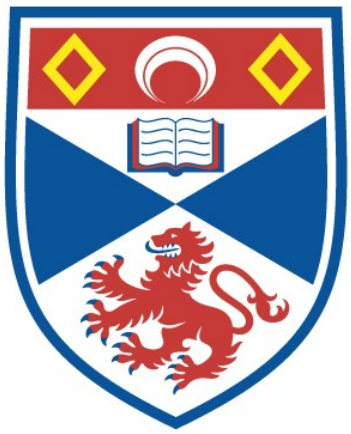
HIGH-LEVEL EFFICIENT CONSTRAINT DOMINANCE PROGRAMMING FOR PATTERN MINING PROBLEMS

Gökberk Koçak

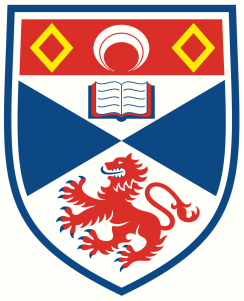
A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2023

High-level Efficient Constraint Dominance Programming for Pattern Mining Problems

Gökberk Koçak



University of
St Andrews

This thesis is submitted in partial fulfilment for the degree of
Doctor of Philosophy (PhD)
at the University of St Andrews

February 2022

ABSTRACT

Pattern mining is a sub-field of data mining that focuses on discovering patterns in data to extract knowledge. There are various techniques to identify different types of patterns in a dataset. Constraint-based mining is a well-known approach to this where additional constraints are introduced to retrieve only interesting patterns. However, in these systems, there are limitations on imposing complex constraints.

Constraint programming is a declarative methodology where the problem is modelled using constraints. Generic solvers can operate on a model to find the solutions. Constraint programming has been shown to be a well-suited and generic framework for various pattern mining problems with a selection of constraints and their combinations. However, a system that handles arbitrary constraints in a generic way has been missing in this field.

In this thesis, we propose a declarative framework where the pattern mining models can be represented in high-level constraint specifications with arbitrary additional constraints. These models can be efficiently solved using underlying optimisations.

The first contribution of this thesis is to determine the key aspects of solving pattern mining problems by creating an ad-hoc solver system. We investigate this further and create Constraint Dominance Programming (CDP) to be able to capture certain behaviours of pattern mining problems in an abstract way. To that end, we integrate CDP into the high-level ESSENCE pipeline. Early empirical evaluation presents that CDP is already competitive with current existing techniques. The second contribution of this thesis is to exploit an additional behaviour, the incomparability, in pattern mining problems. By including the incomparability condition to CDP, we create CDP+I, a more explicit and even more efficient framework to represent these problems. We also prototype an automated system to deduct the optimal incomparability information for a given modelled problem. The third contribution of this thesis is to focus on the underlying solving of CDP+I to bring further efficiency. By creating the Solver Interactive Interface (SII) on SAT and SMT back-ends, we highly optimise not only CDP+I but any iterative modelling and solving, such as optimisation problems. The final contribution of this thesis is to investigate creating an automated configuration selection system to determine the best performing solving methodologies of CDP+I and introduce a portfolio of configurations that can perform better than any single best solver.

In summary, this thesis presents a highly efficient, high-level declarative framework to tackle pattern mining problems.

Acknowledgements

General Acknowledgements

There are so many people that helped me conduct my research and write this thesis. It wasn't easy to finish my thesis during the times of a pandemic. Thank you all!

First of all, I would like to thank my supervisors Özgür Akgün and Ian Miguel for advising me throughout my PhD studies and giving me the help I need to improve my skills and knowledge. They provided me with the constant support and encouragement that I needed when I was in doubt. Thank you for giving your input on any matter and trusting me and my skills, giving me a lot of freedom to investigate my own way.

Coming to St Andrews was not a simple decision. I would like to thank Juliana Bowles for welcoming me into the joint masters programme Dependable Software Systems (DESEM), which led me to stay in St Andrews and continue with a PhD.

Being a member of the Constraints group has been very important for me. Working alongside many people in similar areas in a friendly environment was a great experience. I'm thankful for past and present members of the group Ian Gent, Peter Nightingale, Nguyen Dang, Andras Salamon, Mun See Chang, Chris Jefferson, Ruth Hoffman, Joan Espasa Arxer, Chris Stone, Saad Attieh, Fraser Dunlop, and Jordina Francès de Mas. I would also like include people outside of our group who ocasionally collaborated with us: Mateu Villaret, Tias Guns, and Jordi Coll.

Being part of the general department of St Andrews Computer Science (StACS) was also amazing. I thank all the people I met and became friends with during my studies. I want to thank many, although I'm sure I'll forget some names: Xu, Ryo, Al Dearle, Martin, Gui, Diana, Rozzi and many more. I am also thankful to all my friends outside of St Andrews. Thanks for being with me despite the long distances. I would especially like to thank my gaming buddies, Hasan and Oyil.

Apart from my friends, my family has always give amazing support and encouragement during my studies. I especially thank my parents Belgin and Hikmet for always being supportive. Thank you for creating a great environment for me to follow my ideas. I also thank my sister Begüm for always being an inspiration to me and always taking the first step to guide me.

And finally, Paula, thank you for making my stressful times easier. With your love and support, you kept me from overworking and over-stressing.

Funding Acknowledgements

This work was supported by the University of St Andrews (School of Computer Science).

Digital Outputs

Research software and data of this thesis are available at https://doi.org/10.5281/zenodo.5931360 [Koç22].

DECLARATIONS

Candidate's declaration

I, Gökberk Koçak, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 37,500 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree. I confirm that any appendices included in my thesis contain only material permitted by the 'Assessment of Postgraduate Research Students' policy.

I was admitted as a research student at the University of St Andrews in November 2017.

I received funding from an organisation or institution and have acknowledged the funder(s) in the full text of my thesis.

Date: 2022-11-21 Signature of candidate:

Supervisor's declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree. I confirm that any appendices included in the thesis contain only material permitted by the 'Assessment of Postgraduate Research Students' policy.

Date: 2022-11-21 Signature of supervisor:

Permission for publication

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with

the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, Gökberk Koçak, confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

Printed Copy

No embargo on print copy.

Electronic Copy

No embargo on electronic copy.

Date: 2022-11-21 Signature of candidate:

Date: 2022-11-21 Signature of supervisor:

Underpinning Research Data or Digital Outputs

Candidate's declaration

I, Gökberk Koçak, understand that by declaring that I have original research data or digital outputs, I should make every effort in meeting the University's and research funders' requirements on the deposit and sharing of research data or research digital outputs.

Date: 2022-11-21 Signature of candidate:

Permission for publication of underpinning research data or digital outputs

We understand that for any original research data or digital outputs which are deposited, we are giving permission for them to be made available for use in accordance with the requirements of the University and research funders, for the time being in force.

We also understand that the title and the description will be published, and that the underpinning research data or digital outputs will be electronically accessible for use in accordance with the license specified at the point of deposit, unless exempt by award of an embargo as requested below.

The following is an agreed request by candidate and supervisor regarding the publication of underpinning research data or digital outputs:

No embargo on underpinning research data or digital outputs.

Date: 2022-11-21 Signature of candidate:

Date: 2022-11-21 Signature of supervisor:

CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

List of Acronyms

BDD Binary Decision Diagram 84–88, 90

CDIM Closed Discriminative Itemset Mining xix, 28, 52, 69, 104, 111, 112, 193

CDP Constraint Dominance Programming xv, xvi, 5–8, 37, 49, 51, 53–58, 61, 65, 67–69, 71–74, 78–81, 83, 95, 116, 123, 132, 133, 168, 170

CDP+I Constraint Dominance Programming with Incomparability xvi, xvii, xix, xxiii, 5–8, 37, 68–75, 77–81, 83, 87, 92, 95, 97, 102, 104–108, 111, 112, 116, 126, 129, 131–134, 136, 137, 140, 168–172, 190–194

CFIM Closed Frequent Itemset Mining xvii, xix, xxiii, 26, 46, 52, 54, 57–59, 61, 69, 78, 104, 105, 111, 112, 116–119, 138, 172, 190

COP Constraint Optimisation Problem xv, 14, 15, 19, 49, 53, 54, 56, 67, 73, 74, 123, 143

CP Constraint Programming 7, 13, 14, 20, 22, 33, 34, 37, 40–42, 44, 57, 72, 78, 81, 96, 102, 132, 168, 170, 171

CSP Constraint Satisfaction Problem xv, 13, 14, 19, 21, 33–36, 38, 39, 50, 54, 56

DIG Dominance Incomparability Generator xvi, 5, 86–92, 168, 172

FIM Frequent Itemset Mining 9, 25, 26, 34

GFIM Generator Frequent Itemset Mining xix, 20, 27, 52, 104, 111, 112, 135, 138, 172, 191

MFIM Maximal Frequent Itemset Mining xxiii, 26, 35, 45, 46, 51, 52, 54, 69, 87, 88, 119

MRCPSP Multi-mode Resource Constrained Project Scheduling Problem xvii, 101–103, 109–111, 170

MRIM Minimal Rare Itemset Mining xix, 27, 52, 69, 104, 135, 138, 192

RSD Relevant Subgroups Discovery xvi, xix, 29, 53, 70, 72, 81, 91, 93, 104, 132, 133, 138, 194

SII Solver Interactive Interface 6–8, 95, 102, 133, 134, 140, 169, 171

List of Notations

C Constraints of a CSP-like problem over problem variables denoted with a set such as $C = \{c_1(V), c_2(V), ...\}$ 13, 15, 49, 50, 68

D Domains of a CSP-like problem variables denoted with a set such as $D_{v_1} = \{1, 2, 3\}$ 13, 15, 49, 50, 68

R Dominance relation of a CDP/CDP+I problem represented by a mathematical relation 49, 68

I Incomparability function of a CDP+I problem represented by a Boolean function 68

$\mathbb{I}$ Items of a dataset denoted by a set such as $\mathbb{I} = \{1, 2, ..., n\}$ 9, 24, 35, 39

O Optimisation function of a COP problem represented by an expression 14, 15

P CSP-like problem denoted by a triplet (V,D,C) when it is a CSP, a quadruplet (V,D,C,R) when it is a CDP, a quintuplet (V, D, C, R, I) when it is a CDP+I and a different quadruplet (V,D,C,O) when it is a COP 13, 15

ψ Class of a transaction represented by a Boolean-like integer value such as $\psi_{t_1} = 0$ 28

ϕ Cover of an itemset represented by a set of transactions variable such as $\phi_S = \{t_1, t_3, t_5\}$ 25, 28

$\mathbb{S}$ Solution set of a CSP-like problem represented by a set of solutions such as $\mathbb{S} = \{S_1, S_2, S_3\}$ 40, 41

$|\phi|$ Support of an itemset represented by an integer value such as $|\phi_S| = 3$ 25, 40

$\mathbb{T}$ Transaction ids of a dataset represented by a set such as $\mathbb{T} = \{1, ..., m\}$ 25, 39

V Variables of a CSP-like problem denoted by parentheses such as $V = (v_1, v_2, v_3)$ 13, 15, 49, 50, 68

Chapter One

Introduction

This thesis conducts research on two distinct research fields by combining them to produce an efficient system to solve challenging problems. We will start by introducing two general fields of research, data mining and constraint programming, and their importance towards approaching difficult problems. Afterwards, we will discuss the general contributions of this thesis and its resulting publications. Finally, we will outline the general structure of the thesis.

1.1 Data Mining

Data mining is about discovering knowledge using data. In the current age, the amount of data we generate and store has increased drastically with technological advancements. While the question of how to store big portions of data has mostly been answered, the question of how to make sense of the data through analytic techniques is still an active field. To this end, data mining is a field where the research aims at creating techniques to process available data and extract knowledge from it. The discovery of knowledge can be done in different ways. One specific technique we will be delving into is pattern mining, which focuses on finding patterns (i.e. smaller substructures in the data). Patterns can be found in many contexts, for example, certain words or sentences in texts or certain behaviours in an animal.

When pattern mining is applied with a set of items, it is called itemset mining. This term comes from the shopping market analysis systems that led the initial search on pattern mining. That is why in itemset mining each data entry is called a transaction, while each transaction includes a subset of the available set of items. An example of a transaction within a dataset can be bacon, lettuce, tomato and baguette while another example can be chips, ketchup and mayonnaise. Itemset mining aims to find interesting patterns considering all of the transactions. The discovered patterns can be used for additional analysis and can later lead to changing marketing/shelving strategies. Following the previous example, if the found pattern is chips, ketchup and mayonnaise, the market can place them next to each other for customers' convenience and also make promotions about buying them together with another product, such as a new beer brand.

After getting more popular in the research communities, new itemset mining methodologies have been proposed in different fields such as document analysis [HC99], web usage mining [TK01], bio-informatics [BBJ+02]. Itemset mining can be applied to any field where the system can be defined as a set of items and transactions. For this reason, a general purpose itemset mining system that is independent of any target domain could offer value to a wide variety of settings.

Before generalising the itemset mining system, it is important to consider which patterns are the most interesting. However, defining what patterns are interesting is in itself a challenge. A common approach is to impose constraints on the itemset mining. This is called Constraint-based mining [NLHP98, BAG00, BL07] and it aims to reduce the number of possible patterns with the usage of pattern-based constraints. Another challenge is the application of the constraints during the itemset mining procedure. Different constraints can be applied globally or locally before, after or during the procedure.

Most early approaches to constraint-based mining are problem-specific which makes applying constraints more difficult given the approaches' algorithmic nature. These methods are designed with a certain focus in mind, making it difficult to incorporate additional constraints they did not support originally.

Engineering a general pattern mining system has been regarded as very difficult to achieve in the pattern mining community [Man00]. One interesting approach [IM96] separates the dataset and constraints, similar to an inductive database query system, to make the pattern mining more generic. Later on, other data query language systems were also proposed [MPC98, BGL+09] utilising different approaches to achieve the same objective. However, there is room for improvement in terms of efficiency and the extent to which they can be generalised.

1.2 Constraint Programming

Constraint Programming is a field for solving combinatorial problems. It involves finding a combination of assignments of a problem's variables while satisfying a set of constraints. Constraint programming is being successfully used in many fields and applications including scheduling, planning and logistics [RVBW06].

Constraint programming is composed of two steps: modelling and solving. Modelling is where the parameters of the problem and their constraints are expressed. We can consider the simple constraint example of $\sum_{i=1}^{3} X_i < Y$. This constraint indicates that the sum of 3 X variables should be smaller than a Y variable. The solving component of constraint programming takes the model with its constraints and conducts a search to arrive at one or many solutions, using the given constraints to eliminate parts of the search space. In this example, assuming X_i and Y variables are integers, a satisfying solution would be $X_1 = 1, X_2 = 1, X_3 = 1$, and $Y = 5$.

Constraint programming is a declarative system where the end-user describes the constraints to be satisfied through a human intuitive language. Then the given specification is expressed in a way that solvers can operate on it. Following this, it is the solving system's responsibility to find the solutions.

The user gives the specification of the problem (i.e. the model) using a constraint specification or modelling language. Some examples of these languages are OPL [VH99], ESSENCE [FGJ+05, FJHM05, FHJ+08] and MiniZinc [NSB+07].

Each constraint specification language supports various variable types such as Booleans, integers or a list of these types. Higher levels types such as distinct sets can be also supported in the languages, such as ESSENCE and MiniZinc.

Users can make use of basic constraint types like arithmetic constraints, element constraints, linear constraints over integer variables, and many more. Additionally, the constraints can be also used as building blocks to create more complex constraints, enabling greater generality.

A constraint solver's purpose is to find an assignment for each variable while respecting every given constraint. To do so, the solver needs to reduce the search space using the constraints. The complexity or the size of search space is defined by the number of variables and their respective domains of values. While the constraints of the problem are crucial to reducing the search space, another important factor in the reduction is `propagation`. Propagation is the procedure of reducing the domain of a variable based on the constraints and the domains of other variables. Every type of constraint has a propagator and each propagator on a certain constraint not only ensures that a constraint is not violated but also removes all possible values that would violate the constraint (i.e. takes a domain and gives a possible smaller subset of the domain as a result).

The generality of constraint programming comes from the flexible modelling space where the problem is separated from its solving counterpart. Two distinct models can use similar constraints and can use similar optimal search/propagation strategies without the explicit statement of the user. Alternatively, the user can alter the constraint model without changing the solving methodology.

1.3 Contributions

The main contribution of this thesis is creating an effective and declarative framework to model and solve pattern mining problems in constraint programming systems. To do so, we examine the research question of creating such a system, focusing on two important aspects: improved declarative modelling and effective

solving. By improving the declarative representation of pattern mining problems, we aim to bring more flexibility to the end-user and allow complex problems to be represented in easier, more abstract ways. Following the modelling phase, we also focus on solving these problems in the most efficient way possible.

Improved Declarative Modelling

CDP - To make a declarative system for pattern mining problems, we define a new construct called dominance relation and we integrate this system into the ESSENCE language. We name this system Constraint Dominance Programming (CDP) and we implement the necessary abstractions for the ESSENCE pipeline to make a seamless CDP modelling system. The CDP system allows the modeller to express pattern mining problems in a more explicit way while the inner workings of the system are abstracted from the user. The generalised behaviour of CDP expands its functionality beyond pattern mining problems, including optimisation problems.

CDP+I - We define an additional component for CDP called incomparability condition (CDP+I). This new condition allows the constraint modeller to express additional information about the problem. It also grants the constraint user more control over the flow of execution to solve the given problem. In addition, CDP+I also allows multi-objective optimisation problems to be expressed in a more intuitive way.

DIG - We have created another tool called Dominance Incomparability Generator (DIG) that can automatically generate the primitives of the incomparability condition for a given CDP problem. This tool expands users' flexibility in declarative constraint modelling even further.

Effective Solving

CDP - To tackle the pattern mining problems efficiently, starting from our early experiments with ad-hoc ESSENCE miner, we explored different options resulting

in the development of the CDP framework. With our new declarative framework and its optimised procedure, the user can operate on problems with constraints amongst solutions, such as pattern mining problems, in a more efficient way.

CDP+I - By including incomparability into the CDP primitives, we achieve the goal of giving the constraint modeller a powerful tool to alter the order of the search in a declarative way. However, in addition to this, we also leverage CDP+I to be optimally efficient in some of the pattern mining problems. Moreover, CDP+I also allows expressing the Pareto frontiers in multi-objective optimisation problems almost in a native way for greater efficiency.

SII - We introduce the Solver Interactive Interface (SII) system in the ESSENCE pipeline for SAT and SMT back-ends. With SII, the constraint solving pipeline is optimised further for incremental solving scenarios such as CDP, CDP+I, and objective optimisation problems. Empirically, CDP+I is even more efficient with SII on the pattern mining problems examined in this thesis. Additionally, some optimisation problems can be solved more efficiently using a standard SAT back-end with SII than using MaxSAT or any other back-end.

Automated Configuration Selection - We build a portfolio of solving methodologies with an automated configuration selection system for pattern mining problems expressed with CDP+I in ESSENCE. To do so, we investigate the vast configuration space of possible parameters in the ESSENCE pipeline and CDP+I. We use these configurations in a training system and evaluate them using selected instance features. Through this, the portfolio system is created and used to select configurations to solve unseen instances in an efficient way.

1.4 List of Publications

Part of the research given in this thesis appeared in previous papers where the author of this thesis is the main author. All of these papers were co-authored with Özgür Akgün and Ian Miguel, the two PhD supervisors of the author of this thesis. Other co-authors in these papers are Nguyen Dang, Tias Guns, and Peter

Nightingale. All of the publications listed here are based primarily on the work of the author of the thesis.

[KAMN18b, KAMN18a] are the preliminary publications on the topic of combining pattern mining and constraint programming. They examine and identify key points to represent pattern mining problems in the constraint programming specification space.

[KAGM19] is the first publication where CDP is formalised and tested with some preliminary experiments. This paper also begins to explore the idea of incomparability with CDP (CDP+I) in a limited context.

[KAGM20a] is the main publication where the CDP/CDP+I system is generalised for any problem class and fully implemented in the ESSENCE pipeline. This paper includes an experimental evaluation on 5 different pattern mining problem classes where the performance of CDP+I is predominantly better.

[KADM20] is the publication where the Solver Interface Interaction (SII) system is proposed for incremental modelling and solving situations. This system is integrated into SAVILE ROW for SAT solvers. The experimental evaluation of this paper shows significant improvement for CDP+I and a single optimisation problem.

1.5 Structure of the Thesis

Chapter 2 outlines background information on several subjects: pattern mining, satisfiability, constraint programming, pattern mining on CP and SAT, dominance programming, automated configuration selection, and machine learning.

Chapter 3 discusses the preliminary work done for modelling and solving pattern mining problems on a constraint programming system. It delves into the main difficulties of including arbitrary side constraints in pattern mining problems. It also introduces an ad-hoc iterative solving mechanism in a constraint programming system to tackle the additionally constrained pattern mining problems.

Chapter 4 introduces the core Constraint Dominance Programming (CDP) system.

To do so, it expands on the previously defined iterative ad-hoc solving mechanism and formalises it to be more declarative. The newly defined structure then gets integrated into ESSENCE pipeline as CDP. The CDP framework allows users to model and solve pattern mining and optimisation problems efficiently.

Chapter 5 describes the concept of incomparability for CDP (CDP+I). The addition of incomparability to CDP allows users to exploit certain conditions of the problem specification for effective solving. This additional structure is integrated on top of CDP in the ESSENCE pipeline. Later in the chapter, the usage of multiple incomparabilities are explored enabling multi-objective optimisation problems to be expressed in a more native way and allowing for their Pareto frontier to be identified declaratively.

Chapter 6 presents the systematic generation of incomparabilities for CDP specifications expressed in ESSENCE. We name this the Dominance Incomparability Generator (DIG). This tool generates the optimal incomparability primitives as a step towards a fully automated CDP to CDP+I translation of problems. The inner working of this system is detailed and its application on multiple pattern mining problems is demonstrated.

Chapter 7 investigates the low-level solver interaction between the ESSENCE pipeline and the SAT/SMT solvers and presents the Solver Interactive Interface (SII). This system bridges the constraint model tailoring and encoding to efficiently solve SAT and SMT problems. This chapter is split into two sections where SAT and SMT interactions are investigated separately.

Chapter 8 details the instance generation process and the experimental setup. The generated instances and the experimental setup are used in the empirical evaluations throughout the thesis.

Chapter 9 studies the vast configuration space of pattern mining problems in ESSENCE using instance features. It investigates creating a portfolio of configurations for better efficacy using automated configuration selection systems. It also includes the feature importance analysis on the created portfolio.

Chapter 10 summarises the thesis and discusses possible future work.

CHAPTER TWO

BACKGROUND / LITERATURE REVIEW

This chapter provides the background on pattern mining, satisfiability, constraint programming, constraint programming for pattern mining, algorithm configuration, and machine learning. The concepts and techniques explained in this chapter will be used throughout the rest of the thesis.

2.1 Pattern Mining

Pattern mining is the process of finding interesting patterns in large data sets. There are pattern mining methods that are very fundamental in many applications areas such as market basket analysis, medicine, bio-informatics, web mining, network detection, and DNA research [BDRK+16]. Common pattern mining tasks include the well-known frequent itemset mining (FIM) problem in transactional databases, where the goal is to find sets of items that occur together frequently. FIM is first described in [AIS93] in which the dataset is given in the form of a set of transactions where each transaction consists of a set of items. In other words, if we describe all possible items as $\mathbb{I}$ an itemset S is a subset of $\mathbb{I}$. The idea of a frequent itemset comes from S having a support, $|\phi(S)|$, in the transactional database DB. The itemset S is frequent if $|\phi(S)|$, the number of transactions that contain the itemset S, respects the given minimum threshold support s_{min} (i.e. $|\phi(S)| \geq s_{min}$).

An example transactional dataset for a frequent itemset problem can be seen in the following.

Example 1.

$$DB = \begin{cases} T_1 = \{Bacon, Lettuce, Tomato, Cheese\} \\ T_2 = \{Bacon, Lettuce, Tomato, Onion\} \\ T_3 = \{Lettuce, Tomato, Egg, Fish\} \end{cases}$$

$$s_{min} = 2$$

In this example, $\{Bacon, Lettuce, Tomato\}$ with its subsets pass the required support constraint.

Standard pattern mining tasks that require enumerating all frequent itemsets are best performed using specialised tools and algorithms [Zak00, HPYM04]. The most well known is APRIORI [AS+94], which is included in multiple very efficient implementations such as Eclat [Bor03] and LCM [UKA+04]. However, a complete enumeration of all frequent itemsets is rarely what a user needs since the number of all frequent itemsets can be very large. The main goal of pattern mining is to find a smaller number of interesting patterns for further analysis.

Domain-specific side constraints [BL04] restrict the search with more limitations and use new methods for compactly representing the outcome of a particular pattern mining task [PBTL99, SR14, SNV07]. Using domain-specific side constraints has been proposed to increase the utility of constraint-based pattern mining. While these methods allow us to focus on interesting patterns and represent solution sets compactly, they also result in a significantly more computationally difficult data mining task.

There are constraint-based mining frameworks that try to incorporate specific constraints into the mining system by building the system from the ground up to support side constraints [BL07]. Some of these approaches also incorporate a level-wise search system [CYS03, BGMP03].

2.2 Satisfiability

Satisfiability (SAT) is one of the longest-standing areas in computer science, having been the first proven NP-complete problem [BHvM09].

The problem consists of a set of variables represented in Boolean logic and a propositional formula over these variables. The task is to decide whether it's possible to find a set of assignments for these variables such that the propositional formula becomes true. If it is possible, the task extends to finding one or many sets of assignments as well.

SAT problems are usually expressed in conjunctive normal form (CNF). This representation consists of a conjunction of clauses. Each clause is a disjunction of literals where literals are a variable or a negation of a variable. The following example 2 is a SAT formula represented in CNF.

Example 2.

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$$

This example consists of three SAT variables. One assignment to the variables which would satisfy the above formula is x_1 = true, x_2 = false, and x_3 = false. x_1 being true satisfies the first clause while x_2 being false satisfies the second clause. x_3 is left as false for the third clause to be satisfied.

A systematic search can be applied in the space of assignments to be able to solve SAT instances. A well-known algorithm for this is DPLL [DLL62]. DPLL applies chronological backtracking during the search. In each assignment step, it also applies unit propagation (sometimes called unit resolution) which consists of going through clauses and eliminating infeasible options to reduce the search space. A more complex way of backtracking is to directly backtrack more than one step to leave the infeasible section of the search space [BJS97]. This method was originally proposed as a technique for solving constraint satisfaction problems.

Another powerful technique that can be coupled with non-chronological backtrack is called conflict-driven clause learning (CDCL) [BHvMW09]. The

combination of CDCL and non-chronological backtracking makes sure that unit propagation is strengthened every time a conflict between variables arises. This guarantees that the system does not make the same mistake. The identification of conflict-driven clauses is done through a process called conflict analysis.

Nowadays, highly optimised implementations of SAT solvers use the explained techniques and can tackle large and challenging problems if the problem can be expressed in the CNF and loaded to the solver. A way to supply CNF style SAT instances to the SAT solvers is using the DIMACS format. The DIMACS format is a specialised API for CNF where the first line explains the characteristics of the problem while the rest of the lines represents the clauses. The clauses in DIMACS are separated with 0. Each literal is expressed with non-zero integers; negative integers represent negated variables. Lines starting with "c" are denoted as comments for possible explanations. The problem in example 2 can be expressed in DIMACS like fig. 2.1.

```
c SAT CNF example problem
p cnf 3 3
1 2 3 0
-1 -2 0
2 -3
```

Figure 2.1: A DIMACS representation of the example 2.

Some of the most common SAT solvers are MINISAT [SE05], GLUCOSE [ALS13] and CADICAL/ KISSAT [FH20] which are highly optimised to achieve satisfiability. There is a particular SAT solver which modifies the SAT backtrack algorithm to be able to enumerate all solutions [TS16]. This particular AllSAT solver is a modification of the MINISAT solver that blocks the solution with new clauses (i.e. BC_MINISAT_ALL) or uses a back-jump instead to be non-blocking (i.e. NBC_MINISAT_ALL).

Optimisation problems can be also tackled using satisfiability systems. MaxSAT [LM09], which supports hard and soft constraints, has been suggested to be able to approach optimisation problems in SAT.

2.3 Constraint Programming

Constraint Programming (CP) [RVBW06] is a general-purpose method for specifying decision and optimisation problems in a declarative language and finding solutions to these problems using highly efficient black-box solvers. In CP approaches, problem-solving is divided into 2 phases: modelling and solving. When using CP approaches, the end-user specifies the problem in a declarative manner (i.e. modelling) and the given system finds solutions that satisfy the given constraints (i.e. solving). Additionally, the same constraint model can be applied to multiple parameters. These are called instances of a model, in which the general parameters of the model stays the same and a small portion of the parameters are altered. This ultimately gives high-level control to the user.

Definition 2.3.1. Constraint Satisfaction Problem (CSP) - A constraint problem can be expressed in a constraint specification using three primitives: the variables of the problem V, the domain of all possible values of the variables D, and the constraints of these variables C. Thus, we can define a CSP as $P = (V, D, C)$.

Example 3. Let us consider an example where we need to arrange the order of 3 meetings for 3 time slots. There are some restrictions for the meetings: Meeting 1 M_1 needs to be after Meeting 3 M_3, Meeting 2 M_2 cannot be assigned to the last time slot, and M_3 cannot be assigned to the first time slot.

We can formally define this problem as:

$$P = \begin{cases} V = (M_1, M_2, M_3) \\ D_{M_1} = D_{M_2} = D_{M_3} = \{1, 2, 3\} \\ C = \{M_1 > M_3, M_2 \neq 3, M_3 \neq 1\} \end{cases}$$

The modelled problem then can be supplied to the solver, which is purposed to find solutions that satisfy the given constraint model. CP solvers use the principles of search and propagation. The search methodology used by most CP solvers is to apply a depth-first search alongside propagation [SS08]. Propagation is applied

to reduce the domain of the variables to make the problem consistent within the current state of the search.

An alternative to the Constraint Programming Search Solvers is Local Search Solvers [HS04, HM09]. Instead of employing search and propagation, these solvers operate by starting from a full assignment and gradually reduce the number of constraint violations. This occurs by improving the objective in the current assignment each iteration through a sequence of moves or neighbourhoods.

Considering the same example above, in the first step, the initial propagation will reduce the domains of M_2 and M_3 and make the search space smaller for the system:

$$P = \begin{cases} V = (M_1, M_2, M_3) \\ D_{M_1} = \{1,2,3\} \\ D_{M_2} = \{1,2\} \\ D_{M_3} = \{2,3\} \\ C = \{M_1 > M_3, M_2 \neq 3, M_3 \neq 1\} \end{cases}$$

After this propagation, the constraint $M_1 > M_3$ can propagate the domain of M_1 to be $\{3\}$ and the search can continue from there. Eventually, the problem will be solved at $M_1 = 3$, $M_2 = 1$, and $M_3 = 2$.

Using CP systems, it's possible to address two different sets of problems: 1) constraint satisfaction problems (CSP) where the goal is to find satisfactory solutions, and 2) constraint optimisation problems (COP) (sometimes called optimisation CSPs as well) where the goal is to find an optimal solution depending on certain criteria.

Definition 2.3.2. Constraint Optimisation Problem (COP) - Some problem classes, like scheduling or planning, are represented with CSP primitives and an objective (O) indicated by a function f, which will be optimised. The goal is to minimise or maximise the f function. The COP solving system operates similarly to the CSP one, with the addition of back-tracking on the solution (S) and adding a new constraint on the model ($c < f(S)$ or $c > f(S)$) . This allows the search

to continue while tightening the optimisation value's range to approach to the optimal value [RVBW06]. Thus, COP can be defined as $P = (V, D, C, O)$.

Example 4. Let's consider the knapsack problem, where there are items with monetary value and volume. We have a limited amount of storage and would like to maximise the monetary value we carry. Let's say we have a computer (I_1) with a volume of 3 and value of 10, a rock (I_2) with a volume of 4 and value of 1, a glass figurine (I_3) with a volume of 1 and value of 2, and a pencil case (I_4) with a volume of 2 and value of 2. Our space budget is 4.

We can formally define this problem using Boolean variables to indicate whether each item is present or not:

$$P = \begin{cases} V = (B_1, B_2, B_3, B_4) \\ D_{B_1} = D_{B_2} = D_{B_3} = D_{B_4} = \{0, 1\} \\ Vol_{I_1} = 3, Vol_{I_2} = 4, Vol_{I_3} = 1, Vol_{I_4} = 2 \\ Mon_{I_1} = 10, Mon_{I_2} = 1, Mon_{I_3} = 2, Mon_{I_1} = 2 \\ VolBudget = 4 \\ C = \{\sum_{i=1}^{4}(B_i.Vol_{I_i}) \leq VolBudget\} \\ O = max(\sum_{i=1}^{4}(B_i.Mon_{I_i})) \end{cases}$$

In this simple example, we can see that including the computer (i.e. I_1) gives the most value towards the optimisation function. In this case, it is clear that it should be included, but in any other instance of this problem recognising this may prove challenging.

Most known constraint modelling/specification languages are OPL [VH99], ESSENCE [FHJ+08], and MiniZinc [NSB+07]. While the OPL language goes through the IBM CPlex solver, the MiniZinc language is processed to be compatible with a lower level language, FlatZinc(FZN). FZN can be consumed by solvers such as the constraint solver Gecode [SLT06], the lazy clause generator constraint solver Chuffed [Chu16] or the local search solver Yuck [BMFP15].

2.3.1 Essence Pipeline

While most constraint modelling/specification languages offer support for Boolean, integer and a basic collection of these types, some problems can be easier to explain and describe in higher-level structures such as nested sets. For instance, the itemset mining problem consists of transactions in the form of a list of sets of items. ESSENCE supports abstract types and structures such as arbitrarily nested sets by design, which allows these constructs to be represented in higher level.

An ESSENCE specification comprises: problem class parameters (`given`); combinatorial objects to be found (`find`); constraints the objects must satisfy (`such that`); identifiers declared (`letting`); and an optional objective function (`min/maximising`). As mentioned earlier, the key feature of the language is support for abstract decision variables, such as multiset, relation and function, and high-level *nested* types, such as the list/multiset of sets which directly fits the itemset mining problem. Considering this example, we can deduce that the abstract representation flexibility available in ESSENCE would allow us to model pattern/itemset mining problems more natively in a more abstract way.

Although ESSENCE specification language supports high-level abstract types, most generic constraint solvers do not support operating on them (except local search solver ATHANOR [ADJ+19] which can operate on the ESSENCE level). Thus, the high-level ESSENCE needs to be translated first to be able to be compatible with the solvers. This is where a pipeline of systems comes into play to translate and transform ESSENCE specifications into the formats where solvers can use it.

ESSENCE pipeline overview can be seen fig. 2.2.

CONJURE [AMJ+11, Akg14, AGJ+14] takes as input a specification in ESSENCE and applies a series of successive model *refinement* rules to translate high level problem specifications into ESSENCE PRIME [NR16]. ESSENCE PRIME is a constraint modelling language with similar level abstractions to MiniZinc's.

The refinements made by CONJURE converts abstract types, such as a set with lower-level primitives with a list. There are multiple ways of representing

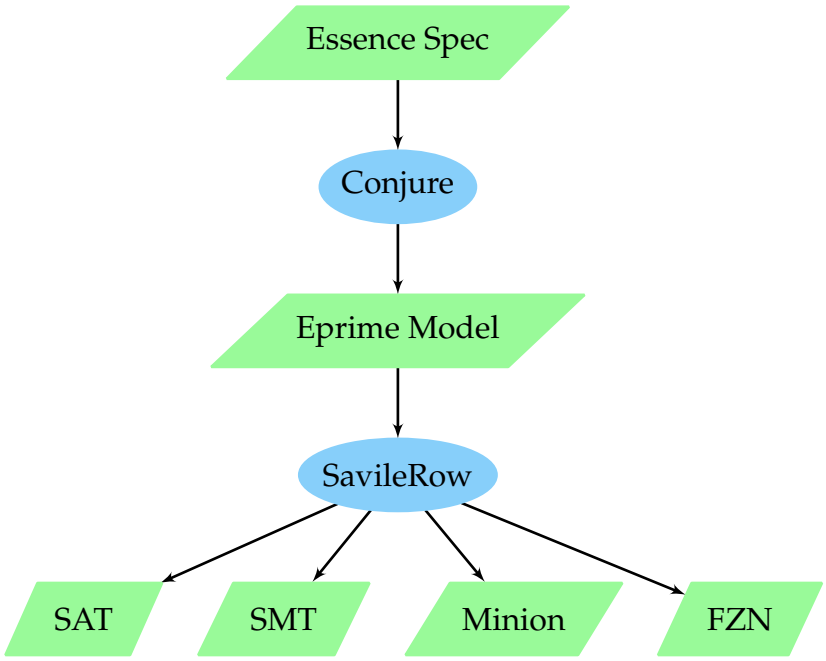


Figure 2.2: ESSENCE pipeline.

higher-level constructs: this is called representation choices. For instance, a set of integers can be represented in two ways: first, using an explicit matrix of integers with additional constraints to satisfy the uniqueness constraint or second, an occurrence list of Boolean values where every possible value in the domain of integers is already included.

A set example in ESSENCE can be seen in the fig. 2.3. CONJURE produces 4 different options to translate this set variable into ESSENCE PRIME: one with occurrence representation and 3 with different explicit representations. The three different explicit representations differ in how to indicate the boundary of the set with dynamic cardinality: one uses a *marker* to indicate the end spot, one uses an additional Boolean array to *flag* values in use, and one uses an additional *dummy* value to fill the rest of the matrix.

```
find test_set : set (maxSize 5) of int(0..10)
```

Figure 2.3: A set example in ESSENCE specification. The set also has a maximum cardinality constraint.

