data structures is often a key element in the design of new PM approaches using CP.* The first element that makes the difference in our approaches was the data structures. In chapter 3, we showed that by using the *"reversible" sparse bitset*, inspired by the data structures used in constraint tables [DHL$^+$16], we can very efficiently perform set operations (intersection, counting,...) which are several times used in the mining process.

For SPM and FEM problems (Chapters 5-7), we showed that our *backtracking-aware trailing-based data structure* is an essential ingredient for efficient management of (projected) sequence databases and memory usage during the mining process. In addition, this data structure can be used in any depth-first search.

Message 3: *Pre-computed structures allow drastically speeding up the mining process.* In chapters 5, 6 and 7, we showed that pre-computing the last position of the items, an idea introduced by [YK05], can lead to several constant-time operations. For example, in $\mathcal{O}(1)$, one can check the presence of an item in a sequence. These pre-computed structures also allow efficiently counting items, a task at the core of apriori-based methods.

Message 4: ***The choice of modelling does matter.*** In Chapter 3, we demonstrated that by exposing a variable that represents frequency, it becomes possible to solve several FIM problems. This ease the use of FIM as a *building block* in solving other problems such as finding the discriminating itemsets (Chapter 3) and, in Chapter 4, the most significant or relevant set of itemsets (rule list).

Message 5: ***The choice of a good measure is essential in the search for more relevant and interpretable approaches.*** In Chapter 4, we showed that by combining several itemsets, we can build a new rule list that captures as much information as possible based on the Minimum description length (MDL) principle (compression of data). Thanks to this principle, one can iteratively build *small-and-good* rule lists using branch-and-bound.

8.2 Discussion and perspectives

In this section, we discuss the perspectives for this research.

Perspective 1: ***Quality of patterns.*** The most important concern in pattern mining is the quality of the patterns. A lot of effort is focused on defining what a “good” pattern is and how to get it. This hot topic opens up many interesting scientific perspectives. We stick to this philosophy in the perspectives we propose here.

In Chapter 3, we proposed a model of frequent itemset problems that allowed outputting patterns which were interesting in terms of their discriminating power. We believe this is an exciting direction to explore in depth. We can highlight two paths to investigate.

The first one is a **modeling effort**. The question is *how to model pattern mining tasks in order to solve several problems?* The CoverSize constraint presented in Chapter 3 is an attempt to answer this question. The model of CoverSize consists of exposing cover size as a variable. This way, we can solve Frequent Itemset Mining (FIM), Closed FIM, and Discriminative FIM problems efficiently. Modeling Pattern-set Mining problem was investigated in Guns et al. [GNDR13]. This work showcases different constraints which can be used to solve number of mining problems. They exemplified their modeling with the concept learning problem [GNR11]. Several other applications are also possible in the same direction. MiningZinc [GDT^{+}13], a language for modeling constraint-based pattern mining problems, is another example of modeling. More generally, many supervised and unsupervised learning problems based on patterns such as the concept learning, rule learning, decision trees amount to parameter learning; where parameters represent decisions to include (or not) a feature (item) in patterns of pattern-set solution. Efforts in that direction can allow solving several problems.

The second track is the **consideration of the continuous aspect of real problems in the modeling of problems in Constraint Programming**. The optimization criteria for DFIM (Information gain, χ^2, Gini index, etc) and rule learning (Minimum Description Length score) problems, presented in Chapters 3 and 4, are examples of functions expressed in a continuous domain. The ZDC constraint presented in Algorithm 3.4 (Chapter 3), is a starting point in this type of modeling of continuous functions in discrete domains. A more global approach, in this direction, is the complete modeling of pattern-based learning tasks as parameter learning problems in discrete domains; similar to neural networks in continuous domains.

It is worth noting that the majority of approaches to finding "interesting" solution are based on pattern-set mining [VT14, ZAV14, ZN14]. Effective pattern-set problem modeling itself is a challenging task. Pattern-set modeling in CP, developed by Guns et al. [GNR11], is based on repeating the modeling of mining a pattern at the size of the pattern-set plus the constraints to link the patterns in the pattern-set to each other. This approach is not very efficient in practice, mainly due to unavoidable sum constraints. We have shown in this thesis that the use of a global constraint and the reduction of Boolean variables (in Chapter 3) can boost overall efficiency.

Perspective 2: *Extension of our work* Given the encouraging work obtained in the fields of Frequent Itemset Mining and Sequential Pattern Mining, it is interesting to consider **building constraints in other mining problems** such as tree mining, graph mining, etc.

In Chapter 4, we used an MDL score to find *small-and-good* rule list. This work can be extended to two aspects. The MDL score can be improved because the two-part version is a basic one. The **more sophisticated scores**, with more gain, are possible. Namely, the Normalised Maximum Likelihood (NML) and the Prequential Plug-In Model can yield models with higher prediction power [Grü07]. Besides, one can investigate a **tighter lower-bound**. We believe that one way of doing this is to find the minimal increase in the local code length obtained by the new itemset to add in the rule list plus the default rule (see (4.7) in Chapter 4).

With regard to the problems of frequent sequence mining discussed in Chapters 5, 6 and 7, a number of future directions is possible.

The type of patterns found by our approach are the commonly used sequences of single events. We do not consider **other related pattern types such as sequences of sets of events** (sequences of itemsets), multivariate temporal patterns [BFH$^+$12]. These require changes to the pattern representation and the embedding (constraint). One could build on the principles and data structures investigated in this paper for those purposes.

It should also be pointed out that in the literature, to the best of our knowledge, **the problems of closed, free, and discriminative frequent sequences have not been addressed using Constraint Programming**. These problems are worth investigating because they are commonly used as a building block in finding interesting patterns [AMS$^+$96, VvLS11, VT14].

Putting all together, throughout this thesis four key ingredients contributed to maximizing the flexibility and effectiveness of frequent itemset and sequence mining tasks using the Constraint Programming (CP) paradigm: Global constraints, appropriate data structures, pre-computed data, and suitable models. As a result, we were able to prove that by combining these ingredients, we can efficiently (in time and memory consumption) enumerate frequent sequences and episodes in various sequence databases (single and long sequence, sequence database with or without timestamps) being flexible. Those approaches become state-of-the-art in sequence mining. For itemset mining problems, this efficiency is undoubtedly reduced, but we are still behind the specialized methods. Our CP-based approaches compared to other existing CP-based approaches, are more efficient but often at the cost of sacrificing a bit of flexibility. It is therefore clear that a further step has been taken in the use of CP in Pattern Mining. However, there is still a long way to go because many questions remain, especially concerning the quality of the patterns obtained. It is not trivial to incorporate quality measurements (which are often continuous functions, see Chapter 3 and 4) as constraints to ensure relevancy and non-redundancy on the obtained patterns, given that CP operates in discrete domains. My conviction is that further efforts are needed in this direction implementing new constraints which can guide the mining process to relevant patterns. Besides, the implementation of new applications might allow us to face real constraints.