One explicit representation with a marker of the fig. 2.3 can be seen in the fig. 2.4. The first constraint ensures that up to the marker point each value is ordered from lower index to higher index, ensuring uniqueness. The second constraint sets the values above the marker point to 0. The third constraint is somewhat redundant, as the domain limitation already ensures the set cardinality does not pass beyond 5.

```
find test_set_ExplicitVarSizeWithMarker_Marker: int(0..5)
find test_set_ExplicitVarSizeWithMarker_Values:
 matrix indexed by [int(1..5)] of int(0..10)
such that
 and([q1 + 1 <= test_set_ExplicitVarSizeWithMarker_Marker ->
  test_set_ExplicitVarSizeWithMarker_Values[q1]
  < test_set_ExplicitVarSizeWithMarker_Values[q1 + 1]
  | q1 : int(1..4)]),
 and([q2 > test_set_ExplicitVarSizeWithMarker_Marker ->
  test_set_ExplicitVarSizeWithMarker_Values[q2] = 0
  | q2 : int(1..5)]),
 test_set_ExplicitVarSizeWithMarker_Marker <= 5
```

Figure 2.4: Explicit with markers ESSENCE PRIME representation of the ESSENCE set variable for the example in fig. 2.3

The occurrence representation of the fig. 2.3 can be seen in the fig. 2.5. In terms of complexity, the occurrence representation looks much simpler since it uses one single matrix structure and a single additional constraint to ensure cardinality. However, because of the potential large domain ranges and sparsity of the set, it may not be as efficient as an explicit representation.

```
find test_set_Occurrence:
 matrix indexed by [int(0..10)] of bool
such that
 sum(
  [toInt(test_set_Occurrence[q1]) | q1 : int(0..10)]
  ) <= 5
```

Figure 2.5: Occurrence ESSENCE PRIME representation of the ESSENCE set variable for the example in fig. 2.3

After ESSENCE to ESSENCE PRIME translation is completed, the constraint modelling assistant SAVILE ROW [NAG+17] takes constraint models written in the ESSENCE PRIME language and translates them to a backend solver capable format (MINION, SAT, SMT or FZN). It employs several reformulations to improve the model, such as common subexpression elimination (CSE) or domain filtering techniques. Domain filtering techniques includes generic arc consistency (GAC), singleton arc consistency (SAC), or double singleton arc consistency (SSAC). [NAG+14]. Applying Arc consistency to the CSPs is a very common methodology which removes any possible values from domains and is very effective [BR97, DB97]. For each solver backend targeted from SAVILE ROW, constraint programming solver MINION is used for domain filtering via GAC, SAC or SSAC.

When encoding to SAT, SAVILE ROW can use direct encoding and order encoding depending on the constraint expression [NAG+17].

SAVILE ROW can translate COP by targeting MaxSAT directly or using an ad-hoc SAT targeting mechanism. To target standard SAT backends, SAVILE ROW uses 3 different strategies: 1) Linear, 2) Unsat and 3) Bisect. All three strategies encode the COP into CSP with a changing optimisation value target. The linear strategy starts from the worst possible value for the optimisation variable and tries to improve it linearly each iteration by searching for the satisfiability of the value given. The unsat strategy operates in the opposite direction, starting from the best possible value of the optimisation variable and trying to identify the first satisfiable point. While the first and second strategies improve their value linearly, the bisect strategy conducts a binary search operation and starts from the midpoint to find the best possible value, performing in $log(n)$.

When targeting SMT, SAVILE ROW can use multiple theories such as Linear Integer Arithmetic (LIA), Non-linear Integer Arithmetic (NIA), Boolvector (BV) or Integer Difference Logic (IDL) [DAEN20].

2.4 CP/SAT for Pattern Mining

Constraint programming for pattern mining, which first appeared in [DRGN08] offers a general means of modelling more sophisticated pattern mining tasks. Its flexibility means that side constraints can easily be added to the basic model of a pattern mining problem, which is difficult to do with a specialised mining tool. We distinguish *local* and *non-local* constraints in modelling pattern mining problems. A local constraint, such as the frequent itemset property, can be expressed simply on a candidate solution, e.g. by constraining the support of a candidate itemset to be equal to or greater than the threshold. Non-local constraints, however, must be expressed *between* candidate solutions and are therefore more challenging to model. Closed frequent itemset mining [PBTL99], which is one approach to representing the full set of frequent itemsets more compactly, is an illustrative example: it stipulates that an itemset is closed frequent if its support exceeds that of all of its supersets.

More recent work demonstrates the utility of CP for performing pattern mining tasks ClosedPattern [LLL+16] and CoverSize [SAG17]. These approaches define new efficient propagators for the closed frequent itemset problem to combine the power of CP with its generality. The idea is that additional side constraints can be introduced without being challenging. In contrast to specialised algorithms, where incorporating domain knowledge is often difficult, side constraints often improve the performance of a black-box constraint solver.

A declarative framework has been established and implemented [Gun15] via MINIZINC. MINIZINC [GDT+13, GDN+17] creates a collection of methods including hybrid approaches to solve pattern mining problems. The problem is given in the form of a MINIZINC syntax and later is taken by the MINIZINC framework. It is possible to generate a multitude of possible solving approaches from pure CP, a hybrid between specialised algorithms with CP/post-processing steps.

More CP propagator approaches appear in recent work to tackle GFIM [BBL19a] to capture the association rule mining problem. A generalisation of these types of

bordering itemset mining problems has also been done recently in [BBL19b]. This work also proves these types of problems are coNP-Hard eliminating the one-shot CSP or SAT approaches.

In the meanwhile, there is a significant amount of related work in the SAT community as well. Using specialised SAT encoding for itemset mining problems has been brought up in recent studies [JSS15, JSS17]. A particular work focuses on maximal itemset mining with side constraints [JMD+18].

2.5 Dominance Programming

Dominance programming has been suggested as a way of formulating constraints amongst solutions in a general way [NDGN13, GST18] such that they are compatible with other arbitrary constraints.

In addition to the main CSP workflow, where the decision variables and constraints relating to a single solution are declared in the usual way, a dominance programming model specifies constraints among solutions using dominance blocking constraints. If we consider two candidate solutions c_1 and c_2, $c_1 \succ c_2$ indicates that candidate solution c_1 dominates the candidate solution c_2 given the constraints amongst solutions.

Every time a solution is found during the search, a new blocking constraint is added. This way, potential solutions that are dominated by a previously found solution are blocked. Following these semantics, this system always finds all non-dominated solutions. However, without a perfect search order guiding the search, the set of solutions can include dominated solutions [NDGN13].

An example can be used to explain the possibility to find dominated solutions. Let's assume *cost* as the criteria for the dominance constraints:

$$C_d = \{\forall s \in S \mid s \leq c_s \iff s \succ c_s\} \tag{2.1}$$

C_d indicates dominance constraints, S is the current solution set and c_s is a candidate solution. This constraint indicates that if the cost of the candidate is

higher than a previous solution, the candidate solution is dominated by previous solution.

Depending on our search order, we can find a solution with a very high cost early on, which gets registered into our solution set. The next solutions can improve this cost value and may not permit any solution with cost similar to the early solution's very high cost. However, since our order of finding these solutions is not perfect, we ended up having a solution with very high cost in our solution set. This is why if the system ever finds dominated solutions, they need to be removed using a post-processing step for correctness in the end result.

2.6 Algorithm Configuration

Algorithms for hard computational problems, including combinatorial search or local search, are often highly parameterised. In local search, for instance, typical parameters include neighbourhoods, percentage of random walk steps, and more. Typical parameters in the combinatorial search include the method of pre-processing, branching rules, usage of any possible learning to perform, and more. As an example, IBM's commercial solver CPLEX [LRSV18] has 76 parameters to optimise its search strategy [HHLB10]. The idea is that optimising the settings of these parameters can bring significant performance improvements. However, doing these optimisations manually can be time-consuming and often impractical.

Defining automated procedures to approach this algorithm configuration problem can be quite useful for many reasons. The most common use case is to use a training set of instances to optimise the parameters of the algorithm, resulting in greater performance on the unseen test instances. Achieving the parameter tuning automatically instead of performing manually saves developer time, converting it into machine time. There are model-free algorithm configuration methods that are flexible and can be applied out-of-the-box to any system. These can lead to drastic performance improvements in a variety of CP problems. While the earliest

example of these approaches start appearing in 90s [MJPL92], they are recently gaining attraction again.

While some of these model-free systems are focused on numerical values [ADL06], some others can work on categorical systems [BSP+02]. Some well-known tools for configuration tuning are F-race [BYBS10] and ParamILS [HHS07].

More recent work in model-based approaches includes Sequential model-based optimisation (SMBO), which iterates between model fitting and making choices about which configurations to investigate [HHLB11]. It is a Bayesian Optimisation [Moc12] approach where we sample new configuration in a sequential manner.

A well-known version of SMBO, *SMAC* (sequential model-based algorithm configuration) is an adapted version of Bayesian Optimisation where Random Forest is used instead of Gaussian Process (the typical choice in Bayesian Optimisation), and with special capping techniques for more efficient search in the context of automated algorithm configuration (when optimising on run-time). SMAC is currently on its third iteration with SMAC3 [1].

One important automated algorithm system that uses a portfolio-based selection system is proposed in Hydra [XHLB10].. The portfolio selection system Hydra uses SMAC for building a portfolio of algorithm configurations with complementary strength.

Another racing-based automated configuration selection is irace [LIDLC+16]. A hyperparameter tuning/optimisation framework with an intuitive python front-end has been published under OPTUNA [ASY+19].

2.7 Machine Learning

Machine learning is a research field where the focus is on learning systems and algorithms. It is a very diverse and interdisciplinary field that interacts with fields of artificial intelligence, statistics, mathematics, and many others [RN02]. As it is

[1] `https://github.com/automl/SMAC3`

in use in many applications, machine learning has been of great importance and features in many scientific domains.

Machine learning has generally been divided into three categories: supervised, unsupervised, and reinforcement learning [RN02]. Supervised learning requires training with tagged/labelled data with specified input and output. Its counterpart unsupervised learning works by not requiring the tagging operation and only feeding input to the system. Reinforcement learning works as a constant learning feedback loop via an interactive external environment.

In our thesis, we will be benefiting from some of the supervised methods. Some well-known, supervised techniques are Support Vector Machines [HDO+98], Hidden Markov model [Edd04], Bayesian Networks [BG08], and Neural Networks [ABB+99].

To be able to use the supervised machine learning techniques, a user-friendly framework with Python frontend scikit-learn [PVG+11] is available. It is one of the most popular machine learning toolkits available. A more black-box approach, where the machine learning algorithms are efficiently selected by a hyperparameter selection system auto-sklearn [FKE+15], is also available to automatically train systems.

2.8 Problem Classes

The problem classes we are going to use in this work all operate on a transactional dataset structure and can be modelled in the form of a multi-set of sets. They are pattern mining problems, more specifically itemset mining problems, which means the inner sets consist of items and the items can be represented as a primitive construct.

Itemsets and transactions can be defined mathematically with n as the number of possible items ($\mathbb{I}$) eq. (2.2)

$$\mathbb{I} = \{1, \ldots, n\} \tag{2.2}$$

The transaction set ($\mathbb{T}$) defines m number of transactions eq. (2.3)

$$\mathbb{T} = \{1, ..., m\} \tag{2.3}$$

By merging these two elements, the database of the transactions DB can be defined as eq. (2.4)

$$\forall t \in \mathbb{T}, DB(t) \subseteq \mathbb{I} \tag{2.4}$$

On the defined transactional dataset, the FIM problem can be defined as identifying itemsets where the *support* (definition 2.8.1) of an itemset is above a given threshold value.

Definition 2.8.1. The cover (ϕ) is defined as the transactions that contain the pattern itemset as a subset eq. (2.5). The support is the number covering transactions (i.e the cardinality of the cover).

$$\phi_{DB}(I) = \bigcup_{t=1}^{m} \{I | I \subseteq DB(t)\} \tag{2.5}$$

To find frequently appearing sets of items in a transaction database, we can define the frequent itemset mining for the itemset I as eq. (2.6)

$$fis(I) \implies |\phi_{DB}(I)| \geq min\_freq \tag{2.6}$$

The cardinality of ϕ_{DB} ($|\phi|$) will give the support of the itemset I. min_freq is the occurrence limit, which is set to define what is frequent. If we want an itemset with at least 10% frequency, this would lead to $min\_freq = m/10$.

While all other problem classes require only the support for a specific pattern, one particular problem class, relevant subgroup discovery problem, requires *identifying* which itemsets cover the pattern. For this problem class, it is necessary to create a distinction between the covering transactions.

Each model denotes similar decision variables. In these models, we always try to find a set of items to represent the pattern. In addition to the pattern, the

goal is to retrieve the pattern's support (using integers) or its cover (using a set of transaction ids).

In the non-mathematical dominance representation of the models, we use *is* to denote the decision variable itemset and $s()$ to access support. Additionally, $prev()$ has been used to indicate previous solution's decision variables. $prev()$ is equivalent to `fromSolution` in ESSENCE syntax.

2.8.1 Maximal Frequent Itemset Mining

Maximal frequent itemset mining (MFIM) dictates that no subset of any frequent itemset should be allowed and only maximal cardinality of any pattern should be included in the final solution set. Mathematically, for a maximal frequent itemset I_a (which has a support higher than min_freq), this can be defined as follows eq. (2.7):

$$mim(I_a) \implies \forall I_b, I_a \subseteq I_b \wedge |\phi_{DB}(I_b)| < min\_freq \tag{2.7}$$

For every possible I_b which supersets I_a, I_b cannot have the minimum superset requirement satisfied. Otherwise, I_a would not be maximal.

If we use the same bacon-lettuce-tomato transactional dataset example from example 1, applying maximality to the frequent itemsets over in that example, we find a single itemset $\{Bacon, Lettuce, Tomato\}$ is maximal.

2.8.2 Closed Frequent Itemset Mining

Closedness is a condition for the whole solution set of a frequent itemset mining task. Itemsets are called *closed* if and only if their support is greater than all of their supersets. Closed frequent itemset mining, CFIM, also acts as a lossless way of compressing the solution set of FIM, since all frequent itemsets can easily be enumerated once the closed frequent itemsets are found [PBTL99].

Mathematically, for a closed frequent itemset I_a, this can be defined as eq. (2.8)

$$cfim(I_a) \implies \forall I_b, I_a \subseteq I_b \wedge |\phi_{DB}(I_b)| < |\phi_{DB}(I_a)| \tag{2.8}$$

For every possible I_b which supersets I_a, I_a should have strictly higher support to be able to satisfy closedness.

Following example 1, we find $\{Bacon, Lettuce, Tomato\}$ and $\{Lettuce, Tomato\}$ are closed. An additional support for $\{Lettuce, Tomato\}$ makes the itemset closed, although this itemset was not in the maximal itemsets previously.

2.8.3 Generator Frequent Itemset Mining

Generator itemsets (also called free itemsets or key itemsets) [BBR00, BJ01] are a related compressed representation of all frequent itemsets. A generator itemset is a frequent itemset that does not have any frequent subsets with the same support.

Generator frequent itemset mining (GFIM) is useful as part of a larger association rule mining task. Together with closed frequent itemsets, these itemsets construct minimal non-redundant association rules [Kry98].

Mathematically, for a generator itemset I_a, this can be defined as eq. (2.9)

$$gim(I_a) \implies \forall I_b, I_a \supseteq I_b \wedge |\phi_{DB}(I_b)| > |\phi_{DB}(I_a)| \tag{2.9}$$

For every possible I_b under I_a's supersets, I_b needs to have strictly higher support than I_a, otherwise I_a would not be a generator.

Following the same example in example 1, we find $\{Lettuce\}$, $\{Tomato\}$ and $\{Bacon, Lettuce, Tomato\}$ are generator itemsets.

2.8.4 Minimal Rare Itemset Mining

A minimal rare itemset is an infrequent itemset whose subsets are all frequent. They are closely related to maximal, closed, and generator itemsets. Minimal rare itemset mining (MRIM) is useful for dense datasets where the number of frequent itemsets may be very large [SNV07].

Mathematically, for a minimal rare itemset I_a, the equation can be defined as in eq. (2.10). The equation benefits from the previous maximal equation eq. (2.7) as

the base point.

$$mri(I_a) \implies \forall I_b, I_a \supseteq I_b \wedge mim(I_b) \wedge |\phi_{DB}(I_a)| < min\_freq \tag{2.10}$$

In the bacon, lettuce, and tomato example (see example 1), since there is only one maximal itemset, the minimal rare itemset set is empty.

2.8.5 Closed Discriminative Itemset Mining

Discriminative itemset mining operates on a slightly different dataset; in addition to transactions, each entry has an associated class label (positive/negative). We calculate two support values for a discriminative itemset: the support among positively labelled itemsets and the support among the negatives. A discriminative itemset is one where the difference between the positive support and the negative support is greater than a frequency threshold [CYHH07]. In closed discriminative itemset mining (CDIM), we also add the additional closedness condition.

We have previously defined the cover set $\phi(I)$ over DB eq. (2.6). We can define the class label feature of each transaction over DB with $\psi(DB(t))$ eq. (2.11).

$$\forall \in \mathbb{T}, \psi(DB(t)) \in \{0,1\} \tag{2.11}$$

The positive cover $\phi^+_{DB}(I)$ can now be defined using $\psi(DB(t))$ by the eq. (2.12).

$$\phi^+_{DB}(I) = \bigcup_{t=1}^{m} \{I | I \subseteq DB(t) \wedge \psi(DB(t)) = 1\} \tag{2.12}$$

Respectively, the negative cover is defined by eq. (2.13).

$$\phi^-_{DB}(I) = \bigcup_{t=1}^{m} \{I | I \subseteq DB(t) \wedge \psi(DB(t)) = 0\} \tag{2.13}$$

The discriminative itemset using positive and negative supports (using the covers) can be defined as eq. (2.14).

$$disc(I) \implies (|\phi^+_{DB}(I)| - |\phi^-_{DB}(I)| \geq min\_freq) \tag{2.14}$$

The closedness on the positive cover can be enforced similarly to eq. (2.8) in eq. (2.15).

$$cdisc(I_a) \implies \forall I_b, I_a \subseteq I_b \wedge |\phi^+_{DB}(I_b)| < |\phi^+_{DB}(I_a)| \tag{2.15}$$

If we want to represent this problem class in ESSENCE space, a multi-set of records should be used. This allows each record to contain the transaction and the class rather than only the transaction.

Considering the example in example 1, we need class information for the transactions first. Expanding the example by giving class 0 to the first two transactions and 1 to the last one, $\{Bacon, Lettuce, Tomato\}$ is found to be closed discriminative.

2.8.6 Relevant Subgroup Discovery

Relevant subgroup discovery (RSD) is related to discriminative itemset mining. While discriminative itemset mining relies on the support numbers of different classes of transactions, relevant subgroup discovery reasons using the actual sets of transactions that provide the support [LRA10, NDGN13]. A relevant subgroup X is an itemset such that at least one of the following conditions hold: 1) for positive transactions, no other itemset covers a superset of the transactions covered by X, 2) for negative transactions, no other itemset covers a subset of the transactions covered by X, or 3) for both kinds of transactions, no other itemset that has the same total cover is a superset of X.

In mathematical notation, considering an itemset I_a to be the relevant subgroup, the logic described in eq. (2.16) must hold.

$$rsd(I_a) \implies \forall I_b \subseteq \mathbb{I}, \bigvee_{i=0}^{2} rsd\_cond(i) \tag{2.16}$$

$$rsd\_cond(i) = \begin{cases} \phi^{+}_{DB}(I_b) \subset \phi^{+}_{DB}(I_a), & i = 0 \\ \phi^{-}_{DB}(I_b) \supset \phi^{-}_{DB}(I_a), & i = 1 \\ \phi_{DB}(I_b) = \phi(I_a) \wedge I_b \subseteq I_a, & i = 2 \end{cases} \tag{2.17}$$

For this problem, we can represent the dataset in ESSENCE using a sequence of records instead of the multi-set that we had in discriminative itemset mining. This is to allow us to store references to transactions using their index in the sequence. The decision variables are modified accordingly to encode the transaction identifiers instead of just the total number of transactions that provide the support.

Considering the example in example 1 and assuming the same class information as closed discriminative itemset mining, $\{Bacon, Lettuce, Tomato\}$ and $\{Lettuce, Tomato\}$ are found to be relevant. $\{Bacon, Lettuce, Tomato\}$ is a maximal itemset for positive covered transactions. Therefore, due to the first condition of eq. (2.17), no other frequent itemset can be included from that clause. $\{Lettuce, Tomato\}$ is found to be relevant since it appears in all transactions and subsets only one transaction: $\{\}$ (i.e. the third clause in eq. (2.17)). The second clause of eq. (2.17) doesn't directly permit any relevant subgroups, since we can not find an itemset to be included in all possible negative covers of all itemsets.

2.9 Benchmark Datasets

Throughout this thesis, we use 16 transactional datasets from the CP4IM [2] which are derived from UCI datasets to benchmark our developed systems.

Table 2.1 represents the datasets used with their characteristics. While total transaction size indicates how big a dataset is, the number of items and density of the dataset affect the difficulty to find patterns.

The figure which has been represented as the number of closed itemsets is achieved without any side constraints. Additionally, for audiology, hypothyroid

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

Dataset	Transactions	Items	Density	Closed Itemsets
Anneal	812	93	0.45	1224754
Audiology	216	148	0.45	167000000
Australian	653	125	0.41	24208803
German	1000	112	0.34	2080153
Heart	296	95	0.47	12774456
Hepatitis	137	68	0.50	1827264
Hypothyroid	3247	88	0.49	56000000
Kr-vs-Kp	3196	73	0.49	59000000
Lymph	148	68	0.40	46802
Mushroom	8124	119	0.18	3287
Soybean	630	50	0.32	2908
Splice	3190	287	0.21	1606
Tic-tac-toe	958	27	0.33	192
Tumor	336	31	0.48	31025
Vote	435	48	0.33	35771
Zoo	101	36	0.44	3292

Table 2.1: Datasets from CP4IM used in benchmarks in the thesis with their characteristics.

and kr-vs-kp datasets, the search is restrained for the first 30 minutes. For these datasets, the given number does not represent the whole count of closed itemsets.

The density is calculated using the average transaction length in comparison to the total number of items available in the dataset.

CHAPTER THREE

PATTERN MINING ON PURE ESSENCE AND ADDING SIDE CONSTRAINTS

This chapter is about modelling pattern mining problems using pure CP approaches. Modelling itemset mining problems has been investigated throughout this chapter to determine the feasibility of representing pattern mining problems in a pure CSP fashion. It includes another approach that involves iterative modelling/solving. This chapter also investigates encoding arbitrary side constraints into pattern mining models and possible interpretation differences which cause discrepancies in the solutions.

Some of the work presented in this chapter has been published in article [KAMN18b]. This chapter includes a more detailed look at the already published work and expands on it on a technical level.

3.1 Pattern Mining on Essence

The pattern mining problems which have been described in section 2.8 do not include any additional constraints. These variations of the problems can be

solved by predefined algorithmic methods given in their respective publications. Additionally, since the engineered algorithms are specifically crafted for limited type of constraints or no constraints at all, in circumstances where no additional constraints are involved they will most likely perform much better than any CP approaches. We will discuss adding additional constraints in section 3.2.

3.1.1 Modelling

It is possible to encode the FIM problem without any conditional constraints in between candidate solutions. This way of representing the problem requires only one decision variable named `freq_items` and makes the problem easy to represent. This model can be executed on a multiple solution enumeration mode.

In the ESSENCE specification, we can represent the given database as `db`, a minimum frequent itemset cardinality as `min_size`, and minimum frequency as `min_freq`. This model can be seen in fig. 3.1.

```
given db : mset of set of int
find freq_items : set (minSize min_size, maxSize max_t_size)
                        of int(db_minValue..db_maxValue)
such that
    (sum entry in db . toInt(freq_items subsetEq entry))
        >= min_freq
```

Figure 3.1: Frequent itemset mining problem modeled in CSP

In fig. 3.1, `max_t_size` represents the longest transaction in the database. This value can be automatically calculated from the database input directly since list comprehensions on givens is supported in ESSENCE. The calculation itself can be seen in fig. 3.2.

```
letting max_t_size be max([ |entry| | entry <- db ])
```

Figure 3.2: Calculating the maximum transaction size in ESSENCE.

However, the actual calculation of the longest transaction is done on the ESSENCE level. CONJURE's translated model to ESSENCE PRIME delegates to SAVILE ROW to find the value. Having ad-hoc calculations such as this one should work better in less dense databases where the database has fewer items in its transactions compared to the total set of items. However, for dense databases, using a value such as the total number of items n can be more CPU efficient with less memory overhead on representation.

Using the same principle, the calculation of the minimum and maximum item identification numbers `db_minValue` and `db_maxValue` can be calculated as seen in fig. 3.3.

```
letting db_minValue be
    min([val | entry <- db, val <- entry])
letting db_maxValue be
    max([val | entry <- db, val <- entry])
```

Figure 3.3: Calculating the minimum and maximum items in the given database in ESSENCE.

Similar to the maximum transaction size, the calculation of these two values will also bring overhead to the system. If we can guarantee that every item in $\mathbb{I}$ exists in the database, the calculations of these values can be avoided and hard-coded to the model directly with $I_{min} = 1$ and $I_{max} = n$. However, since we don't have this necessary information about any potential givens, we do not incorporate these assumptions into the model.

3.1.1.1 Representing All Solutions in a Set Variable

With a single set decision variable we can represent all of the solutions as one set of solutions. This methodology makes the CSP model complete and able to find all possible itemset solutions in a set variable.

If we would like to model MFIM this way, we can wrap the $freq\_items$ we have described earlier with another set layer. The properties of the MFIM problem can

be encoded using an additional restriction to make any itemset *i* and *i*2 not have a subset relationship (fig. 3.4).

```
given db : mset of set of int
find max_itemsets :
    set of set of int(db_minValue..db_maxValue)
such that
    forAll item_set in max_itemsets .
        (sum entry in db . toInt(item_set subsetEq entry))
            >= min_freq
such that
    forAll i in max_itemsets_so_far .
        forAll i2  in max_itemsets_so_far .
            !(i subsetEq i2)
```

Figure 3.4: Representing the maximal itemset mining problem using pure CSP with a set of solutions.

This model can produce all the solutions in a single run. However, the propagation of the model can be abysmal considering the complexity of the decision variable (i.e. set of set of integers). The expansion of this decision variable in lower-level representation can be considerably large and sub-optimal.

Evaluating this model, we examine that the model performs very badly, as expected. This model is not even able to finish the search for most examples, apart from some exceptionally small examples. Additionally, it should be noted that these findings are achieved on the maximal itemset mining model, where the complexity of the in-between candidate solutions is considered to be the weakest or easiest. Thus, it can be argued that the other pattern mining problems will be even more challenging for this single set representation approach. Cleverer methodologies to encode in-between itemset relations are necessary.

3.1.1.2 Iterative Ad-hoc Approach

As a possible approach to represent solutions efficiently, we can create an ad-hoc given variable to encode (`solutions_so_far`) and operate on a multi-run system to update this variable after each run. The multi-run system will make the solution set grow each iteration using the decision variables of the previous run.

We can direct the search by applying additional rules to split the search space into smaller batches. The pattern cardinality is a good measurement to divide the search space. The order in which itemsets are found is also important and will indicate if post-processing is necessary. With the usage of proper techniques for the order of search, we can try to eliminate the post-processing requirement. This is the methodology that sparked CDP and CDP+I. We will go into more detail about these in their relevant chapters chapter 4 and chapter 5.

The flow process of the ad-hoc iterative approach can be seen in algorithm 1.

Algorithm 1 Ad-hoc Iterative pattern mining with CP for MFIM and CFIM.

```
1: procedure AdHocIterativeMiner(D)
2:     ub ← SmartStart(D)
3:     solutions ← {}
4:     size ← ub
5:     while size ≥ 0 do
6:         s ← Solve(D, solutions, size)
7:         solutions ← solutions ∪ s
8:         size ← size − 1
9:     return solutions
```

For maximal or closed itemset mining, the ad-hoc procedure can use a maximum cardinality and reduce its value one by one. The maximum transaction size `max_t_size` is a guaranteed maximum cardinality for any itemset. However, to further shrink the domain of the upper bound of the cardinality, we can define another ad-hoc feature called `SmartStart`. This feature benefits using the efficient algorithm Eclat [Bor03]. Instead of running the algorithm to find itemsets, we run it just to find the upper bound of the largest itemset's cardinality. Without the inclusion of any side constraints, this makes our ad-hoc CP approach do redundant work since Eclat can already do the job. `SmartStart` procedure can be logical to use when there are side constraints involved.

3.1.1.2.1 Maximal Itemset Mining For maximal itemsets, the ad-hoc iterative search mechanism will be applied on the variable called `max_itemsets_so_far`. Each found solution will update this model parameter to be able to satisfy the maximality requirement for the next solutions. This can be seen in fig. 3.5.

```
given db : mset of set of int
given max_itemsets_so_far :
    set of set of int(db_minValue..db_maxValue)
find freq_items :
    set (size current_size) of int(db_minValue..db_maxValue)
such that
    (sum entry in db . toInt(freq_items subsetEq entry))
        >= min_freq
such that
    forAll item_set in max_itemsets_so_far .
        !(freq_items subsetEq i)
```

Figure 3.5: Ad-hoc CSP model for the Maximal itemset mining.

`current_size` is the pattern cardinality variable used in an ad-hoc fashion. After each search, found maximal itemsets will be added to the `max_item_sets_so_far` set. If there is no solution left on a particular cardinality, the ad-hoc variable `current_size` will be reduced by one. Otherwise, the search will continue on the same cardinality to find more maximal itemsets.

3.1.1.2.2 Closed Itemset Mining Using the same methodology, we can model the closed itemset mining problem. However, we additionally need to keep track of the support size of each set we find to compare in the next iteration. The ad-hoc modelling can be seen in fig. 3.6.

In this ad-hoc model, `freq_item` is represented by a tuple. The additional second part of the tuple represents the support value for the itemsets. `closed_item_sets_so_far` is also updated to hold a set of tuples to keep the support values with the same logic. The maximum value the support can take equals the number of transactions ($|db|$ in ESSENCE).

3.2 Adding Side Constraints

On pattern mining applications, it is crucial to consider side constraints to identify important patterns. An example of a side constraint on a pattern mining problem would be having a cost attached to each item and requiring certain limitations

```
given db : mset of set of int
given closed_item_sets_so_far :
    set of (set of int(db_minValue..db_maxValue), int(1..m))
find freq_items :
    (set
        (size current_size)
            of int(db_minValue..db_maxValue),
        int(1..db_row_size))
such that
    (sum entry in db . toInt(freq_items[1] subsetEq entry))
        = freq_items[2],
    freq_items[2] >= min_freq,
    forAll (i,support_size) in closed_item_sets_so_far .
        ((freq_items[1] subsetEq i) ->
            (freq_items[2] > support_size))
```

Figure 3.6: Ad-hoc CSP model for the closed itemset mining.

on the total cost of the pattern. As mentioned in section 3.1, specialised pattern mining algorithms can incorporate some basic side constraints into their system. However, adding arbitrary constraints is not directly feasible. This is due to specific properties of the constraints such as monotonicity and anti-monotonicity. In this section, we will look at two constraints: one monotonic (section 3.2.2) and one non-monotonic (section 3.2.3).

3.2.1 Working Example

We will first start by defining an example case and work on this example to illustrate information about side constraints. The example, which consists of a transactional database (i.e. $\mathbb{T}$) over four items (i.e. $\mathbb{I}$) (eq. (3.1)), can be seen in eq. (3.2). On this database, when we search for the frequent itemsets, we find 10 itemsets that can be seen in eq. (3.3).

$$\mathbb{I} = \{1,2,3,4\} \tag{3.1}$$

$$\mathbb{T} = \{\{1,2,4\},\{1,2,3,4\},\{3,4\}\} \tag{3.2}$$

$$FIS = \{\{\},\{1\},\{2\},\{3\},\{4\},\{1,2\},\{1,4\},\{2,4\},\{3,4\},\{1,2,4\}\} \tag{3.3}$$

We will consider the closed frequent itemset mining problem for the given example. Without any side constraints at hand, if the occurrence threshold $|\phi| = 2$, there are 2 closed frequent itemsets. The solution set ($\mathbb{S}$) can be seen in eq. (3.4).

$$\mathbb{S}_{un} = \begin{cases} \{1,2,4\}, \\ \{3,4\} \end{cases} \tag{3.4}$$

We can set side constraints using the same utility and cost values (eq. (3.5)) with a limitation on minimum utility (eq. (3.6)) and a maximum cost (eq. (3.7)).

$$UC(i) = uc[i] \mid uc = [1,2,2,1] \tag{3.5}$$

$$\sum_{i=1}^{|I|} > MIN_u \mid MIN_u = 3 \tag{3.6}$$

$$\sum_{i=1}^{|I|} < MAX_c \mid MAX_c = 3 \tag{3.7}$$

3.2.2 A Monotonic Side Constraint - Minimum Utility

If we only consider minimum utility side constraint, the correct amount of solutions is 2. The solution set ($\mathbb{S}$) can be seen in eq. (3.8).

$$\mathbb{S}_{ut} = \begin{cases} \{1,2,4\}, \\ \{3,4\} \end{cases} \tag{3.8}$$

When using other CP methods, such as MININGZINC, some of the options produce additional candidate solutions as well. Those two additional candidate

solutions in a solution set ($\mathbb{S}$) are in eq. (3.9).

$$\bar{\mathbb{S}}^{m}_{ut} = \begin{cases} \{2,4\}, \\ \{1,2\} \end{cases} \tag{3.9}$$

These offered solutions are not closed since both of them have the same support size 2 as their super-set $\{1,2,4\}$.

This discrepancy occurs because the minimum utility constraint is a monotonic constraint, which indicates that if set S violates the constraint, the subsets of S will also violate that. Subsequently, closure can be applied anywhere in the process. Still, some other MININGZINC options provide incorrect results due to incorrect closure.

3.2.3 A Non Monotonic Side Constraint - Maximum Cost

When we consider maximum cost constraint, we uncover interesting findings in the comparison with other methods. The correct result count is 5 both when calculated manually and with our iterative approach. The correct results (solution set $\mathbb{S}$) are available at eq. (3.10).

$$\mathbb{S}_{c} = \begin{cases} \{1,2\}, \\ \{1,4\}, \\ \{2,4\}, \\ \{3,4\}, \\ \{4\} \end{cases} \tag{3.10}$$

Other itemset mining approaches with CP apply the closure constraint first and then apply side constraints. Similarly, a system that uses the specialised algorithm LCM with CP post-processing can initially find $\{1,2,4\}$ as closed itemset. However, the post-processing filters it out since the pattern does not satisfy the side constraint. This eliminates some sub-patterns under $\{1,2,4\}$. This two-step system only finds two solutions (solution set $\mathbb{S}$), which are in eq. (3.11).

$$\mathbb{S}_c^m = \begin{cases} \{3,4\}, \\ \{4\} \end{cases} \tag{3.11}$$

The reason behind this behaviour is that the maximum cost constraint is not monotonic and does not carry any information about the subsets of S if the constraints are violated by S [BL04]. Meanwhile, the max cost constraint is also anti-monotonic, meaning if the pattern set S satisfies the constraint, the sub-sets directly satisfies it. This means if the maximum cost is not reached by the set S, the subsets are not going to reach it either. Due to the non-monotonic property (and more strictly anti-monotonic), two-step approaches only keep $\{3,4\}$ and $\{4\}$ correctly in the solution set.

Additionally, some MININGZINC options do not end up with the correct number of solutions either, as they provide $\{\}$, $\{1\}$, $\{2\}$, $\{3\}$ as correct patterns. These solutions are not closed since a super-set of this patterns offers the same support size. $\{1\}$ and $\{2\}$ have the same support size as $\{1,2\}$, which is 2. $\{3\}$ has the same support size as $\{3,4\}$, which is 2 again. The empty set has the same support size $\{4\}$, which is 3.

Having a different number of solutions on different CP approaches with maximum cost constraints is not desired. However, the discrepancy needs to be identified to detect why this is the case.

Difference in behaviour between our approach and other approaches boils down to the order of application of the different constraints. This inevitably leads to solving two different problems. In this circumstance, we can indicate two possible definitions when we consider the closed frequent itemset mining with arbitrary side constraints: 1) all closed frequent itemsets that also satisfy the side constraint, and 2) all frequent itemsets that satisfy the side constraint and are closed within this solution set.

The former is strictly less useful since when we remove a closed frequent itemset from the solution set due to the side constraint, we might also remove several of its subsets. In other words, we remove an itemset which was compressed

in a lossless way. By removing the compressed version directly, we lose all the information it contains as well.

We can also demonstrate the difference between the two order of operations with a side by side comparison in Figure 3.7.

(Step 1) Database	(Step 2) Closed Itemsets	(Step 3) Closed and Low-Cost
{{1,2,4}, {1,2,3,4}, {3,4}}	{{4}, {3,4}, {1,2,4}}	{{4},{3,4}}

(Step 1) Database	(Step 2) Frequent Itemsets	(Step 3) Low-Cost	(Step 4) Low-Cost and Closed
{{1,2,4}, {1,2,3,4}, {3,4}}	{{},{1},{2},{3}, {4},{1,2},{1,4}, {2,4},{3,4}, {1,2,4}}	{{},{1},{2},{3}, {4},{1,2},{1,4}, {2,4},{3,4}}	{{4},{1,2},{1,4}, {2,4},{3,4}}

Figure 3.7: Visualisation of the difference between the order of application of side constraints and the closedness constraint.

As mentioned earlier, losing $\{1,2,4\}$ in the first approach leads to losing $\{1,2\},\{1,4\}$, and $\{2,4\}$, which we deem important. They satisfy the available side constraints and they are closed since all of their supersets are eliminated via the closedness property or side constraints.

3.3 Preliminary Results

Some initial experiments are conducted using the ad-hoc iterative system and a couple of options from MININGZINC. The experiments are done on a handful of instances using the hypothyroid dataset. The instances and their generation will be explained in section 8.1. From the generated instances, we used different transaction size samples to measure the effect of the scalability of each option.

Due to the possible discrepancies in the number of solutions, a non-monotonic max-cost constraint has not been used in these experiments. Starting from the next chapter, where we develop a new framework (chapter 4), we will take non-monotonic side constraints into consideration.

In our experiment, to further reduce the number of solutions, we limit the minimum pattern size so the CP systems can handle the problem more easily. This constraint can either be enforced during the ad-hoc process or added to the ESSENCE specification by adding a constraint or using the supported set parameter `minSize`.

We also used multiple solver backends to get an initial idea about which backend and solvers are more suitable for the frequent itemset mining problem.

As stated earlier, we have used two distinct solving platforms to conduct the experiments: our ad-hoc iterative miner (AD) and MININGZINC (MZN).

For the ad-hoc iterative miner, we considered two pre-processing options available in SAVILE ROW: 1) no preprocessing (None) and 2) default pre-processing with SACBounds (S). As targeted solvers, we used two AllSAT solvers NBC_MINISAT_ALL (NBC) and BC_MINISAT_ALL (BC); one standard SAT solver LINGELING (LIN); and one CP solver MINION (MIN).

For MININGZINC options, we have used the five different options where MININGZINC offers them as the best candidate. These are: LCM5 for whole process(`MZN_LCM5`), LCM5 + Gecode postprocessing (`MZN_LCM5_HYB`), LCM2 + Gecode post-processing (`MZN_LCM2_HYB`), Gecode with reify and rewrites (`MZN_GECODE`), and Plain Gecode (`MZN_GECODE_NOWR`).

The results can be seen in table 3.1. The configurations which always timeout has been omitted from the table. The full plot with all configurations can be seen in fig. A.1 at appendix A.

The results show that using a moderately large dataset like hypothyroid can be challenging even for options with specialised tools like LCM. A reason why hybrid options are under-performing is that they try to enumerate millions of candidate solutions before the post-processing step, ending in a timeout. Using

Size	MZN_GECODE	AD_MIN_None	AD_NBC_None	AD_NBC_SAC
500	**12.94**	5493.582	151.421	130.985
1000	*	*	**730.37**	754.559
1500	*	*	**1767.64**	3184.326
2000	*	*	**4495.803**	4643.016
2500	*	*	7898.161	**7104.976**
3000	*	*	9566.493	**8763.414**

Table 3.1: Preliminary results for the ad-hoc iterative miner and its comparison to a handful of MININGZINC options. The size value indicates the number of transactions in the subset of the dataset. Time values are represented in seconds. The timeout threshold is 3 hours.

only Gecode is a sensible option only for the smallest subset of the instance. Scaling using Gecode is not possible as we can see in the later instance sizes. Our ad-hoc approach with MINION solver does not look feasible after the initial smallest instance either.

Regarding SAT solvers, one SAT solver, NBC_MINISAT_ALL, out-performs through the experiments while LINGELING and BC_MINISAT_ALL struggle. We believe the reason behind this is the usage of blocking clauses overwhelms the solver and makes scaling difficult.

The pre-processing options on the SAVILE ROW level are attached with overhead cost. This leads to having no pre-processing option `AD_NBC_None` out-performing `AD_NBC_SAC` on small subset instances. However, with the growing size of the instance, we can see that the pre-processing starts to be more beneficial. With its effects, `AD_NBC_SAC` and `AD_NBC_None` switch places.

3.3.1 A Case Study on the Number of Solutions

In addition to having a solver base comparison, we have decided to look into the solution characteristics as well.

Since we are not comparing solvers in this section, we have used the default ESSENCE pipeline with MINION solver backend. We performed this experiment with a relatively smaller dataset Tumor to achieve a faster turnover.

For MFIM, the tumor dataset has 2043 maximal itemsets. The size of these

itemsets can be seen in the table 3.2.

Cardinality	Number of Itemsets	Solver Time (s)
11	51	0.7
10	269	0.9
9	446	1.6
8	330	2.8
7	241	4.7
6	330	8.8
5	128	6.8
4	43	5.4
3	7	3.3
2	1	0.4
1	0	0.4
0	0	0

Table 3.2: Maximal itemset finding on Tumor Database using CONJURE SAVILE ROW MINION pipeline

The ad-hoc methodology creates additional translation time. This step concludes with SAVILE ROW and CONJURE taking significantly longer times. While the solver is always bound to 10 seconds as a maximum, an additional SAVILE ROW time is also around 10 seconds each step. Furthermore, CONJURE also takes a significant amount of time for solutions to be translated back into ESSENCE space.

For CFIM, the same Tumor dataset has 31024 closed itemsets available. Itemsets found in each step and the solver time can be seen in the table 3.3. SAVILE ROW times are maximum 50 seconds each steps and again CONJURE takes quite some time for solution translations.

While the solver times are much greater than the MFIM equivalent, the SAVILE ROW time stays around the same at each level. This indicates a constant overhead in SAVILE ROW when using the ad-hoc iterative system.

Both of these results indicate that our ad-hoc iterative approach has room for improvement.

Cardinality	Number of Itemsets	Solver Time
11	51	0.4
10	605	0.8
9	2449	3.4
8	5360	29.5
7	7179	126.4
6	6772	264.5
5	4841	305.4
4	2587	229.2
3	945	111.2
2	211	38.4
1	45	15.8
0	0	0

Table 3.3: Closed itemset finding on Tumor Database using CONJURE SAVILE ROW MINION pipeline

3.3.2 Summary

The experiments with our ad-hoc approach show that our proposed system is capable to solving some of the pattern mining problems with a handful of instances and already has a competitive advantage over the options in the MININGZINC platform. However, the results also show that the system is not efficient enough due to its ad-hoc nature. An improved system with a better layout and better formalisation could be more practical for research purposes and end-user use cases.

CHAPTER FOUR

PATTERN MINING ON ESSENCE WITH CDP

On the previous chapter 3, we explored pure ESSENCE mining on certain pattern mining problems and conducted some preliminary experiments (section 3.3). These results show that using pure ESSENCE constructs is not efficient enough to tackle the pattern mining problems.

This chapter is a continuation of our efforts to solve pattern mining problems which cannot be tackled efficiently. To do so, in this chapter, we introduce a new formalisation and generalise expressing solution dominance relations with Constraint Dominance Programming (CDP). Using CDP, we focus on pattern mining problems and Constraint Optimisation Problems (COP).

Some of the work presented in this chapter has been published in articles [KAMN18b, KAMN18a]. This chapter improves and expands the published work at a technical level to make it applicable to optimisation problems and more.

4.1 CDP

A constraint satisfaction problem is a triple (V, D, C) of decision variables, domains and constraints. A constraint dominance problem extends a constraint satisfaction problem to a quadruple (V, D, C, R) by adding a dominance relation R [NDGN13]. Dominance blocking constraints are generated from an existing solution using a

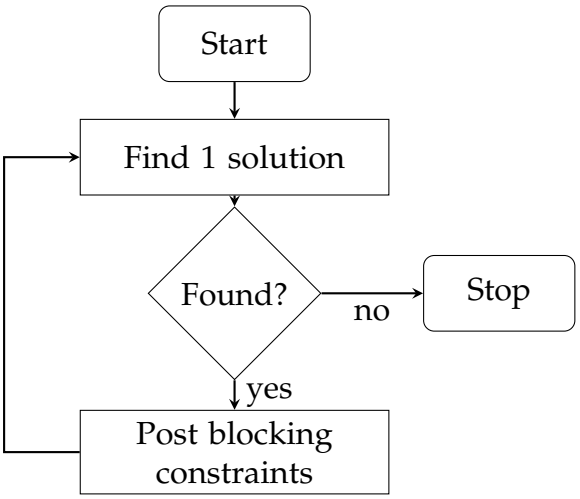


Figure 4.1: Procedure of Constraint Dominance Programming in flowchart form.

template provided by the modeller. They are used to prune all *dominated* solutions by the solution at hand.

When solving a constraint dominance problem, the goal is to enumerate all non-dominated solutions. Semantically, a solution is non-dominated if the dominance relation statement evaluates to true in comparison with every other solution. Operationally, this is achieved by iteratively finding a solution s, posting dominance blocking constraints to disallow solutions dominated by s, and using the modified model to find the next solution [GST18]. This procedure creates as many dominance blocking constraints as there are solutions and requires one solver call per solution.

Moreover, without a perfect search order, the system may produce solutions that gets dominated by the solutions later on, which a post-processing step is required to remove. A perfect search order guarantees that we only find all non-dominated solutions by arranging the order of finding solutions where an early solution can never get dominated by a later one.

The flow of the procedure can be seen in Figure 4.1. The algorithm shows that it is possible to apply pre-processing methods on the standard sub-CSP problem, modifying (V, D, C) before the iterative process.

To understand the dominance problem we can use a mathematical example.

This example problem is equivalent to MFIM and it can be seen on Equation (4.1). The dominance condition indicates that the future possible solutions should not be a subset of any found solution. After finding one of the solutions, (e.g. $\{1,2\}$, subsets $\{1\},\{2\}$) dominated candidate solutions are blocked from the possibilities.

$$\left\{\begin{array}{c} \text{Example Solutions} = \{1,2\},\{1,3\} \\ \text{Dominance Condition} = \quad \nexists S_{future}\ \forall S \mid S_{future} \subseteq S \\ \text{Dominated Solutions} = \{1\},\{2\},\{3\} \end{array}\right\} \tag{4.1}$$

Pure CDP has a number of shortcomings in the context of constraint-based mining:

- When the number of solutions is large, CDP's requirement of calling the solvers once per solution creates an overhead, as it incrementally adds a new dominance blocking constraint after each solution. This creates an unnecessary overhead, and in addition, it reduces the utility of learnt clauses in a learning solver. It might also evaluate the same sub-trees of searches multiple times since this information is lost at the end of the solver call.
- Without a good search ordering, CDP might enumerate dominated solutions as well. The number of dominated solutions is typically orders of magnitude greater than non-dominated solutions.
- Post-processing is required to remove dominated solutions found during search.

4.2 Modelling Problems in CDP

To be able to use the CDP framework, the selected problem needs to be examined and determined to be feasible for CDP primitives. The section analyses the pattern mining problems from section 2.8 in section 4.2.1 and a general optimisation problem in section 4.2.2.

4.2.1 Modelling of Pattern Mining Problems

The maximal itemset mining MFIM (section 2.8.1) can be represented in the dominance logic by definition 4.2.1 using *is* as the candidate itemset. *prev()* denotes accessing any previous assignments.

Definition 4.2.1. The dominance relation of maximality is: $\neg(is \subseteq prev(is)))$.

For closed itemset mining, CFIM (section 2.8.2), the representation is definition 4.2.2.

Definition 4.2.2. The dominance relation of closedness is: $(is \subseteq prev(is)) \implies (s(is) \neq prev(s(is)))$.

For generator itemset mining, GFIM (section 2.8.3), the representation is definition 4.2.3.

Definition 4.2.3. The dominance relation, $(prev(is) \subseteq is) \implies (s(is) \neq prev(s(is)))$ follows the definition very closely. This verifies that a frequent itemset is a generator itemset if its support is not equal to the support of any of its subsets.

For minimal rare itemset mining, MRIM (section 2.8.4), the dominance representation is definition 4.2.4.

Definition 4.2.4. The dominance relation follows from the definition of minimal rare itemsets: an itemset is a solution if none of its subsets are solutions: $\neg(prev(is) \subseteq is)$.

For closed discriminative itemset mining CDIM (section 2.8.5), the dominance representation is definition 4.2.5.

Definition 4.2.5. After calculating the positive and negative support of the itemset with limiting constraints, we can apply the closure dominance on positive cover as follows: $(is \subseteq prev(is)) \implies (s_+(is) > prev(s_+(is))$, where $s_+()$ is used to access the positive support.

Lastly, for relevant subgroups discovery problem RSD (section 2.8.6), the dominance representation is given in definition 4.2.6 where c_+ and c_- is used for positive and negative cover on transactions.

Definition 4.2.6. The dominance relation for RSD is fairly more complex, but it follows the definition given in eq. (4.2).

$$\bigvee \begin{cases} \neg(c_+(is) \subseteq prev(c_+(is))) \\ \neg(prev(c_-(is)) \subseteq c_-(is)) \\ \neg((c_+(is) \cup c_-(is) = prev(c_+(is)) \cup prev(c_-(is))) \implies is \subseteq prev(is)) \end{cases} \tag{4.2}$$

4.2.2 Modelling Optimisation Problems using CDP

While we can model the pattern mining problems with CDP directly, it is also possible to model COP using CDP primitives indirectly.

Definition 4.2.7. A minimisation optimisation can be expressed using the dominance relation with $f(S) \leq f(prev(S))$ where $f(S)$ is the optimisation function for feasible assignment S

By enforcing the optimisation variable to reduce at each iteration, we can find the optimal solution at the last solution. This procedure can also produce optimally equivalent solutions that have the same optimisation value.

The disadvantage of modelling an optimisation problem using CDP is that the intermediate solutions found by CDP are actually not useful for COP. In other words, we are spending extra effort to find the intermediate solutions unnecessarily. However, they might be useful on certain statistical analysis where the goal is to check the distribution of the optimisation variable among upcoming potential solutions. Another possible space where CDP can be preferred over COP is interactive scenarios. By its abstract declarative nature, it outputs more

information compared to COP solving systems. This can be useful for interactive solving systems.

4.3 Implementation in Essence

To bring CDP to ESSENCE specification, a couple of new concepts and their related keywords should be defined.

```
language Essence 1.3
letting ITEM be domain int(...)
letting SUPPORT be domain int(...)
given db : mset of set of ITEM
given minSupport : int
find itemset: set of ITEM
find support: SUPPORT
such that
 support = sum entry in db .
            toInt(itemset subsetEq entry),
 support >= minSupport,
 AdditionalSideConstraints
```

```
dominanceRelation
  (itemset subsetEq fromSolution(itemset))
  -> (support != fromSolution(support))
```

Figure 4.2: Closed Frequent Itemset Mining modelled in ESSENCE with CDP constructs. The dominance relation defines the closedness property between the currently sought solution and the previous solutions via `fromSolution`.

An ESSENCE specification translation of a similar CDP specification to Equation (4.1), which is CFIM instead of MFIM, can be seen in Figure 4.2. While the top segment of the model is the same as a pure CSP based model to find frequent itemsets, the dominance relation defines the constraint in-between candidate solutions. In the second segment, the dominance relation condition is denoted

as *dominanceRelation*, while a new keyword *fromSolution* is defined to access previously found solutions.

These concepts are implemented in the ESSENCE pipeline, beginning with CONJURE. CONJURE refines the dominance relation condition in the same way as the model to transform ESSENCE level abstractions to ESSENCE PRIME level.

As pointed out in section 2.3.1, there are multiple ways to translate a high-level ESSENCE type and its operation. A single high-level ESSENCE type can be represented in multiple ways. For instance, a set of integers can be represented via different options: an occurrence representation or any of the multiple explicit representations. With the addition of the CDP to the ESSENCE language, these representation choices are applied to the dominance relation condition as well.

Each representation choice has its benefits and downsides. While one representation can lead to a smaller encoding size in the final model, another model might perform faster. Since we are using an iterative structure in CDP, the effect of the representation is even more important. This is due to the effect being magnified with each successive iteration as either performance improvement stays steady or all possible bottlenecks accumulate for the final result.

One example translation of the dominance relation from the Figure 4.2 can be seen in Figure 4.3. This translation uses the occurrence representation for the high-level ESSENCE set type. The set is represented by a matrix of Booleans and the subset operation becomes the implication of all possible domain values on an iteration. If a candidate value is present in the left part of the implication (ie. *freq_items* list), it is possible to have the same value present or not present on the right side (ie. *fromSolution* counterpart) to be able to comply with the subset rule. But if the left part of the implication indicates that value is not present, the right side also needs to not have the value.

After the model transformation to ESSENCE PRIME, if any parameters exist, they are also translated to ESSENCE PRIME level. The next step in the process is the ESSENCE PRIME level CDP model is taken by SAVILE ROW and processed for solver usage.

```
dominanceRelation(
    and([
        freq_items_1_Occurrence[q3] ->
            fromSolution(freq_items_1_Occurrence)[q3]
                        | q3 : int(db_minValue..db_maxValue)
    ]) -> freq_items_2 > fromSolution(freq_items_2))
```

Figure 4.3: One possible ESSENCE PRIME translation of the dominance relation from the ESSENCE level specification of the CDP for the model in fig. 4.2. This possible translation uses the occurrence representation for the ESSENCE level set type.

Another example of dominance relation, which is a COP this time, in ESSENCE can be seen in fig. 4.4.

```
    ...
find totalCost : int(...)
    ...
```

```
minimising totalCost
```

```
dominanceRelation (totalCost <= fromSolution(totalCost))
```

Figure 4.4: Modelling a COP using CDP assuming the decision variable totalCost is the optimisation variable. The second block of minimising can be replaced with the dominanceRelation to express the criteria with dominance.

Algorithm 2 CDP

1: $(V,D,C,R) \leftarrow CDP$
2: $AC(V,D,C)$ ▷ Singleton Arc Consistency preprocessing
3: **while** $True$ **do**
4: $\quad CSP \leftarrow (V,D,C)$
5: $\quad S \leftarrow findSolution(CSP)$
6: $\quad$ **if** $S = \emptyset$ **then**
7: $\quad\quad break$
8: $\quad B \leftarrow generateDominance(R,S)$
9: $\quad C \leftarrow C \cup B$

Applying the Algorithm 2 in principle, SAVILE ROW reduces the domains using Singleton Arc Consistency (SAC) [NAG+14] (first mentioned in section 2.3.1) to the core CSP model and creates an initial solver level encoding of the CDP model

without the dominance relation. Afterwards, SAVILE ROW operates the low-level solver to find solutions one by one. With each solution, a new dominance blocking (nogood) constraint is generated. This new constraint can be encoded into the low-level solver to enforce dominance relation rules.

On low-level solvers, new constraints can be encoded in different ways. The first and the most trivial way is to access low-level solvers using a written model on a file. For some CP solvers like MINION, it is possible to append new constraints at the end of the CP model. Some other solver encodings like flatzinc require a strict order in the model file where the location of the constraints cannot be arbitrary. This requirement enforces a CDP model to be rewritten in each step on flatzinc. Whereas, on the MINION level, the cost of updating the model is only the I/O attached to the append process.

A more complex way of interfacing low-level solvers to eliminate the cost of updating is using a memory level alteration of the running model. We will talk more about this method in detail in chapter 7.

4.4 Comparison with Current Techniques

We have tested our methodology CDP on pattern mining problems with sixteen datasets listed on the CP4IM website for one problem class CFIM. For more details on instances, refer to chapter 8.

We design two closed itemset mining experiments with side constraints. We compare the performance of our method against already existing methods that are also capable of handling any type of arbitrary side constraints correctly. This includes monotonic side constraints such as minimum utility in the section 3.2.2 and non-monotonic side constraints such as maximum cost in the section 3.2.3. Thus the comparison includes multiple methods available on MiningZinc. However, some other methods are not viable to apply due to how the non-monotonic constraints are handled. The effect of monotonicity of side constraints to the problem has been explained in section 3.2.3 with more detail in [BL04].

Additionally, some other methodologies such as CoverSize [SAG17] and ClosedPattern [LLL+16], which provide direct propagators for closedness, are also not included in this comparison due to not being directly applicable with non-monotonic side constraints. However, these additional propagation techniques can be fruitfully integrated into CDP to gain additional performance.

The comparison point MiningZinc works with a declarative model and calculates a number of execution plans after inspecting a given model. It is hard to estimate the relative performance of these execution plans, so in the interest of fairness, we ran every execution plan offered by MiningZinc in our experiments.

The first experiment uses a minimum utility constraint. We compare our computational results against the MiningZinc model given in Figure 8 of [GDN+17]. The second experiment uses a max-cost constraint in addition to the minimum utility constraint. The max-cost constraint can be written in the MiningZinc language in a similar way to the minimum utility constraint in the model of Figure 8 of [GDN+17]. MiningZinc produces a number of execution plans when provided with this model. However, all of these execution plans produce faulty answers. The max-cost constraint is not monotonic, so the MiningZinc execution plans suffer from the problem we describe in Section 3.2.3.

Experiments were performed with 16 processes in parallel on a 32-core AMD Opteron 6272 at 2.1 GHz with 256 GB RAM (for more details see chapter 8). By default, MiningZinc uses a fixed directory for its temporary files, which precludes us from running multiple MiningZinc processes at the same time. Some modifications have been done on the MiningZinc source code to use a different temporary directory for each of its invocations.

4.4.1 CFIM with Minimum Utility Only

In order to experiment with CFIM with a minimum utility side constraint, utility values for each item and a threshold value for total utility has been assigned. A more detailed explanation of instance generation can be seen at the section 8.1. The modelling of the problem in ESSENCE specification, which has been given

in Figure 4.2, is altered with the addition of the minimum utility side constraint. We add two parameters and a single arithmetic constraint as listed in fig. 4.5. We experiment with the same frequency thresholds (10%, 20%, 30%, 40% and 50%) as [GDN+17].

```
given utility_values : matrix indexed by [ITEM] of int(0..5)
given min_utility : int
such that
 (sum item in itemset . utility_values[item]) >= min_utility
```

Figure 4.5: Minimum utility side constraint in ESSENCE specification for itemset mining problem given in fig. 4.2.

As mentioned previously in section 4.4, MiningZinc calculates multiple execution plans, and in this case eighteen execution plans, for CFIM. Plans 1–2 run a specialised algorithm (i.e. Eclat) to find closed itemsets, followed by different ways of post-processing/filtering the closed itemsets using a constraint programming-based approach. Plans 3–6 run a pre-processing step followed by different configurations of Gecode [SLT06]. Plans 7–14 run Gecode without a pre-processing step. Plans 15–18 run LCM (v2 and v5) to find closed itemsets, followed by a constraint programming based approach to filtering the side constraints.

All of these execution plans either calculate closed itemsets first and then apply side constraints, or contain the closedness constraint inside a constraint model and apply it simultaneously with the side constraints.

For sixteen datasets and five different frequency levels, we use 80 instances. With a 3-hour time limit, we are able to solve all but 6 of these instances. Since the minimum utility constraint is monotonic, all MININGZINC options should be feasible to use in this circumstance. Thus, we run all execution plans produced by MiningZinc for comparison. However, 12 plans out of 18 produce an incorrect number of solutions for at least one instance. Therefore, we have only included 6 plans out of 18 in our comparison. Our contact with the creators of MiningZinc has led us to believe this is due to software-related bugs, but the situation has not

Instance	MZ-11	MZ-13	MZ-12	MZ-14	MZ-6	MZ-5	CDP	CDPs	NbSols
lymph-50	15	**5**	10	10	**5**	9	16	1	1,492
lymph-40	33	**8**	23	23	9	20	17	2	1,711
lymph-30	103	29	93	91	32	80	**24**	6	3,902
lymph-20	194	60	202	203	64	188	**38**	17	6,322
lymph-10	920	526	987	1,263	447	1,002	**165**	126	22,606
krvskp-50	*	3,896	282	281	3,911	**274**	1,001	872	381
krvskp-40	*	6,774	794	792	6,901	**761**	2,221	2,079	1,035
krvskp-30	*	*	3,057	3,057	*	**2,971**	4,735	4,531	4,371
krvskp-20	*	*	*	*	*	*	*	*	*
krvskp-10	*	*	*	*	*	*	*	*	*
hypo-50	*	*	*	*	*	*	**4,984**	4,498	30,950
hypo-40	*	*	*	*	*	*	**2,455**	2,281	1,629
hypo-30	*	*	*	*	*	*	**7,800**	7,574	2,748
hypo-20	*	*	*	*	*	*	**5,585**	5,358	39
hypo-10	*	*	*	*	*	*	**6,036**	5,727	481
hepatitis-50	15	**9**	15	15	**9**	15	21	6	3,590
hepatitis-40	67	**37**	72	73	39	68	54	31	12,587
hepatitis-30	322	195	256	254	200	237	**83**	54	18,139
hepatitis-20	1,862	1,402	1,192	1,191	1,452	1,119	**214**	175	23,379
hepatitis-10	10,368	8,614	3,283	3,356	9,457	2,962	**251**	211	17,685
heart-50	55	**16**	18	17	**16**	**16**	27	8	483
heart-40	254	107	122	120	110	113	**92**	68	3,630
heart-30	1,469	760	521	516	774	485	**298**	264	7,165
heart-20	*	8,219	7,561	7,585	8,082	6,564	**3,367**	3,224	68,040
heart-10	*	*	*	*	*	*	**4,380**	4,296	19,053
german-50	98	36	**23**	**23**	35	**23**	72	27	377
german-40	354	136	127	128	140	**120**	182	125	1,969
german-30	1,479	662	893	883	671	844	**620**	519	9,087
german-20	7,049	3,532	6,371	6,369	3,618	5,683	**1,904**	1,614	40,311
german-10	*	*	6,780	7,288	*	6,485	**4,730**	4,633	5,581
australian-50	479	200	182	177	204	177	**166**	118	2,093
australian-40	2,860	1,688	1,182	1,174	1,764	1,094	**553**	492	4,474
australian-30	*	*	10,672	10,647	*	10,272	**2,131**	2,003	16,020
australian-20	*	*	*	*	*	*	**9,044**	8,862	22,374
australian-10	*	*	*	*	*	*	*	*	*
audiology-50	*	*	1,123	1,115	*	1,025	**265**	17	6,699
audiology-40	*	*	436	433	*	370	**274**	14	8,283
audiology-30	*	4,688	423	417	4,751	349	**259**	14	7,357
audiology-20	*	*	1,940	1,950	*	1,647	**260**	28	9,667
audiology-10	*	*	5,081	5,113	*	4,591	**254**	33	6,761
anneal-50	*	*	*	*	*	*	**411**	335	17,456
anneal-40	*	*	*	*	*	*	**1,595**	1,474	36,041
anneal-30	*	*	*	*	*	*	**1,150**	1,048	25,487
anneal-20	*	*	*	*	*	*	**1,672**	1,572	24,728
anneal-10	*	*	*	*	*	*	**2,809**	2,683	32,689

Table 4.1: Closed Itemset Mining with minimum utility side constraint experiment results on 9 datasets.
Times are in seconds (* indicates a 3-hour timeout).

been resolved.

We compare CDP run-time against the run-times of the MiningZinc execution plans and indicate the winner using a bold font. In addition, we provide the time taken by only the AllSAT solver for our method in the CDPs column. In general, the modelling overhead is small. However, in some cases (for example in the audiology dataset) there is a significant difference between the total time and the solver time.

On the harder instances of the `hypothyroid` dataset our method is the only one that finishes before the time limit.

Overall, our method, which does not require the user to select among a large set of execution plans, reliably finds correct results and is competitive with the six of the eighteen execution plans produced by MiningZinc. This is despite not exploiting the monotonicity of the minimum utility constraint. However, the greatest benefit of our approach is its generality. In the next section, we analyse our performance in the presence of a constraint that is not monotone.

4.4.2 CFIS with Minimum Utility and Maximum Cost Side Constraints

We experiment with a combined minimum utility and max-cost side constraint on CFIM. To generate costs per item and a cost threshold, we follow a similar procedure to that of generating utility values and the utility threshold.

The cost values are uniformly randomly chosen to a value between 0 and 5. We also maintain the utility values that were previously generated. A cost threshold and a utility threshold is chosen to limit the number of closed frequent itemsets to at most tens of thousands of closed frequent itemsets.

We experiment with the same frequency thresholds (10%, 20%, 30%, 40% and 50%).

Extending our ESSENCE problem specification on fig. 4.2 to work with the max-cost constraint is trivial. We add two more parameters and another arithmetic constraint as listed in fig. 4.6. We add the necessary statements for the parameter

Instance	MZ-11	MZ-13	MZ-12	MZ-14	MZ-6	MZ-5	CDP	CDPs	NbSols
lymph-50	15	**5**	10	10	**5**	9	16	1	1,492
lymph-40	33	**8**	23	23	9	20	17	2	1,711
lymph-30	103	29	93	91	32	80	**24**	6	3,902
lymph-20	194	60	202	203	64	188	**38**	17	6,322
lymph-10	920	526	987	1,263	447	1,002	**165**	126	22,606
krvskp-50	*	3,896	282	281	3,911	**274**	1,001	872	381
krvskp-40	*	6,774	794	792	6,901	**761**	2,221	2,079	1,035
krvskp-30	*	*	3,057	3,057	*	**2,971**	4,735	4,531	4,371
krvskp-20	*	*	*	*	*	*	*	*	*
krvskp-10	*	*	*	*	*	*	*	*	*
hypo-50	*	*	*	*	*	*	**4,984**	4,498	30,950
hypo-40	*	*	*	*	*	*	**2,455**	2,281	1,629
hypo-30	*	*	*	*	*	*	**7,800**	7,574	2,748
hypo-20	*	*	*	*	*	*	**5,585**	5,358	39
hypo-10	*	*	*	*	*	*	**6,036**	5,727	481
hepatitis-50	15	**9**	15	15	**9**	15	21	6	3,590
hepatitis-40	67	**37**	72	73	39	68	54	31	12,587
hepatitis-30	322	195	256	254	200	237	**83**	54	18,139
hepatitis-20	1,862	1,402	1,192	1,191	1,452	1,119	**214**	175	23,379
hepatitis-10	10,368	8,614	3,283	3,356	9,457	2,962	**251**	211	17,685
heart-50	55	**16**	18	17	**16**	**16**	27	8	483
heart-40	254	107	122	120	110	113	**92**	68	3,630
heart-30	1,469	760	521	516	774	485	**298**	264	7,165
heart-20	*	8,219	7,561	7,585	8,082	6,564	**3,367**	3,224	68,040
heart-10	*	*	*	*	*	*	**4,380**	4,296	19,053
german-50	98	36	**23**	**23**	35	**23**	72	27	377
german-40	354	136	127	128	140	**120**	182	125	1,969
german-30	1,479	662	893	883	671	844	**620**	519	9,087
german-20	7,049	3,532	6,371	6,369	3,618	5,683	**1,904**	1,614	40,311
german-10	*	*	6,780	7,288	*	6,485	**4,730**	4,633	5,581
australian-50	479	200	182	177	204	177	**166**	118	2,093
australian-40	2,860	1,688	1,182	1,174	1,764	1,094	**553**	492	4,474
australian-30	*	*	10,672	10,647	*	10,272	**2,131**	2,003	16,020
australian-20	*	*	*	*	*	*	**9,044**	8,862	22,374
australian-10	*	*	*	*	*	*	*	*	*
audiology-50	*	*	1,123	1,115	*	1,025	**265**	17	6,699
audiology-40	*	*	436	433	*	370	**274**	14	8,283
audiology-30	*	4,688	423	417	4,751	349	**259**	14	7,357
audiology-20	*	*	1,940	1,950	*	1,647	**260**	28	9,667
audiology-10	*	*	5,081	5,113	*	4,591	**254**	33	6,761
anneal-50	*	*	*	*	*	*	**411**	335	17,456
anneal-40	*	*	*	*	*	*	**1,595**	1,474	36,041
anneal-30	*	*	*	*	*	*	**1,150**	1,048	25,487
anneal-20	*	*	*	*	*	*	**1,672**	1,572	24,728
anneal-10	*	*	*	*	*	*	**2,809**	2,683	32,689

Table 4.2: Closed minimum utility and maximum cost Itemset Mining on 9 datasets. Times are in seconds (* indicates a 3-hour timeout).

```
given cost_values : matrix indexed by [ITEM] of int(0..5)
given max_cost : int
such that
 (sum item in itemset . cost_values[item]) <= max_cost
```

Figure 4.6: ESSENCE specification for max-cost constraint for itemset mining problem given in fig. 4.2.

values and post an arithmetic constraint that requires the total cost of an itemset to be less than the cost threshold.

We also tried extending the MiningZinc model (from Figure 8 of [GDN+17]) similarly. Due to the issue that was explained in section 3.2.3 the extended model gives incorrect results: it misses a large number of solutions. Hence, comparing our performance to this model is not sensible.

Following the analysis of [BL04] we decided to relax the closedness condition for MiningZinc and find all frequent itemsets that satisfy the side constraints. This is the only sensible comparison since the full set of closed frequent itemsets is a lossless compression of the full set of frequent itemsets, and thus the two contain identical amounts of information. In other words, we use MiningZinc to perform the first three steps in the second table of Figure 3.7. To achieve the same results as our method, another procedure would also be needed. We were also unable to validate the results found by MININGZINC with respect to our results for this experiment since the last step (calculating the set of closed itemsets) is missing.

Table 4.2 contains the results of this experiment. We limited this experiment to the six execution plans that gave consistent results in Section 4.4.1. Often the number of closed frequent itemsets is much smaller than the number of all frequent itemsets. Therefore, directly calculating the set of closed itemsets may have less overhead. However, it is important to note that from the full set of closed itemsets (our results), it is trivial to generate the set of all frequent itemsets.

Lymphography (`lymph`) is a relatively small dataset that has 10M frequent itemsets and 47K closed itemsets (at 10% frequency). The results show that there

is an overhead in our system for easy instances. For `lymph-50` even though the solver spends 1 second, the total time is above 15 seconds. This overhead quickly becomes negligible for harder instances. In the rest of the `lymph-X` instances, the speed-up achieved by our method progressively increases as the instance becomes harder.

Kr-vs-kp (`krvskp`) is the only instance where our method is consistently slower than MiningZinc. This dataset is both very large and also seems to have a very large ratio between the number of frequent itemsets and closed frequent itemsets. Intuitively, the set of frequent itemsets of this dataset does not compress very well. Since the number of closed itemsets is very close to the number of frequent itemsets, it is not surprising that a more specialised tool that focuses on mining tasks handles this case slightly better.

Hypothyroid (`hypo`) is a medical dataset that is of a similar size to Kr-vs-kp but seems to have a lower ratio of closed itemsets to frequent itemsets. Our approach is the only one that can complete the task within the 3-hour time limit. The performance does not seem to directly depend on the number of solutions or the frequency threshold.

Hepatitis is another medical dataset that has a low ratio of frequent itemsets to close itemsets (less than 0.01%). We observe a similar phenomenon here to Lymphography. As the instances get harder (and the frequency threshold gets lower) our approach becomes much faster in comparison. We show that this behaviour does not stop at the frequency level 10% and gives results for lower frequency levels in section Section 4.4.3.

Heart-Cleveland (`heart`) is a third medical dataset that is very similar to the Hepatitis dataset in its behaviour. One difference between this dataset and the Hepatitis dataset is that this dataset has around one order of magnitude more closed frequent itemsets. The performance of our method progresses similarly to Hepatitis at a slightly larger scale.

The German-credit (`german`) and Australian-credit (`australian`) datasets contain attributes related to credit risks and credit card transactions. The German

dataset is smaller and has a ratio of (roughly) 1/20 between frequent itemsets and closed frequent itemsets. For easy instances of this dataset, the modelling overhead of our method causes us to be slower than MiningZinc. However, for harder instances and for all instances of the Australian dataset our approach is comfortably faster.

Audiology is one of the largest datasets with more than 167M closed frequent itemsets (at 10%). We were not able to calculate the ratio of the number of frequent itemsets to the number of closed itemsets since there are a very large number of both. Our method is much faster for all five instances that we tested. being challenging for this problem and always triggering a timeout inside SAVILE ROW without much gain.

Finally, the Anneal dataset contains data about steel annealing. This dataset has a very small ratio of frequent itemsets to closed frequent itemsets and hence is likely to be amenable to our method. For all instances of this dataset, all execution plans offered by MiningZinc reach the time limit, whereas our method finishes the task in less than one-third of the time limit.

Overall, in all of our experiments, we sometimes find a trade-off between different execution plans offered by MiningZinc. No single execution plan seems to give the best results for all instances. However, our CDP approach gives consistent results all around the board.

4.4.3 Lower Frequency Thresholds on Selection of Datasets

In the Lymphography, Hepatitis and Audiology datasets, our method provides the best performance for all instances by roughly one order of magnitude for the five frequency levels. In order to demonstrate the performance of our method at lower frequency levels, for these three datasets, we ran additional experiments. We compare our method with the MiningZinc execution plan 5 (since this is the best option overall in our earlier experiments). Figure 4.7 contains the run-times of the two solvers on these instances. We see that at lower frequency levels, our method presents an even bigger advantage. This is due to the decreased ratio of

closed itemsets to frequent itemsets at lower frequency levels.

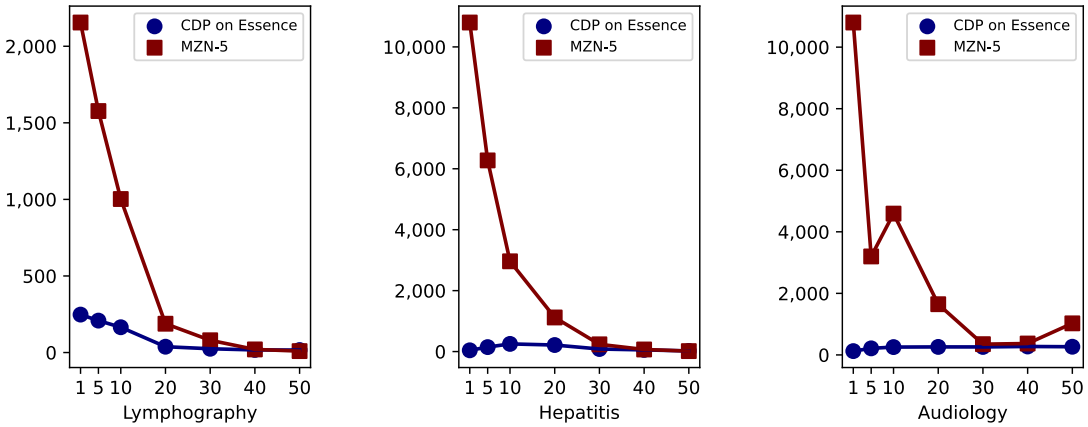


Figure 4.7: Closed Min-Utility and Max-Cost itemset mining for Lymphography, Hepatitis, and Audiology datasets at lower levels of frequency thresholds. The horizontal axes contain the frequency level that we use, and the vertical axis is time in seconds.

Chapter Five

CDP with Incomparability

This chapter builds on top of the previous chapter, where Constraint Dominance Programming (CDP) was introduced. It declares a new parameter called an incomparability condition on top of CDP to exploit solutions that are incomparable to each other. This chapter also provides a detailed empirical evaluation of the incomparability condition for CDP problems and COP.

Some of the work presented in this chapter has been published in articles [KAGM19, KAGM20a, KAGM20b]. This chapter improves the already published work and extends it in terms of technicality and applicability.

5.1 Incomparability Condition

By observing incomparable solutions and explicitly stating them in CDP, we might be able to aid the search when we enumerate all solutions. Incomparable solutions are defined as follows.

Definition 5.1.1. Solutions A and B are incomparable if neither A dominates B, nor B dominates A.

A significant number of pairs of solutions A and B in the solution set of a CDP tend to be incomparable with each other. For any pair of solutions, the dominance

blocking constraint generated from either solution is irrelevant when searching for the other solution.

We can define a new type of statement to specify incomparable solutions explicitly. This statement is only valid in a CDP problem specification, i.e. one containing a dominance relation statement. The dominance relation statement defines the dominance relation itself, similar to the dominance blocking constraints introduced in [GST18]. The incomparability function statement provides a function I mapping any solution to a single value that has an orderable ESSENCE type (typically an integer). Two solutions A and B such that $I(A) = I(B)$ are said to be incomparable.