Appendix A

Algorithms

A.1 Example of Implementation of Discriminative FIM in OscaR

Listing A.1. DFIM Runner

```
package fim.examples
import java.io.File
import fim.constraints._
import fim.utils.ZDCScaled
import oscar.cp._
import scala.io.Source

object DiscriminativeCoverSizeZDCRunner extends App {

  printHead()

  //parser
  val parser = argsParser()

  //parsing
  parser.parse(args, Config()) match {
    case Some(config) =>

      //initialisation
      System.err.println("Start␣CoverZize+ZDC␣(Optimal␣
         Discriminative␣FIM)␣on␣" + config.tdbFile.
  getAbsolutePath)

      //get data from file
      val fileLines = Source.fromFile(config.tdbFile.getAbsolutePath).getLines

      //set horizontal dataset with the positive and negative datasets
      val tdbHorizontal: Array[Array[Int]] = fileLines.map { line => line.mkString.split("\\s+").map(_.toInt) }.toArray
      val tdbHorizontalP: Array[Array[Int]] = tdbHorizontal.filter(_.last == 1).map(line => line.dropRight(1))
      val tdbHorizontalN: Array[Array[Int]] = tdbHorizontal.filter(_.last == 0).map(line => line.dropRight(1))
      val hTdb: Array[Array[Int]] = tdbHorizontal.map(line => line.dropRight(1))

      //transform horizontal to vertical hTdb -> vTdb
      val max: Int = tdbHorizontal.map(_.max).max
      val tdbVerticalP: Array[Set[Int]] = Array.tabulate(max + 1)(i => tdbHorizontalP.indices.filter(t => tdbHorizontalP(t).contains(i)).toSet)
      val tdbVerticalN: Array[Set[Int]] = Array.tabulate(max + 1)(i => tdbHorizontalN.indices.filter(t => tdbHorizontalN(t).contains(i)).toSet)
      val vTdb: Array[Set[Int]] = Array.tabulate(max + 1)(i => hTdb.indices.filter(t => hTdb(t).contains(i)).toSet)

      println("------start------")
      //initialize size of dataset (Positive -> P, Negative -> N)
      val nTrans = tdbHorizontal.length
      val nTransP = tdbHorizontalP.length
      val nTransN = tdbHorizontalN.length
      val nItems = max + 1

      // -------------------------------

      implicit val cp = CPSolver()

      ////Variables I , P, N, Minsup
      val I = Array.fill(nItems)(CPBoolVar())
      var P = CPIntVar(0 to nTransP)
      var N = CPIntVar(0 to nTransN)

      var score = CPIntVar(0 to 1000000)

      ////BEGIN
      val zdc = new ZDCScaled(new fim.utils.InfGain(),
         100000)

      ///Create Constraints

      //--//MAIN: (CoverSize-) + (CoverSize+) + (ZDC)
      val consP = new CoverSizeVar(I, P, nItems, nTransP,
         tdbVerticalP)
      val consN = new CoverSizeVar(I, N, nItems, nTransN,
         tdbVerticalN)

      ///Post it
      add(consP)
      add(consN)

      //COP, optimum
      maximize(score)

      //Discriminative eval function, look utils/
         ZDCFunction for the availability functions
      val zdcConstraint = new ZDC(P, N, nTransP, nTransN,
         zdc, score)
      add(zdcConstraint)

      val Isorted: Array[CPIntVar] = I.indices.sortBy(vTdb
         (_).size).map(I(_)).toArray[CPIntVar]


      search {
        if (config.algo == 1) {
          println("With␣Binary␣Static␣Search␣")
          binaryStatic(Isorted, _.min)
        }
        else {
          println("With␣Conflict␣Search␣")
          new MyConflictOrderingSearch(Isorted, i => i, i
             => I(i).min)
        }
      }


      onSolution {
        println("items:" + (0 until nItems).filter(I(_).
           isTrue) + s"␣pos:$P/$nTransP␣neg:$N/$nTransN␣
           score:" + score)
      }

      val stat = start()
      println(stat)

    case None =>
    // parser.showUsage
    // arguments are bad, error message will have been
       displayed
  }

  }
```

A.2 Reversible sparseBitSet data structure

Listing A.2. RSparseBitSet

```
/**
  * Data structure (Reversible Sparse BitSet)
  *
  * @author John Aoga johnaoga@gmail.com
  * @author Pierre Schaus pschaus@gmail.com
  *         Relevant paper: Compact table and http://
     becool.info.ucl.ac.be/biblio/coversize-global-
     constraint-frequency-based-itemset-mining
  *
  */

package fim.utils

```

```
object BitSetOp2 {

  def bitLength(size: Int): Int = (size + 63) >>> 6

  // = size / 64 + 1
  def oneBitLong(pos: Int): Long = 1L << pos

  // = pos / 64 (64 = 2^6)
  def bitOffset(pos: Int): Int = pos >>> 6

  // = pos % 64
  def bitPos(pos: Int): Int = pos & 63

  // = pos % 63
  def setBit(bitSet: Array[Long], pos: Int): Unit = {
    bitSet(bitOffset(pos)) |= oneBitLong(bitPos(pos))
  }

}


import oscar.algo.reversible.BitSetOp._
import oscar.algo.reversible.{ReversibleContext,
    TrailEntry}

/* Trailable entry to restore the value of the ith Long of
     the valid tuples */
final class ReversibleSparseBitSetEntry(set:
    ReversibleSparseBitSet2, numberOfValues: Int) extends
    TrailEntry {
  @inline override def restore(): Unit = set.restore(
      numberOfValues)
}

/**
  * A reversible set with an internal bit-set
      representation.
  * This set can remove efficiently its elements from
      another bit-set
  * This set can compute efficiently its intersection with
       another bit-set
  *
  * @param context
  * @param n initial values must be taken from {0,...,n-1}
  * @param initialValues the initial values contained in
      the set
  * @author Pierre Schaus pschaus@gmail.com
  */
class ReversibleSparseBitSet2(val context:
    ReversibleContext, val n: Int, val initialValues:
    Iterable[Int]) {

  /**
    * Immutable bit-set that can be used to remove/
        intersect
    * with the the ReversibleSparseBitSet
    *
    * @param values initial values, they must be in
        {0,...,n-1}
    */
  class BitSet(values: Iterable[Int]) {
    protected[ReversibleSparseBitSet2] var lastSupport = 0
    protected[ReversibleSparseBitSet2] var words: Array[
        Long] = Array.fill(nWords)(0L)
    assert(values.forall(v => v < n && v >= 0))
    values.foreach(v => setBit(words, v))

    def &=(bs: BitSet) = {
      var i = words.length
      while (i > 0) {
        i -= 1
        words(i) = words(i) & bs.words(i)
      }
    }

    def &~=(bs: BitSet) = {
      var i = words.length
      while (i > 0) {
        i -= 1
        words(i) = words(i) & ~bs.words(i)
      }
    }

    def isZero: Boolean = {
      words.forall(_ == 0)
    }

    def |=(bs: BitSet) = {
      var i = words.length
      while (i > 0) {
        i -= 1
        words(i) = words(i) | bs.words(i)
      }
    }

    val mask: Long = ~0L >>> (64 - (n % 64))
    def unary_~ = {
      var i = words.length
      while (i > 0) {
        i -= 1
        words(i) = ~words(i)
      }
      words(words.length - 1) = words(words.length - 1) &
          mask
    }

    def indexOfFirstNonZero: Int = {
      var i = 0
      while (i < words.length && words(i) == 0) {
        i += 1
      }
      (i + 1) * 64 - java.lang.Long.numberOfLeadingZeros(
          words(i)) - 1
    }

    override def toString: String = {
      val size = n min 64
      words.map(e => String.format(s"%${size}s", java.lang
          .Long.toBinaryString(e)).replace('␣', '0')).
          mkString("␣")
    }
  }

  private[this] var timeStamp = -1L

  /* Compute number of Long in a bitset */
  private[this] var nWords = bitLength(n)

  private[this] val words: Array[Long] = Array.fill(nWords
      )(0L)
  private[this] val lastMagics = Array.fill(nWords)(-1L)
  private[this] var nonZeroIdx: Array[Int] = Array.
      tabulate(nWords)(i => i)
  private[this] var nNonZero: Int = nWords
  private[this] val tempMask = Array.fill(nWords)(0L)

  assert(initialValues.forall(v => v < n && v >= 0))
  initialValues.foreach(v => setBit(words, v))

  private[this] var innerTrailSize = 1000
  private[this] var nTrailEntries = 0
  private[this] var wordIndex = Array.ofDim[Int](
      innerTrailSize)
  private[this] var wordValue = Array.ofDim[Long](
      innerTrailSize)

  @inline private[this] def growInnerTrail(): Unit = {
    val newWordIndex = new Array[Int](innerTrailSize * 2)
    val newWordValue = new Array[Long](innerTrailSize * 2)
    System.arraycopy(wordIndex, 0, newWordIndex, 0,
        innerTrailSize)
    System.arraycopy(wordValue, 0, newWordValue, 0,
        innerTrailSize)
    wordIndex = newWordIndex
    wordValue = newWordValue
    innerTrailSize *= 2
  }

  // Remove the zero words from sparse set
  var i: Int = nNonZero
  while (i > 0) {
    i -= 1
    if (words(nonZeroIdx(i)) == 0L) {
      nNonZero -= 1
      nonZeroIdx(i) = nonZeroIdx(nNonZero)
      nonZeroIdx(nNonZero) = i
    }
  }

  def isEmpty(): Boolean = {
    nNonZero == 0
  }

  override def toString(): String = {
    val size = n min 64

    def format(l: Long) = String.format(s"%${size}s", java
        .lang.Long.toBinaryString(l)).replace('␣', '0')

    "NonZeroWords:" + nNonZero + "␣words:" + words.map(
        format(_)).mkString("␣,␣")
  }

  @inline final def restore(numberOfValues: Int): Unit = {
    var k = numberOfValues
    while (k > 0) {
      var pos = nTrailEntries - k
      words(wordIndex(pos)) = wordValue(pos)
      k -= 1
    }
    nTrailEntries -= numberOfValues
    nNonZero = numberOfValues
  }

  private[this] def trail(): Unit = {
    while (nTrailEntries + nNonZero > innerTrailSize)
        growInnerTrail()
    var i: Int = nNonZero
    while (i > 0) {
      i -= 1
      val offset = nonZeroIdx(i)
      val word = words(offset)
      wordIndex(nTrailEntries) = offset
      wordValue(nTrailEntries) = word
      nTrailEntries += 1
    }
    val trailEntry = new ReversibleSparseBitSetEntry(this,
        nNonZero)
    context.trail(trailEntry)
  }

  /**
    * Clear all the collected elements
    */
```

```
def clearCollected(): Unit = {
  var i: Int = nNonZero
  while (i > 0) {
    i -= 1
    tempMask(nonZeroIdx(i)) = 0L
  }
}

/**
  * Add the elements in set in the set of collected
     elements
  * to be used with a subsequent intersectCollected() or
      removeCollected() operation
  *
  * @param set
  */
def collect(set: BitSet): Unit = {
  var i: Int = nNonZero
  while (i > 0) {
    i -= 1
    val offset = nonZeroIdx(i)
    tempMask(offset) |= set.words(offset)
  }
}

def intersectWith(set: BitSet): Boolean = {
  if (context.magic != timeStamp) {
    trail()
    timeStamp = context.magic
  }
  var changed = false
  var i: Int = nNonZero
  while (i > 0) {
    i -= 1
    val offset = nonZeroIdx(i)
    val oldLong: Long = words(offset)
    val setLong: Long = set.words(offset)
    val newLong: Long = (if (setLong != 0L) oldLong &
        setLong else 0L)
    words(offset) = newLong
    /* Remove the word from the sparse set if equal to 0
        */

    if (newLong == 0L) {
      nNonZero -= 1
      nonZeroIdx(i) = nonZeroIdx(nNonZero)
      nonZeroIdx(nNonZero) = offset
    }
    changed |= oldLong != newLong
  }
  changed
}

/**
  * @param set
  * @return the number of bits of the intersection of
     the bitSet and this
  */
def intersectCount(set: BitSet): Int = {
  var count = 0
  var i: Int = nNonZero
  while (i > 0) {
    i -= 1
    val offset = nonZeroIdx(i)
    count += java.lang.Long.bitCount(words(offset) & set
        .words(offset))
  }
  count
}

def intersectCount(set: BitSet, minsup: Int): Int = {
  var count = 0
  //Constantes.nLoops += nNonZero
  var i: Int = nNonZero
  while (i > 0 && i * 64 >= minsup - count) {
    //Constantes.curSupport += 1
    i -= 1
    val offset = nonZeroIdx(i)
    val setLong: Long = set.words(offset)
    if (setLong != 0L) {
     count += java.lang.Long.bitCount(words(offset) &
         setLong)
    }
  }
  count
}

def intersectCountAll(sets: Array[BitSet], itemIdx:
    Array[Int], limit: Int): Int = {
  var count = 0
  var i: Int = nNonZero
  while (i > 0) {
    i -= 1
    val offset = nonZeroIdx(i)
    var j = limit
    var myWord: Long = ~0L
    while (j > 0 && myWord != 0) {
     j -= 1
     myWord = myWord & sets(itemIdx(j)).words(offset)
    }
    if (myWord != 0L) count += java.lang.Long.bitCount(
        words(offset) & myWord)
  }
  count
}

def count(): Int = {
  var count = 0
  var i: Int = nNonZero
  while (i > 0) {
    i -= 1
    val offset = nonZeroIdx(i)
    count += java.lang.Long.bitCount(words(offset))
  }
  count
}

def isSubSetOf(set: BitSet): Boolean = {
  var i: Int = nNonZero
  while (i > 0) {
    i -= 1
    val offset = nonZeroIdx(i)
    if ((words(offset) & ~set.words(offset)) != 0L) {
     return false
    }
  }
  return true
}
}
```

A.3 Frequent Itemset Mining: CoverSize

Listing A.3. CoverSize

```
package fim.constraints

import fim.utils.ReversibleSparseBitSet2
import oscar.algo.reversible._
import oscar.cp._
import oscar.cp.core.CPPropagStrength

/**
  *
  * CoverSize with Support as a CP variable (same for all
     contraints end with Var)
  *
  * @author John Aoga johnaoga@gmail.com
  * @author Pierre Schaus pschaus@gmail.com
  *          Relevant paper: Compact table and http://
     becool.info.ucl.ac.be/biblio/coversize-global-
     constraint-frequency-based-itemset-mining
  *
  *          PART OF SOLVER OSCAR (https://bitbucket.org/
     oscarlib/oscar/wiki/Home)
  */
class CoverSizeVar(val I: Array[CPBoolVar], Sup: CPIntVar,
    val nItems: Int, val nTrans: Int, TDB: Array[Set[Int
   ]]) extends Constraint(I(0).store, "CoverSizeVar") {

  private[this] val coverage = new ReversibleSparseBitSet2
     (s, nTrans, 0 until nTrans)
  ///Create matrix B (nItems x nTrans) (i = item, j =
     transaction)
  //Is such that columns(i) is the coverage of item i.
  private[this] val columns = Array.tabulate(nItems) { x
     => new coverage.BitSet(TDB(x)) }
  ///contains all the unbound variables that are not in
     the closure of the current itemset.
  //closure => freq(I(D)U{i}) = freq(I(D))
  private[this] val unboundNotInClosure = Array.tabulate(I
     .length)(i => i)
  private[this] val nUnboundNotInClosure = new
      ReversibleInt(s, I.length)
  private[this] val updateSupLB = new ReversibleBoolean(s,
      true)
  private[this] var supLB = 0

  /**
    *
    * @param l
    * @return CPOutcome state
    */
  override def setup(l: CPPropagStrength): Unit = {
    for (i <- 0 until nItems; if !I(i).isBound) {
      I(i).callPropagateWhenBind(this)
    }
    if (!Sup.isBound) Sup.callPropagateWhenBoundsChange(
        this)
    propagate()
  }

  /**
    *
    * @return CPOutcome state
    */
  override def propagate(): Unit = {
    coverage.clearCollected()
    var coverChanged = false

    // update the coverage
    var nU = nUnboundNotInClosure.value
    var i = nU
    while (i > 0) {
      i -= 1
      val idx = unboundNotInClosure(i)
      if (I(idx).isBound) {
        nU = removeItem(i, nU, idx)
        //when bound to 1, then idx in coverage, make
            intersection
        if (I(idx).min == 1) {
          coverChanged |= coverage.intersectWith(columns(
              idx))
        } else {
          // need to recompute upper bound only if one
              item is newly set to 0
          updateSupLB.value = true
        }
      }
    }

    // remove items that if included induce a too small
        coverage
    i = nU
    while (i > 0) {
      i -= 1
      val idx = unboundNotInClosure(i)
      //cond1: frequency condition
      if (coverage.intersectCount(columns(idx)) < Sup.min)
          {
        nU = removeItem(i, nU, idx)
        I(idx).assignFalse()
        updateSupLB.value = true
      }
    }

    val supUB = coverage.count()
    Sup.updateMax(supUB)
    // update the Sup.min by intersect all the remaining
        unbound items
    //if (updateSupLB.value) {
    supLB = coverage.intersectCountAll(columns,
        unboundNotInClosure, nU)
    Sup.updateMin(supLB)
    updateSupLB.value = false
    //}

    if (supUB == Sup.min) {
      // remove all unbound items that are not super sets
          of coverage
      // because these would necessarily decrease the
          coverage
      i = nU
      while (i > 0) {
        i -= 1
        val idx = unboundNotInClosure(i)
        if (!coverage.isSubSetOf(columns(idx))) {
          nU = removeItem(i, nU, idx)
          I(idx).assignFalse()
          updateSupLB.value = true
        }
      }
    }
    nUnboundNotInClosure.value = nU
  }

  /**
    *
    * @param item
    * @param nU    the number of not unbound item which
        are not in the current closure
    * @param index the index of current item
    * @return
    */
  def removeItem(item: Int, nU: Int, index: Int): Int = {
    val lastU = nU - 1
    unboundNotInClosure(item) = unboundNotInClosure(lastU)
    unboundNotInClosure(lastU) = index
    lastU
  }
}
```