Definition 5.1.2. A CDP+I problem specification is a quintuple (V, D, C, R, I) where R is the dominance relation similar to that of CDP and I is the incomparability function.

We can use the domain-specific information of the incomparability function I to separate the search space into distinct blocks (levels) and use them.

Definition 5.1.3. A level in a CDP+I problem comes from the discrete domain of the incomparability function I. Each discrete value which incomparability function maps to (ie. in $I(X) \to Y$ where Y is the discrete domain) can be defined as levels and they together create a complete partition of the search space.

Using levels, the CDP iterative procedure is altered and, instead of the number of solutions as the iteration point, the number of levels will be used. The new procedure can be seen in fig. 5.1.

The incomparability function partitions the search space with non-overlapping parts. Since we enumerate all solutions for each part of the partition, it necessarily checks the whole search space. However, it is not guaranteed to eliminate non dominated solutions. The reason behind this is the same reason which is examined in the section 2.5. This time instead of examining individual solutions, we can examine the level/partition of the search space as a whole and an early level can be dominated by a later level if the order of the partitions is perfectly chosen.

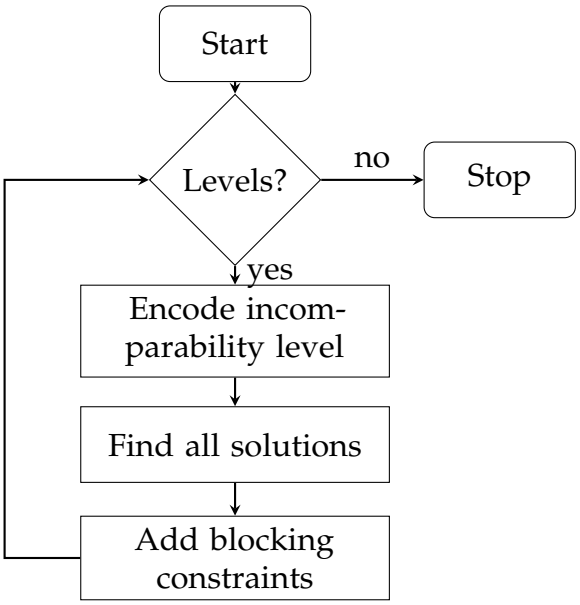


Figure 5.1: Procedure for CDP enhanced with Incomparability (CDP+I) in flowchart form.

5.2 Choosing Incomparability for Problem Classes

Once the incomparable solutions are identified, an analysis of the dominance relation can point out a sensible incomparability condition.

For MFIM, CFIM and CDIM, we can use the cardinality of the itemset is, $|is|$, as the incomparability function, since two itemsets with the same cardinality cannot dominate each other. The reason behind this lies on the left-hand side of the implication on CFIM and CDIM and the only operator on MFIM: **subset**. Different itemsets with the same cardinality cannot be subsets of each other unless they are the same set. For these problems, descending order for subset operator is the correct order to avoid dominated solutions.

The incomparability condition for generator itemsets is almost the same as for CFIM: it uses the itemset cardinality. This condition is complete when paired with an ascending direction of search on the itemset cardinality. In contrast to CFIM, smaller itemsets dominate larger ones by definition since they do not have any frequent subsets. Then, dominance blocking constraints are added and we only find generator itemsets in ascending order.

For MRIM, we can use cardinality as the incomparability condition as well. It

is complete when the search order is ascending on the itemset cardinality, since we first find small itemsets which cannot have any infrequent subsets. Then, dominance blocking constraints are added and we only find minimal rare itemsets in the successive levels.

For RSD, the descending order on the cardinality of the itemset can be applied. This incomparability function is not complete though: some dominated solutions may be found when using it. Even though it is not complete, it helps CDP+I in solution performance because it still eliminates a large number of dominated solutions.

Example 5.

$$DB = \begin{cases} T_1 = \{1,2,3,4\}, Class_1 = 1 \\ T_2 = \{1,2,3,4\}, Class_2 = 1 \\ T_3 = \{1,2,3,5\}, Class_3 = 1 \end{cases}$$

An example follows to demonstrate that the cardinality is not complete for incomparability. The itemset $\{1,2\}$ dominates $\{3,4\}$ because the positive cover of the former includes all transactions whereas the positive cover of the latter only includes the first two. This violates the first component of the dominance relation given above.

5.3 Integration in Essence

We make use of this explicit incomparability statement by enumerating all solutions that have an equal incomparability function value. This avoids the need to add any blocking constraints after each solution that has the same incomparability value. Then, all of the necessary blocking constraints are added at once before moving to the next incomparability level. This reduces the number of solver calls required, reduces the total number of dominance blocking constraints maintained, and allows the usage of efficient solution-enumeration solvers.

To do so, we also extend ESSENCE to add an `incomparabilityFunction`

statement, which allows the modeller to specify the incomparability condition. This field takes a function that points to a discrete domain such as integers or a list of discrete domains which we call multi-layered incomparability (see section 5.4). Using the same model example from the previous chapter fig. 4.2, the incomparability function would be as follows fig. 5.2.

```
incomparabilityFunction descending |itemset|
```

Figure 5.2: Incomparability function statement in ESSENCE for the model in fig. 4.2

The function is followed by an `ascending` or `descending`, which indicates in which order the values in the defined domain should be explored.

Thanks to the explicit search order specified in the incomparability function statement, we produce fewer (or in the best case no) dominated solutions. CDP enhanced with an explicit incomparability statement is implemented in CONJURE and SAVILE ROW, which will be explained in section 5.3.1.

Algorithm 3 presents the CDP+I algorithm we propose. In contrast to the pure CDP algorithm proposed in [GST18], which iterates over the solution set, our algorithm iterates over levels jointly defined. The levels correspond to the set of values that the incomparability function takes. All solutions at a particular level *i* are known to be incomparable to each other, and we exploit this by running the solver to enumerate all solutions at that level. We then generate one dominance blocking constraint per solution using the template provided by the modeller in the `dominanceRelation` statement of the model. Having access to a set of blocking constraints generated from the same template presents an opportunity that is unique to level-wise search. We optionally perform a model reformulation step provided by SAVILE ROW (line 8) to reduce the size and number of constraints and to achieve better propagation. In this work, we use the partial evaluator and the common subexpression elimination methods (Section 5.3.1).

The difference from the standard CDP algorithm is that the loop is running around the captured incomparability level rather than around an individual

Algorithm 3 CDP+I

```
1: (V,D,C,R,I) ← CDP+I
2: SAC(V,D,C)
3: levels ← getLevels(I)
4: for l ← levels do
5:     CSP ← (V,D,C + levelRestriction(l))
6:     S ← findAllSolutions(CSP)
7:     B ← generateDominance(R,S)
8:     B ← reformulate(B)                    ▷ Optional
9:     C ← C ∪ B
```

solution. Additionally, the solver enumerates all solutions.

We create CDP+I models for five problem classes from Section 2.8. Four of these models have complete incomparability functions whereas one RSD has a partial incomparability function which does not completely cover the whole incomparable cases. In the chapter 6, we will discuss a complete incomparability function for the RSD problem class. The full ESSENCE models can be seen in appendix B.

5.3.1 Implementation

CDP+I Algorithm 3 is implemented in a similar fashion to CDP. Here, at each step, we enumerate all solutions. The CP solver we use (MINION) supports finding a single solution as well as enumerating all solutions natively. For SAT, we use a fast SAT solver in the context of CDP (`glucose` [AS09]) and a non-blocking AllSAT solver (`nbc_minisat_all` [TS16]) in the context of CDP+I. `nbc_minisat_all` is much faster at enumerating solutions than repeated calls to a standard SAT solver since it is specifically crafted for this purpose using a non-blocking jumping mechanism.

SAVILE ROW is capable of translating solutions back from SAT/MINION automatically into ESSENCE PRIME. For each solution, dominance blocking constraints are generated. For CDP+I, we apply CSE on the set of dominance blocking constraints. Eliminating common sub-expressions is a very effective model reformulation in previous work [NSM15]. Having a set of similar con-

straints presents a natural opportunity for them to arise. We demonstrate the effect of applying CSE on a part of an instance on the hepatitis dataset from our experiments.

Example 6.

$$
\begin{aligned}
&((is_{29} \vee is_{37} \vee is_{55} \vee is_{56} \vee is_{58} \vee is_{60} \vee is_{62} \vee (76 < sup)) \\
&\wedge \\
&(is_{29} \vee is_{37} \vee is_{53} \vee is_{55} \vee is_{56} \vee is_{58} \vee is_{60} \vee (80 < sup))) \\
&\rightsquigarrow \\
&(aux = is_{37} \vee is_{55} \vee is_{56} \vee is_{58} \vee is_{60} \\
&\wedge \\
&(aux \vee is_{62} \vee (76 < sup)) \wedge (aux \vee is_{37} \vee (80 < sup)))
\end{aligned}
$$

In this example, a disjunction with 5 literals is common to two dominance blocking constraints. It is extracted by the CSE algorithm by introducing an auxiliary Boolean decision variable. In larger examples, we typically identify many more common sub-expressions.

5.3.2 Modelling Optimisation Problems using Incomparability

Incomparability allows us to create a partition on required criteria. It is possible to use the more advanced CDP+I to model COP problems while eliminating all possible sub-optimal solutions.

Considering the previous COP to CDP example in fig. 4.4, we can add the following incomparability function fig. 5.3 to only enumerate optimal solutions.

```
incomparabilityFunction ascending totalCost
```

Figure 5.3: CDP+I incomparability function of the COP modelled in CDP in fig. 4.4.

The way this model operates is similar to the CDP equivalent. The only difference is that the incomparability function ensures we start exploring the partition from the best possible value. It aims to find the best satisfiable value by reducing the optimisation value similar to the unsat strategy in SAVILE ROW for COP SAT encodings (explained in section 2.3.1). It also operates for finding multiple solutions in the optimal value. In some cases, this may be desired but can bring additional overhead if the number of solutions is large.

5.4 Multi-Layer Incomparability

We mentioned in section 5.3 that the incomparability function either points to a discrete domain or a list of discrete domains. In the case of the latter, we define multiple incomparabilities on complex dominance relations to split the search space into more discrete partitions. An example situation can be seen in section 5.4.1.

CDP+I can be also used in modelling multi-objective optimisation problems. It allows defining the Pareto frontier by using multi-layer incomparabilities. We will discuss this in section 5.4.2.

In a multi-layer incomparability environment, it is necessary to generate all the possible combinations of the different layer incomparabilities. This can be done through a recursive algorithm, which can be seen in algorithm 4.

In this recursive algorithm, starting c and t as empty lists, we generate all the possible combinations with $O(\binom{n}{d})$ in time and $O(nd)$ in space, where n is the max length of any domain and d the number of layers or total depth. As the time complexity is $\binom{n}{d} = \frac{n!}{d!(n-d)!}$, it is generally expensive to calculate all combinations. However, with small numbers of d, the procedure does not create a big overhead for the CDP+I system. For $d = 2$, its behaviour is quadratic and for $d = 3$ its behaviour is cubic complexity.

Algorithm 4 Multi-layer Incomparability level generation in CDP+I

```
1: Input
2:     md  List of Multi-layer domains (assuming already ordered)
3:     d   Depth of the multi-layer
4:     c   Current combination
5:     t   Total list of combinations
6: procedure GENERATOR(md, d, c, t)
7:     if d == |md| then
8:         total ← c
9:         return
10:    currentDomain ← md[d]
11:    for value ← currentDomain do
12:        c ← value
13:        generator(md, d + 1, c, t)
```

5.4.1 On a Hypothetical Example

Considering a complex dominance example where there are multiple criteria on different sets, it is possible to set a multi-layer incomparability which will go through each of the defined statements in order.

```
find s1 : set (minSize 2) of int(1..5)
find s2 : set (minSize 3) of int(1..6)
find s3 : set (minSize 4) of int(1..7)
find s4 : set (minSize 5) of int(1..8)

such that sum([i | i <- s1]) >= 3
such that sum([i | i <- s2]) >= 5
such that sum([i | i <- s3]) = 15
such that sum([i | i <- s4]) <= 18
```

```
dominanceRelation
    !( s1 subsetEq fromSolution(s1)
    /\ s2 subsetEq fromSolution(s2)
    /\ s3 subsetEq fromSolution(s3)
    /\ s4 subsetEq fromSolution(s4) )
```

```
incomparabilityFunction descending [|s1|, |s2|, |s3|, |s4|]
```

Figure 5.4: An example CDP+I problem which uses multi-layer incomparability

The incomparability values in the multi-layered incomparability will be

```
Looking for solution on incomp value(s): [5, 6, 6, 7]
No solutions.
Looking for solution on incomp value(s): [5, 6, 6, 6]
No solutions.
Looking for solution on incomp value(s): [5, 6, 6, 5]
No solutions.
Looking for solution on incomp value(s): [5, 6, 5, 7]
No solutions.
Looking for solution on incomp value(s): [5, 6, 5, 6]
No solutions.
Looking for solution on incomp value(s): [5, 6, 5, 5]
Found 7 Solutions.
Looking for solution on incomp value(s): [5, 6, 4, 7]
No solutions.
Looking for solution on incomp value(s): [5, 6, 4, 6]
No solutions.
Looking for solution on incomp value(s): [5, 6, 4, 5]
Found 28 Solutions.
Looking for solution on incomp value(s): [5, 5, 6, 7]
No solutions.
...
```

Figure 5.5: An exempt of the SAVILE ROW output of the problem defined in fig. 5.4.

explored using all the possible combinations. Since it's defined as descending, each of the levels on each layer will start from their maximum value and decrease in order of appearance in the incomparability list.

For this particular example, a part of the simplified output from SAVILE ROW is shown at fig. 5.5. This particular problem has 35 solutions that have been found in 9 multi-layered levels. Thanks to the singleton arc consistency pre-processing, the domains of the incomparability values are automatically shrunk and some levels without any solutions are skipped directly.

5.4.2 A Multi Objective Optimisation Example

We can use the multi-layer incomparability to encode multi-objective optimisation problems. An example problem can be seen in fig. 5.6.

In this example, both objective criteria depend on the same decision variable. However, in a real-life application, this might not be the case. The incomparability function takes both `totalCost` and `totalUtility` to crawl through the space

```
find s1 : set (minSize 5) of int(1..10)

find totalCost    : int(0..25)
find totalUtility : int(75..150)

such that sum([i | i <- s1]) = totalCost
such that sum([i * i | i <- s1]) = totalUtility
```

```
dominanceRelation
    !( totalCost > fromSolution(totalCost)
    /\ totalUtility < fromSolution(totalUtility) )
```

```
incomparabilityFunction ascending [totalCost, -totalUtility]
```

Figure 5.6: A multi objective optimisation problem represented as a CDP+I problem which uses multi layer incomparability

depending on these two variables. Since our definition of the incomparability in the ESSENCE specification takes one single **ascending** or **descending** statement, it is necessary to negate `totalUtility` and use it in the same ascending context as the `totalCost`.

The model taken by SAVILE ROW will have similar output to the fig. 5.5 with the first solution available at [20, -130] layer and continuing to enumerate the whole Pareto frontier.

5.5 CDP vs CDP+I results

In our experiments, we use 12 transactional datasets from CP4IM in which we use the generated instances from the section 8.1.

We included minimum value and maximum cost side constraints in all of our models to demonstrate their ability to handle arbitrary side constraints. Due to the inclusion of side constraints, specialised data mining algorithms are inapplicable to these tasks. We generate values between 0 and 5 for item values and costs using uniform randomness. In addition, we generate a threshold for the minimum value and the maximum cost constraints as well.

Specifically for this experiment, we systematically generate several candidate instances and choose instances that have a reasonable number of solutions (in the order 10,000 at most for all problem classes except minimal rare itemset mining and in the order of 100,000 for minimal rare). We also limit the instances that can be solved within our time limit of 6 hours[1]. We generate instances at 5 frequency levels: 10%, 20%, 30%, 40% and, 50%. For CFIM we use the instances published in the appendix of [KAMN18a].

For each problem class, we solve each instance using 4 pairs of solver configurations. The first two configurations are for standard CDP with the default search order provided by the solver, one with a CP solver (MINION), and one with a SAT solver (`glucose`). The second two configurations are for CDP with the search order specified in the incomparability function statement. The third two configurations are for CDP+I with CP/SAT and the last two are for CDP+I with reformulation enabled. We use a CP solver (MINION) and an AllSAT solver (`nbc_minisat_all`) for CDP+I configurations.

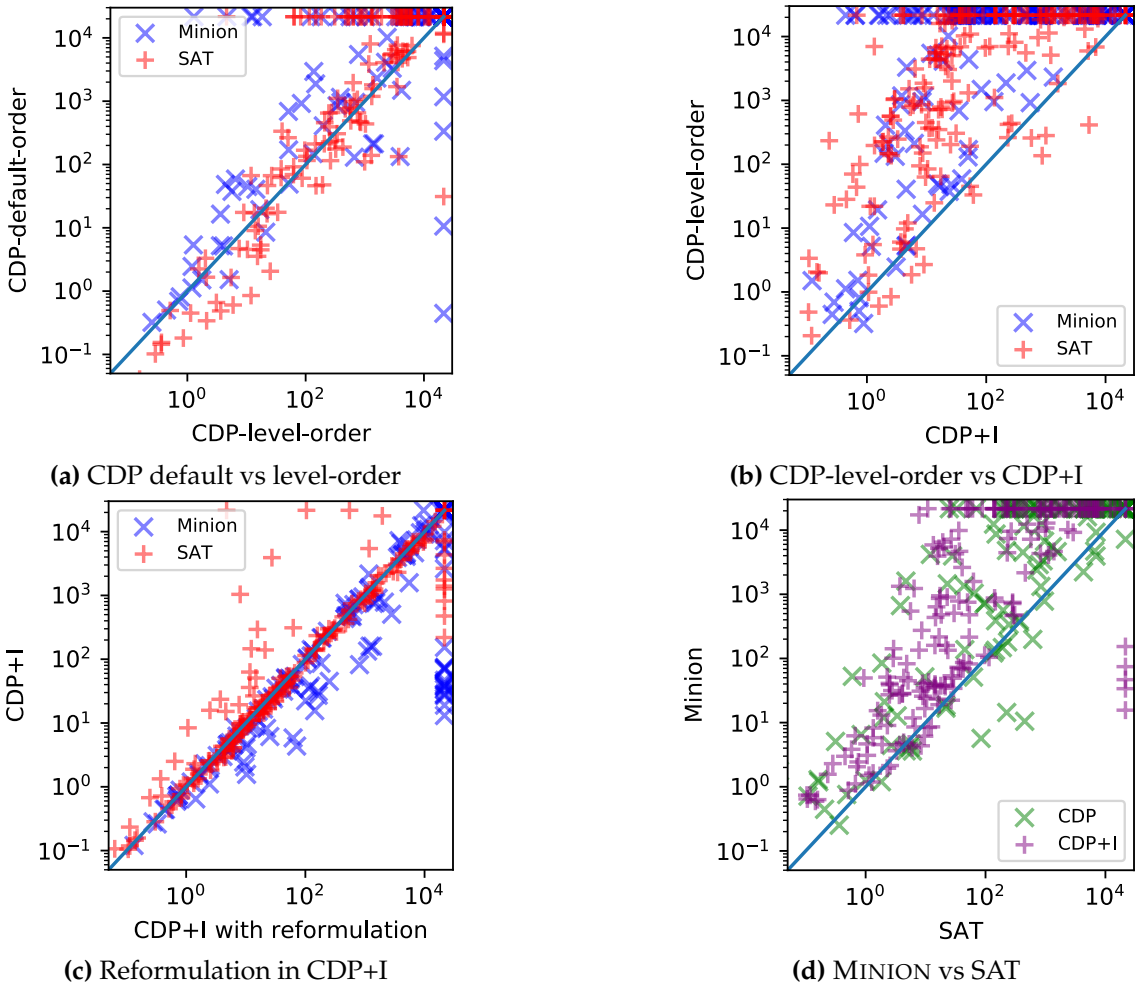
We run every SAT instance 3 times with different seeds and present averages. We run MINION instances only once with a deterministic static search ordering.

We run our experiments on two identical 32 core AMD Opteron 6272 machines, at 2.1 GHz and with 256GB memory. Experiments are run according to the section 8.2.

The experiments are conducted in a way to examine and explain the following:

- Impact of adding a level-wise search order to CDP
  (CDP-default-order vs CDP-level-order, see Figure 5.7a)
- Impact of CDP+I in general with respect to standard CDP
  (CDP-level-order vs CDP+I, see Figure 5.7b)
- Impact of model reformulations in CDP+I (CDP+I vs CDP+I with model reformulation, see Figure 5.7c)

[1]Our Github repository of the data and results is available at **https://doi.org/10.5281/zenodo.3675340** [KAGM20b]



(a) CDP default vs level-order

(b) CDP-level-order vs CDP+I

(c) Reformulation in CDP+I

(d) MINION vs SAT

Figure 5.7: Comparison plots for 4 CDP/CDP+I variants and 2 solver back-ends (i.e. 8 solving configurations). All plots present solver times (in seconds) with a 6-hour timeout. Timed-out instances are near the top and the right borders.

- Comparison of the solving methodology on CDP-level-order and CDP+I (MINION vs SAT, see Figure 5.7d)

For a general picture, Figure 5.8 shows the time spent solving all instances of the five problem classes using the 8 configurations. The results are sorted by time spent by SAT CDP+I without reformulation. The timed out instances are also included at the 6-hour mark near the top of the plot. The results show that CDP+I configurations are significantly better than CDP configurations in most instances. While SAT CDP+I configurations require the least time in the majority of the cases, for a small number of instances the SAT CDP configuration is the fastest; these

instances have a very small number of levels of solutions. For a small number of instances, Minion CDP+I is the best configuration; these instances contain large amounts of data and the size of the SAT encoding gets prohibitively large.

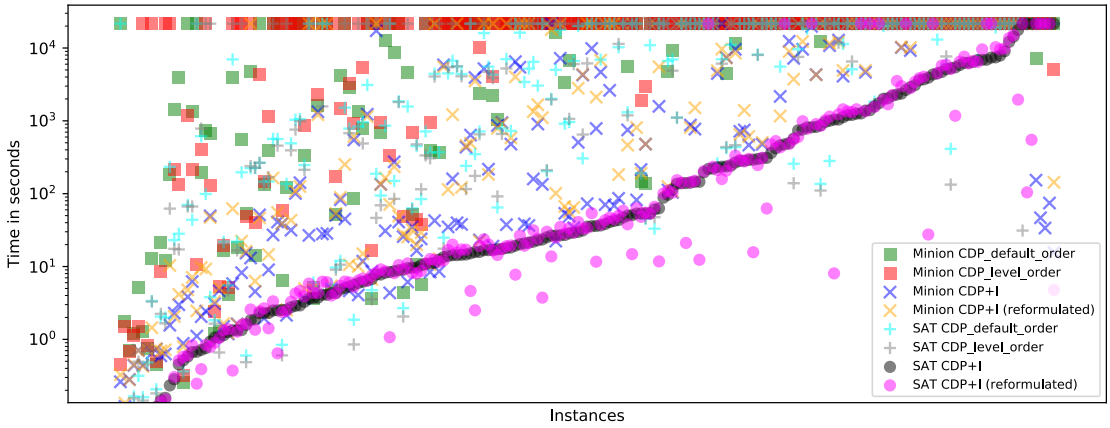


Figure 5.8: Solver time for all instances, sorted by SAT CDP+I. Timeouts are also shown at the top of the plot.

The optional model reformulation brings a negligible overhead on most instances and on some instances it helps significantly.

For a small number of instances, the default search order performs better for SAT; these instances have a very small number of solutions, all of which are non-dominated. In these cases, the small overhead of applying a specific search order does not pay off.

Effect of CDP order Figure 5.7a presents a comparison plot between two different CDP configurations with MINION and SAT. In easy instances, level ordering creates an overhead. However, for difficult instances using the same search order defined in the incomparability function helps. CDP-level-order solves many more instances than CDP-default order, where a large number of instances time out.

CDP vs CDP+I Figure 5.7b presents a comparison plot between CDP-level-order and CDP+I on the same instances, using both MINION and SAT. We only compare against CDP-level-order since it performs better than CDP-default order in general. This plot shows the direct effect of using a level-wise search and adding

the dominance blocking constraints in batches. A point above the diagonal line means CDP+I performs better, which is a significant majority of the instances. Both solving methodologies clearly benefit from CDP+I thanks to typically fewer solver calls (once per level as opposed to once per solution) and retaining learnt clauses for SAT.

Effect of reformulation on CDP+I Figure 5.7c presents a comparison plot between two different CDP+I configurations for MINION and SAT: with and without the optional model reformulation. The results show that the reformulation does not consistently help solve time. The reformulation hurts the performance of the CP solver MINION more. However, the performance of the SAT solver `nbc_minisat_all` up to median difficulty is improved. For the most difficult instances, the reformulation has mixed effects. The auxiliary variables added by CSE create a connection between the dominance blocking constraints and this can help propagation only for a subset of the instances. When the help is not significant enough, they create unnecessary overhead.

MINION vs SAT Figure 5.7d presents a comparison plot between MINION and SAT for CDP and CDP+I. We use CDP-level-order and CDP+I without reformulations in this plot. SAT performs better for most instances and benefits from using incomparability more. This is likely due to the learning employed by SAT solvers. In contrast, MINION is not a learning solver.

In Table 5.1, we focus on the five problem classes separately to see the effect of the complexity of the dominance relation. Results show that SAT is always significantly better for all problem classes except the RSD problem, where the complexity of dominance relation is the highest.

Number of solver calls CDP variants perform better than CDP+I equivalents for 25 instances. All of these instances have a shared characteristic: they have a small number of solutions (less than 10, which is smaller than the number of levels). In the extreme case of instances with significantly fewer solutions than levels using CDP+I has a large overhead. In these cases the cost of making one solver call per solution becomes negligible.

Problem	CP wins	SAT wins	Similar time
Closed	0	40	18
Generator	0	36	17
Minimal	0	17	13
Discriminating	0	18	14
Relevant	5	19	19

Table 5.1: Solver time comparison on CP vs SAT on CDP+I. Substantial differences ($> 50s$) are reported as wins. Similar time indicates the CP and the SAT solver reached the solutions approximately around the same time ($\pm$50 seconds).

CHAPTER SIX

SYSTEMATIC INCOMPARABILITY DEDUCTION

While CDP defines a relation to create restrictions in between potential solutions, by introducing CDP+I, we aim to capture one of the problem's key aspects: incomparability of solutions. By exploiting these solutions, we achieve greater efficiency with the cost of incorporating a function called an incomparability function. So far, we determined the incomparability conditions of the problem theoretically and through testing. In this chapter, we will look into making the determination of the incomparability process systematic and possibly automated.

6.1 Motivation

In an environment where the dominance relation is defined as a comparator of two candidate solutions (A and B), the condition for incomparability function to hold requires that neither A dominate B nor B dominate A.

Taking A dominates B and B dominates A as a logical formula, a procedure can be defined to determine which logical bits in A and B are required to hold true.

In the context of ESSENCE specification, two candidate solutions to compare are one candidate solution A and any previously found solution fromSol(A). Thus,

we can take the logical step and conjunct both dominance relations in eq. (6.1). We use the same $\succ$ notation from section 2.5 to denote dominance.

$$\neg(A \succ fromSol(A)) \wedge \neg(fromSol(A) \succ A) \tag{6.1}$$

If we can find an equivalent statement or a one-sided implication statement, we can systemically determine the incomparability function for any given dominance statement. We will mark this potential function as $i()$. $i()$ gives the same result for A and $fromSol(A)$. The function needs to verify the following eq. (6.2).

$$i(A) = i(fromSol(A)) \implies \neg(A \succ fromSol(A)) \wedge \neg(fromSol(A) \succ A) \tag{6.2}$$

To find a function which satisfies the condition, we can expand, simplify, and find an equivalent representation for the right-hand side of eq. (6.2).

It may not be feasible to find a single function as the incomparability since the problem may have more complexity and require a multi-layer incomparability (e.g. multi-objective optimisation). This equation can be altered to include n number of functions to represent incomparability (eq. (6.3)).

$$\bigvee_{k=0}^{n} i_k(A) = i_k(fromSol(A)) \implies \neg(A \succ fromSol(A)) \wedge \neg(fromSol(A) \succ A) \tag{6.3}$$

6.2 Simplifying the Incomparability Logical Formula and Semantic Analysis

To simplify the incomparability logical expression, we can eagerly apply logical laws as rewrite rules (i.e DeMorgan's law, distribution of $\wedge$ over $\vee$, expanding *neg* and eliminating constants) or we can generate a Binary Decision Diagram (BDD) and find an equivalent but simpler version of the BDD.

To be able to apply logical rules of propositional logic or to search in a BDD, all the high-level constructs of the incomparability expression need to be expressed in propositional logic. Thus, the atomic parts of the incomparability expression that cannot be split into more expressions in propositional logic are taken as terminals.

For instance, if we consider the multi-objective optimisation dominance relation from fig. 5.6 in section 5.4.2, we can define $a = totalCost > fromSolution(totalCost)$ and $b = totalUtility < fromSolution(totalUtility)$. Then, a and b can be considered atomic logical pieces since their sub-structures cannot be represented in propositional logical form. Taking them as the terminals, we achieve eq. (6.4) and by applying distribution of the negation we achieve eq. (6.5).

$$\neg(a \wedge b) \wedge \neg(a' \wedge b') \tag{6.4}$$

$$\iff (\neg a \vee \neg b) \wedge (\neg a' \vee \neg b') \tag{6.5}$$

Later, from the simplified expression, assignments for the chosen terminal points can aid to determine the full incomparability function(s) and find their order signature.

To solve the example in eq. (6.5), we can assign all a, a‘, b and b‘ to false and satisfy the expression. By applying semantic analysis on the final equations, we arrive at eq. (6.6). This gives us two incomparability functions that can be used in a multi-layer context.

$$\left.\begin{array}{l} totalCost \not\succ fromSol(totalCost), \\ fromSol(totalCost) \not\succ totalCost, \\ totalUtility \not\prec fromSol(totalUtility), \\ fromSol(totalUtility) \not\prec totalUtility \end{array}\right\} \Longrightarrow \begin{cases} totalCost = fromSol(totalCost), \\ totalUtility = fromSol(totalUtility) \end{cases} \tag{6.6}$$

6.3 Implementation

The system is implemented in Rust using a Boolean expression manipulation library that supports BDD representation and offers SAT compatibility for the solving stage. We have named this system `DIG`, which stands for Dominance Incomparability Generator.

The flow of the `DIG` is as follows:

- The system starts the process by parsing the *dominance_relation* in the ESSENCE specification and constructing the dominance nogood constraint as a logical expression. In this step, the dominance constraint in the ESSENCE specification is translated into a propositional formula, while anything complex to represent in the propositional logic formula is taken as a terminal.
- The transpose of the dominance expression is determined by a $A \iff B$ replacement. Then, two expressions are conjuncted.
- The propositional logical formula is simplified by either logical rewrite rules or by constructing a BDD.
- Search of a satisfiable solution is done by a SAT solver. (The BDD graph can also be plotted at this stage).
- The system exports the possible simplified final resolution expression with its SAT assignment in a JSON format for further analysis.

At the end of the `DIG`, the user can look at the resulting solution. By applying higher-level semantic analysis, they can derive the function(s) that satisfies the given properties similar to eq. (6.6).

6.4 Application on Pattern Mining Problems

Now we will look into applying the DIG system into some of our pattern mining problem classes.

6.4.1 Maximal Frequent Itemset Mining

The easiest example in the mining context can be given as maximal frequent itemset mining (MFIM).

In this problem class, the dominance nogood constraint is defined in ESSENCE. The dominance expression and its transpose can be seen in fig. 6.1.

```
dominanceRelation !(fis subsetEq fromSolution(fis))
```

```
fis subsetEq fromSolution(fis)
fromSolution(fis) subsetEq fis
```

Figure 6.1: Dominance relation for the maximal itemset mining problem in ESSENCE specification and the dominance expression with its transpose.

We represent the subset operations as atomic terminals in propositional logic. We rename them as a and a'. The representation of MFIM in the DIG system can be seen in fig. 6.2.

```
{
    "translation_table": {
        "a": "freq_items[1] subsetEq from_solution_freq_items[1]"
    },
    "resolve_with_bdd": false,
    "resolution_expression_str": "And(Terminal(\"a\"), Terminal(\"a_prime\"))",
    "sat_assignment": {
        "a": true
        "a_prime": true,
    }
}
```

Figure 6.2: Maximal itemset mining problem represented and resolved in the DIG system.

Additionally, we can also look at the BDD at fig. 6.3.

The result indicates that a and a' should both be true. That means a candidate solution must be a $\subseteq$ and a $\supseteq$ at the same time. This condition can only be validated where both the candidate solution and the previously found solution have the same cardinality. This can be translated to the ESSENCE CDP+I as in fig. 6.4.

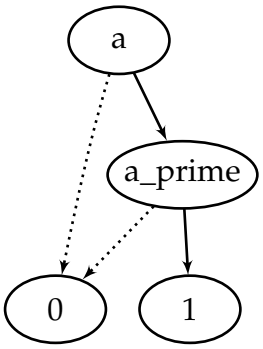


Figure 6.3: BDD for the maximal itemset problem generated by the DIG system.

```
incomparabilityFunction descending |fis|
```

Figure 6.4: Optimal incomparability function of the maximal itemset mining problem generated by the dig system expressed in ESSENCE specification.

6.4.2 Closed Frequent Itemset Mining

In this problem class, the dominance nogood constraint can be seen on fig. 6.5. The first operation (i.e. left hand side of the implication) can be represented as a, while the second one (i.e. right hand side of the implication) with b.

```
dominanceRelation
    (freq_items[1] subsetEq fromSolution(freq_items[1]))
    -> (freq_items[2] > fromSolution(freq_items[2]))
```

Figure 6.5: Dominance relation of the closed itemset mining problem.

The DIG representation and its solution can be seen in fig. 6.6. The optional BDD representation can be seen in Figure 6.7.

The final expression is $a \wedge a' \wedge \neg b \wedge \neg b'$ which translates into: 1) The candidate solution should be $\subseteq$ or $\supseteq$ of the prev solution and 2) Neither of the solutions can have greater cardinality.

This boils down to the same argument that they should have the same cardinality, like the MFIM problem. It will have a similar deduced incomparability function, which is available in fig. 6.8.

```
{
    "translation_table": {
        "a": "freq_items[1] subsetEq from_solution_freq_items[1]",
        "b": "freq_items[2] > from_solution_freq_items[2]"
    },
    "resolve_with_bdd": false,
    "resolution_expression_str": "...",
    "sat_assignment": {
        "a_prime": true,
        "a": true,
        "b": false,
        "b_prime": false
    }
}
```

Figure 6.6: Closed itemset mining problem represented and resolved in the DIG system.

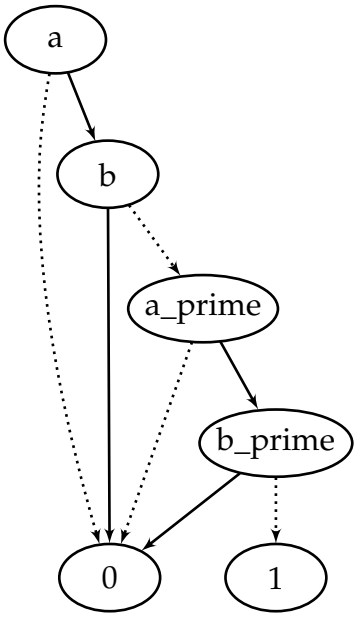


Figure 6.7: BDD for the closed itemset problem generated by the DIG system.

```
incomparabilityFunction descending |freq_items[1]|
```

Figure 6.8: Optimal incomparability function of the closed itemset mining problem generated by the DIG system expressed in ESSENCE specification.

6.4.3 Relevant Subgroups Discovery

This problem class consists of the most complex dominance relation we have. It can be seen in fig. 6.9.

Since this problem consists of class information on transactions, the final

```
dominanceRelation !((freq_items[cover_pos]
    subsetEq fromSolution(freq_items[cover_pos]))
    /\ ( freq_items[cover_neg]
        supsetEq fromSolution(freq_items[cover_neg]) )
    /\ ((freq_items[cover_pos] union freq_items[cover_neg]
        = fromSolution(freq_items[cover_pos]) union
            fromSolution(freq_items[cover_neg]) )
        -> (freq_items[itemset] subsetEq
            fromSolution(freq_items[itemset]) )))
```

Figure 6.9: Dominance relation of the relevant subgroups discovery problem.

pattern is also examined in positive coverage and negative coverage. Thus, the number of variables in the equation is increased.

After creating the representation and the resolution, the expression becomes as fig. 6.10.

```
{
    "translation_table": {
        "a": "freq_items[cover_pos] subsetEq from_solution_freq_items[cover_pos]",
        "b": "freq_items[cover_neg] supsetEq from_solution_freq_items[cover_neg]"
        "c": "freq_items[cover_pos] union freq_items[cover_neg] \
            = from_solution_freq_items[cover_pos]
                union from_solution_freq_items[cover_neg]",
        "d": "freq_items[itemset] subsetEq from_solution_freq_items[itemset]",
    },
    "resolve_with_bdd": false,
    "resolution_expression_str": "...",
    "sat_assignment": {
        "a": true,
        "a_prime": true,
        "b": true,
        "b_prime": true
        "c": false,
        "c_prime": false,
    }
}
```

Figure 6.10: Relevant subgroups discovery problem represented and resolved in the DIG system.

The resolution expression in BDD form can be seen in Figure 6.12.

From BDD, we can see that there are actually three possible assignments. However, upon closer examination, we can see that two of these possible solutions require $(c \wedge \neg c')$ or $(\neg c \wedge c')$. Since c terminal dictates that $(fis_+ \cup fis_-)$ (which is

the total pattern frequent itemsets) = $fromSol(fis_+ \cup fis_-)$, c' is the same as c. Thus, the opposite assignments for c and c' are not possible.

The only assignment combination which puts c and c' in the same assignment also puts them on `false`. Thus, the candidate solution can be different from the prev solution. This assignment combination is independent from d: that means the incomparability_function does not necessarily rely on the cardinality of the whole itemset.