A.4 Sequential Pattern Mining: PPIC

Listing A.4. PPIC

```
package cp4d.spm.constraints

import oscar.algo.reversible.{ ReversibleInt}
import oscar.cp.core.CPOutcome._
import oscar.cp.core._
import oscar.cp.core.variables._

/**
  * PPIC [Constraint Programming & Sequential Pattern
      Mining with Prefix projection method]
  * is the CP version of Prefix projection method of
      Sequential Pattern Mining (with several improvements
    )
 * which is based on projected database (We use here
    pseudo-projected-database \{ (sid, pos) \}).
 *
 * This constraint generate all available solution given
    such parameters
 *
 * @param P, is pattern where $P_i$ is the item in
    position $i$ in $P$
 * @param SDB, [sequence database] it is a set of
    sequences. Each line $SDB_i$ or $t_i$ represent a
    sequence
 *           s1 abcbc
 *           s2 babc
 *           s3 ab
 *           s4 bcd
 *
 * @param lastPosOfItem is the last real position of an
    item in a sequence, if 0 it is not present
 *                         s1  s2  s3  s4
 *                      a  1   2   1   0
 *                      b  4   3   2   1
 *                      c  5   4   0   2
 *                      d  0   0   0   3
 * @param itemsSupport is the initial support (number of
    sequences where a item is appeared) of all items
 *                       a : 3, b : 4, c : 3, d : 1
 * @param minsup is a threshold support, item must appear
     in at least minsup sequences $support(item)>=
    minsup$
 * @param nItems is the number of items in SDB
 *
 * @author John Aoga (johnaoga@gmail.com) and Pierre
    Schaus (pschaus@gmail.com)
 */

// VSDB = vertical database
class PPIC(val P: Array[CPIntVar], val SDB: Array[Array[
    Int]], val SDBlastPos: Array[Array[Int]], val
    firstPosOfItem: Array[Array[Int]], val lastPosOfItem :
     Array[Array[Int]], val itemsSupport : Array[Int], val
     minsup : Int, val nItems : Int) extends Constraint(P
    (0).store, "PPIC"){
```

```

  idempotent = true

  private[this] val epsilon = 0  //this is for empty item
  private[this] val lenSDB = SDB.size
  private[this] val patternSeq = P.clone()
  private[this] val lenPatternSeq = P.length

  ///representation of pseudo-projected-database
  private[this] var innerTrailSize = lenSDB*5
  private[this] var psdbSeqId = Array.tabulate(
      innerTrailSize)(i => i) //the No of Sequence (sid)
  private[this] var psdbPosInSeq = Array.tabulate(
      innerTrailSize)(i => -1) //position of prefix in
      this sid
  private[this] val psdbStart = new ReversibleInt(s,0)  //
      current size of trail
  private[this] val psdbSize = new ReversibleInt(s,lenSDB)
       //current position in trail

  ///when InnerTrail is full, it allows to double size of
      trail
  @inline private def growInnerTrail(): Unit = {
    val newPsdbSeqId = new Array[Int](innerTrailSize*2)
    val newPsdbPosInSeq = new Array[Int](innerTrailSize*2)
    System.arraycopy(psdbSeqId, 0,newPsdbSeqId, 0,
        innerTrailSize)
    System.arraycopy(psdbPosInSeq, 0,newPsdbPosInSeq, 0,
        innerTrailSize)
    psdbSeqId = newPsdbSeqId
    psdbPosInSeq = newPsdbPosInSeq
    innerTrailSize *= 2
  }

  ///support counter contain support for each item, it is
      reversible for efficient backtrack
  private[this] var supportCounter = itemsSupport
  var curPrefixSupport : Int = 0

  ///current position in P $P_i = P[curPosInP.value]$
  private[this] val curPosInP = new ReversibleInt(s,0)

  ///check if pruning is done successfully
  private[this] var pruneSuccess = true

  /**
    * Entry in constraint, function for all init
    * @param l
    * @return The outcome of the first propagation and
        consistency check
    */
  final override def setup(l: CPPropagStrength): CPOutcome
      = {
    if (propagate() == Failure) Failure
    else {
      var i = patternSeq.length
      while (i > 0) {
        i -= 1
        patternSeq(i).callPropagateWhenBind(this)
      }
      Suspend
    }
  }

  /**
    * propagate
    * @return the outcome i.e. Failure, Success or Suspend
    */
  final override def propagate(): CPOutcome = {
    var v = curPosInP.value

    if (P(v).isBoundTo(epsilon)) {
      if (!P(v-1).isBoundTo(epsilon)) {
        enforceEpsilonFrom(v)
      }
      return Success
    }

    while (v < P.length && P(v).isBound && P(v).min !=
        epsilon) {
      if(!filterPrefixProjection(P(v).getMin)) return
          Failure
      curPosInP.incr()
      v = curPosInP.value
    }

    if (v > 0 && v < P.length && P(v).isBoundTo(epsilon)) {
      enforceEpsilonFrom(v)
    }
    Suspend
  }

  /**
    * when $P_i = epsilon$, then $P_i+1 = epsilon$
    * @param i current position in P
    */
  def enforceEpsilonFrom(i: Int): Unit = {
    var j = i
    while (j < lenPatternSeq) {
      P(j).assign(epsilon)
      j += 1
    }
  }

  /**
    * P[curPosInP.value] has just been bound to "prefix"
    *  all the indices before (< currPosInP) are already bound
    *
    *  if prefix is not epsilon we can compute next pseudo-projected-database
    *  with projectSDB function
    *
    * @param prefix
    * @return the Boolean is to say if current prefix is a solution or not
    */
  private def filterPrefixProjection(prefix : Int) : Boolean = {
    val i = curPosInP.value + 1
    //////println("filter prefix "+prefix+" at position "+i)
    if (i >= 2 && prefix == epsilon) { return true }
    else {
      val sup = projectSDB(prefix)

      if ( sup < minsup) { return false }
      else {
        pruneSuccess = true
        ///Prune next position pattern P domain if it
            exists unfrequent items
        prune(i)
        return pruneSuccess
      }
    }
  }

  ///initialisation of domain
  val dom = Array.ofDim[Int](nItems)

  /**
    * pruning strategy
    * @param i current position in P
    */
  private def prune(i : Int) : Unit = {
    val j = i
    if (j >= lenPatternSeq) return
    var k = 0
    val len = P(j).fillArray(dom)
    while (k < len){
      val item = dom(k)
      if ( item != epsilon && supportCounter(item) <
          minsup ) {
        if(P(j).removeValue(item) == Failure) {
          pruneSuccess = false
          return
        }
      }
      k += 1
    }
  }

  /**
    * Computing of next pseudo projected database
    * @param prefix
    * @return
    */
  private def projectSDB(prefix : Int) : Int = {
    val startInit = psdbStart.value
    val sizeInit = psdbSize.value

    //Count sequences validated for next step
    curPrefixSupport = 0

    //allow to predict failed sid (sequence) and remove it
    val nbAddedTarget = itemsSupport(prefix)

    //reset support to 0
    supportCounter = Array.fill[Int](nItems)(0)

    //
    var i = startInit
    var j = startInit + sizeInit
    var nbAdded = 0
    //////println("startInit = "+startInit+" sizeInit = "+
       sizeInit)
    // Tias optim: nbAdded < nbAddedTarget
    // because we know how many need to be added so we can
        stop when this target is reached
    while (i < startInit + sizeInit && nbAdded <
       nbAddedTarget) {

      val sid = psdbSeqId(i)
      val ti = SDB(sid)
      val lti = ti.length
      val start = psdbPosInSeq(i)
      var pos = start

      if (lastPosOfItem(sid)(prefix) != 0) {
        // here we know at least that prefix is present in
            sequence sid

        // search for next value "prefix" in the sequence
            starting from
        if (lastPosOfItem(sid)(prefix) - 1 >= pos) {
          // we are sure prefix next position is available
              and so we add the sequence in the new
             projected data base
          nbAdded += 1

          // find next position of prefix
          if (start == -1) { pos = firstPosOfItem(sid)(
             prefix) - 1}
            else {while (pos < lti && prefix != ti(pos)) {
                pos += 1}}
            //update pseudo projected database and support
            psdbSeqId(j) = sid
            psdbPosInSeq(j) = pos + 1
            j += 1
            if (j >= innerTrailSize) growInnerTrail()

            curPrefixSupport += 1

            //recompute support
            var break = false
            val tiLast = SDBlastPos(sid)

            var c = 0
            while ( tiLast(c)-1 > pos ) {
              supportCounter ( ti(tiLast(c)-1) )+= 1
              c += 1
            }
          }
        }

        i += 1
      }

      psdbStart.value = startInit + sizeInit
      psdbSize.value = curPrefixSupport

      return curPrefixSupport
  }
}
```

Bibliography

[] *Scala standard library 2.11.6*, http://www.scala-lang.org/api/current/index.html#scala.Long.