The final incomparability expression comes to the point of two conditions: positive covers having the same cardinality and negative covers having the same cardinality. Since the incomparability function supports multiple layers, we can put them in either order and achieve the optimal incomparability in ESSENCE as in fig. 6.11. We can use either ascending or descending contexts while negating the opposite context. In the figure, we used a descending context and negated the negative cover incomparability.

```
incomparabilityFunction descending
    [|freq_items[cover_pos]|, -|freq_items[cover_neg]|]
```

Figure 6.11: Optimal incomparability function of the relevant subgroups discovery problem generated by the DIG system expressed in ESSENCE specification.

6.5 Experiments using the Newly Generated Incomparability Function for RSD

On one dataset (tumor), using different frequency thresholds, the newly defined incomparability function is tested. The experiments show that it performs similarly in terms of node count. However, the number of solver calls increases drastically.

We can expose the cardinality of the positive or negative patterns to variables to make pre-processing shrink their domains. On the default incomparability_function chosen for RSD, this allows the system to perform 14 solver calls in total (on one frequency threshold), representing 14 different itemset cardinalities.

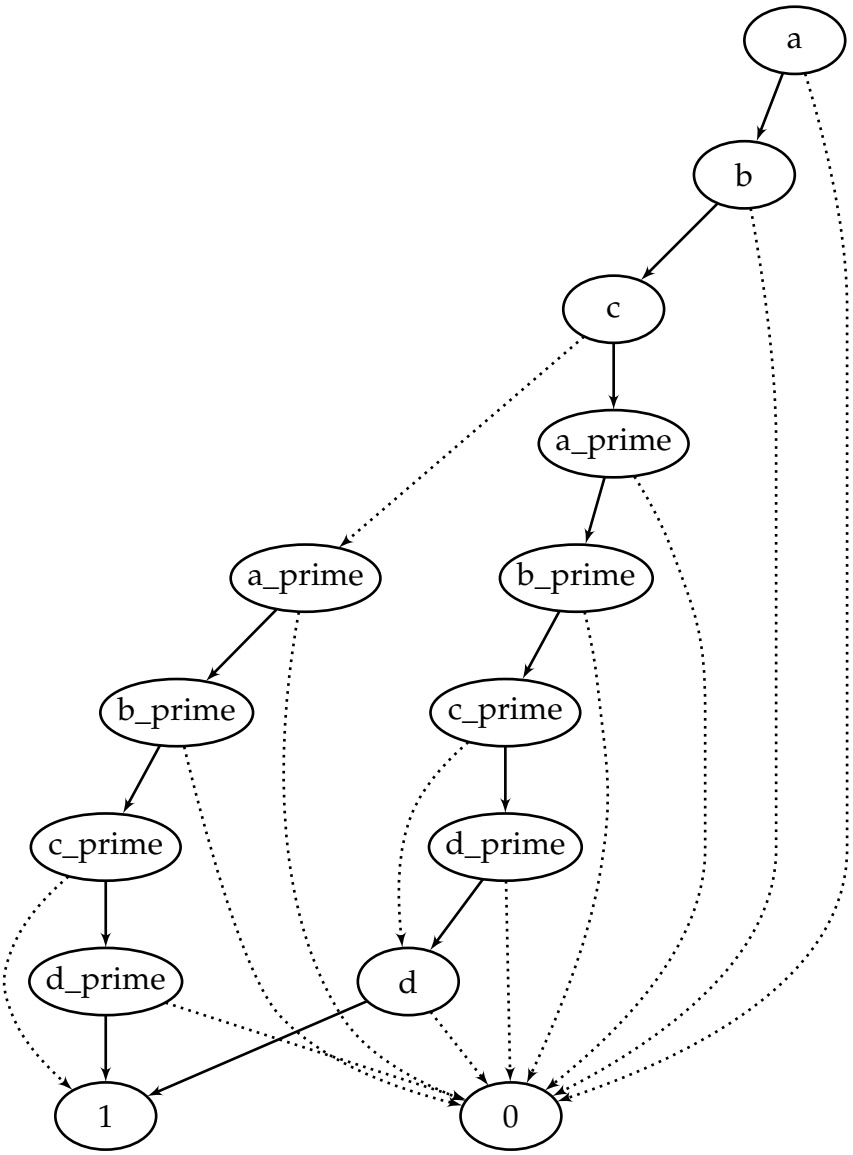


Figure 6.12: BDD for the relevant subgroups discovery problem generated by the DIG system.

Without exporting positive and negative cardinalities, the CDP+I model with the new multi-layer incomparability condition performs the combination of $255x75 = 19125$ solver calls. This number is unreasonably large. While we try to achieve perfect incomparability functions to eliminate any dominated solution, the system can generate unnecessary overhead as well.

After exporting the positive and negative cardinalities, the number of solver calls drops down to $100x70 = 7000$. This number is still large, but an improvement.

Other preprocessing options can be tested. In the experiments, using a more stricter preprocessing SSAC causes one dimension of the multi-layer incomparability to occasionally disappear. However, the multi-layer incomparability function does not perform better than the sub-optimal pattern cardinality. The results can be seen in fig. 6.13.

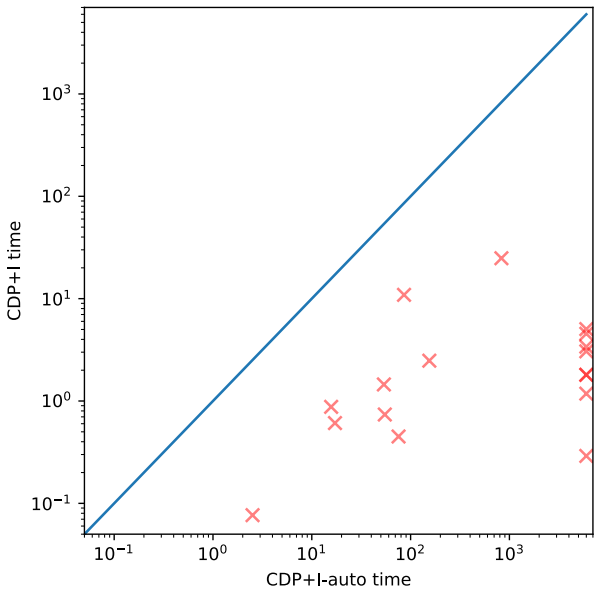


Figure 6.13: CDP+I vs CDP+I-auto which uses systematically generated incomparability on RSD problem class. Times are in seconds.

CHAPTER SEVEN

SOLVER INTERFACE INTERACTION

By defining CDP and CDP+I we have achieved a significant improvement in modelling/solving some pattern mining problems. The next step is to improve the inner workings of how the iterations in CDP and CDP+I are handled in the ESSENCE pipeline. This chapter will focus on two distinct improvements of the Solver Interface Interaction (SII) on two different solver backends. The first section (section 7.1) dwells on creating a native interaction interface for SAT solvers. The second section (section 7.2) is about interfacing SMT and its improvements.

Some of the work presented in this chapter has been published in article [KADM20]. This chapter extends the published work by giving a more detailed look and improves the presented work by adding another backend to the system.

7.1 Native Interaction on SAT

The main CDP+I algorithm (Algorithm 3) and the SAT optimisation backend require multiple solver calls. For CDP+I, each solver call occurs once per level when using an AllSAT solver and once per solution when using a standard SAT solver. Solutions from a level are used to produce dominance blocking constraints for the next level. Furthermore, level restriction constraints are both added and removed between levels. Likewise, for optimisation problems using a standard

SAT backend, multiple solver calls occur to apply three optimisation strategies to reach an optimal value. In addition to adding temporary constraints, the ability to remove added constraints is also required.

Adding constraints during the search is relatively common even without an incremental process such as in symmetry breaking for CP [GS99] or in conflict-driven clause learning for SAT [MSLM09]. However, removing constraints requires special treatment by the solver in question. A direct implementation of these algorithms would indeed call the solver several times and consequently would not benefit from any learnt clauses between solver calls.

There are two main ways of maintaining learnt clauses between solver calls. The first option works by extracting learnt clauses once the solver finishes the search and post-processing them to keep a relevant subset for a future solver invocation. [SMTdlB16] uses a similar approach to learn candidate implied constraints from a learning solver. The second option works by keeping the solver active, modifying the current model by posting additional constraints and restarting the search. Adding new variables and constraints in this way is a relatively common operation, available in `ipasir`, an incrementality API for SAT solvers used in SAT competitions [JLBRS12]. Removing constraints requires the *assumptions* machinery that is available in most modern SAT solvers. Constraints that are going to be removed are posted as conditional new clauses dependent on an assumption. Hence, when the assumption is lifted (and the constraint is removed) any learnt clauses which depend on that assumption can be deactivated.

We define a new API for SAT solvers that shares most of the functionality of `ipasir`, including methods for adding new clauses, adding assumptions, solving, and retrieving solutions. We extend this basic API to also include methods for reporting detailed statistics about learnt clauses and the solver's state, in addition to triggering solution callbacks. Our extended API is implemented using the Rust programming language. It works with SAT solvers GLUCOSE, CADICAL, and MINISAT and the AllSAT solver NBC_MINISAT_ALL. Our Rust implementation encapsulates the required functionality of these solvers and compiles them into a

shared library.

The entire pipeline of tools starts with CONJURE, which produces an ESSENCE PRIME model for each problem class. A modified SAVILE ROW is then used to instantiate the problem class model using a given data file, preprocessing it using MINION to shave domains, and then encoding it into SAT using the standard encodings found in SAVILE ROW [NAG+17]. Prior to our work, SAVILE ROW worked by producing a DIMACS file that has the entire encoding in it and calling a SAT solver on this file. Thanks to the new API we define and implement, SAVILE ROW now skips building this file and directly makes calls to the SAT solver to create the model.

7.1.1 Implementation

Our solver API layer is implemented in Rust, while SAVILE ROW was implemented in Java. We use the Java Native Interface (JNI) to integrate the API layer into SAVILE ROW. In this way, the modelled problem which exists in the ASTNode structure in the SAVILE ROW domain can be piped into the solver directly. An example of adding a new SAT clause from Java level SAVILE ROW to AllSAT solver NBC_MINISAT_ALL can be seen in fig. 7.1. To use JNI, we declare the native functions in SAVILE ROW space while implementing them in a C ABI compatible language (in this case Rust). Using the `JNI` module available in Rust, we can capture the declared native functions in the `ipasir` like API. The naming of the internal function names are dictated by Java and by using `no_mangle` we direct the Rust compiler to not optimise any function naming. Afterwards, by using unsafe pointer casting operations we can access the solver construct (which is created beforehand in a similar fashion) to access and execute preexisting code with our given parameters.

Defining assumptions that are useful to CDP+I can be done similarly by adding a clause to the underlining solver accessed with the Rust API.

Additionally, we can define new functions to retrieve additional information from the solver. One of the most important examples is the number of learnt

```
public class InteractiveAllMinisatSATSolver
    extends InteractiveSATSolver {
    ..
    private static native long
        addClauseToSolver(long solver, int[] clause);
    ..
}
```

```
#[no_mangle]
pub extern "system"
    fn Java_savilerow_InteractiveAllMinisatSATSolver_addClauseToSolver(
        env: JNIEnv,
        _class: JClass,
        s: jlong,
        clause: jintArray,
) -> jlong {
    let s = s as *mut solver;
    let v_size = env.get_array_length(clause).unwrap() as usize;
    let mut rust_clause = vec![0; v_size];
    env.get_int_array_region(clause, 0.into(), rust_clause.as_mut_slice()).unwrap();
    add_clause_to_nbc_solver(s, rust_clause);
    0 as jlong
}
```

```
pub fn add_clause_to_nbc_solver(s : *mut solver, given : Vec<i32>) {
    unsafe{
        ..
        solver_addclause(s, begin, last);
        ..
    }
}
```

```
bool solver_addclause(solver* s, lit* begin, lit* end){
    ..
}
```

Figure 7.1: Code execution example of adding a new SAT clause for AllSAT Solver NBC_MINISAT_ALL in SAVILE ROW with native interaction.

clauses available at the solver. This information is usually kept guarded in the solver level, which is private to the outside. However, with our solver level API, we can expose it to the Java level to be collected and given to us to use in our experiments as statistics.

```
public class InteractiveAllMinisatSATSolver
    extends InteractiveSATSolver {
    ..
    private static native long getNbLearntClauses(long solver);
    ..
}
```

```
#[no_mangle]
pub extern "system"
    fn Java_savilerow_InteractiveAllMinisatSATSolver_getNbLearntClauses(
        _env: JNIEnv,
        _class: JClass,
        s: jlong
) -> jlong {
    let s = s as *mut solver;
    let l = get_nbc_nb_learnt_clauses(s);
    l as jlong
}
```

```
pub fn get_nbc_nb_learnt_clauses(s: *mut solver) -> u64 {
    let l = unsafe{ (*s).stats.learnts    };
    l
}
```

Figure 7.2: Code execution example of accessing the number of SAT learnt clauses for AllSAT Solver NBC_MINISAT_ALL in SAVILE ROW with native interaction.

Most operations are straightforward and sequential in the java-rust-c/c++ system. Where Java level requests one native function to execute to then be interpreted as a success or a failure, there is one more improvement that can be implemented only for AllSAT solvers. Since AllSAT solvers work differently from a standard SAT solver and can find multiple solutions in one execution, there is a chance that we can implement a solution callback directly to the solver itself. With this callback, we can directly collect a solution at the SAVILE ROW level rather than waiting for the whole execution to finish while storing each solution at the solver level. This additional callback system optimises the low-level memory of the solver level and grants further improvements to the system.

This callback system execution can be seen in fig. 7.3 and fig. 7.4. The first solver call registers the Java environment and the Java class object in a data

structure and gives the pointer to the low-level solver. Thus, when a solution is found later in the process, the C/C++ level solver can provide the solution and necessary tools to access the Java level to the mid-level Rust layer. Using the passed information, the Rust mid-level layer accesses the Java run-time environment to pass the information to SAVILE ROW for the solution to be processed. The memory ownership model that Rust enforces requires holding the Java run-time reference in the mid-layer. This ensures better safety in terms of avoiding possible bugs related to applying pointer operations.

```
typedef void (*rust_callback)(void*, solver*);
void* rcb_java_data;
rust_callback rcb;

extern int nbc_register_callback
    (void* callback_target, rust_callback callback);
```

```
int nbc_register_callback
    (void* callback_target, rust_callback callback) {
        rcb_java_data = callback_target;
        rcb = callback;
        return 1;
}
static lbool solver_search(solver* s, int noc, int nol) {
    ..
    rcb(rcb_java_data, s);
    ..
}
```

Figure 7.3: Registering the solver callback in the AllSAT solver NBC_MINISAT_ALL and calling it when a solution is found

The API layer is open source and will be available in the next release of SAVILE ROW[1]. The new mechanism to access SAT solvers with this API layer is available under the flag `-interactive-solver`. SAT interface APIs are also open source and available for NBC_MINISAT_ALL[2], GLUCOSE[3] and CADICAL[4]. All the software

[1] SAVILE ROW releases are available at: https://savilerow.cs.st-andrews.ac.uk/releases.html

[2] https://github.com/gokberkkocak/rust_nbc

[3] https://github.com/gokberkkocak/rust_glucose

[4] https://github.com/gokberkkocak/rust_cadical

```
const JAVA_CALLBACK_METHOD_DYN: &str = "handleFreshSolution";
const JAVA_CALLBACK_METHOD_SIGN: &str = "([I)V";
#[repr(C)]
struct JavaData {
    java_env: *mut jni::sys::JNIEnv,
    solution: jintArray,
    java_callback_object: jobject,
}
extern "C" fn nbc_callback(java_data: *mut JavaData, s: *mut solver) {
    ..
    let _sol = j_env.call_method(
            JObject::from(j_callback_object), JAVA_CALLBACK_METHOD_DYN,
            JAVA_CALLBACK_METHOD_SIGN,
            &[JValue::from(JObject::from(j_solution))],
        )
        .unwrap();
    ..
}
extern "C" {
    fn nbc_register_callback(
        target: *mut JavaData,
        cb: extern "C" fn(*mut JavaData, *mut solver),
    ) -> ::std::os::raw::c_int;
}
```

Figure 7.4: Rust level of solution callback for the AllSAT solver NBC_MINISAT_ALL and how it handles the call the upper Java level.

code in this thesis is available in a supplementary archive with DOI [Koç22].

7.1.2 Results

7.1.2.1 On an Optimisation Problem

The example problem we have chosen to experiment on is a real-life problem and is called the Multi-mode Resource-Constrained Project scheduling problem (MRCPSP). It is a variant of the project scheduling problem [KS97], a classical and well-known optimisation problem in operations research. The givens are a number of activities and a set of renewable resources. Each activity is associated with a duration and demands for some resources. The activities are non-interrupted

and there are precedence constraints that state that some activities can only start once some others have finished. The variant considered in this thesis is the *multi-mode* [MT97], where each activity may have multiple modes. Each mode dictates the duration and resource demands of the activity. The goal is to schedule the activities and choose a mode for each of them such that the makespan (the latest completion time) is minimised. An ESSENCE specification of this problem is presented in Appendix C (Figure C.1).

To demonstrate the effectiveness of keeping SAT learnt clauses between levels during the optimisation process using native interaction, we evaluate the three optimisation strategies explained in section 2.3.1 on 928 MRCPSP instances from the PSPlib [KS97]. The SAT solver GLUCOSE [AS18] is combined with each of the three optimisation strategies. We also compare the resulting performance with Open-WBO [MML14], a MaxSAT solver, and with `Chuffed` [CS], a learning CP solver.

Each run on an instance is given a time limit of one CPU hour and is repeated three times. The average solving time is recorded. The comparison of the usage of native interaction (SII) on GLUCOSE is shown in Figure 7.5. Results suggest that for all three strategies, the native interaction boosts the efficiency significantly on all tested instances.

Comparison against Open-WBO and Chuffed is plotted in Figure 7.6. While the first figure only includes the default SAT strategies, the second figure replaces them with their native equivalents. Results suggest that the native interaction create a drastic performance improvement for the SAT backend GLUCOSE. Results on these problem instances are competitive against the two established optimisation solvers.

7.1.2.2 CDP+I Experiments

7.1.2.2.1 Computational Evaluation with AllSAT Solver In order to evaluate the effectiveness of maintaining learnt clauses and using SAT assumptions between CDP+I levels, we solve 240 instances across 5 problem classes (see Section

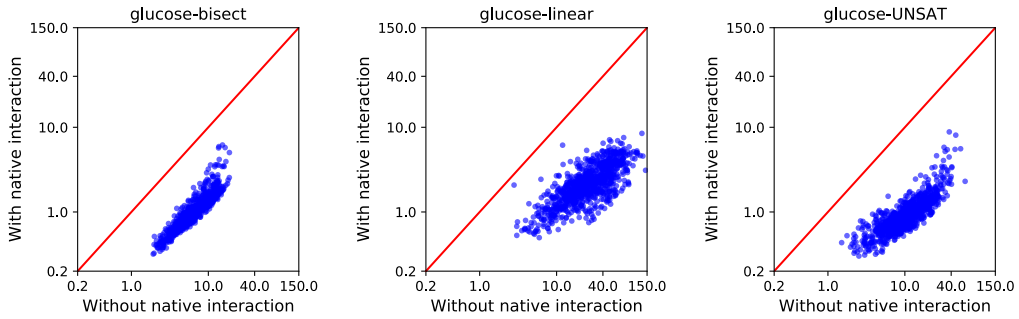
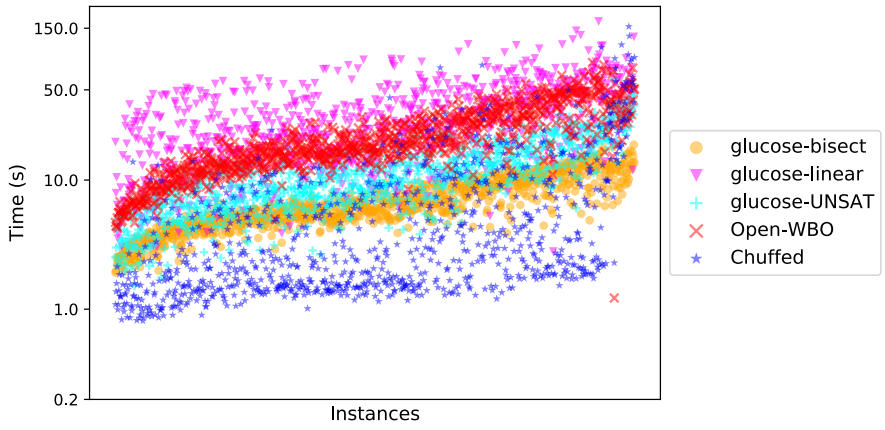
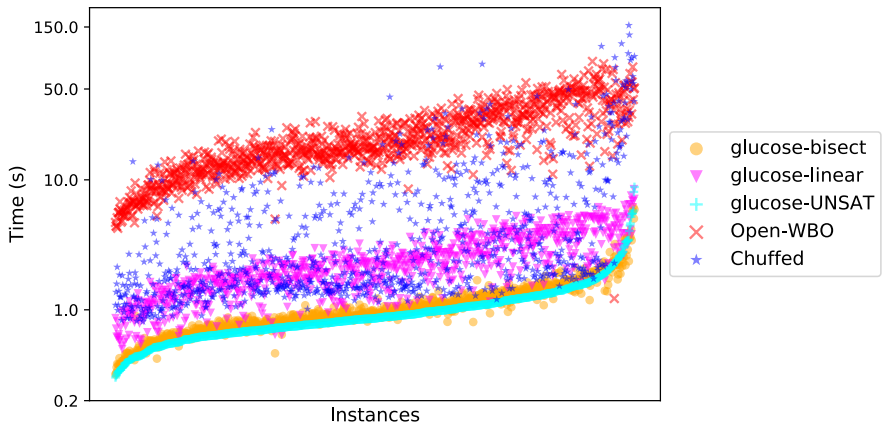


Figure 7.5: Solving time of GLUCOSE with versus without native interaction on 928 MRCPSP instances. Times are in seconds.



(a) Without native interaction



(b) With native interaction

Figure 7.6: Solving time of GLUCOSE with three settings (bisect, linear and UNSAT), Open-WBO and Chuffed on 928 MRCPSP instances. GLUCOSE's results are shown without (top) and with (bottom) native interaction.

section 2.8). Within a 6-hour time limit, the native version solves 210, instances whereas pure CDP+I solves only 173 instances. We believe this is due to needing fewer search nodes, which is made possible by pruning large parts of the search tree via the learnt clauses.

Figure 7.7 presents the median number of search nodes per level. Since instances have different numbers of levels, we normalise the number of levels on the horizontal axis. The plot also shows that the default CDP+I's performance can vary amongst different instances, while CDP+I-native's performance has more stability, indicating that CDP+I-native is more robust.

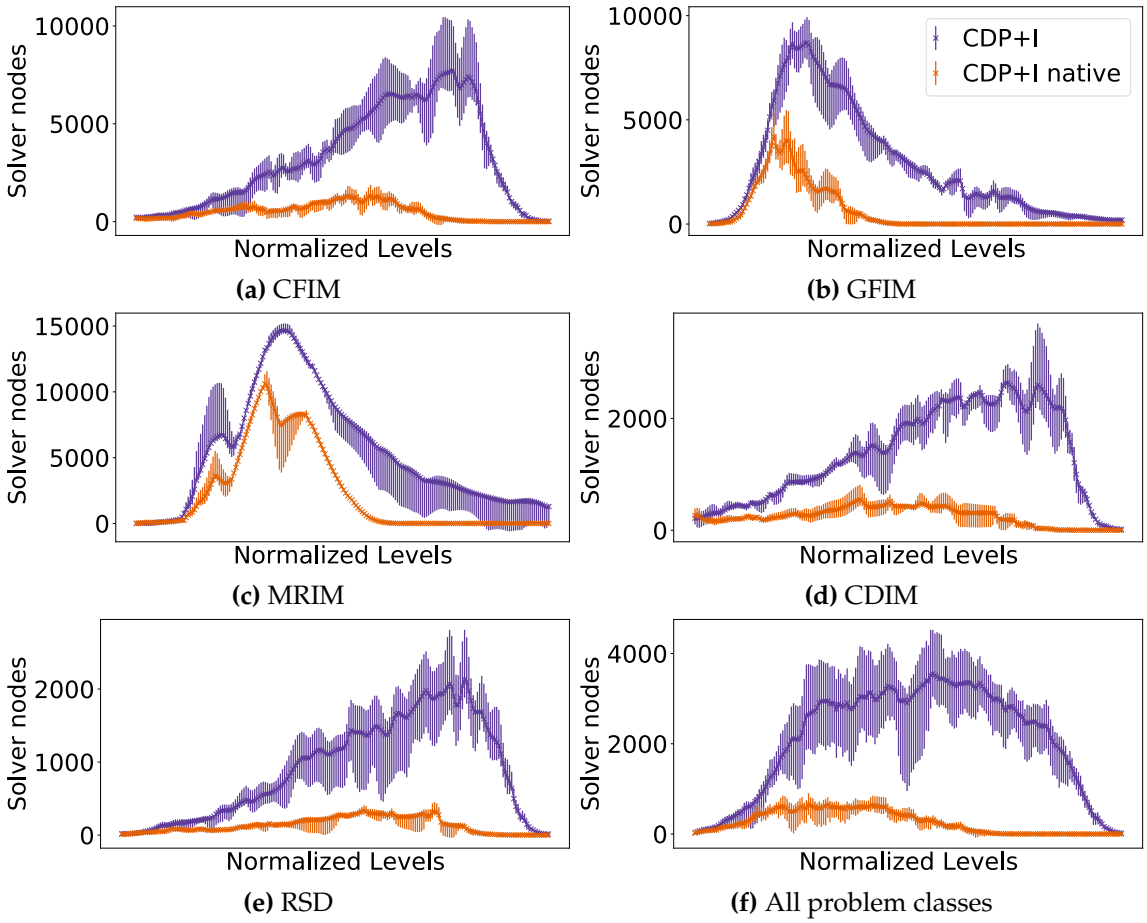


Figure 7.7: Median solver nodes per CDP+I level. Error bars range between the 45th and the 55th percentile. The horizontal axis represents normalised levels between instances. Native CDP+I uses significantly fewer search nodes, thanks to accumulated learnt clauses between levels.

CDP+I-native uses fewer search nodes than pure CDP+I, due to maintaining

a subset of learnt clauses between levels. Figure 7.9a presents a comparison of total solver run time of the two CDP+I variants on NBC_MINISAT_ALL and shows that native interaction clearly results in faster run times as well. On PAR2 average, CDP+I-native spends 493 seconds per instance, whereas pure CDP+I spends 8,210 seconds.

7.1.2.2.2 A Case Study on CFIM Tumor 20% Instance To evaluate whether keeping learnt clauses improves efficiency, we will demonstrate this by examining one particular instance in detail as a case study.

Figure 7.8 presents two plots. The first shows that CDP+I-native uses fewer search nodes on each level. The second illustrates the increased number of SAT clauses in each level that result from keeping learnt clauses. The improved efficiency seen on the first plot is a direct result of the restricted search space from having more clauses.

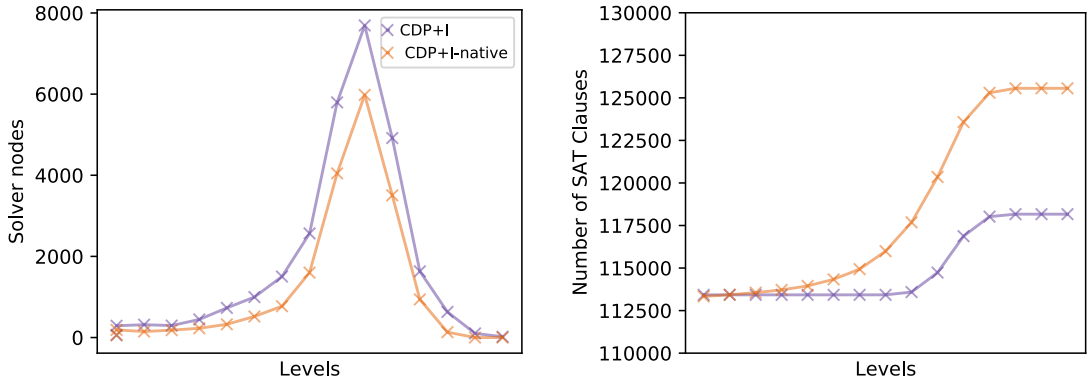
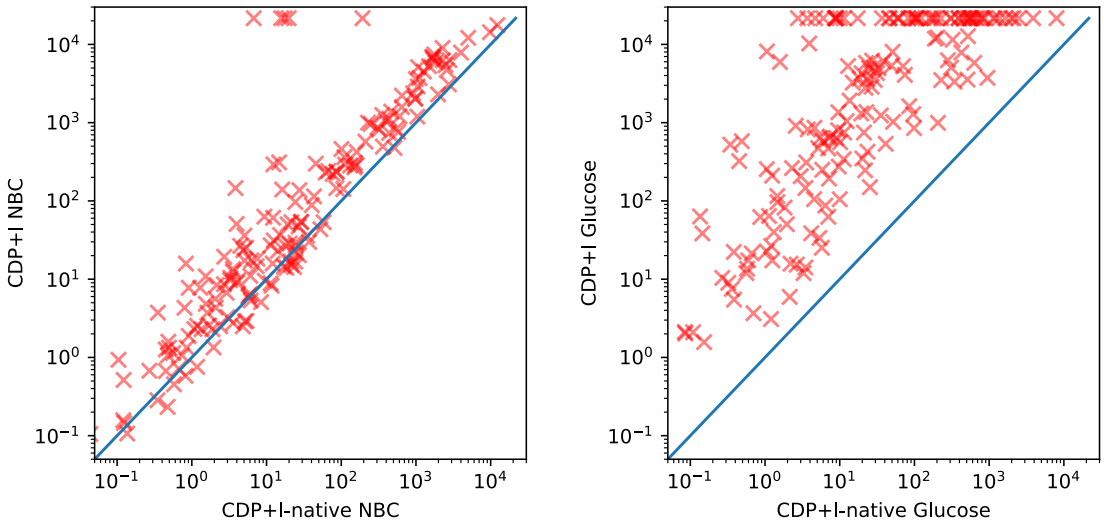


Figure 7.8: A comparison on one CDP+I instance with and without native interaction using NBC_MINISAT_ALL AllSAT solver. The example instance is CFIM Tumor with 20% frequency. Each plot is averaged out from a single model and multiple random seeds. The plot on the left shows the number of solver nodes on each level, while the plot on the right shows the total number of SAT clauses on each level.

7.1.2.2.3 Computational Evaluation with a Standard SAT Solver CDP+I on a standard SAT solver operates by generating solution blocking clauses between each solver call in a level. Once a level is completed, the dominance blocking clauses generated by SAVILE ROW are encoded and passed on to the next level.

The solution blocking clauses are not encoded again since they are redundant and already implied in the dominance blocking constraints.

Implementing a native interactive system on a standard SAT solver will bring both costs and benefits to its performance. AllSAT solvers are already capable of keeping learnt information at one level due to their all solution enumeration behaviour. The native interaction will grant the standard SAT solver this capability, in addition to making the learnt information persistent between levels. Thus, the increase of the standard SAT solver's performance will be relatively much higher than the increase of the AllSAT solver's performance. However, since we will still be using solution blocking clauses in a level and since the system cannot eliminate the redundant solution blocking clauses once the level is done, the standard SAT model might expand far beyond its non-native equivalent. AllSAT solver NBC_MINISAT_ALL is not susceptible to this because it can operate without the use of solution blocking clauses, regardless of whether it uses native interaction.



(a) Comparing total solver time using the AllSAT solver NBC_MINISAT_ALL.

(b) Comparing total solver time using the standard SAT solver GLUCOSE.

Figure 7.9: Comparison plot between pure CDP+I and CDP+I-native. The time limit is 6 hours per instance. Each data point is averaged out from a single model and multiple random seeds. Times are in seconds.

Figure 7.9b illustrates a comparison of CDP+I with and without native

interaction using the standard SAT solver GLUCOSE. Native interaction increases the performance amongst all instances significantly. The results also suggest that the anticipated decrease in performance due to the expansion of the model did not outweigh the increase provided by native interaction.

In this section, we have evaluated the effect of native interaction on the performance of CDP+I. We conducted our analysis on an AllSAT solver and a standard SAT solver.

7.2 Interactive Interface Usage on SMT

Similarly to the SAT backend, The main CDP+I algorithm (Algorithm 3) and the optimisations on the SMT backend also require multiple solver calls. Considering CDP+I, each solver call results in a single solution since most mainstream SMT solver does not support all solution enumeration. To properly apply CDP+I, solution blocking constraints are added and another solver call occurs within a CDP+I level. There is one SMT solver that supports all solution enumeration by performing greedy search [CGSS13]. However, the all solution enumeration feature of this solver is experimental and sometimes give unpredictable results; thus, we did not include them in our work.

After each solution, a dominance blocking constraint is generated and posted to the SMT solver to prevent getting the same solution again. The same principle applies to the level restriction constraints which are added and removed between levels, similar to the inner workings of SAT backend handling CDP+I.

On optimisation problems, a standard SMT solver can also be used with the three optimisation strategies mentioned, similar to the SAT. The assumption mechanism that allows revoking constraints that are already posted can be useful for the bisect strategy where a guessed partition can be revoked without an issue.

By keeping the solver alive between solver calls, we can guarantee the learnt information of the solver persists, and potential performance improvements are expected.

To achieve this system, we propose altering the ESSENCE pipeline execution plan once more, where SAVILE ROW generates the intermediate SMT file format then calls the solver. Instead, the altered version of SAVILE ROW initiates the solver on a virtual terminal interface in the interactive mode and pipes the generated information to the solver directly in real-time. This way, we can achieve three things: 1) avoid additional I/O overheads, 2) keep solver state persistent after each call, and 3) use assumptions on each call.

7.2.1 Implementation

The SMT solver interactive interface is implemented by adding additional multi-threaded functionality to SAVILE ROW. The general application flow of SAVILE ROW is pretty straightforward and single-threaded. However, keeping the interactive solver interface alive while processing other information requires an additional worker thread. This additional thread is used to pass the translated SMT model to the interactive SMT solver interface and wait for the solver to do a search to retrieve results. To guarantee proper concurrency, the worker thread where the solver interface is handled uses a `ConcurrentLinkedQueue` to pass the retrieved information to the main SAVILE ROW thread. This thread also uses `CountDownLatch` to inform the main thread when the solver changes its state and an `AtomicInteger` for the solver state itself, which the main thread can access freely if necessary without concurrency issues.

When passing the translated model to the solver interface, we can also pass any optional constraints that we would like to mark as assumptions. The working system can be seen in fig. 7.10. The modified system uses the `check-sat-assuming` SMT functionality to indicate Boolean assumptions, which can be seen in fig. 7.11. The assumptions' predicates can be encoded as constants with `declare-const`; these identifiers can be passed as assumptions later on. CDP+I and optimisation problems can use the modified SAVILE ROW with assumptions in a multi solver call structure.

```
public class InteractiveSMT extends SMT {
..
    public void addSMTAssumption(String clause) .. {
        String n = "sr_assumption_" + aC;
        outstream.write("(declare-const␣"+n+"␣Bool)");
        outstream.newLine();
        ..
        outstream.write("(assert␣(=␣" + n + "␣");
        outstream.write(clause);
        outstream.write("))");
        outstream.newLine();
        activeAssumptions.add(n);
        ..
    }
    ..
}
```

Figure 7.10: Code example of encoding SMT assumptions to the interactive SMT assumptions as boolean constants.

7.2.2 Results

7.2.2.1 On an Optimisation Problem

We use the same MRCPSP problem from our interactive SAT experiments (section 7.1.2.1).

Once more, to show the effectiveness of keeping learnt clauses and the strength of assumptions, we use the three optimisation strategies with and without native interaction on the 928 instances. We use the standard SMT solver YICES2 [Dut14] and the BitVector (BV) theory encoding. All experiments are conducted with cgroups as indicated in section 8.2.1, with one CPU core quota each experiment. This is an important factor (especially on this multi-threaded system) to eliminate any potential bias, since we compare two systems with different characteristics.

Each run on an instance is given a time limit of one CPU hour and is repeated three times. The average solving time is recorded.

The comparison of the usage of native interaction on YICES2 is shown in Figure 7.12. Results suggest that for all three strategies, the native interaction boosts the efficiency significantly on all tested instances.

```
..
    private String getCheckCommand(){
        if (model.satModel.activeAssumptions.size() > 0){
            StringBuilder sb = new StringBuilder();
            sb.append("(check-sat-assuming␣(");
            for(String a : model.satModel.activeAssumptions){
                sb.append(a);
                sb.append("␣");
            }
            sb.setLength(sb.length() - 1);
            sb.append("))");
            return sb.toString();
        }
        else {
            return "(check-sat)";
        }
    }
..
```

Figure 7.11: Code example of executing the solver with assumptions with the usage of `check-sat-assuming` rather than `check-sat`.