[ADF16] John O. R. Aoga, Théophile K. Dagba, and Codjo C. Fanou, *Integration of yoruba language into marytts*, I. J. Speech Technology **19** (2016), no. 1, 151–158.

[AFGY02] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu, *Sequential pattern mining using a bitmap representation*, Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada, ACM, 2002, pp. 429–435.

[Agg14a] CC Aggarwal, *An introduction to frequent pattern mining*, pp. 1–17, Springer, 2014.

[Agg14b] Charu C. Aggarwal, *Applications of frequent pattern mining*, in Aggarwal and Han [AH14], pp. 443–467.

[AGNS18] John O. R. Aoga, Tias Guns, Siegfried Nijssen, and Pierre Schaus, *Finding probabilistic rule lists using the minimum description length principle*, Discovery Science - 21st International Conference, DS 2018, Limassol, Cyprus, October 29-31, 2018, Proceedings (Larisa N. Soldatova, Joaquin Vanschoren, George A. Papadopoulos, and Michelangelo Ceci, eds.), Lecture Notes in Computer Science, vol. 11198, Springer, 2018, pp. 66–82.

[AGS16] John O. R. Aoga, Tias Guns, and Pierre Schaus, *An efficient algorithm for mining frequent sequence with constraint programming*, Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19-23, 2016, Proceedings, Part II (Paolo Frasconi, Niels Landwehr, Giuseppe Manco, and Jilles Vreeken, eds.), Lecture Notes in Computer Science, vol. 9852, Springer, 2016, pp. 315–330.

[AGS17] ———, *Mining time-constrained sequential patterns with constraint programming*, Constraints **22** (2017), no. 4, 548–570.

[AH14] Charu C. Aggarwal and Jiawei Han (eds.), *Frequent pattern mining*, Springer, 2014.

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami, *Mining association rules between sets of items in large databases*, International Conference on Management of Data (SIGMOD) **22** (1993), no. 2, 207–216.

[AMS$^+$96] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al., *Fast discovery of association rules.*, Advances in knowledge discovery and data mining **12** (1996), no. 1, 307–328.

[AO03] Cláudia Antunes and Arlindo L. Oliveira, *Generalization of pattern-growth methods for sequential pattern mining with gap constraints*, Machine Learning and Data Mining in Pattern Recognition: 3rd International Conference, MLDM 2003 Leipzig, Germany, July 5–7, 2003 Proceedings (Berlin, Heidelberg) (Petra Perner and Azriel Rosenfeld, eds.), Springer Berlin Heidelberg, 2003, pp. 239–251.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant, *Mining sequential patterns*, Data Engineering, 1995. Proceedings of the Eleventh International Conference on, IEEE, 1995, pp. 3–14.

[BC94] Nicolas Beldiceanu and Evelyne Contejean, *Introducing global constraints in chip*, Mathematical and computer Modelling **20** (1994), no. 12, 97–123.

[BCG01] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke, *MAFIA: A maximal frequent itemset algorithm for transactional databases*, Data Engineering, 2001. Proceedings. 17th International Conference on, IEEE, 2001, pp. 443–452.

[BFH$^+$12] Iyad Batal, Dmitriy Fradkin, James Harrison, Fabian Moerchen, and Milos Hauskrecht, *Mining recent temporal patterns for event detection in multivariate time series data*, Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2012, pp. 280–288.

[BH03] Christian Bessière and Pascal Van Hentenryck, *To be or not to be ... a global constraint*, Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings (Francesca Rossi, ed.), Lecture Notes in Computer Science, vol. 2833, Springer, 2003, pp. 789–794.

[BL04] F. Bonchi and C. Lucchese, *On closed constrained frequent pattern mining*, ICDM '04. Fourth IEEE International Conference on Data Mining, Nov 2004, pp. 35–42.

[Bor03] Christian Borgelt, *Efficient implementations of Apriori and Eclat*, FIMI: Workshop on Frequent Itemset Mining Implementations, 2003.

[Bor12] ______, *Frequent item set mining*, Wiley Interdisciplinary Reviews: Data Mining and Knowl. Discovery **2** (2012), no. 6, 437–456.

[BR97] Christian Bessiere and Jean-Charles Régin, *Arc consistency for general constraint networks: preliminary results*, International Joint Conference on Artificial Intelligence (IJCAI), 1997.

[BRY98] Andrew R. Barron, Jorma Rissanen, and Bin Yu, *The minimum description length principle in coding and modeling*, IEEE Trans. Information Theory **44** (1998), no. 6, 2743–2760.

[BZ05] Björn Bringmann and Albrecht Zimmermann, *Tree 2–Decision Trees for tree structured data*, European Conference on Principles of Data Mining and Knowledge Discovery, Springer, 2005, pp. 46–58.

[BZDRN06] Björn Bringmann, Albrecht Zimmermann, Luc De Raedt, and Siegfried Nijssen, *Don't be afraid of simpler patterns*, PKDD, vol. 4213, Springer, 2006, pp. 55–66.

[C$^+$08] UniProt Consortium et al., *The universal protein resource (uniprot)*, Nucleic acids research **36** (2008), no. suppl 1, D190–D195.

[CAS18] Quentin Cappart, John O. R. Aoga, and Pierre Schaus, *Episodesupport: A global constraint for mining frequent patterns in a long sequence of events*, Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings (Willem Jan van Hoeve, ed.), Lecture Notes in Computer Science, vol. 10848, Springer, 2018, pp. 82–99.

[CDG07] Toon Calders, Nele Dexters, and Bart Goethals, *Mining frequent itemsets in a stream*, Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on, IEEE, 2007, pp. 83–92.

[CGR09] Boris Cule, Bart Goethals, and Céline Robardet, *A new constraint for mining sets in sequences*, Proceedings of the 2009 SIAM International Conference on Data Mining, SIAM, 2009, pp. 317–328.

[CJSS12] Emmanuel Coquery, Saïd Jabbour, Lakhdar Saïs, and Yakoub Salhi, *A sat-based approach for discovering frequent, closed and maximal patterns in a sequence*, ECAI 2012 - 20th European Conference on Artificial Intelligence. Montpellier, France, August 27-31 , 2012 (Luc De Raedt, Christian Bessière, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas, eds.), Frontiers in Artificial Intelligence and Applications, vol. 242, IOS Press, 2012, pp. 258–263.

[CMS97] Robert Cooley, Bamshad Mobasher, and Jaideep Srivastava, *Web mining: Information and pattern discovery on the world wide web*, 9th International Conference on Tools with Artificial Intelligence, ICTAI '97, Newport Beach, CA, USA, November 3-8, 1997, IEEE Computer Society, 1997, pp. 558–567.

[CT06] Thomas M. Cover and Joy A. Thomas, *Elements of information theory (2. ed.)*, Wiley, 2006.

[CY10] Kenil CK Cheng and Roland HC Yap, *An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints*, Constraints **15** (2010), no. 2, 265–304.

[CYHP08] Hong Cheng, Xifeng Yan, Jiawei Han, and S Yu Philip, *Direct discriminative pattern mining for effective classification*, Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, IEEE, 2008, pp. 169–178.

[DAF16] Théophile K. Dagba, John O. R. Aoga, and Codjo C. Fanou, *Design of a yoruba language speech corpus for the purposes of text-to-speech (TTS) synthesis*, Intelligent Information and Database Systems - 8th Asian Conference, ACIIDS 2016, Da Nang, Vietnam, March 14-16, 2016, Proceedings, Part I (Ngoc Thanh Nguyen, Bogdan Trawinski, Hamido Fujita, and Tzung-Pei Hong, eds.), Lecture Notes in Computer Science, vol. 9621, Springer, 2016, pp. 161–169.

[DFL+15] Yiheng Duan, Xiao Fu, Bin Luo, Ziqi Wang, Jin Shi, and Xiaojiang Du, *Detective: Automatically identify and analyze malware processes in forensic scenarios via dlls*, 2015 IEEE International Conference on Communications, ICC 2015, London, United Kingdom, June 8-12, 2015, IEEE, 2015, pp. 5691–5696.

[DG15] Niti Ashish Kumar Desai and Amit Ganatra, *Efficient constraint-based sequential pattern mining (spm) algorithm to understand customers buying behaviour from time stamp-based sequence dataset*, Cogent Engineering **2** (2015), no. 1, 1072292.

[DHL+16] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus, *Compact-Table: Efficiently filtering table constraints with reversible sparse bit-sets*, Principles and Practice of Constraint Programming - 22nd International Conference, CP'16, Proceedings (Michel Rueher, ed.), Lecture Notes in Computer Science, vol. 9892, Springer, 2016, pp. 207–223.

[DKWK05] Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis, *Frequent substructure-based approaches for classifying chemical compounds*, IEEE Transactions on Knowledge and Data Engineering **17** (2005), no. 8, 1036–1050.

[DLM+98] Gautam Das, King-Ip Lin, Heikki Mannila, Gopal Renganathan, and Padhraic Smyth, *Rule discovery from time series*, KDD, vol. 98, 1998, pp. 16–22.

[DM02] Elizabeth D. Dolan and Jorge J. Moré, *Benchmarking optimization software with performance profiles*, Math. Program. **91** (2002), no. 2, 201–213.

[DRGN08] Luc De Raedt, Tias Guns, and Siegfried Nijssen, *Constraint programming for itemset mining*, International Conference on Knowledge Discovery and Data Mining (SIGKDD), ACM, 2008, pp. 204–212.

[dSMSSL13] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre, *Sparse-sets for domain implementation*, CP workshop on - Techniques foR Implementing Constraint programming Systems (TRICS), 2013, pp. 1–10.

[ERP18] N Esipova, J Ray, and A Pugliese, *Number of potential migrants worldwide tops 700 million*, Gallup (2018).

[Fed86] Meir Feder, *Maximum entropy as a special case of the minimum description length criterion*, IEEE Trans. Information Theory **32** (1986), no. 6, 847–849.

[FGL14] Johannes Fürnkranz, Dragan Gamberger, and Nada Lavrač, *Foundations of rule learning*, Springer Publishing Company, Incorporated, 2014.

[FKE+15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter, *Efficient and robust automated machine learning*, Advances in Neural Information Processing Systems 28 (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), Curran Associates, Inc., 2015, pp. 2962–2970.

[FLV+17] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Bay Vo, Tin Chi Truong, Ji Zhang, and Hoai Bac Le, *A survey of itemset mining*, Wiley Interdiscip. Rev. Data Min. Knowl. Discov. **7** (2017), no. 4, e1207.

[Fre97] Eugene C. Freuder, *In pursuit of the holy grail*, Constraints **2** (1997), no. 1, 57–61.

[FVLK$^+$17] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, and Rincy Thomas, *A survey of sequential pattern mining*, Data Science and Pattern Recognition **1** (2017), no. 1, 54–77.

[FVWT13] Philippe Fournier-Viger, Cheng-Wei Wu, and Vincent S Tseng, *Mining maximal sequential patterns without candidate maintenance*, Advanced Data Mining and Applications, Springer, 2013, pp. 169–180.

[FZC$^+$08] Wei Fan, Kun Zhang, Hong Cheng, Jing Gao, Xifeng Yan, Jiawei Han, Philip Yu, and Olivier Verscheure, *Direct mining of discriminative and essential frequent patterns via model-based search tree*, Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2008, pp. 230–238.

[GDT$^+$13] Tias Guns, Anton Dries, Guido Tack, Siegfried Nijssen, and Luc De Raedt, *Miningzinc: A modeling language for constraint-based mining*, Proceedings of the Twenty-Third international joint conference on Artificial Intelligence, AAAI Press, 2013, pp. 1365–1372.

[Gec06] Gecode Team, *Gecode: Generic constraint development environment*, 2006, Available from http://www.gecode.org.

[GLF$^+$18] Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, and Philip S. Yu, *A survey of parallel sequential pattern mining*, CoRR **abs/1805.10515** (2018).

[GNDR11] Tias Guns, Siegfried Nijssen, and Luc De Raedt, *Itemset mining: A constraint programming perspective*, Artificial Intelligence **175** (2011), no. 12-13, 1951–1983.

[GNDR13] ———, *k-Pattern set mining under constraints*, IEEE Transactions on Knowl. and Data Eng. **25** (2013), no. 2, 402–418.

[GNR11] Tias Guns, Siegfried Nijssen, and Luc De Raedt, *Evaluating pattern set mining strategies in a constraint programming framework*, Advances in Knowledge Discovery and Data Mining - 15th Pacific-Asia Conference, PAKDD 2011, Shenzhen, China, May 24-27, 2011, Proceedings, Part II (Joshua Zhexue Huang, Longbing Cao, and Jaideep Srivastava, eds.), Lecture Notes in Computer Science, vol. 6635, Springer, 2011, pp. 382–394.

[Goo15] Google, *Google optimization tools*, 2015, Available from https://developers.google.com/optimization/.

[Grü07] Peter D Grünwald, *The minimum description length principle*, MIT press, 2007.

[HAM14] Rui Henriques, Cláudia Antunes, and Sara C. Madeira, *Methods for the efficient discovery of large item-indexable sequential patterns*, New Frontiers in Mining Complex Patterns: Second International Workshop, NFMCP 2013, Held in Conjunction with ECML-PKDD 2013, Prague, Czech Republic, September 27, 2013, Revised Selected Papers (Cham) (Annalisa Appice, Michelangelo Ceci, Corrado Loglisci, Giuseppe Manco, Elio Masciari, and Zbigniew W. Ras, eds.), Springer International Publishing, 2014, pp. 100–116.

[HC08] Kuo-Yu Huang and Chia-Hui Chang, *Efficient mining of frequent episodes from complex sequences*, Information Systems **33** (2008), no. 1, 96–114.

[HFPZ13] Jun He, Pierre Flener, Justin Pearson, and Wei Ming Zhang, *Solving string constraints: The case for constraint programming*, International Conference on Principles and Practice of Constraint Programming, Springer, 2013, pp. 381–397.

[HKV19] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (eds.), *Automated machine learning - methods, systems, challenges*, The Springer Series on Challenges in Machine Learning, Springer, 2019.

[HM14] Rui Henriques and Sara C. Madeira, *Bicspam: flexible biclustering using sequential patterns*, BMC Bioinformatics **15** (2014), no. 1, 130.

[HPMA$^+$01] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and MC Hsu, *Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth*, proceedings of the 17th international conference on data engineering, 2001, pp. 215–224.

[HPYM04] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao, *Mining frequent patterns without candidate generation: A frequent-pattern tree approach*, Data mining and knowledge discovery **8** (2004), no. 1, 53–87.

[ITN04] Koji Iwanuma, Yo Takano, and Hidetomo Nabeshima, *On anti-monotone frequency measures for extracting sequential patterns from a single very-long data sequence*, Cybernetics and Intelligent Systems, 2004 IEEE Conference on, vol. 1, IEEE, 2004, pp. 213–217.

[JSS13] Said Jabbour, Lakhdar Sais, and Yakoub Salhi, *The top-k frequent closed itemset mining using top-k sat problem*, Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2013, pp. 403–418.

[JSS17] Saïd Jabbour, Lakhdar Sais, and Yakoub Salhi, *Mining top-k motifs with a sat-based framework*, Artif. Intell. **244** (2017), 30–47.

[KLL$^+$15] Amina Kemmar, Samir Loudni, Yahia Lebbah, Patrice Boizumault, and Thierry Charnois, *Prefix-projection global constraint for sequential pattern mining*, Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31 – September 4, 2015, Proceedings (Cham) (Gilles Pesant, ed.), Springer, Springer International Publishing, 2015, pp. 226–243.

[KLL$^+$16] Amina Kemmar, Samir Loudni, Yahia Lebbah, Patrice Boizumault, and Thierry Charnois, *A global constraint for mining sequential patterns with GAP constraint*, Integration of AI and OR Techniques in Constraint Programming - 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings (Claude-Guy Quimper, ed.), Lecture Notes in Computer Science, vol. 9676, Springer, Springer, 2016, pp. 198–215.

[KLL$^+$17] Amina Kemmar, Yahia Lebbah, Samir Loudni, Patrice Boizumault, and Thierry Charnois, *Prefix-projection global constraint and top-k approach for sequential pattern mining*, Constraints **22** (2017), no. 2, 265–306.