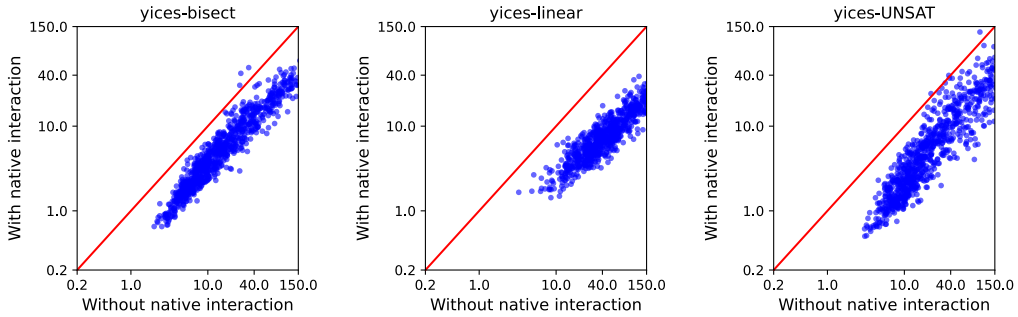
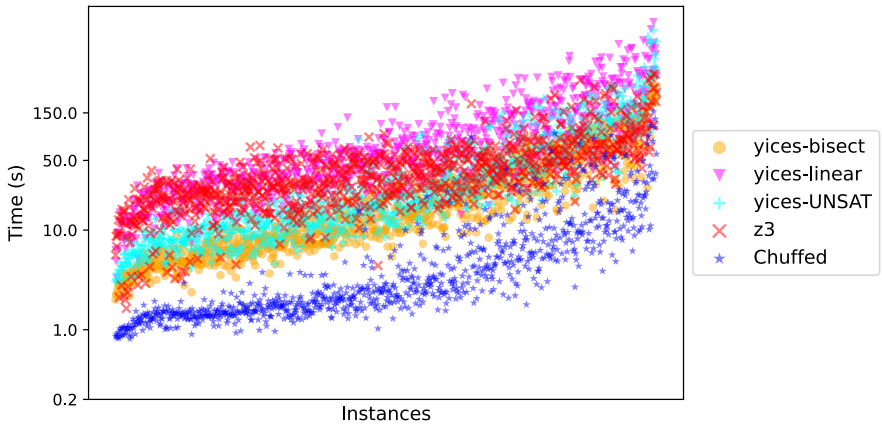
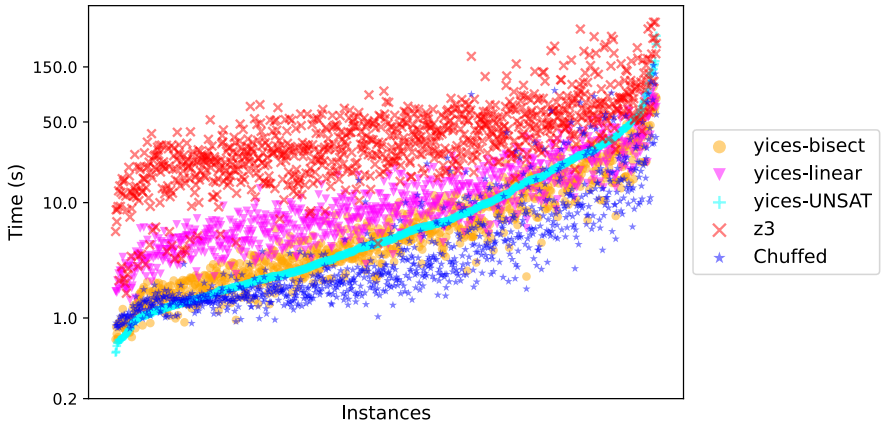


Figure 7.12: Solving time of YICES2 with versus without native interaction on 928 MRCPSP instances. Times are in seconds.

The results can be seen in fig. 7.12. As a comparison point, we used the `z3` SMT solver [DMB08] that already directly supports optimisation problems. Since we are testing out the SMT solvers with the same instances and in the same circumstances from fig. 7.6, we can bring CHUFFED solver as a comparison point as well. The results show that our interactive solver interface improves the performance of YICES2 significantly, allowing it to out-perform z3 in almost all instances.



(a) Without interactive solver interface



(b) With interactive solver interface

Figure 7.13: Solving time of YICES2 SMT solver with three settings (bisect, linear and UNSAT), z3 and Chuffed on 928 MRCPSP instances. YICES2' results are shown without (top) and with (bottom) native interaction.

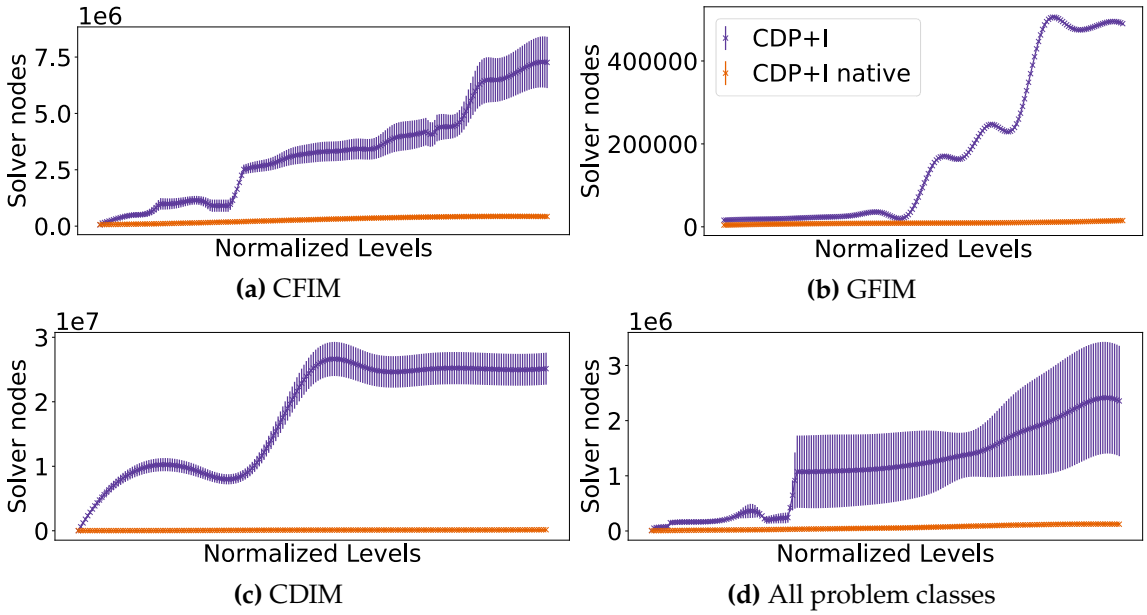
7.2.2.2 CDP+I Experiments

To evaluate the effectiveness of keeping learnt clauses and SMT assumptions between CDP+I levels, we test a small subset of instances across 3 of the problem classes: CFIM, CDIM, and GFIM from section 2.8, applying a 6 hour time limit. In our experiments, we use the same pre-processing options with BV encoding and the same standard SMT solver YICES2 with and without the interactive support.

Figure 7.14 presents the median number of search nodes per level. Since each instance has different numbers of levels, the levels are normalised with the number of levels on the horizontal axis, in a similar fashion to fig. 7.7f. The graphs show

that CDP+I with the SMT native interactive interface performs much better than regular CDP+I.

The results also illustrate that when using SMT on CDP+I, the problems do not get less challenging than the SAT equivalent ones as levels are completed and the problem approaches UNSAT. We believe the reason behind this can be related to how SMT solvers operate compared to SAT solvers in terms of usage of learnt clauses.



(a) CFIM **(b)** GFIM **(c)** CDIM **(d)** All problem classes

Figure 7.14: Median solver nodes per CDP+I level. Error bars range between the 45th and the 55th percentile. The horizontal axis represents normalised levels between instances. Native CDP+I uses significantly fewer search nodes, thanks to accumulated learnt clauses between levels.

Figure 7.15 illustrates a comparison of CDP+I with and without native interactive SMT solver interface using the standard SMT solver YICES2. Native interaction increases the performance amongst all given instances significantly, except some instances that have under a 10-second runtime.

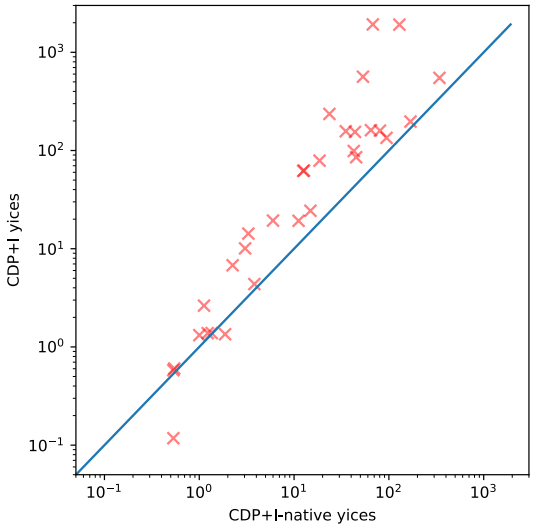


Figure 7.15: Comparison plot between pure CDP+I and CDP+I-native for SMT solver YICES2. Each data point is averaged out from a single model and multiple random seeds while a subset of all instances has been used. Times are in seconds.

CHAPTER EIGHT

INSTANCE GENERATION AND EXPERIMENTAL SETUP

This chapter is to explain how we constructed our experimental structure and generated the necessary instances, which is a complementary piece of our research. Section 8.1 investigates the instance generation methodologies we have used. Section 8.2 gives details on how experimental evaluation throughout the thesis has been handled.

8.1 Instance Generation

In our experiments, we use 16 transactional datasets from the CP4IM website. They are derived from UCI datasets, meant to be used for constraint-based itemset mining. The datasets which have been used with their characteristics can be seen in section 2.9.

To generate instances for 5 pattern mining problem classes with 2 side constraints (i.e. minimum utility and maximum cost) on 16 data-sets, for each item we uniformly randomly assign values between 0 and 5 to cost and utilities. We then use different techniques to find sensible and challenging minimum utility and maximum cost thresholds. These techniques include manual brute force

enumeration of the 2D search space section 8.1.1 and using a hyperparameter optimisation framework OPTUNA section 8.1.2. The experiments are conducted on mostly 5 different frequency thresholds: 10%, 20%, 30%, 40%, and 50%. For some case studies, we also generated some instances below 10%, namely 5% and 1%.

8.1.1 Manual Brute Force Instance Generation

The brute force instance generation is done by generating all possible instances in certain bounds for min-utility and max-cost values and picking the promising ones. For each dataset and each frequency, we determined predetermined costs and utility values. Using these, we generated a matrix of all possible instances and crawled the search space to select some as good instances.

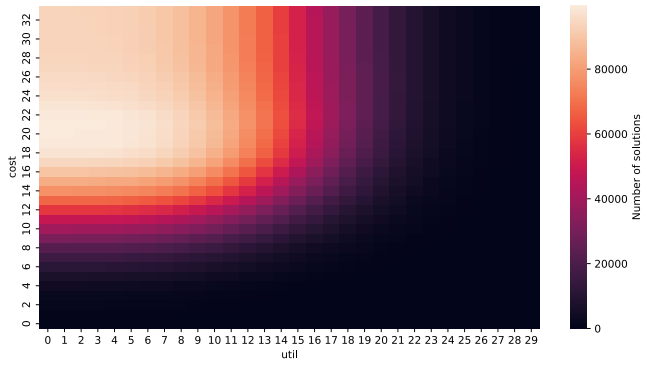
8.1.1.1 Two Solution Frontiers

Two 2D explorations of the search space for hepatitis 30% and lymph 10% on CFIM can be seen in fig. 8.1 and fig. 8.2.

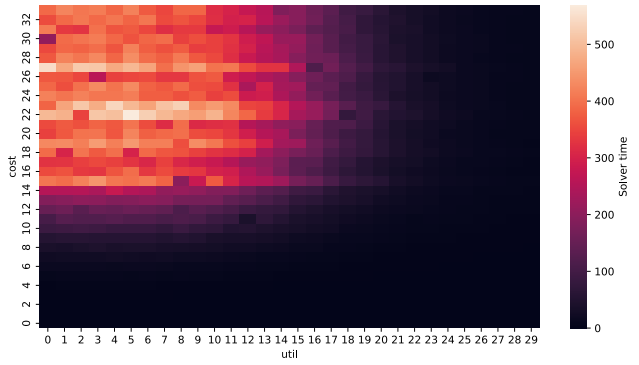
These figures show that it is trivial to recognise the frontier where the problem starts to permit numerous solutions. Additionally, at the same frontier, solver time syncs up with the number of solutions. However, if we also examine the solver time per solution, we can see another frontier that shows us where the problem becomes difficult. We can choose our criteria to be in between these two frontiers. Thanks to the first frontier, we can have multiple solutions to test our CDP/CDP+I framework while trying to stay close to the second frontier to select challenging instances.

8.1.1.2 Solution Compression Ratio with Expander

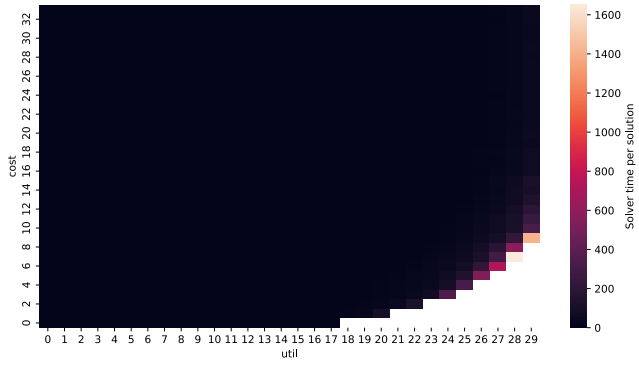
An important factor for the solutions is the compression ratio with its unconditioned total frequent itemset mining. While we can only apply this for most pattern mining problems we have, the calculation of this ratio gets more challenging with the complexity of the problem class. This metric shows how generative or



(a) Number of Solutions



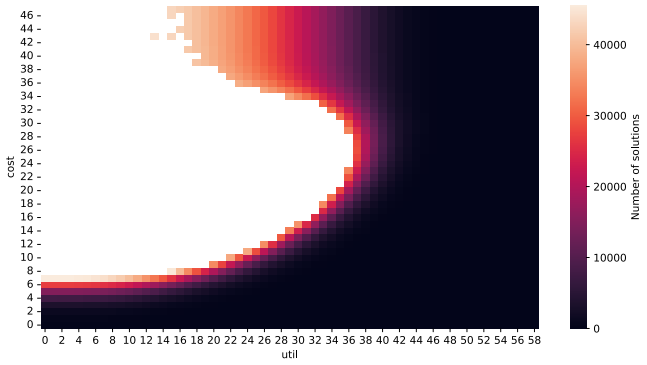
(b) Solver time (s)



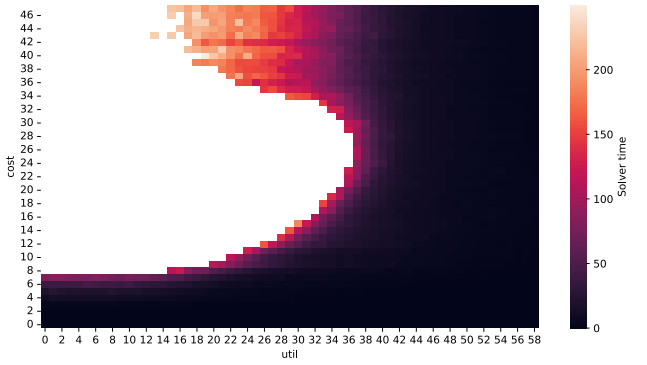
(c) Solver time (s) per solution

Figure 8.1: An example of manual instance tryout representation for the CFIM Hepatitis dataset at 30% frequency.

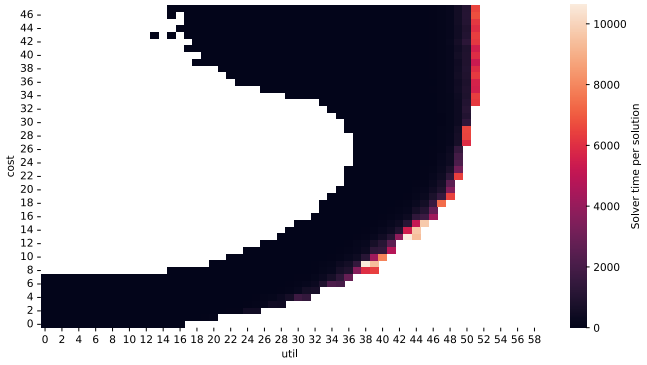
distinct the compressed patterns are through the amount of unfiltered patterns they capture. Generative patterns hold more information with higher compression while distinct pattern contain less information. We can define a good instance where the compression ratio is higher and a dense amount of information is packed in a small number of patterns.



(a) Number of Solutions



(b) Solver time (s)



(c) Solver time (s) per solution

Figure 8.2: An example of manual instance tryout representation for the CFIM Lymph dataset at 10% frequency.

Calculating the compression ratio of an instance requires finding the number of total patterns of the same instance. This can be done in two ways: 1) solving the same problem without in-between solution constraints or 2) solving the problem normally then applying a post-processing step. Since the number of frequent patterns can be exceedingly large, the first option might require enumerating

millions of solutions, which can be challenging for the ESSENCE pipeline to handle. The second option requires storing all of the compressed solutions and applying an expansion system. Storing all the solutions will increase the space complexity of the system drastically; however, storing a couple of thousand solutions should not create a big bottleneck.

For the expansion process, we developed another tool that works for MFIM and CFIM. This tool has been named `expander` [1] and is implemented in Rust with a focus of high optimisation with the purpose of expanding integer patterns with compact memory usage.

The general flow of `expander` follows as algorithm 5. This recursive algorithm goes through all possible subsets of a given solution and expands them while adding every new itemset to the total frequent itemsets. This procedure uses memoisation by using an ever-growing total itemset pool to automatically skip already explored paths. While both the time and space complexities of this system are $O(2^n)$, the memoisation reduces the number of cases where the upper bound worst-case time complexity is reached on consecutive runs of the algorithm with different solutions.

Algorithm 5 Expander for MFIM and CFIM

```
1:  Input
2:      c     Current Itemset
3:      t     The growing set of total frequent itemsets
4:  procedure EXPANDER(c, t)
5:      total ← c
6:      if |c| ≥ 1 then
7:          for item ← c do
8:              c ← c \ item
9:              if c ⊄ t ∧ C(c) then          ▷ C indicates side-constraints
10:                 expander(c, t)
11:             c ← c ∪ item
```

This system benefits from representing integer sets as Bit Vectors in memory.

[1] https://github.com/gokberkkocak/expander-rs

An example of Bit Vector representation of an itemset can be seen in eq. (8.1).

$$\{0,1,3,4\} \iff [1,1,0,1,1,0,0,..] \tag{8.1}$$

In this example, the integer set with 4 elements can be represented with a minimum of 5 bits since the maximum value we need to represent is 4. For datasets with up to 64 items, we can represent each itemset with a standard 64-bit integer. For the example itemset in eq. (8.1), the default set representation where index values are also encoded, 4 8-byte integers take 64 bytes in total. The same set in the BitVector form is only a single byte. This highly optimises memory usage and enables CPU specific vector operations as well.

However, having up to 64 items is not realistic. To represent itemsets in bigger datasets where more items are available, we need to use bigger integer constructs such as 128-bit integers or 256-bits. These may create additional CPU overhead on typical 64-bit systems.

The itemset expander tool is equipped with the ability to use different hashing mechanisms such as Fowler-Noll-Vo (FNV) hashing [2], Swiss Tables HashMap [3], and AHash with AES hashing [4]. Our preliminary testing concluded that using FNV hashing is the most efficient for our itemset mining problem thanks to its performance on small value hashing.

The real goal of the `expander` is to count the number of frequent itemsets. Consequently, the resulting itemsets are not wanted. Therefore, to boost the efficiency, we can eliminate storing the itemsets and only store the corresponding hash values. This behaviour can be dangerous due to the potential hash collisions. Typically, on normal sets hash collisions are avoided as the values stored to check hash collisions. In our experiments, we tested both our hash-only and normal versions of the program in multiple examples. Although, we did not experience any hash collisions in our tests, we decided to make `expander`'s default behaviour

[2] http://www.isthe.com/chongo/tech/comp/fnv/
[3] https://abseil.io/blog/20180927-swisstables
[4] https://github.com/tkaitchuck/ahash

a normal set rather than a faster hash-only set to prioritise safety. We left the hash-only feature as an experimental feature available in the tool.

To go even further with memory efficiency, an optional mimalloc [5] support is also available in `expander`. With it, it is possible to deactivate heap security features. Heap-security features allow a global allocator to encrypt the contents of the heap allocations with a performance overhead. Since we conduct our experiments using server machines on a private network and we deal with non-sensitive data, we can deactivate this feature for a performance boost.

Using the defined `expander` system, for the second lymph %10 example in fig. 8.2 the compression ratio of the grid can be seen in fig. 8.3. The compression ratio mostly stays close to 1 in many areas; however, in the upper region of the graph between the two frontiers, the compression is higher. We can choose select instances from this area.

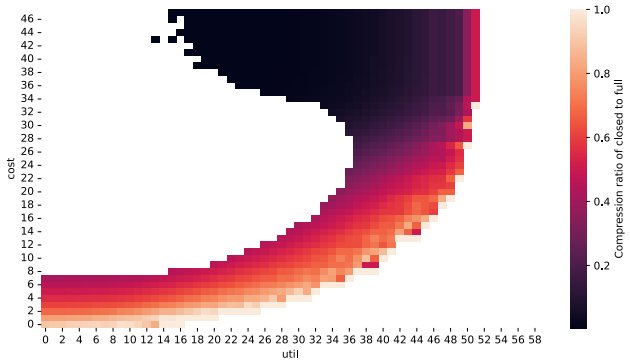


Figure 8.3: Compression ratio of closed itemsets to frequent itemsets for the example in fig. 8.2.

Even though one specific region might show more interesting instances, to test our instance in all possible ways we still generate some instances from other compression ratio regions and UNSAT instances for statistical purposes.

[5] https://github.com/microsoft/mimalloc

8.1.2 Using OPTUNA for Instance Generation

Instead of manually and brute force exploring the search space, it is possible to use hyperparameter optimisation frameworks to aid us in systematically crawling through the search space to generate instances. In our work, we use the OPTUNA framework, which we talked about in section 2.7.

An example code excerpt can be seen in fig. 8.4. The way OPTUNA works for a given parameter space is to operate on each trial's suggested values. By using two trial parameters for `min_util` and `max_cost`, we eliminate the need for a brute-force search in 2D util-cost plane and apply a depth-first search instead.

```
args = [..]
study = optuna.create_study(direction='maximize')
study.optimize(objective,
    n_trials=TRIALS, n_jobs=JOBS, timeout=TIMEOUT)

def objective(trial):
    opt_args["min_util"] = trial.suggest_int("min_util", 0, 15)
    opt_args["max_cost"] = trial.suggest_int("max_cost", 0, 15)
    output_file = to_essence_param(args, opt_args)
    info_file = miner_lite.solve(output_file)
    ..
    if nb is not nb > 1:
        return sr_total_time / float(nb)
    else:
        return 0
```

Figure 8.4: Example code to use Optuna to generate instances on the 2D util-cost plane.

With the usage of OPTUNA, it is also possible to increase the dimensionality beyond 2D. We can include other parameters previously taken as constant givens, such as the utility and the cost values. If there are n number of items, this will bring the instance dimensionality to $2n+2$ from 2, which is a drastic increase. In our experiments, generating instances while also taking utility and cost values into consideration did not perform well: one single instance generation can take several CPU days, excluding timeouts.

8.2 Experimental Setup

8.2.1 Experiment Management

Solving COP or CDP problems can be tasked off to one single CPU core/thread since the search base solving does not require any concurrency. By keeping every task to one single CPU it is possible to run multiple tasks if the machine has and supports multiple CPU cores/threads, which is standard in any personal computer after the 2010s. For our timed experiments, we used two identical 32-core AMD Opteron 6272 at 2.1 GHz with 256 GB RAM machines.

To enforce a more strict CPU core usage, we can designate each task to be in a control group supported by the Linux kernel (i.e. `cgroups`). This utility might be necessary since the ESSENCE pipeline includes SAVILE ROW, which is written in Java. The Java run-time is a multi-threaded application which might create bias positive in the experimentation process. Applying hard CPU core limits with `cgroups` to a process and its children processes, we eliminate the impact of using Java.

The next step is to run a large number of independent experiments by efficiently dispatching them into the machines. Since these experiments are time-consuming, any optimisation on running these experiments is beneficial for research to be conducted in faster iterations. For this reason, it is necessary to divide tasks into separate CPU cores in a clever fashion.

8.2.1.1 Using In-house Experiment Manager

To manage our experiments, we have developed an experiment handling system named Distributed Parallel Experiment Manager (DPEM) in Python [6]. Later, we ported this utility to Rust [7].

This tool expands on top of gnu-parallel [Tan15], which is capable of smart dispatching of tasks to reduce the wasted CPU and total wall-time. For a single

[6] https://github.com/gokberkkocak/dpem/
[7] https://github.com/gokberkkocak/dpemr/

machine dispatching of experiments, GNU-parallel is one of the most efficient tools we can benefit from. However, since we have access to multiple machines, it is necessary to use remote dispatching on multiple computers. GNU-parallel supports remote dispatching by using the `-sshlogin` flag. However, the number of tasks to be dispatched is always assumed to be the number of cores available. This behaviour is not desired since we would like to use a subset of the cores (e.g 20 out of 32 cores) and reserve the rest to eliminate any OS-level discrepancies.

An easy solution to having multiple machines is to distribute the tasks into the machine preemptively. This will assign each experiment to a machine at the start of the whole experimentation process. Doing so might lead to some machines running at full time while other machines are idle. To overcome this, we propose a centralised task dispatching and running system. A visualisation of this system can be seen in fig. 8.5. A system like this will dispatch the tasks to each machine on run-time depending on the load of each machine. To achieve this, we need to have a separate SQL database management entity to fetch and register tasks. Upon research, we found an existing tool called parallel-SQL [8] that extends from gnu-parallel while using a PostgresSQL for task management. However, usage of parallel-SQL requires an additional controller machine that runs a PostgreSQL server, which might be overkill for a small scale application like this one. A lighter MySQL or MariaDB can be more suitable for this application. Another alternative is using SQLite if there is a shared file system between machines and the database and can be kept in a local file with tight write and read locks on the database file.

A second possible problem with parallel-SQL is that it is not actively maintained, with the latest commit on Github going back to 2017. This can create potential problems with later versions of the kernel or libraries. Freezing the state of the OS and packages just for parallel-SQL on a highly maintained and up-to-date server machine is not feasible.

In the end, we implemented a MySQL/MariaDB backend to fetch and register experiments. Since the the University of St Andrews Computer Science students

[8] https://github.com/stephen-fralich/parallel-sql

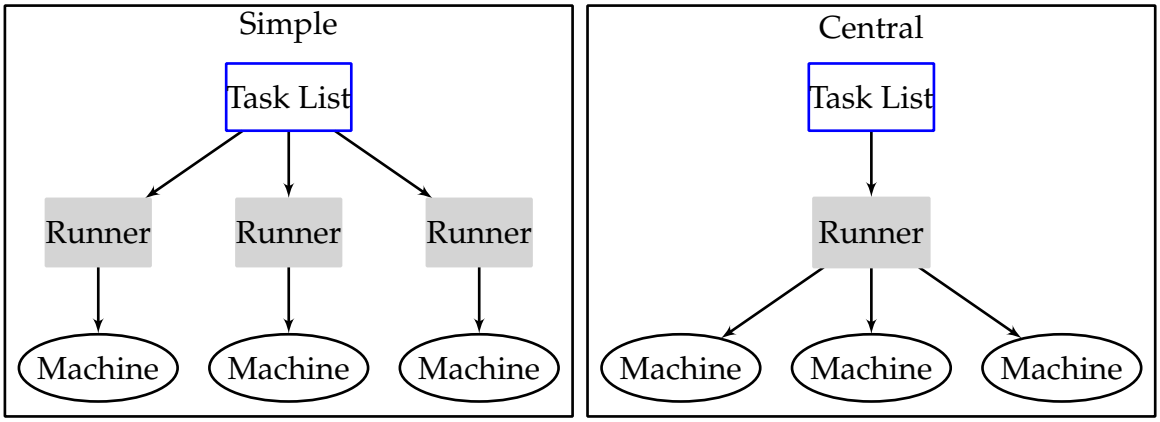


Figure 8.5: Two execution paths for parallel experiment dispatching system. While the system on the left assigns tasks to each runner, the right alternative uses a centralised runner.

can access an active MariaDB database, this would not require running a separate SQL server.

We also incorporated additional features to DPEM where each registered task reports back to the database with the result code of the experiment. The result codes indicate success, timeout within the given time, or crash due to any possible reason. This allowed us to implement a useful feature called *automatic timeout increase* in our experiments. If desired, this structure allows re-running some of the experiments without altering the timeout value manually.

While we can register the result of the experiment task, as the database is deliberately being kept light it does not contain all the information about the experiments necessary to make a statistical analysis. Instead, we dump each experiment result in a JSON file to later collect and process the information.

8.2.2 Experiment Collection and Processing

As indicated in section 8.2.1.1, each experiment result is dumped as a JSON file to be later collected for processing. The experiment results contain: 1) time-specific information such as solver time and total time, 2) the number of solutions and, 3) the solutions themselves if they are required for `expander` (section 8.1.1.2). Additionally, level-specific solving information (e.g solver time on each level and

the number of solutions on each level) is also dumped for CDP+I experiments.

Recording all this information for all experiments can be quite space-intensive; any processing done on this information can be CPU intensive as well. Any possible optimisations on this structure will allow research to be conducted more efficiently.

Initially, we used simple Python scripts to parse and collect information from the JSON files. Considering we collect over 200K distinct experiments with repetitions, using unoptimised un-typed data parsing can be resource-intensive and unfeasible to work with. To overcome this performance bottleneck, we have developed a tool called result-reader-on-rust (rrr) [9]. Using strongly typed data representations and tagged enum structures for heterogeneous result types (given in serde crate family [10]), it is possible to speed up the processing by up to 100 times while keeping the memory usage minimal. The heterogeneous solver information structure can be seen in fig. D.1 in appendix D.

In addition to optimising and matching the internal structure of the experiment results, we also added a database integration system to `rrr`, similar to the one in section 8.2.1.1, so that we could retrieve some basic information about the experiment result in real-time. The representation of this structure is available in fig. D.2 in appendix D.

Having some of the experiment results available in real-time immediately after the experiment is completed potentially opens up useful paths. One particular example is to conclude race type experiments early if any of the experiments are finished. One other example is to determine a *dynamic timeout* where if a configuration of an instance is being tested while any other configuration of the same instance is concluded, we can alter the timeout of the first experiment automatically in real-time. In theory, the *dynamic timeout* feature can allow us to lower the timeouts of experiments that have already been outperformed, saving CPU time.

For plotting purposes, we implemented a feature on `rrr` called *plotting views*.

[9] https://github.com/gokberkkocak/rrr
[10] https://serde.rs/

When we generate necessary plots from our experimental data, the complexity of the data can be considerably large. We experienced significant time and memory issues during this step. Thus, we decided to create this view system. Using the strongly structured experiment results as a base, we can define an almost zero-copy view using borrowed structures operating on the large JSON construct with minimal additional memory usage. An example structure can be seen in fig. D.3 in appendix D.

While the almost complete plot view can be used to generate statistical plots with ease, it avoids exposing all of the experiments' results by applying the mean of multiple results upon query. More verbose complete data can be used for more detailed statistical analysis.

One last important feature, which has been implemented on `rrr`, is compressing the storage of the experiments. Since the data's entropy can be smaller in the JSON file format with repeated header information, to be space-efficient we incorporated the option of using zstandard (zstd) [11] compression into our system, saving up to 10 times more space with minimal CPU overhead.

[11] **https://facebook.github.io/zstd/**

CHAPTER NINE

AUTOMATED CONFIGURATION SELECTION

With the CDP+I framework and SII, we have improved the efficiency of solving pattern mining problems. These improvements are based on using a single model for a given instance. The next logical step is to consider multiple modelling/solving pipelines as possible configurations and create an efficient portfolio. To do so, in this chapter, we firstly discuss the vast configuration space of the ESSENCE pipeline with CDP+I and SII. Then, we create two distinct automated configuration selection and portfolio building systems, one that uses SMAC [HHLB11] (mentioned in section 2.6) and another that defines our method using instance features. Lastly, we look into feature importance in a constructed portfolio.

9.1 Configuration Space

When solving a pattern mining problem that has been represented in ESSENCE, there are many parameters to consider both for the ESSENCE pipeline and the solving process. Depending on parameter choices, the solving time of these problems can change drastically for better or for worse. To guarantee achieving

high performance, we plan to use the combinations of the possible parameters and create a vast configuration space. Then, we aim to create a configuration selection system to choose efficient configurations in a systematic way.

9.1.1 Model Representation

The first parameter, namely `model`, comes from the translation of an ESSENCE specification to the lower-level modelling language ESSENCE PRIME using CONJURE. During the translation of the high-level specification to the lower-level ones, the high-level types such as sets can be expressed in multiple ways. Thus, CONJURE can produce several possible models using different *refinement* rules (see section 2.3.1 for more detail). We consider two representation choices: *Explicit* and *Occurrence*. Explicit rules represent memberships of elements in high-level data types directly in the lower level with additional constructs like flags, markers, or a dummy value. Occurrence representation rules use a Boolean matrix to indicate whether an element belongs to the corresponding variable. Two examples of the translation of the freq_items variable of an itemset mining problem from ESSENCE to ESSENCE PRIME can be seen in fig. 9.1. One has an explicit (with flags) representation while the other has an occurrence representation.

Representation choices have to be made for each of the given parameters and decision variables of a problem specification. To simplify things and reduce the number of configurations, we decided to use either use explicit or occurrence for all parameters and decision variables. This decision gives us four possible values for the `model` parameter: Explicit-Explicit (EE), Occurrence-Occurrence (OO), Explicit-Occurrence (EO), and Occurrence-Explicit (OE).

9.1.2 Pre-processing

`preprocessing` comes from the use of SAVILE ROW where the ESSENCE PRIME level constraint model is translated into a low-level solver specific format. If desired, SAVILE ROW can apply optimisation techniques [NAG+17] during its

```
find freq_items :
    (set (maxSize db_maxEntrySize) of
        int(db_minValue..db_maxValue), int(1..db_row_size))
```

```
find freq_items_1_ExplicitVarSizeWithFlags_Flags:
    matrix indexed by [int(1..db_maxEntrySize)] of bool
find freq_items_1_ExplicitVarSizeWithFlags_Values:
        matrix indexed by [int(1..db_maxEntrySize)]
            of int(db_minValue..db_maxValue)
```

```
find freq_items_1_Occurrence:
    matrix indexed by
        [int(db_minValue..db_maxValue)] of bool
```

Figure 9.1: ESSENCE and two refined ESSENCE PRIME versions of the freq_items construct from a pattern mining problem. The first figure shows the ESSENCE specification which uses sets while the others use a matrix in ESSENCE PRIME with explicit or occurrence representation

translation process. Among these, the choice of pre-processing methods potentially has a great impact. Since the variables' domains are possibly shrunk due to pre-processing using arc-consistency methods, this will affect the number of levels on CDP+I by removing unfeasible levels. This can give a great performance boost to the solving system.

We consider four choices for the `preprocessing` parameter. The first one is *SACBounds* (S), a Singleton Arc Consistency-based method [BCDL11] to prune the bounds of decision variables. The second choice, *SSACBounds* (SS), is a two-layer application of SACBounds. The third is the Generalised Arc Consistency [RVBW06] (GAC). The last choice applies no arc-consistency preprocessing (None). There are potential trade-offs to be considered when using high level preprocessing, such as SACBounds or SSACBounds, as opposed to simpler approaches like GAC or None when considering the total running time of the whole solving process. High-level preprocessing reduces the number of levels but the reformulation process itself can take a lot of time in some cases.

9.1.3 Solving Back-end and Solver

SAVILE ROW is capable of targeting multiple solvers with different back-ends. While the MINION solver back-end is the default option, SAVILE ROW has a well-established SAT back-end [NSM15]. With SAVILE ROW, it is also possible to target SMT solvers [DAEN20].

While we use the standard SAT encoding, different SMT theories can be enabled if the solver supports it. Four major SMT logics are: BV (Bit Vector), LIA (Linear Integer Arithmetic), NIA (Non-Linear Integer Arithmetic), and IDL (Integer Difference Logic).

Considering the possible options given, the list of solvers we can target on that work with CDP/CDP+I framework becomes: NBC_MINISAT_ALL [TS16] (NBC) - an AllSAT solver, GLUCOSE [AS18] (GLU) - a standard SAT solver, Cadical (CAD) - a standard SAT solver, Z3 - a SMT solver with BV, LIA, NIA or IDL support, Boolector (BOL) - a SMT solver specialised on BV encoding, YICES2 (Y2) - a SMT solver with BV and LIA support, and MINION CP solver (MIN).

Thus, for the parameter `solver`, we have 11 options: NBC, GLU, CAD, Z3-BV, Z3-LIA, Z3-NIA, Z3-IDL, BOL, Y2-BV, Y2-LIA, and MIN.

9.1.4 CDP/CDP+I

We can consider the usage of the CDP+I in exchange for CDP as a parameter as well. While in the earlier chapters, we experiment in detail and conclude that CDP+I statistically outperforms CDP in almost all cases, for certain edge cases CDP can still be competitive. Thus, CDP can still be considered an option for the configuration space.

9.1.4.1 Different Incomparabilites for RSD

While for most pattern mining problems we have defined one single incomparability, RSD problem class can be actually used with another incomparability defined

in section 6.4.3. This can be considered an optional parameter for this problem class.

9.1.5 Solver Interaction Scheme

For SAT and SMT back-ends, we have defined native interaction schemes (SII) over in section 7.1 and section 7.2. While it is not possible to use this native interaction system on MINION, these schemes can be used as options on SAT and SMT solvers.

9.1.6 Reformulation

On the implementation of CDP+I in section 5.3.1, we pointed out that it is possible to apply additional reformulation rules on the generated constraints to potentially boost the performance of the system. This also can be used as an optional scheme for the configuration space.

9.1.7 Final Configuration Space

With 4 pre-processing options, 4 model encodings and, 11 solvers we have 176 possible combinations without interactivity and reformulations. We can only use 10 solver options with SII. With SII and reformulations, this results in 320 configurations in general and 640 for RSD. The total number of configurations is 496 in general and 816 for RSD.

9.2 Automated Configuration Selection Using SMAC

In this section, we explore some of the configuration space of CDP/CDP+I by tuning those choices using the automated algorithm configuration tool SMAC [HHLB11]. We first describe the details of the parameters being tuned and the values considered in this study. Tuning results across all problem classes are then presented. Finally, we discuss the potential of building a portfolio

of configurations with complementary strengths to achieve the best possible performance.

For these experiments, we used a subset of all possible configurations (section 9.1) that have incomparability and SII enabled (CDP+I-native) on SAT backend with GLU and NBC solvers. The reason we used a subset is to reduce computation space while using a highly competitive configuration (i.e CDP+I-native with NBC/GLU) as a starting point. Since all experiments use CDP+I-native in this setup, incomparability and interactivity indicators are omitted from the configuration names. We named the configurations with model-preprocessing-solver triplets.

9.2.1 Tuning Results

We use the automated algorithm configuration tool SMAC[1] [HHLB11] to find the best configuration across all problem classes. SMAC is a sequential optimisation method based on Bayesian Optimisation [Moc12]. The tool has a wide range of applications and has been shown to efficiently find well-performing algorithm configurations in several case studies [HHLB11].