[KNGO15] Lars Kotthoff, Mirco Nanni, Riccardo Guidotti, and Barry O'Sullivan, *Find your way back: Mobility profile mining with constraints*, International Conference on Principles and Practice of Constraint Programming, Springer, 2015, pp. 638–653.

[Knu15] D.E. Knuth, *The art of computer programming: Combinatorial algorithms*, vol. 4, Addison-Wesley, 2015.

[KS10] Serdar Kadioglu and Meinolf Sellmann, *Grammar constraints*, Constraints **15** (2010), no. 1, 117–144.

[KTH$^+$17] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown, *Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka*, Journal of Machine Learning Research **17** (2017), 1–5.

[Lau18] Laurent Michel, Pierre Schaus, Pascal Van Hentenryck, *MiniCP: A lightweight solver for constraint programming*, 2018, Available from `https://minicp.bitbucket.io`.

[LC05] Congnan Luo and Soon Myoung Chung, *Efficient mining of maximal sequential patterns using multiple samples*, Proceedings of the 2005 SIAM International Conference on Data Mining, SDM 2005, Newport Beach, CA, USA, April 21-23, 2005 (Hillol Kargupta, Jaideep Srivastava, Chandrika Kamath, and Arnold Goodman, eds.), SIAM, 2005, pp. 415–426.

[Lec11] Christophe Lecoutre, *STR2: optimized simple tabular reduction for table constraints*, Constraints **16** (2011), no. 4, 341–371.

[Lho05] Olivier Lhomme, *Quick shaving*, Proceedings of the 20th national conference on Artificial intelligence-Volume 1, AAAI Press, 2005, pp. 411–415.

[Lic13] M. Lichman, *UCI machine learning repository*, 2013.

[LKFT04] Nada Lavrac, Branko Kavsek, Peter A. Flach, and Ljupco Todorovski, *Subgroup discovery with CN2-SD*, Journal of Machine Learning Research **5** (2004), 153–188.

[LL04] S Lu and C Li, *Aprioriadjust: An efficient algorithm for discovering the maximum sequential patterns*, Proc. Intern. Workshop Knowl. Grid and Grid Intell, 2004.

[LLL$^+$16] Nadjib Lazaar, Yahia Lebbah, Samir Loudni, Mehdi Maamar, Valentin Lemière, Christian Bessiere, and Patrice Boizumault, *A global constraint for closed frequent pattern mining*, International Conference on Principles and Practice of Constraint Programmingn (CP), Springer, 2016, pp. 333–349.

[LSU07] Srivatsan Laxman, PS Sastry, and KP Unnikrishnan, *A fast algorithm for finding frequent episodes in event streams*, Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2007, pp. 410–419.

[LW08] Chun Li and Jianyong Wang, *Efficiently mining closed subsequences with gap constraints*, Proceedings of the SIAM International Conference on Data Mining, SDM 2008, April 24-26, 2008, Atlanta, Georgia, USA, SIAM, 2008, pp. 313–322.

[MBC$^+$11] J Metivier, Patrice Boizumault, Bruno Crémilleux, Mehdi Khiari, and Samir Loudni, *A constraint-based language for declarative pattern discovery*, Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on, IEEE, 2011, pp. 1112–1119.

[MDLN01] Bamshad Mobasher, Honghua Dai, Tao Luo, and Miki Nakagawa, *Effective personalization based on association rule discovery from web usage data*, 3rd International Workshop on Web Information and Data Management (WIDM 2001), Friday, 9 November 2001, In Conjunction with ACM CIKM 2001, Doubletree Hotel Atlanta-Buckhead, Atlanta, Georgia, USA. ACM, 2001 (Roger H. L. Chiang and Ee-Peng Lim, eds.), ACM, 2001, pp. 9–15.

[ME10] Nizar R. Mabroukeh and C. I. Ezeife, *A taxonomy of sequential pattern mining algorithms*, ACM Comput. Surv. **43** (2010), no. 1, 3:1–3:41.

[MR04] Nicolas Méger and Christophe Rigotti, *Constraint-based mining of episode rules and optimal window sizes*, European Conference on Principles of Data Mining and Knowledge Discovery, Springer, 2004, pp. 313–324.

[MS00] Shinichi Morishita and Jun Sese, *Traversing itemset lattice with statistical metric pruning*, Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA (Victor Vianu and Georg Gottlob, eds.), ACM, 2000, pp. 226–236.

[MT96] Heikki Mannila and Hannu Toivonen, *Discovering generalized episodes using minimal occurrences.*, KDD, vol. 96, 1996, pp. 146–151.

[MTV95] Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo, *Discovering frequent episodes in sequences extended abstract*, 1st Conference on Knowledge Discovery and Data Mining, 1995.

[MTV97] ______, *Discovery of frequent episodes in event sequences*, Data mining and knowledge discovery **1** (1997), no. 3, 259–289.

[NDGN13] Benjamin Negrevergne, Anton Dries, Tias Guns, and Siegfried Nijssen, *Dominance programming for itemset mining*, Data Mining (ICDM), 2013 IEEE 13th International Conference on Data Mining, IEEE, 2013, pp. 557–566.

[NG10] Siegfried Nijssen and Tias Guns, *Integrating constraint programming and itemset mining*, ECML PKDD 2010 European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, 2010, pp. 467–482.

[NG15] Benjamin Négrevergne and Tias Guns, *Constraint-based sequence mining using constraint programming*, Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings (Laurent Michel, ed.), Lecture Notes in Computer Science, vol. 9075, Springer, 2015, pp. 288–305.

[NGDR09] Siegfried Nijssen, Tias Guns, and Luc De Raedt, *Correlated itemset mining in ROC space: a constraint programming approach*, International Conference on Knowledge Discovery and Data Mining (SIGKDD), ACM, 2009, pp. 647–656.

[NLHP98] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang, *Exploratory mining and pruning optimizations of constrained association rules*, SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA. (Laura M. Haas and Ashutosh Tiwary, eds.), ACM Press, 1998, pp. 13–24.

[NMB$^+$15] Stefan Naulaerts, Pieter Meysman, Wout Bittremieux, Trung-Nghia Vu, Wim Vanden Berghe, Bart Goethals, and Kris Laukens, *A primer to frequent itemset mining for bioinformatics*, Briefings in Bioinformatics **16** (2015), no. 2, 216–231.

[NZ14] Siegfried Nijssen and Albrecht Zimmermann, *Constraint-based pattern mining*, pp. 147–163, Springer, 2014.

[Osc12] OscaR Team, *OscaR: Scala in OR*, 2012, Available from `https://bitbucket.org/oscarlib/oscar`.

[Pes04] Gilles Pesant, *A regular language membership constraint for finite sequences of variables*, International conference on principles and practice of constraint programming, Springer, 2004, pp. 482–495.

[PFM16] Yao Jean Marc Pokou, Philippe Fournier-Viger, and Chadia Moghrabi, *Authorship attribution using small sets of frequent part-of-speech skip-grams*, Proceedings of the Twenty-Ninth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2016, Key Largo, Florida, USA, May 16-18, 2016. (Zdravko Markov and Ingrid Russell, eds.), AAAI Press, 2016, pp. 86–91.

[PHMA$^+$01] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu, *Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth*, proceedings of the 17th international conference on data engineering, IEEE, 2001, pp. 215–224.

[PHP$^+$01] Helen Pinto, Jiawei Han, Jian Pei, Ke Wang, Qiming Chen, and Umeshwar Dayal, *Multi-dimensional sequential pattern mining*, Proceedings of the tenth international conference on Information and knowledge management, ACM, 2001, pp. 81–88.

[PHW07] Jian Pei, Jiawei Han, and Wei Wang, *Constraint-based sequential pattern mining: the pattern-growth methods*, Journal of Intelligent Information Systems **28** (2007), no. 2, 133–160.

[PR14] Guillaume Perez and Jean-Charles Régin, *Improving GAC-4 for table and MDD constraints*, International Conference on Principles and Practice of Constraint Programmingn (CP), Springer, 2014, pp. 606–621.

[PZOD99] Srinivasan Parthasarathy, Mohammed Javeed Zaki, Mitsunori Ogihara, and Sandhya Dwarkadas, *Incremental and interactive sequence mining*, Proceedings of the 8th international conference on Information and knowledge management, ACM, 1999, pp. 251–258.

[QLvBG04] Claude-Guy Quimper, Alejandro López-Ortiz, Peter van Beek, and Alexander Golynski, *Improved algorithms for the global cardinality constraint*, Principles and Practice of Constraint Programming - CP 2004, 10th International

Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings (Mark Wallace, ed.), Lecture Notes in Computer Science, vol. 3258, Springer, 2004, pp. 542–556.

[QW06] Claude-Guy Quimper and Toby Walsh, *Global grammar constraints*, International Conference on Principles and Practice of Constraint Programming, Springer, 2006, pp. 751–755.

[Rác04] Balázs Rácz, *nonordfp: An FP-growth variation without rebuilding the FP-tree.*, FIMI: Workshop on Frequent Itemset Mining Implementations, 2004.

[Rég94] Jean-Charles Régin, *A filtering algorithm for constraints of difference in csps*, Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1. (Barbara Hayes-Roth and Richard E. Korf, eds.), AAAI Press / The MIT Press, 1994, pp. 362–367.

[Rég96] Jean-Charles Régin, *Generalized arc consistency for global cardinality constraint*, Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1, AAAI Press, 1996, pp. 209–215.

[Ris78] Jorma Rissanen, *Modeling by shortest data description*, Automatica **14** (1978), no. 5, 465–471.

[RMD$^+$15] Reza Rawassizadeh, Elaheh Momeni, Chelsea Dobbins, Pejman Mirza-Babaei, and Ramin Rahnamoun, *Lesson learned from collecting quantified self information via mobile and wearable devices*, Journal of Sensor and Actuator Networks **4** (2015), no. 4, 315–335.

[RTWT13] Reza Rawassizadeh, Martin Tomitsch, Katarzyna Wac, and A Min Tjoa, *Ubiqlog: a generic mobile phone-based life-log framework*, Personal and ubiquitous computing **17** (2013), no. 4, 621–637.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh (eds.), *Handbook of constraint programming*, Foundations of Artificial Intelligence, vol. 2, Elsevier, 2006.

[SA96] Ramakrishnan Srikant and Rakesh Agrawal, *Mining sequential patterns: Generalizations and performance improvements*, Springer, 1996.

[SAG17] Pierre Schaus, John O. R. Aoga, and Tias Guns, *Coversize: A global constraint for frequency-based itemset mining*, Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings (J. Christopher Beck, ed.), Lecture Notes in Computer Science, vol. 10416, Springer, 2017, pp. 529–546.

[SBF10] Peter J Stuckey, Ralph Becket, and Julien Fischer, *Philosophy of the minizinc challenge*, Constraints **15** (2010), no. 3, 307–316.

[SC06] Christian Schulte and Mats Carlsson, *Finite domain constraint programming systems*, in Rossi et al. [RvBW06], pp. 495–526.

[SD14] Dan A. Simovici and Chabane Djeraba, *Mathematical tools for data mining - set theory, partial orders, combinatorics. second edition*, Advanced Information and Knowledge Processing, Springer, 2014.

[SNK07] Laszlo Szathmary, Amedeo Napoli, and Sergei O. Kuznetsov, *ZART: A multifunctional itemset mining algorithm*, Proceedings of the 5th International Conference on Concept Lattices and Their Applications, CLA 2007 (Peter W. Eklund, Jean Diatta, and Michel Liquiere, eds.), vol. 331, 2007.

[SR14] Arnaud Soulet and François Rioult, *Efficiently depth-first minimal pattern mining*, Advances in Knowledge Discovery and Data Mining - 18th Pacific-Asia Conference, PAKDD 2014, Tainan, Taiwan, May 13-16, 2014. Proceedings, Part I (Vincent S. Tseng, Tu Bao Ho, Zhi-Hua Zhou, Arbee L. P. Chen, and Hung-Yu Kao, eds.), Lecture Notes in Computer Science, vol. 8443, Springer, 2014, pp. 28–39.

[SYCC$^+$15] Mohammad Shokoohi-Yekta, Yanping Chen, Bilson Campana, Bing Hu, Jesin Zakaria, and Eamonn Keogh, *Discovery of meaningful rules in time series*, Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining, ACM, 2015, pp. 1085–1094.

[TBG08] Roberto Trasarti, Francesco Bonchi, and Bart Goethals, *Sequence mining automata: A new technique for mining frequent sequences under regular expressions*, Data Mining, 2008. ICDM'08. 8th IEEE International Conference on, IEEE, 2008, pp. 1061–1066.

[TC10] Nikolaj Tatti and Boris Cule, *Mining closed strict episodes*, Data Mining (ICDM), 2010 IEEE 10th International Conference on, IEEE, 2010, pp. 501–510.

[Tea12] OscaR Team, *OscaR: Scala in OR*, 2012.

[UKA05] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura, *LCM Ver.3: Collaboration of array, bitmap and prefix tree for frequent itemset mining*, Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations (OSDM '05), OSDM '05, ACM, 2005, pp. 77–86.

[vB06] Peter van Beek, *Backtracking search algorithms*, in Rossi et al. [RvBW06], pp. 85–134.

[vHK06] Willem-Jan van Hoeve and Irit Katriel, *Global constraints*, in Rossi et al. [RvBW06], pp. 169–208.

[VT14] Jilles Vreeken and Nikolaj Tatti, *Interesting patterns*, in Aggarwal and Han [AH14], pp. 105–134.

[VvLS11] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes, *Krimp: mining itemsets that compress*, Data Min. Knowl. Discov. **23** (2011), no. 1, 169–214.

[WHL07] Jianyong Wang, Jiawei Han, and Chun Li, *Frequent closed sequence mining without candidate maintenance*, IEEE Transactions on Knowledge and Data Engineering **19** (2007), no. 8, 1042–1056.

[YHA03] Xifeng Yan, Jiawei Han, and Ramin Afshar, *Clospan: Mining: Closed sequential patterns in large datasets*, Proceedings of the 2003 SIAM International Conference on Data Mining, SIAM, 2003, pp. 166–177.

[YK05] Zhenglu Yang and Masaru Kitsuregawa, *LAPIN-SPAM: An Improved Algorithm for Mining Sequential Pattern*, International Conference on Data Engineering, 2005.

[YRS17] Hongyu Yang, Cynthia Rudin, and Margo Seltzer, *Scalable bayesian rule lists*, Proceedings of the 34th International Conference on Machine Learning, ICML'17 (Doina Precup and Yee Whye Teh, eds.), Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 3921–3930.

[YW06] Qiang Yang and Xindong Wu, *10 challenging problems in data mining research*, International Journal of Information Technology & Decision Making **5** (2006), no. 04, 597–604.

[YWK07] Zhenglu Yang, Yitong Wang, and Masaru Kitsuregawa, *LAPIN: effective sequential pattern mining algorithms by last position induction for dense databases*, Advances in Databases: Concepts, Systems and Applications, 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, Thailand, April 9-12, 2007, Proceedings (Kotagiri Ramamohanarao, P. Radha Krishna, Mukesh K. Mohania, and Ekawit Nantajeewarawat, eds.), Lecture Notes in Computer Science, vol. 4443, Springer, 2007, pp. 1020–1023.

[Zak98] Mohammed J Zaki, *Efficient enumeration of frequent sequences*, Proceedings of the seventh international conference on Information and knowledge management, ACM, 1998, pp. 68–75.

[Zak00] ———, *Sequence mining in categorical domains: incorporating constraints*, Proceedings of the ninth international conference on Information and knowledge management, ACM, 2000, pp. 422–429.

[Zak01] ———, *Spade: An efficient algorithm for mining frequent sequences*, Machine learning **42** (2001), no. 1-2, 31–60.

[ZAV14] Arthur Zimek, Ira Assent, and Jilles Vreeken, *Frequent pattern mining algorithms for data clustering*, in Aggarwal and Han [AH14], pp. 403–423.

[ZB03] Qiankun Zhao and Sourav S Bhowmick, *Sequential pattern mining: A survey*, ITechnical Report CAIS Nayang Technological University Singapore (2003), 1–26.

[ZjH02] Mohammed J. Zaki and Ching jui Hsiao, *CHARM: An efficient algorithm for closed itemset mining*, SIAM 2002, 2002, pp. 457–473.

[ZLC10] Wenzhi Zhou, Hongyan Liu, and Hong Cheng, *Mining closed episodes from event sequences efficiently*, Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2010, pp. 310–318.

[ZN14] Albrecht Zimmermann and Siegfried Nijssen, *Supervised pattern mining and applications to classification*, in Aggarwal and Han [AH14], pp. 425–442.