We launched 24 SMAC runs in parallel on our local experiment runner machines (explained in section 8.2.1). Each SMAC run took one core and an average of five CPU days. These resources were sufficient due to the use of an experiment database consisting of configuration-instance results, which allowed SMAC runs to directly use them if available, leading to reduced CPU time. A time limit of 6 hours was given to each configuration evaluation on an instance. Performance data was shared among those runs during the tuning using the shared-model setting of SMAC [HHLB12]. We split instances into training and test sets with a ratio of approximately 3:1, i.e., 172 training instances were given to SMAC and 53 test instances were used for the test phase.

After the training phase, the multiple SMAC runs return seven configurations: OO-GAC-NBC, OO-GAC-GLU, OO-None-NBC, EO-GAC-NBC, EO-None-NBC,

[1] https://github.com/automl/SMAC3

EO-GAC-GLU, and OO-None-GLU. These configurations are evaluated on the test set with 3 runs per instance. The non-parametric Friedman test [Fri37] with Nemenyi's All-Pairs Comparisons Test [Nem62] for post-hoc analysis to identify the statistically significantly different group of configurations (with a confidence level of 99%) are applied on the test performance data[2]. Results indicate that the two configurations OO-None-NBC and OO-GAC-NBC are statistically significantly better than the others.

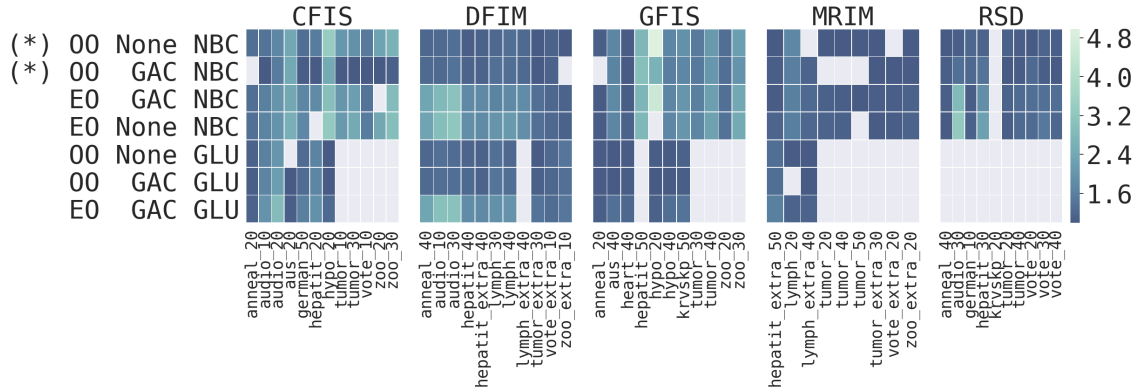


Figure 9.2: Normalised solving time (s) (on test instances) of configurations returned by SMAC. The normalisation is calculated as the original solving time divided by the best solving time observed on the corresponding instance. All timeout runs within 6 hours are marked with light-grey colour. The configurations are sorted according to their average ranks across all test instances (from best to worst). Statistically, significantly better configurations are marked with (*).

A more detailed view of the test performance of those configurations is shown in Figure 9.2, where the solving time of a configuration on an instance is normalised by dividing it by the best solving time observed on that instance. Although the two configurations OO-None-NBC and OO-GAC-NBC give the best overall performance, the best-performing configurations per problem class can be different. For example, for MRIM, EO-GAC-NBC gives the best performance, followed by OO-None-GLU. Even within the same problem class, some configurations might perform very well on a subset of instances. This is illustrated in GFIM, where OO-None-GLU, OO-GAC-GLU and EO-GAC-GLU show the best performance on

[2]For statistical analysis we use the implementation provided by the R package `PMCMRplus` (**https://cran.r-project.org/web/packages/PMCMRplus/index.html**)

six instances while the rest time out. These observations indicate the potential of using a portfolio of CDP+I-native configurations to further improve the overall performance of the solving process.

9.2.2 Configuration Space Analysis

We further investigate the configuration space by testing all 32 possible configurations on the test instances using a capped time limit for each instance. This limit is calculated as twice the best solving time on each instance obtained from previously evaluated runs for the tuned configurations in Figure 9.2. Using capped times instead of the original 6-hour time limit is important to make the computational cost manageable, since bad configurations may timeout on several or all instances, thus consuming lots of time. As we are mostly interested in the reasonably-performing regions of the configuration space, it is unnecessary to spend too much time evaluating poorly-performing configurations.

Figure 9.3 shows the normalised capped-solving time of all configurations on the test instances. The two configurations OO-GAC-NBC and OO-None-NBC are again among the highest-ranked configurations. Interestingly, configuration OO-S-NBC also performs very well but was not returned by SMAC. This can be explained by the fact that we are using the shared-mode of SMAC, which may make the parallel tuning runs converge to similar regions of the configuration space, therefore missing out the configuration OO-S-NBC.

Presented results clearly show the distinction between two groups of configurations: the ones that use the Occurrence representation for decision variables (`model` $\in$ {OO, EO}) and the ones using the Explicit decision-representation (`model` $\in$ {OE, EE}). The latter exceeds the capped time limit on all test instances, which indicates their consistently poor performance across all problem classes. An explanation for this observation is that variable cardinality explicit representations can create a large number of conditional constraints, especially when the cardinality of the set is not restricted. This is because they need to create enough variables to store a very dense set, whereas in practice many solutions

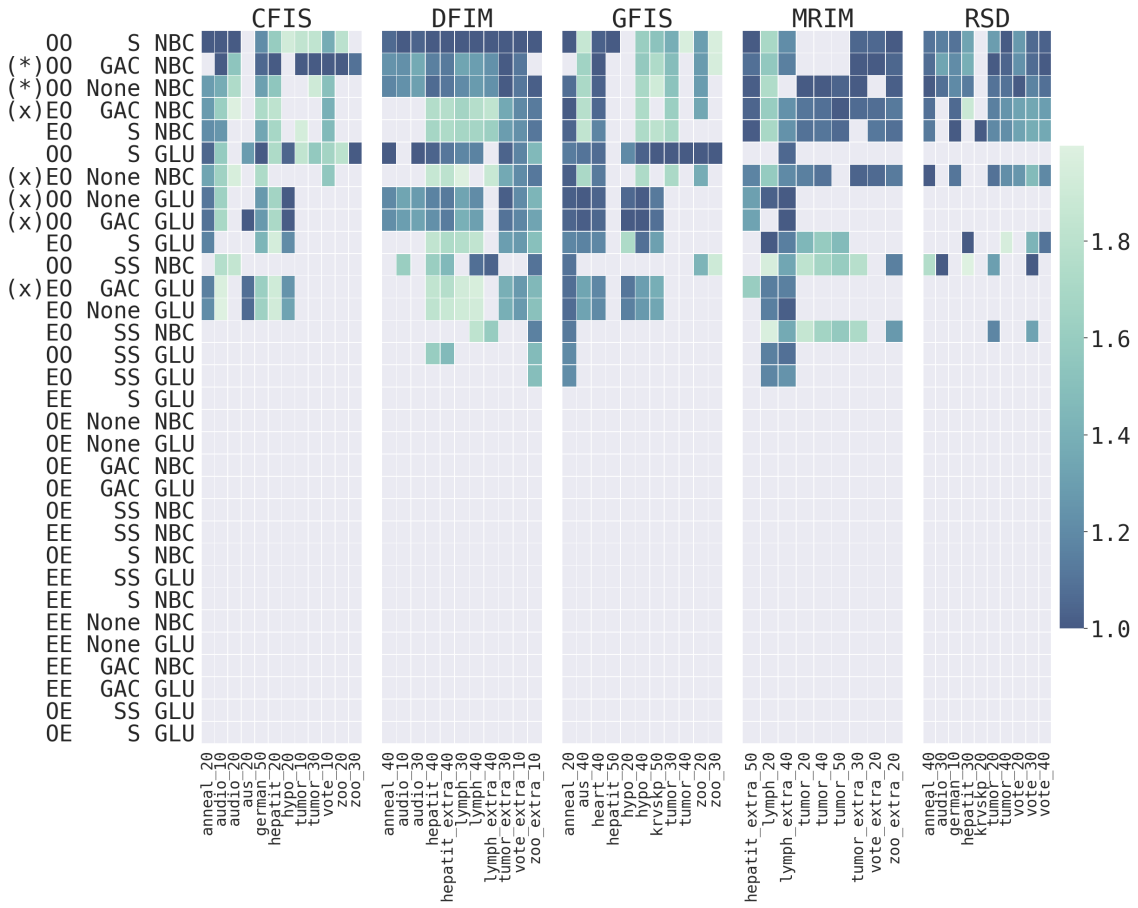


Figure 9.3: Normalised capped-solving time (s) (on test instances) of all 32 configurations. All runs exceeding the capped limit are marked with light-grey colour. The configurations are sorted according to their average ranks across all test instances (from best to worst). The two statistically significantly better configurations returned by SMAC are marked with (*) and, the other tuned configurations are marked with (x).

(frequent itemsets) are likely to be sparse. The representation choice for problem parameters, on the other hand, does not have a large impact on the performance compared to their decision-variable counterpart, as both OO and EO are present in the top configurations, although OO slightly dominates. For the second parameter, `preprocessing`, the three choices S, GAC and None are alternatively chosen in the top configurations. The remaining choice, SS, is associated with lower-performing configurations. This is expected, as SS (Double SACBounds) is the most expensive preprocessing approach among the four choices.

The detailed performance values in Figure 9.3 once again confirm our previous observation on the complementary strengths of different CDP+I-native config-

urations. For example, OO-S-GLU performs strongly on GFIM while showing poor performance on MRIM and RSD. Within the same problem class, the best overall configuration can be dominated by lower-ranked configurations on certain instances. A supporting example can be seen on CFIM, where OO-GAC-GLU, EO-GAC-GLU and EO-None-GLU achieve the lowest solving time on instance `aus_20`.

To facilitate the idea of building a portfolio of configurations and dynamically choosing the best ones to be applied based on characteristics of a given problem instance, we need *instance features*. These features should represent the high-level characteristics of an itemset mining problem and its instance parameters. Some examples of instance-level features include statistics (min, max, mean, standard deviation, etc) of the utility and cost distributions, the density of the given dataset, and how irregular the transactions are in terms of length.

9.3 Portfolio Building and Automated Configuration Selection

Results achieved from the configuration sub-space in the previous section indicate the potential of having a well-performing *portfolio of configurations* with complementary strength. In this section, we investigate that potential on the full configuration space and propose an automated approach for instance-specific configuration selection. Our suggested portfolio building process can be seen in fig. 9.4.

The process starts with pre-processing the configuration space. Filtering the configuration space is necessary to reduce the number of configurations in the portfolio to a reasonable number. Instead of focusing on a sub-set of the configuration space (done in section 9.2), we take the full configuration space and apply filtering techniques to eliminate poor configurations. We first define the competitiveness measurement and its threshold parameter. Afterwards, using the competitiveness we can remove any non-competitive or any dominated

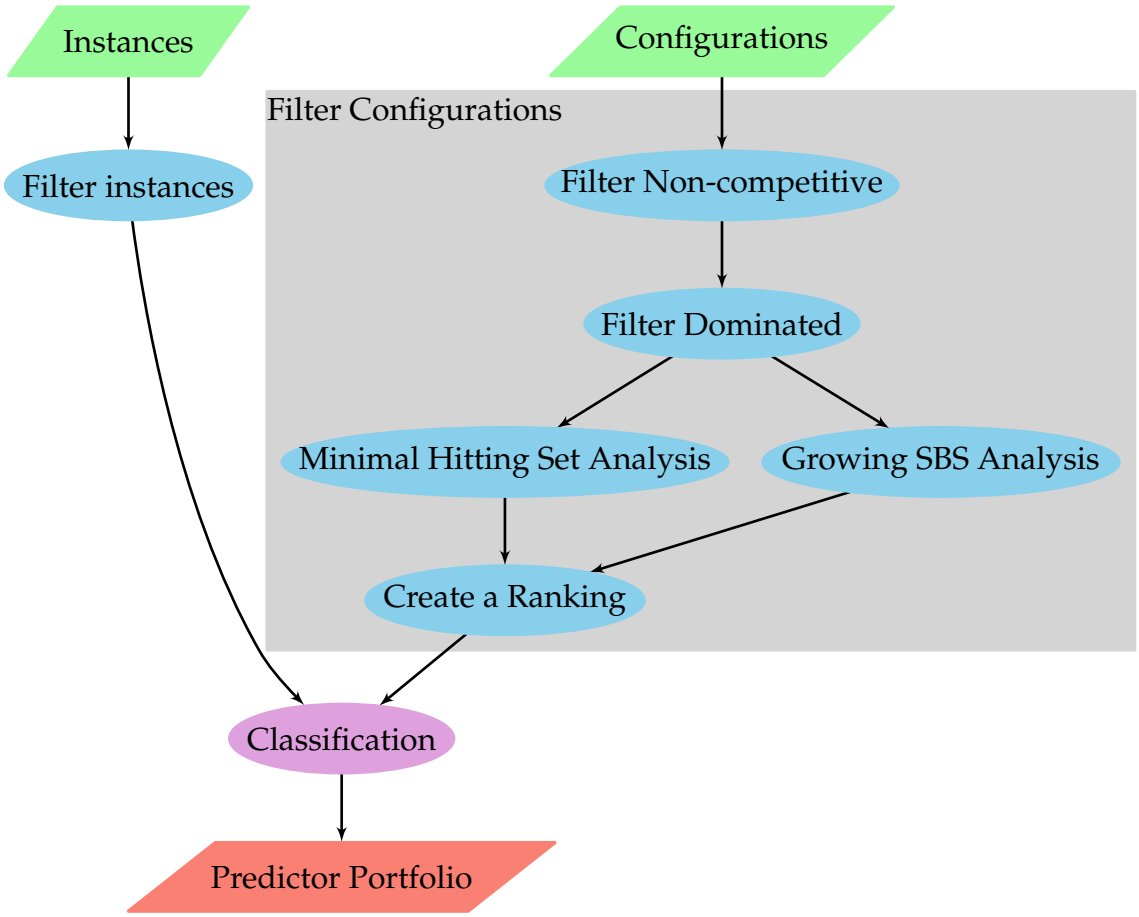


Figure 9.4: Procedure of the Portfolio Building.

configurations. Then, we simultaneously apply Minimal Hitting Set and Growing SBS analyses to filter the configuration space even further. As the last step in the configuration pre-processing, we create a ranking for the remaining configurations. In the meanwhile, we also pre-process our instance space to remove any unfeasible instances. In the end, we supply the remaining configurations and instances into the classification system where we create one classification predictor per configuration.

9.3.1 Competitiveness Metrics

Before any pre-processing, we need to define a criteria for a configuration to become competitive.

Collected results contains the raw solving time of every configuration on

each instance. While a single solving time can indicate that a configuration is competitive amongst all the configurations on a single instance, this does not generalise the competitiveness. A particular configuration might perform differently on different instances. Thus, a well-defined metric is necessary to determine competitiveness across different instances.

To do so, we use two different competitiveness metrics:

A) Threshold based,

B) Run-time based.

We define our first competitiveness metric (A), which regards configurations within a certain threshold of the best configuration range as competitive. This threshold is a variable in our experiments. We used +25%, +50%, +66%, +100%, and +150% as different threshold options in our portfolio building system.

As an example, we can consider two configurations on one instance: c_1 and c_2 with solving times t and $1.75t$ respectively. Since c_1 achieves the best solving time with t, the time t will be the point of comparison for different thresholds. For the thresholds +25%, +50% and +66%, the time given by c_2 ($1.75t$) is above the required threshold. Therefore, on these thresholds, c_2 would not be considered competitive on this instance. On the thresholds +100% and +150%, c_2 is now below the required threshold and is considered competitive alongside with c_1.

Using different competitiveness thresholds allows us to define the competitiveness point dynamically so that we can try to tighten or relax the parameter to find an optimal point. However, if an instance is too easy to solve (such as in a second), the competitiveness at any threshold can be considered harsh on all other configurations by only allowing a small margin. To avoid penalising configurations on such instances, we added another rule to consider any configurations under 10 seconds competitive.

The following is an excerpt of solving time data in Table 9.1. This table shows three instances for three configurations. Although the configurations are equipped with CDP+I-native (with SII), it is omitted from their names for plotting purposes.

experiment_names	GAC-EO-yices2	GAC-EO-nbc	GAC-OO-cadical
disc_zoo_30_30.0	12.60	10.51	8.06
gen_zoo_50_50.0	32.67	14.79	18.71
closed_zoo_50_50.0	11.67	9.16	8.55

Table 9.1: An excerpt of the run-time data (in s) for random 3 instances with 3 random configurations.

As an example, the data excerpt from Table 9.1 becomes Table 9.2 when the competitiveness metric (A) is calculated on the lowest threshold point +25%.

experiment	GAC-EO-yices2	GAC-EO-nbc	GAC-OO-cadical
disc_zoo_30_30.0	1	1	1
gen_zoo_50_50.0	1	0	1
closed_zoo_50_50.0	1	1	1

Table 9.2: An excerpt of the competitiveness(A) data with 3 random configurations selected. GAC-EO-nbc is not competitive on gen_zoo_50_50.0 due to another configuration in the pool disallowing it.

The second competitiveness metric (B) we define uses average run-time (or total run-time) directly. This will allow us to better represent the effectiveness of the portfolio using run-time performance at the potential cost of increased CPU time on any operation on configurations. If a configuration could not finish a particular instance (i.e. does not have a time attached to an instance), the time is interpreted as the maximum timeout, which is 6 hours. This will penalise timed-out configurations on that particular instance for run-time competitive measurements.

We mainly use cheaper competitiveness metric (A) in our system. However, in various places we still use the competitiveness metric (B). In those places, we also mark the operations as *run-time based* to explicitly differentiate.

9.3.2 Filtering Instances

As a starting procedure, we apply an early pre-processing pass to remove any infeasible instances for all configurations. This requires running all configurations

on all instances. With our experiment management and collection system with *dynamic timeout* (see section 8.2), we already have this information.

Applying this filter pass, out of 312 generated instances, 74 instances never seem to be solvable by any configuration. Thus, they will not be contributing towards differentiating configurations and can be removed from the experimentation.

9.3.3 Filtering Configurations

As mentioned in section 9.1, we have a vast number of configurations. To be able to create an effective portfolio, we need to filter the number of configurations to a handful of configurations first.

9.3.3.1 Remove Non-Competitive Configurations

With our competitiveness metric (A), it is possible to reduce the number of configurations by eliminating any configuration that never managed to be competitive.

The defined competitiveness threshold for measure A directly impacts the number of competitive configurations. If we continue with the +25% competitiveness threshold as our running example, we can see that out of 496 configurations, 78 configurations remains competitive.

9.3.3.2 Configuration Domination Analysis

Using our main competitiveness metric (A), we can do a domination analysis in between configurations. By applying a pair-wise comparison on all configurations, we can determine if a configuration subsumes another configuration in terms of being competitive. We can define this dominance system similarly to the dominance programming (see section 2.5). A configuration Y is dominated by X if X is competitive everywhere Y is also competitive. Additionally, X is also competitive in some other instances where Y is not (i.e. X's competitive instances supersets Y's ones). We can remove the dominated configurations to reduce the portfolio size further to only have a handful of configurations.

Using our 25% threshold example, after this elimination process, only 30 configurations remain.

9.3.3.3 Minimal Hitting Set Analysis

Minimal hitting set or set cover problem is defined as: given a set of elements from $E = \{1..n\}$ and a collection of S where $\bigcup_i S_i = E$ (i.e. all elements are covered with the collection S), find the smallest sub-collection of S that still covers the whole elements.

In our context, the instances are the elements and the set of configurations is the collection. Our goal is to find the minimal sub-set of configurations where all the instances are covered competitively. We can apply this analysis on both of our competitiveness metrics, A and B. Minimal hitting set for any satisfying solution is an NP-complete decision problem, whereas the optimisation version is NP-hard. We can directly try to solve with optimality using COP models.

The ESSENCE specification for the minimal hitting set using the competitiveness metric (A) can be seen in fig. 9.5.

```
letting config_domain be domain int(0..n)
given total_set: set of int
given configs :
    function (total) config_domain --> set of int
find result: set of int(0..n)

minimising |result|

such that
    forAll i in total_set .
        exists row in result .
            i in configs(row)
```

Figure 9.5: ESSENCE specification for the Minimal hitting set problem using the competitiveness metric A.

The given *total_set* represents the instances as a set of integers. Every instance and every configuration are denoted with integers with a total of n number of configurations to choose from.

The model ensures that for every instance, there exists at least one configuration that covers that instance. The optimisation goal is to minimise the number of configurations that cover all instances. After finding a lower bound for the number of configurations required, we can search for all possible solutions on this bound. Looking at all the found solutions from the minimal hitting set, we can see which configurations are never included in any set and which are included in any set at least once. To reduce the portfolio space further, we remove the configurations that are never included in any of the minimal hitting sets (i.e. we union all of the resulting sets of configurations).

For our running example +25% threshold, the minimal hitting model is solvable almost instantly and we achieve the lower bound of 16 as the minimum number of configurations. Setting 16 as the cardinality, when we look for all possible solutions, we find 4 different solutions which include 19 different configurations out of all 30.

Since the minimal hitting set problem can be crucial the reduce the portfolio space, we would like to conduct the same experiment using the competitiveness metric (B) as well. Modelling the minimal hitting set problem is more challenging since we need to use a total run-time as the threshold. This value can be chosen somewhat close to the oracle/virtual-best to ensure a reasonable portfolio performance.

The ESSENCE specification for the minimal hitting set using the competitiveness metric (B) can be seen in fig. 9.6.

The model looks similar to the other minimal hitting set model. However, this time configurations map to each instance's run-time. The time values are multiplied by 10 to represent the first floating-point in ESSENCE. The goal is to find the minimal configuration portfolio with a good total run-time.

Running this model is challenging. For most circumstances, finding a solution with MINION or local search solver Athanor is not possible, as it can take days. Using CHUFFED, we managed to solve the problem approximately in one hour depending on `THRESHOLD`.

```
letting config_domain be domain int(0..n)
given total_set: set of int
given THRESHOLD: int(1..vbest-100000)
given configs : function (total) config_domain
    --> sequence (size 238) of int(0..216000)
find result: set of int(0..n)
find result_mat: sequence (size 238) of int(0..216000)
find sum_av_time : int(0..MAX)

minimising |result|

such that
    forAll i in total_set .
        exists row in result .
            configs(row)(i) = result_mat(i)
such that
    sum_av_time = (sum i : int(1..238) . result_mat(i))
such that
    sum_av_time < THRESHOLD
```

Figure 9.6: ESSENCE specification for the Minimal hitting set problem using the competitiveness metric B with additional run-time threshold side constraint. Time values are altered by multiplication of 10 to represent the first floating point in ESSENCE.

```
language Essence 1.3

letting result be {3, 5, 9, 13, 19, 21, 23}
letting result_mat be ...
letting sum_av_time be 1999999
```

Figure 9.7: Result of the minimal hitting set on metric B. The resulting matrix where instance times are shown has been redacted.

Continuing with our +25% threshold example, if we set the THRESHOLD to around 200K secs (2M in the ESSENCE parameter), we can find a 7 configuration portfolio which can be seen in Figure 9.7. This 7 configuration portfolio can work to within 95% accuracy of the oracle's performance.

On the same competitiveness threshold +25% example, assigning run-time THRESHOLD to an even tighter 190K seconds, we can find 11 configurations as the result. They can be seen in fig. 9.8. This portfolio performs within 99% of the oracle's performance.

```
language Essence 1.3

letting result be {3, 5, 8, 9, 12, 13, 15, 19, 21, 23, 28}
letting result_mat be ...
letting sum_av_time be 1899999
```

Figure 9.8: Result of the minimal hitting set on metric B with a tighter threshold. The resulting matrix where instance times are shown has been redacted.

We can use the same approach we used in the first minimal hitting set analysis with metric (A) and remove any configurations that are never included in any of the minimal hitting sets. Doing so, we filter the configuration space and reduce the portfolio even further using run-time metric (B).

9.3.3.4 Growing Single Best Solver Approach

The growing Single Best Solver (SBS) approach is a cheap alternative technique to the *Minimal hitting set*. It is a greedy approach where the portfolio starts from the best candidate and grows over time with iterations. Since it is cheap, we can use this technique as a companion piece to the *Minimal hitting set*. We can remove any of the configurations that are not selected by the growing single best solver approach (i.e. union the configurations coming from the growing single best solver and those previously coming from the minimal hitting set).

Using the competitiveness metric (A), growing SBS starts from the best configuration where the greatest amount of instances are covered (the best configuration being competitive). In each iteration, the procedure looks for the next best configuration for the remaining uncovered instances.

Using +25% as an example again, we can see how many instances are covered with this technique in fig. 9.9.

After 20 iterations, there are still instances without assigned configurations, which is undesirable.

The growing SBS approach acts differently in metric (B). With this metric, on each iteration the goal is to reduce the total run-time performance of the chosen

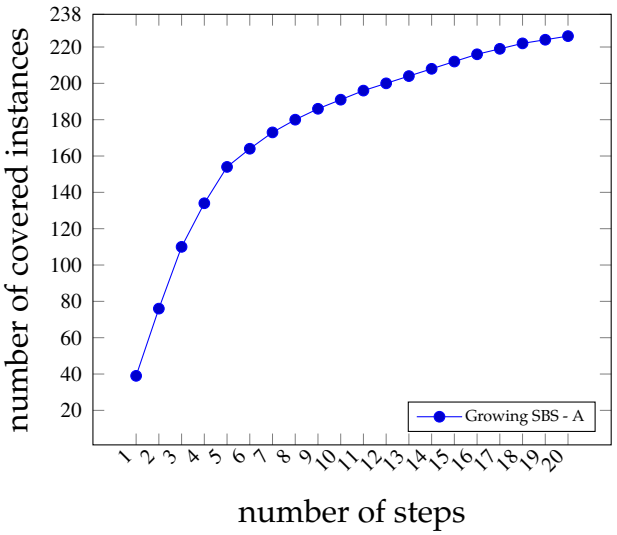


Figure 9.9: Visualisation of the growing SBS method using +25% competitiveness with metric A. The number of instances covered is shown at each step for the total 238 instances.

collective of configurations, starting from the best total run-time performing configuration.

Again with +25% competitiveness threshold, applying the growing SBS approach, we can reach the 200K seconds threshold on 8 iterations (95% of the virtual best solver performance). On 12 iterations we go below 190K seconds (99% of the virtual best solver performance) Figure 9.10.

9.3.3.5 Ranking Configurations

As a final step, we can create a ranking on the remaining configurations based on how many times they are competitive (A) or their run-time performance (B). The rankings of the configurations can be used for tie-breaking purposes later on in the process if required.

For our running +25% threshold example, the first five of these configurations using the competitiveness metric (A) can be seen in Table 9.3.

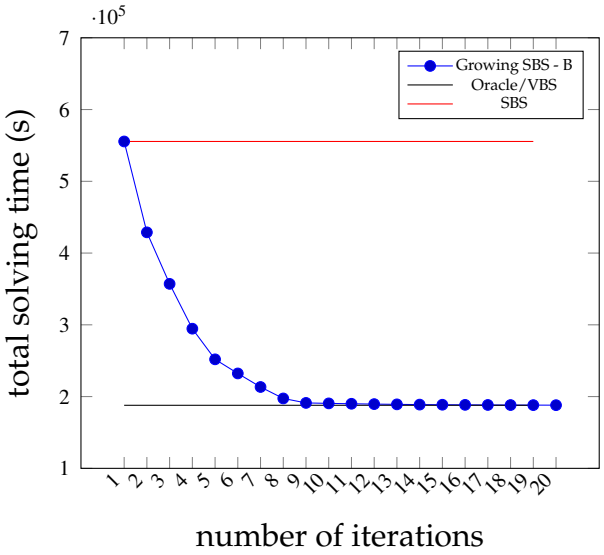


Figure 9.10: Visualisation of the growing SBS method using the +25% competitiveness with metric B. The total run time is shown at each iteration.

conf	count
SACBounds_limit_oo_nbc	99
GAC_oo_nbc	79
SACBounds_limit_oo_glucose	72
None_oo_nbc	67
SACBounds_limit_oo_nbc	60

Table 9.3: Five most competitive configurations on the +25% competitiveness with metric A in all instances.

9.3.4 Classification Prediction Models

To be able to train a classification system on the selected configurations we need to determine what features we should be extracting from the instances and would like to use on training.

9.3.4.1 Features

Firstly, we have determined 23 different mining-related features that can be cheaply calculated. The calculation of most of these features relies on using specialised mining algorithms in an unconstrained way with small timeouts to determine the number of the selected type of frequent itemsets.

The first three of these features are `items`, `nb_transactions`, and `size`.

They represent generic metadata of the mining instance that can be fetched without any significant calculations.

The remaining 20 features are generated by running 4 unconstrained itemset mining problems as pre-processing: standard, maximal, closed, and generator frequent itemsets. For each task, we use the Eclat tool on all instances with 5 different timeout thresholds, which are 10, 30, 60, 120 and 600 seconds. For each instance, the number of itemsets found in every given problem type and timeout has been taken as an instance feature.

The second feature set we consider is the features coming from the ESSENCE space. After a problem instance is written as an ESSENCE parameter file, CONJURE supports extracting simple statistics for each instance parameter, which can be used as instance features. Using this system, CONJURE can generate 90+ features for each mining problem instance. The whole feature set can be seen in appendix E. Some of these features can be duplicates of our defined mining features and some of them can be also meaningless in our context. By filtering duplicate and not useful features (such as $max-cost-isEven$ since our max cost is always fixed to the value 5) and with some pre-processing on the remaining features to eliminate infinite values, we arrive at 16 essence features for all problem classes. After all the pre-processing, all of these features are related to the side constraints (utility and cost values). This also results in having the same amount of features for all problem classes.

The final set of ESSENCE features can be seen in fig. 9.11.

While the first two features (i.e `min_utility_intValue` and `max_cost_intValue`) directly come from their ESSENCE variable counterparts, the rest comes from the lists of utility and cost values for the items. The features in this second category are statistical metrics calculated from both lists.

9.3.4.2 Classifier Metric

For the classification metric, we decided to use our competitiveness metric (A) as a binary classifier. By using binary classification, we aim to have faster training

- "min_utility_intValue",
- "utility_values_1_median",
- "utility_values_1_mean",
- "utility_values_1_stdDev",
- "utility_values_1_harmonicMean",
- "utility_values_1_geometricMean",
- "utility_values_1_skewness",
- "utility_values_1_kurtosis",
- "max_cost_intValue",
- "cost_values_1_median",
- "cost_values_1_mean",
- "cost_values_1_stdDev",
- "cost_values_1_harmonicMean",
- "cost_values_1_geometricMean",
- "cost_values_1_skewness",
- "cost_values_1_kurtosis".

Figure 9.11: Final set of ESSENCE features to use in the classification of the portfolio building.

times.

9.3.4.3 Training and Test sets

To avoid any bias in data and to eliminate the possibility of over-fitting, we have split our instances into separate training and test sets with a 3:1 ratio.

A code snippet example of this with SKLearn in Python can be seen in fig. F.1 at appendix F.

9.3.4.4 Classification with SKLearn - Random-Forest Classifier

As an initial training method, a Random-Forest classifier on SKLearn [PVG+11] has been used.

A code snippet example for the parameters of the classifier can be seen in fig. F.2 at appendix F.

Without using many parameters (with only setting the number of estimators to 100 and with a random seed), we can construct our Random-Forest classifier.

9.3.4.5 Classification with Auto-SKLearn

Auto-SKLearn is an automated machine learning toolkit that applies different machine learning techniques directly with an automated algorithm/classification

selection mechanism [FKE+15].

A code snippet example for the parameters of an Auto-SKLearn classifier with its options can be seen in fig. F.3 at appendix F.

Auto-SKLearn uses 17 different data pre-processing methods and 15 different classification methods (see Table 1 in [FKE+15]). By mixing and matching pre-processing and classification methods, Auto-SKLearn creates $17 * 15 = 255$ possible routes (called pipelines) and uses them as its own configuration space to determine well-performing pipelines. Auto-SKLearn also groups pipelines together into ensembles and then uses a weighted voting system on an ensemble to decide predictions. By default, Auto-SKLearn will create an ensemble of 5 classifications in a pipeline.

9.3.5 Automated Configuration Selection

After the classification training has been done, we can use the classification systems' outputs as predictors. In a test/unseen instance, we can ask our predictors to retrieve which configurations are predicted to be competitive in an instance.

In a scenario of multiple configurations claiming to be competitive, we can break the ties with the ranking we have established in section 9.3.3.5. We can then use the higher ranking configuration as the main candidate for that particular instance to run that configuration.

9.3.6 Results

Using our portfolio building method described above, we aim to build an efficient portfolio of configurations to perform better on total run-time.

As mentioned in section 9.3.1, we experiment on different competitiveness thresholds, namely +25%, +50%, +66%, +100%, and +150%. Before doing any processing, we split our instance space into train and test instances to only test our system on unseen instances. Therefore, all the configuration filtering occurs

after the train/test split. This means our configuration filtering techniques will only rely on the information on the training instances.

We can do the test/train split *multiple times* with different random seeds to remove any possible problems related to one particular split or random bias. We use 6 different train/set splits in each separate experiment.

Afterwards, we pre-process all instances and the configurations with the procedures described earlier in section 9.3.2 and section 9.3.3. For configuration filtering, we eliminate any configuration that does not appear either in any of the growing SBS or minimal set approaches using both competitiveness metrics (A) and (B).

As a result, we achieve k number of configurations, which is smaller than 30 for each competitiveness threshold. By reducing the number of configurations to k, we save precious computational time spend on training.

After the pre-processing of the configurations, we use *SKLearn* Random-Forest and *Auto-SKLearn* on the k configurations using the competitiveness data.

Now we have k predictors, which corresponds to k different configurations. To choose which configuration is the most suitable for one test instance, we can ask each configuration predictor and interpret its response as the availability to be a candidate for that particular instance. If a configuration is predicted to be competitive on the given instance, we place it in the pool of candidates for the instance.

For the default Auto-SKLearn classification training system, the results from the 6 different train/test splits and 2 random seeds on 5 different competitiveness thresholds (i.e. +25%, +50%, +66%, +100%, and +150%) can be seen in fig. 9.12.

These results show that +100% outperforms all the other competitiveness thresholds we tried. Additionally, it consistently beats the single solver's performance. From the results, we can speculate that lower threshold values such as +25% or +50% might be restricting the competitiveness range to be too small, causing very few configurations to be competitive. Thus, the classifier system is predicting configurations to be non-competitive more than competitive.

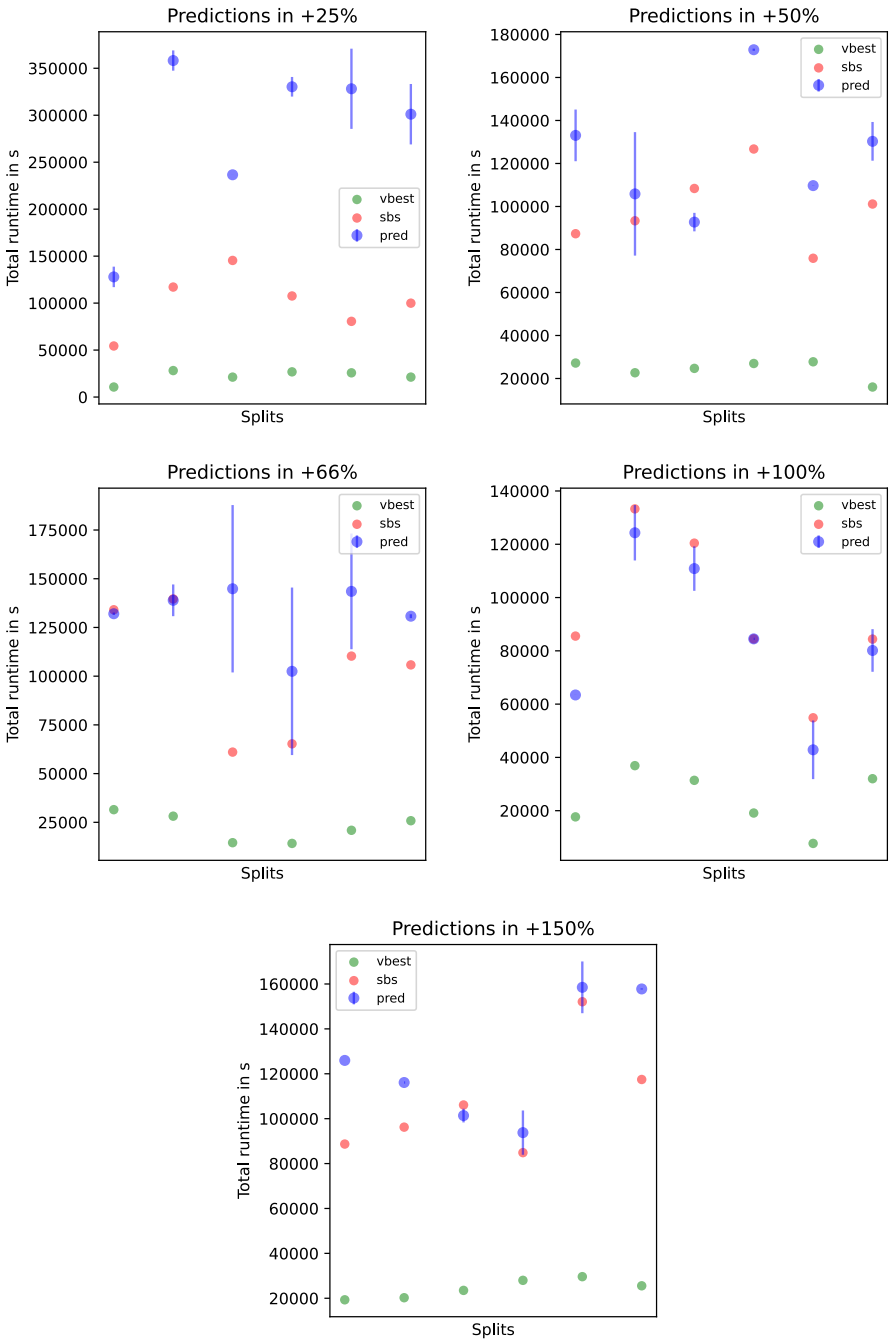


Figure 9.12: Prediction results of the in-house portfolio with different competitiveness threshold values. +100% performs the best amongst them while also beating the single best solver performance. Vbest indicates virtual best solver performance, while sbs represents single best solver performance. Pred is shorthand for the prediction system.

This is resulting in instances with no competitive configurations assigned. In these types of cases, the highest-ranking configuration would be chosen, which does

not perform well in the end.

For the higher +150% threshold, we can likewise speculate that the competitiveness range is too lenient and leads the classification system to predict more competitive values than non-competitive values. In these cases, the ranking among the candidates will break the ties. However, a higher ranking configuration might result in worse performance.

Even when the best predictor threshold is +100%, the results are still far off from the possible virtual-best performance of all configurations. This shows that the features we have extracted might not be sufficient to identify the potential of each configuration.

While the performance of the systems with different competitiveness thresholds can vary, the accuracy values of each classifier attached to a classification method are in the range of 60% to 90%. This indicates that the whole predictor system might not be as efficient as desired, even if the individual classifiers are highly accurate.

With promising results from the +100% competitiveness threshold, we would like to repeat the same experiment for only Random-Forest classifications to see if we can avoid the cost of using Auto-SKLearn. The results can be seen in fig. 9.13.

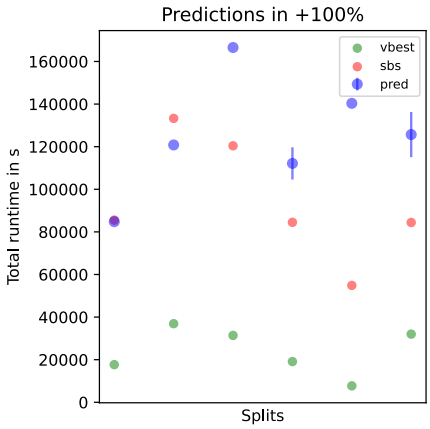


Figure 9.13: Prediction results of the in-house portfolio for single Random-Forest classifier on +100% competitiveness.

The results show that the default Random-Forest classifier performs worse than

the previous +100% Auto-SKLearn model. While the Auto-SKLearn results for +100% are consistently below the single best solver time on all splits, the default Random-Forest classifier performs on par in one split while performing worse in 4 different splits.

9.4 Feature Importance Analysis

In the previous section, we created a portfolio of configurations using different methodologies with certain variables attached to them. At the end of our portfolio building, we concluded that using Auto-SKLearn approach with the +100% threshold is the most efficient choice. In this section, we will analyse how the instance features impact the prediction models in the portfolio. The goal of this analysis is to determine which instance features contribute to a good prediction model in the portfolio. We also would like to examine the relationship between instance features to see how they contribute to each other.

To carry out the importance analysis, multiple approaches can be taken: 1) fANOVA [HLB14], 2) automated iterative feature selection approaches [FPHK94], and 3) our feature selection approach which aims to optimise portfolio run-time. By applying multiple analyses, we plan to compare and contrast different methods that give different insights to achieve a consistent and robust meta-analysis.

9.4.1 fANOVA

fANOVA is a feature analysis method that determines the importance of each feature in a classification system. fANOVA takes the feature set from a classification system and the output of that system as givens to create another machine learning system. Using this information, it calculates the importance of each feature and its interactions. The importance is defined as the contribution to the total variance of the performance.

fANOVA supports pairwise analysis on features as well. This can be used to extract feature pairs (or even triplets) that have a significant impact on performance

together. The pairwise (or triplet-wise) analysis works by looking at every possible pair (or trio) of features together to analyse their effects together. This method can be used to identify possible interactions between the feature set. However, looking into the pairwise interaction of the features comes with a cost. While the pair analysis can be done in n features with $\binom{n}{2}$ being relatively small, trio analysis $\binom{n}{3}$ can be rather large to check for every interaction.

We conducted our fANOVA experiments on our best performing +100% competitiveness threshold with an Auto-SKLearn classifier. With 2 random seeds and 6 splits on k configurations, we apply a fANOVA importance measurement analysis on above 250 different trained models for 45 features.

We have decided to include pair-wise interaction analysis even with the increased cost of computational time to get the potential benefit from it. However, the triplet analysis is significantly more expensive, as mentioned earlier. Thus, it is not included in our fANOVA experiments.

9.4.1.1 Results

We have retrieved 250 fANOVA analyses for each configuration. By averaging each single/pair importance value, we can statistically get the 5 most important features and 5 most important pairwise interactions.

The first 5 individual features with their average importance can be seen in table 9.4.

feature	average importance (%)
min_utility_intValue	4.77
s_t_10	3.84
g_t_1	3.73
s_t_1	3.71
max_cost_intValue	3.11
sum	**19.16**

Table 9.4: Five most important individual features determined by fANOVA and intersected over different classifications.

The first 5 feature-pairs with their average importance can be seen in table 9.5.

feature pair	average importance (%)
s_t_1 & s_t_10	0.27
c_t_1 & s_t_10	0.23
c_t_1 & s_t_3	0.20
c_t_1 & s_t_1	0.19
min_utility_intValue & s_t_1	0.18
sum	**1.07**

Table 9.5: Five most important feature pairs determined by fANOVA and intersected over different classifications.

Looking at these two analyses, we can see that some of the individual features can have an impact over 5% on average, while the pairwise interactions are less present in the data. The basic mining features that are calculated by solving an unconstrained mining problem for a single second such as `s_t_1` or `c_t_1` seem to have a high impact on both individual and pairwise importance.

9.4.2 Automated Feature selection

Alternative to the fANOVA importance analysis, we can use automated feature selection systems. Instead of generating an importance value, these methods work on the classifiers by sub-selecting from the feature set each time, trying to find the most important features. They operate by optimising the accuracy of the classifier at each step.

For automated feature selection, we can use the predetermined methods available in SKLearn. These methods are:

- Sequential feature selection (forwards or backwards),
- Recursive Feature Elimination,
- Recursive Feature Elimination with Cross Validation

Sequential feature selection operates by taking an estimator instance of a trained model and selectively removing or inserting features [3]. By selectively

[3] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SequentialFeatureSelector.html

inserting or removing, this feature selection system can operate on forwards or backwards. It operates by targeting a number of features n. The system tries to eventually arrive at this number of features in the final set. In forwards feature selection, the first n features that appear in the feature set are the most important ones. Conversely, on backwards feature selection, the last remaining n features are the most important ones.

The forwards feature selection is a cheaper procedure since it iterates n times while backwards feature selection does $total - n$ times (where $total$ is the total number of features), which can be more expensive depending on n. However, since the forwards feature selection model adds one feature each iteration, it can fail to capture important feature interactions. The backwards feature selection does not have this short-falling as it can keep important features together.

Recursive Feature Elimination (RFE) is a procedure similar to sequential feature selection's backwards method. Given a prediction model, the goal of RFE is to select features by recursively considering smaller sets of features. It works towards a predetermined n number of features at the end [4].

Recursive Feature Elimination with Cross-Validation (RFECV) is very similar to RFE. However, instead of taking the number of features n as a parameter, it can automatically arrive at the optimal number of features. It finds the optimal number by applying cross-validation. Using this cross-validation loop, it can determine where to stop [5].

On +100% competitiveness threshold, we can apply the 4 approaches from 3 different techniques to each of the configuration classifiers to get the most predominant features. While sequential feature selection and RFE take n as the number of features to select, RFECV can automatically determine this number, which varies from configuration to configuration.

[4] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html

[5] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html

9.4.2.1 Results

Similarly to the fANOVA approach in section 9.4.1, we can intersect all the possible results among configuration classifiers for each of the feature selection methods.

Firstly, if we focus on the number of important features determined by the RFECV method, throughout different configurations RFECV shows between 1 and 30 features to be important. If we eliminate any outliers with very large values and calculate the mean, we arrive at 5 as the average number of important features.

The 5 most important features coming from the intersected RFECV, forwards, backwards, and RFE results can be seen in the given order in table 9.6, table 9.7, table 9.8, and table 9.9. In the tables, the right columns indicate how many times the feature appears in the final feature set of any instance of a classifier.

feature	average appearance (%)
model	51.28
g_t_30	43.59
c_t_30	41.03
s_t_1	41.03
max_cost_intValue	38.46

Table 9.6: Five most important features by RFECV with their average appearance.

feature	average appearance (%)
utility_values_1_median	57.58
model	53.79
min_utility_intValue	47.73
max_cost_intValue	42.42
utility_values_1_mean	37.88

Table 9.7: Five most important features by forwards sequential feature selection with their average appearance.

Looking at 4 different feature selection models, we can see the *model* parameter has been found to be very important by these feature selection methods. For backwards selection, the mining features are the most predominant ones following *model*. Contrarily, the mining features do not seem to appear in the forwards selection model. We can also see the importance of the ESSENCE features of the

feature	average appearance (%)
model	61.65
s_t_5	34.58
s_t_30	23.32
s_t_3	23.31
s_t_10	22.55

Table 9.8: Five most important features by backwards sequential feature selection with their average appearance.

feature	average appearance (%)
model	58.93
min_utility_intValue	42.86
max_cost_intValue	32.14
utility_values_1_stdDev	25.01
c_t_30	23.21

Table 9.9: Five most important features by RFE with their average appearance.

side constraint variables (i.e. utility and cost values), which are predominantly represented in the feature selection models. In RFE and RFECV, we see mixed results with both mining features and ESSENCE features being represented at the same time. Dominant ESSENCE features are found to be related to min utility and max cost variables while some of the utility and cost values are also present.

9.4.3 Run-time based Feature Selection

While previous fANOVA (section 9.4.1) and automated feature selection (section 9.4.2) approaches give us the most significant features depending on the individual configuration classifiers, they operate on an ad-hoc level for our portfolio system. We can implement another feature selection system that is based on the total run-time performance (competitiveness metric (B) in section 9.3.1) of the portfolio instead.

To create a feature selection system that uses the run-time information instead of individual classifier accuracy, we can re-train each classifier in the portfolio system on a different sub-set of the whole feature set. Then we can measure the portfolio's performance on the sub-set of features with the total run-time of the

test set. For this experiment, we use our +100% Auto-SKLearn portfolio from our main analysis.

A basic incremental forwards feature insertion method takes the initial training classifier. Starting from no features, it tries to insert each feature one by one to identify which features are affecting the classifier's performance relative to the previous step. However, on the portfolio scale, to apply the incremental forwards feature insertion system, all the classifiers in the portfolio need to be re-trained with the currently tested feature added into their classifier. The feature that gives the maximum performance (i.e. minimum average running time) in the portfolio is kept. In the next step, the system tries to insert one of the remaining features.

The goal of this method is to use the portfolio's test run-time as the performance metric. Additionally, it also aims to identify the most beneficial features while also determining the point of diminishing return in terms of run-time performance. The point of diminishing return indicates a point where the number of features is enough to perform as close as possible to the full set.

As mentioned earlier (see section 9.4.2), an issue of using forwards feature analysis is that it cannot identify the pairwise importance of the features. If a pair of features interact with each other and impact performance, it is not possible to capture this by forwards feature insertion analysis. To overcome this we can also do a backwards version of this analysis on the removal of features.

The backwards feature removal operates in a reverse direction to the forwards feature insertion procedure. In each step, features are removed one by one while trying to remove the least impactful feature in terms of the run-time performance of the portfolio.

As indicated in the section 9.4.2, backwards feature removal can be quite expensive compared to the forwards version.

Similarly to the incremental equivalent, this procedure will also help us identify the core set of features that have a greater impact. It is possible to cross-validate these two analyses to identify interactive features that perform better together.

9.4.3.1 Implementation

The iterative procedure uses our auto-tuned baseline Auto-SKLearn model generation for the initial classification. Unlike traditional classification methods, the Auto-SKLearn model includes not only a single classification model, but an ensemble of classification pipelines. A classification pipeline includes one pre-processing and one estimator.

Each pipeline is given a weight to determine its significance for this particular training. Later in the prediction phase, a voting system is applied to determine the final prediction depending on individual predictions in the ensemble. This already tuned ensemble can be later stored to be reused on the iterative procedures.

To be able to apply iterative forwards or backwards procedures, we can store classifier models and retrain on a sub-feature set (i.e. altered input dimensions) when needed. Our early experiments show that some of the pre-processors and estimators in the Auto-SKLearn system have limited or no support for a change in dimensions on the input data (e.g. Multi-Layer Perceptron - MLP).

To overcome the issue of not being able to change input dimensions, we can replace the features which are not present in the sub-feature set with dummy values to trick to training system into making those features unusable while keeping the input dimensions the same.

Since our full feature set includes 45 features, this indicates a forwards or backwards iterative procedure is necessary to do 45 iterations with decreasing number of options. Thus, $45x46/2 = 1035$ additional retraining is required for every trained classifier model for each direction. Considering we have a large number of configurations and multiple train splits with random seeds, the amount of computation power required is significant.

To reduce the number of computations, a memoisation approach has been designed in the iterative system. Assuming we can run the experiments in any direction within a shared network, we can cache each sub-feature set experiment result for each configuration classifier. This will allow the cache to be available for other experiments with the same characteristics that use the reverse order.

For example, we assume 0, 2 and 8 have been chosen as the first three important features for a forwards procedure. On the next iteration, the system will try features 0..45, not including the ones that already exist, and will store all possible details. On a parallel process, if a backwards process wants to access 0, 2, 3, and 8, the retraining of this sub-feature-set will not be repeated since this information is already available in the system; the relevant results can be fetched directly.

In an ideal situation, if both directions identify sub-feature set order as exactly the opposite order, the number of experiments for retraining gets reduced to half (from 2070 to 1035). However, since pairwise feature importance impacts are also considered, this ideal case scenario is highly unlikely. Even though the time consumption will not be halved, the system should avoid repeating some experiments thanks to cache usage.

In addition to the performance improvements in general, for the forwards selection system, we can use an early stopping mechanism as well. We can use the average cross-validated number of features 5 from RFECV earlier and focus on the first 5 features directly. By doing so, the amount of retraining can be reduced to around 10%. A natural question would be how much of the performance is lost by stopping at 5 features only. We can analyse this in the following results.

9.4.3.2 Results

The experiments are repeated for each of the six different train/test splits, with the best performing Auto-SKLearn classifier on +100% competitiveness threshold. Similarly to the previous importance analysis (fANOVA and classifier level feature selection), the results from 6 runs are intersected to retrieve average appearance as feature importance. The intersection is done on the 5 most important features of each run depending on the order of the feature selection.

Before looking into which features are included in the sub-feature-set systems, we have a brief look at the performance of the system on each split. We see that a system with a limited 5 most important features can achieve $\geq 95\%$ of the performance of the total 45 feature set system.

For forwards feature selection, the occurrences of the first five features on each of the six splits can be seen in fig. 9.14.

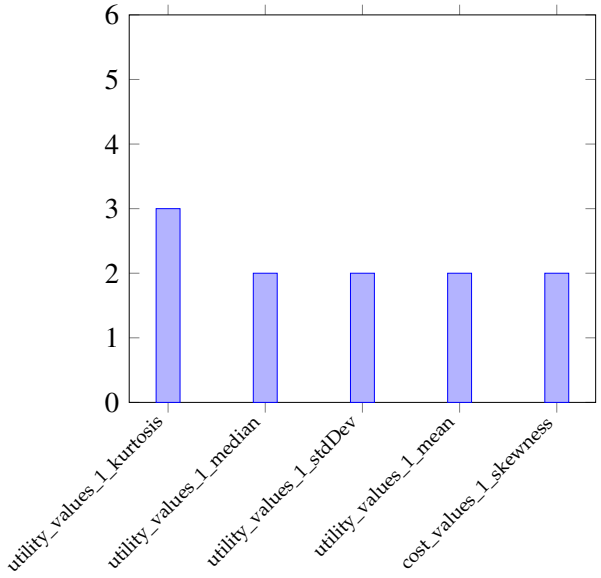


Figure 9.14: Occurrence statistics of the most important features of the 6 different forwards feature selection runs on the whole portfolio.

The results show that the side constraint information coming from ESSENCE features is more valued for the forwards selection system. This behaviour matches the results from classifier level forwards selection in table 9.7 and section 9.4.2, while our recent results show even more emphasis on side constraint features.

Respectively, if we do the same analysis going backwards by looking at the last 5 features as the key features only, we achieve the results in fig. 9.15.

The results indicate similar findings to the classifier level backwards selection results in table 9.7 and section 9.4.2. All of the features are directly related to the mining pre-processing. However, there is one difference between these two sets of results: the model parameter is not present in the most important features.

This may also indicate the importance of the pairwise interactions between mining features is even more present when we consider the system as a whole. They are very crucial to the performance of the whole portfolio, even more than the individual configuration classifiers.

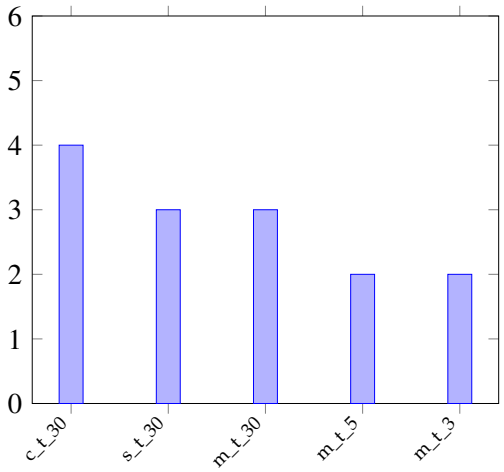


Figure 9.15: Occurrence statistics of the most important features of the 6 different backwards feature selection runs on the whole portfolio.

9.4.4 Summary

Throughout this section, we have defined and used 3 different feature analysis systems. Combined results indicate that some of the ESSENCE level features such as `min_utility` or `max_utility`, and mining features in general, are important in a configuration selection and portfolio system.

With multiple forwards/backwards feature selection analyses, we also examine that mining features in general indicate pair-wise interaction among each other, thus affecting the performance of the system according to the presence of said features.

We also see some discrepancies amongst different methods such as the *model* feature being very important on sequential feature selection systems, while fANOVA and our run-time based system does not find the model parameter crucial for the prediction system. From this point, we can speculate that the *model* parameter has a slight impact on the performance of the system since it is identified by one feature importance analysis. However, this impact is not significant enough to be visible in the other analyses as it is outranked by other important features.

10

CHAPTER TEN

CONCLUSION AND FUTURE WORK

This chapter gives a conclusion to the thesis and adds further discussion on possible future work.

10.1 Conclusion

The motivation of this thesis was to tackle pattern mining problems with arbitrary side constraints. While adding specific side constraints is common in pattern mining algorithms and applications, generalising these side constraints to any arbitrary constraints is very much challenging. Thus, constraint programming approaches are a very suitable candidate for such a task due to their generalised behaviour. The most important aspect of constraint programming that helps achieve this generalised behaviour is that modelling and solving are distinctly separated. This allows making improvements on either end while abstracting over the counterpart without requiring too much change in the flow of the system. Any constraint that can be added to the model can be independently tackled by the solving scheme using specific propagators.

Another motivation of this thesis was to use high-level structures to represent pattern mining problems. Pattern mining problems by their nature include higher-level structures. Approaching these problems using high-level constraint

specification languages such as ESSENCE will bring much more flexibility in modelling while enabling more possible optimisations for the underlying solving systems.

The main contribution of this thesis is to create a generalised framework in CP to solve pattern mining problems. Starting from our early ad-hoc iterative approach, we have experimented with different ways to represent these pattern mining problems in the CP space. By using dominance programming as a baseline, we have created Constraint Dominance Programming (CDP) and integrated it into the high-level ESSENCE pipeline. By representing pattern mining problems in high-level ESSENCE with CDP's dominance relations, we have generated very compact models and focused on solving these models optimally. Our empirical evaluation on CDP with a comparison to MININGZINC indicates that CDP is already capable of being highly efficient and competitive. Additionally, due to its generic nature, CDP is suited to represent optimisation problems natively as well.

Furthermore, we have improved CDP by exploiting another structure commonly occurring in pattern mining problems: incomparability. With this new structure, we have extended CDP to CDP+I, also integrating it into the ESSENCE pipeline. The newly added incomparability condition grants tighter control to the modeller while bringing additional solving optimisations. The empirical evaluation of CDP+I proves these improvements make CDP+I highly competitive. With CDP+I, we demonstrate the ability to represent multi-objective optimisation problems as well.

While CDP+I is highly efficient with the new incomparability condition, including this information to the high-level model is initially left to the end-user. As an improvement, we created the DIG system to systematically generate the optimal incomparability condition for a given dominance relation. The goal of this system is to bring high-level abstractions of the ESSENCE pipeline to the CDP+I framework.

With the creation of CDP+I, we aimed to deliver a generic modelling framework for pattern mining problems. The initial solving mechanism of CDP+I

relies directly on how consecutive/iterative solving is handled in a constraint programming pipeline. To improve this, we proposed the Solver Interaction Interface (SII), a new method to further optimise any iterative solving procedure, including CDP+I. We implemented SII into the ESSENCE pipeline directly for SAT and SMT solvers. With this new interaction system, we significantly improved the efficiency of CDP+I on the ESSENCE pipeline.

While we use high-level ESSENCE structures to model pattern mining problems in a CDP+I framework with improved specific solving methodologies, we initially used a single promising representation of a high-level model generated by CONJURE. Later, we identified any modelling and solving choices as possible configurations and focused on creating a configuration selection system using classification methods. To that end, we used two systems: SMAC and our configuration selection system. We used our configuration selection system to create a portfolio of configurations for further improvements in solving pattern mining problems. The results showed that our portfolio is consistently better than the best performing configuration.

10.2 Future Work

We conclude this thesis with a brief discussion on future work. Future work includes: 1) the application of our system to a wider range of problem classes, both in data mining and beyond, 2) different and more efficient model encodings and better solving mechanisms, and 3) other possible model refinements and a more verbose configuration selection.

10.2.1 Other Problem Classes

In this thesis, we mainly focused on some of the itemset mining problems as a subset of pattern mining. Other itemset mining tasks such as association rule mining [AIS93] can be modelled into CDP+I as well. During this research, there were attempts to model the association rule mining problem into CDP+I. The

modelling is done by separating the association rule property into: 1) closedness and, 2) generator properties. Unfortunately, this attempt to model and solve this problem class failed at the model expansion phase where the ESSENCE PRIME level representation is parsed into an internal structure. To be able to model this problem class properly, the internal representation of certain structures in SAVILE ROW such as complex nested sets should be handled differently and optimised further to be able to make this model work.

Additionally, we can consider other types of pattern mining problems where patterns can be more complex compared to single items such as sequences or graphs. Sequential pattern mining [FVLK+17], for instance, is a very good candidate problem where the CDP+I framework can be applied without too much additional work. The high-level type support of ESSENCE would allow these problems to be represented in a constraint programming specification language natively. However, solving these problems can be challenging with additional complexity brought to the solver scheme. During this research, some attempts are made to model sequential pattern mining problems. However, since the sequential pattern mining problem requires deeper nested set representations compared to frequent itemset mining, the expansion of the model created bottlenecks as well. Following the same reasoning as association rule mining, the internal representation of some structures can be optimised further to facilitate modelling this problem.

Other mining problems can be investigated to be used in the CP space, such as text mining and clustering. Constrained-based clustering [THLN01] would be a good fit for constraint programming approaches that can bring additional flexibility. While a direct SAT-based approach already exists [DRS10], CP techniques can bring abstraction to the modelling and solving.

While our main focus was on pattern mining problems, we demonstrated the possibility of using CDP/CDP+I on single/multi-objective optimisation problems. In [KADM20], with the usage of the solver interactive interface system, we demonstrated that a CP approach targeting SAT can be very competitive on MRCPSP

single optimisation problems. Additionally, many other optimisation problems such as knapsack or social golfers problems are deemed competitive using CP approaches [RVBW06]. We believe multi-objective optimisation problems will be a natural next application area for CDP+I with SII.

10.2.2 Model Encoding and Solving

In this thesis, we used the ESSENCE constraint specification language for modelling pattern mining problems. To solve these problems, we mainly benefited from SAT solvers. In various places, we also used SMT solvers and the CP solver MINION. In chapter 7, we presented the solver interactive interface (SII) on SAT and SMT solvers, where our experiments showed great improvements for our problem classes. A natural next step would be taking SII into CP solvers such as MINION and CHUFFED. By keeping the solver state, we believe we can improve the performance of these solvers on iterative modelling/solving situations drastically. Using CHUFFED, the information learnt from lazy clause generation techniques [OSC07] can bring additional benefit to interactive solver systems to keep even more information to reduce the search even further.

We believe CP solvers, such as lazy clause generation solver CHUFFED, would be a great candidate to work alongside the CDP+I framework. While pattern mining problems have a significant amount of constraints that can be represented with Boolean-like constraints (such as subset operation) that are naturally represented in SAT, they also require some arithmetic constraints which require indirect encoding. Additionally, data mining problems contain a big amount of information that needs to be encoded into the SAT space. This can be quite a bottleneck for the SAT solving process. Moreover, using CP approaches enable the usage of more efficient encodings for large data structures such as BitVectors or Sparse BitSets [dSMSSL13]. With more compact encodings and by targeting CP solvers, it should be feasible to tackle bigger datasets and more complex problem classes previously deemed challenging.

For the CP backend, our framework does not particularly benefit from

specialised propagators for itemset mining problems where such propagators exist for CFIM and GFIM [LLL+16, SAG17, BBL19b]. Including these propagators in our framework and/or creating new propagators for the pattern mining-related constraints can improve the performance of our work even further.

It is possible to improve SAT encodings as well. For general CNF formulas, finding an equivalent CNF formula that is minimal in the number of literals is NP-hard [Uma01]. However, there are cheaper alternative methods of reducing the size of CNF formulas [EMG08]. Specific techniques to find shorter and/or more efficient CNF formulas such as decomposition-based encoding [JSS15] can be applied to improve the efficiency of the general framework.

It is entirely possible to look for alternatives instead of using traditional depth-first search solving. Any problem class with a large space that cannot be solved using exhaustive search methods can be investigated using the heuristic search or local search solvers [HM09]. The power of using heuristic and local search could remove the size barrier where the traditional exhaustive search methods seems to struggle. We believe targeting local/heuristic search solvers can increase the applicability of constraint programming to any data mining problem. Furthermore, since the local search solver ATHANOR [ADJ+19] directly operates on ESSENCE, we believe some additional benefits of directly working on ESSENCE primitives can be gained while any possible bottlenecks of low-level translation can be avoided.

Another future work includes full integration of the DIG incomparability generation to the ESSENCE pipeline by adding it to CONJURE. Currently, it is possible to systematically infer the incomparability condition from the dominance relation in ESSENCE. However, the CDP+I ESSENCE model is generated afterwards without automation.

10.2.3 Model Refinements and Configuration Selection

In this thesis, we have focused on a subset of the refinement options available in CONJURE, namely explicit and occurrence representations for givens and decision variables. With this, we reduced the number of possible configurations we

evaluated. A more complete variety of possible refinements can be used for more configurations: this can be incorporated into the portfolio building system described in chapter 9.

For automated configuration selection, we used SMAC and our custom portfolio building system. A natural next step to try out other portfolio building and automated configuration selection systems such as irace [LIDLC+16] or Hydra [XHLB10].

In our portfolio building mechanism (section 9.3), we used Auto-SKLearn, an automated machine learning toolkit, as a direct replacement for a single classification method. With the usage of this tool, we abstracted our system to automatically choose sensible classification predictors with good hyperparameters. While the accuracy values of the predictors are high enough, we believe there is an opportunity to further investigate to choose classification methods with better hyperparameters. This could possibly increase the performance of the portfolio of configurations further.

Again in section 9.3, after building the portfolio of configurations, we used a ranking system determined from the configuration selection system to break any ties where multiple configurations are found to be competitive. A more complex system to replace the ranking system can be investigated. For instance, a vote-based ensemble [KR14] similar to what is available in Auto-SKLearn can be a good candidate. This could improve the performance of the portfolio drastically by bringing the run-time performance closer to the virtual-best solver results.

REFERENCES

[ABB+99] Martin Anthony, Peter L Bartlett, Peter L Bartlett, et al., *Neural network learning: Theoretical foundations*, vol. 9, cambridge university press Cambridge, 1999.

[ADJ+19] Saad Attieh, Nguyen Dang, Christopher Jefferson, Ian Miguel, and Peter Nightingale, *Athanor: high-level local search over abstract constraint specifications in essence*, Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19), International Joint Conferences on Artificial Intelligence, 2019.

[ADL06] Belarmino Adenso-Diaz and Manuel Laguna, *Fine-tuning of algorithms using fractional experimental designs and local search*, Operations research **54** (2006), no. 1, 99–114.

[AGJ+14] Ozgur Akgun, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale, *Breaking conditional symmetry in automated constraint modelling with conjure.*, ECAI, 2014, pp. 3–8.

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami, *Mining association rules between sets of items in large databases*, Proceedings of the 1993 ACM SIGMOD international conference on Management of data, 1993, pp. 207–216.

[Akg14] Özgür Akgün, *Extensible automated constraint modelling via refinement of abstract problem specifications*, Ph.D. thesis, University of St Andrews, 2014.

[ALS13] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon, *Improving glucose for incremental sat solving with assumptions: Application to mus extraction*, International conference on theory and applications of satisfiability testing, Springer, 2013, pp. 309–317.

[AMJ+11] Ozgur Akgun, Ian Miguel, Chris Jefferson, Alan M Frisch, and Brahim Hnich, *Extensible automated constraint modelling*, Twenty-Fifth AAAI Conference on Artificial Intelligence, 2011.

[AS+94] Rakesh Agrawal, Ramakrishnan Srikant, et al., *Fast algorithms for mining association rules*, 20th int. conf. very large data bases, VLDB, vol. 1215, 1994, pp. 487–499.

[AS09] Gilles Audemard and Laurent Simon, *Predicting learnt clauses quality in modern sat solvers*, Twenty-first International Joint Conference on Artificial Intelligence, 2009.

[AS18] _____, *On the glucose sat solver*, International Journal on Artificial Intelligence Tools **27** (2018), no. 01, 1840001.

[ASY+19] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama, *Optuna: A next-generation hyperparameter optimization framework*, Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2019.

[BAG00] Roberto J Bayardo, Rakesh Agrawal, and Dimitrios Gunopulos, *Constraint-based rule mining in large, dense databases*, Data mining and knowledge discovery **4** (2000), no. 2, 217–240.

[BBJ+02] Celine Becquet, Sylvain Blachon, Baptiste Jeudy, Jean-Francois Boulicaut, and Olivier Gandrillon, *Strong-association-rule mining for large-scale gene-expression data analysis: a case study on human sage data*, Genome Biology **3** (2002), no. 12, 1–16.

[BBL19a] Mohamed-Bachir Belaid, Christian Bessiere, and Nadjib Lazaar, *Constraint programming for association rules*, Proc. of the 2019 SIAM International Conference on Data Mining, SIAM, 2019, pp. 127–135.

[BBL19b] _____, *Constraint programming for mining borders of frequent itemsets*, IJCAI: International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence Organization, 2019, pp. 1064–1070.

[BBR00] Jean-Francois Boulicaut, Artur Bykowski, and Christophe Rigotti, *Approximation of frequency queries by means of free-sets*, European Conference on Principles of Data Mining and Knowledge Discovery, Springer, 2000, pp. 75–85.

[BCDL11] Christian Bessiere, Stéphane Cardon, Romuald Debruyne, and Christophe Lecoutre, *Efficient algorithms for singleton arc consistency*, Constraints **16** (2011), no. 1, 25–53.

[BDRK+16] Christian Bessiere, Luc De Raedt, Lars Kotthoff, Siegfried Nijssen, Barry O'Sullivan, and Dino Pedreschi, *Data mining and constraint programming*, Springer, 2016.

[BG08] Irad Ben-Gal, *Bayesian networks*, Encyclopedia of statistics in quality and reliability **1** (2008).

[BGL+09] Francesco Bonchi, Fosca Giannotti, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Roberto Trasarti, *A constraint-based querying system for exploratory pattern discovery*, Information Systems **34** (2009), no. 1, 3–27.

[BGMP03] Francesco Bonchi, Fosca Giannotti, Alessio Mazzanti, and Dino Pedreschi, *Examiner: Optimized level-wise frequent pattern mining with monotone constraints*, International Conference on Data Mining, IEEE, 2003, pp. 11–18.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren, *Handbook of satisfiability*, vol. 185, IOS press, 2009.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, *Conflict-driven clause learning sat solvers*, Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications (2009), 131–153.

[BJ01] J-F Boulicaut and Baptiste Jeudy, *Mining free itemsets under constraints*, International Database Engineering and Applications Symposium, IEEE, 2001, pp. 322–329.

[BJS97] Roberto J Bayardo Jr and Robert Schrag, *Using csp look-back techniques to solve real-world sat instances*, Aaai/iaai, Providence, RI, 1997, pp. 203–208.

[BL04] Francesco Bonchi and Claudio Lucchese, *On closed constrained frequent pattern mining*, Fourth IEEE International Conference on Data Mining (ICDM'04), IEEE, 2004, pp. 35–42.

[BL07] _____, *Extending the state-of-the-art of constraint-based pattern discovery*, Data & Knowledge Engineering **60** (2007), no. 2, 377–399.

[BMFP15] Gustav Björdal, Jean-Noël Monette, Pierre Flener, and Justin Pearson, *A constraint-based local search backend for minizinc*, Constraints **20** (2015), no. 3, 325–345.

[Bor03] Christian Borgelt, *Efficient implementations of apriori and eclat*, FIMI'03: Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations, 2003, p. 90.

[BR97] Christian Bessiere and Jean-Charles Régin, *Arc consistency for general constraint networks: preliminary results*.

[BSP+02] Mauro Birattari, Thomas Stützle, Luis Paquete, Klaus Varrentrapp, et al., *A racing algorithm for configuring metaheuristics.*, Gecco, vol. 2, 2002.

[BYBS10] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle, *F-race and iterated f-race: An overview*, Experimental methods for the analysis of optimization algorithms (2010), 311–336.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani, *The MathSAT5 SMT Solver*, Proceedings of TACAS (Nir Piterman and Scott Smolka, eds.), LNCS, vol. 7795, Springer, 2013.

[Chu16] Geoffrey Chu, *Chuffed, a lazy clause generation solver*, 2016, Available from `https://github.com/chuffed/chuffed`.

[CS] Geoffrey Chu and Peter J Stuckey, *Chuffed solver description, 2014*.

[CYHH07] Hong Cheng, Xifeng Yan, Jiawei Han, and Chih-Wei Hsu, *Discriminative frequent pattern analysis for effective classification*, 2007 IEEE 23rd International Conference on Data Engineering, IEEE, 2007, pp. 716–725.

[CYS03] Raymond Chan, Qiang Yang, and Yi-Dong Shen, *Mining high utility itemsets*, Third IEEE international conference on data mining, IEEE, 2003, pp. 19–26.

[DAEN20] Ewan Davidson, Özgür Akgün, Joan Espasa, and Peter Nightingale, *Effective encodings of constraint programming models to smt*, International Conference on Principles and Practice of Constraint Programming, Springer, 2020, pp. 143–159.

[DB97] Romuald Debruyne and Christian Bessiere, *Some practicable filtering techniques for the constraint satisfaction problem*, In Proceedings of IJCAI'97, Citeseer, 1997.

[DLL62] Martin Davis, George Logemann, and Donald Loveland, *A machine program for theorem-proving*, Communications of the ACM **5** (1962), no. 7, 394–397.

[DMB08] Leonardo De Moura and Nikolaj Bjørner, *Z3: An efficient smt solver*, International conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.

[DRGN08] Luc De Raedt, Tias Guns, and Siegfried Nijssen, *Constraint programming for itemset mining*, SIGKDD international conference on Knowledge discovery and data mining, ACM, 2008, pp. 204–212.

[DRS10] Ian Davidson, SS Ravi, and Leonid Shamis, *A sat-based framework for efficient constrained clustering*, Proceedings of the 2010 SIAM international conference on data mining, SIAM, 2010, pp. 94–105.

[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.

[Dut14] Bruno Dutertre, *Yices 2.2*, International Conference on Computer Aided Verification, Springer, 2014, pp. 737–744.

[Edd04] Sean R Eddy, *What is a hidden markov model?*, Nature biotechnology **22** (2004), no. 10, 1315–1316.

[EMG08] Thomas Eiter, Kazuhisa Makino, and Georg Gottlob, *Computational aspects of monotone dualization: A brief survey*, Discrete Applied Mathematics **156** (2008), no. 11, 2035–2049.

[FGJ+05] Alan M Frisch, Matthew Grum, Chris Jefferson, Bernadette M Hernández, and Ian Miguel, *The essence of essence*, Modelling and Reformulating Constraint Satisfaction Problems **73** (2005).

[FH20] Armin Biere Katalin Fazekas Mathias Fleury and Maximilian Heisinger, *Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020*, SAT COMPETITION 2020 (2020), 50.

[FHJ+08] Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel, *Essence: A constraint language for specifying combinatorial problems*, Constraints **13** (2008), no. 3, 268–306.

[FJHM05] Alan M Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel, *The rules of constraint modelling*, IJCAI, 2005, pp. 109–116.

[FKE+15] Matthias Feurer, Aaron Klein, Jost Eggensperger, Katharina Springenberg, Manuel Blum, and Frank Hutter, *Efficient and robust automated machine learning*, Advances in Neural Information Processing Systems 28 (2015), 2015, pp. 2962–2970.

[FPHK94] Francesc J Ferri, Pavel Pudil, Mohamad Hatef, and Josef Kittler, *Comparative study of techniques for large-scale feature selection*, Machine Intelligence and Pattern Recognition, vol. 16, Elsevier, 1994, pp. 403–413.

[Fri37] Milton Friedman, *The use of ranks to avoid the assumption of normality implicit in the analysis of variance*, Journal of the american statistical association **32** (1937), no. 200, 675–701.

[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.

[GDN+17] Tias Guns, Anton Dries, Siegfried Nijssen, Guido Tack, and Luc De Raedt, *Miningzinc: A declarative framework for constraint-based mining*, Artificial Intelligence **244** (2017), 6–29.

[GDT+13] Tias Guns, Anton Dries, Guido Tack, Siegfried Nijssen, and Luc De Raedt, *Miningzinc: A modeling language for constraint-based mining*, Twenty-Third International Joint Conference on Artificial Intelligence, 2013.

[GS99] Ian P Gent and Barbara Smith, *Symmetry breaking during search in constraint programming*, Citeseer, 1999.

[GST18] Tias Guns, Peter J Stuckey, and Guido Tack, *Solution dominance over constraint satisfaction problems*, arXiv:1812.09207 (2018).

[Gun15] Tias Guns, *Declarative pattern mining using constraint programming*, Constraints **20** (2015), no. 4, 492–493.

[HC99] John D Holt and Soon M Chung, *Efficient mining of association rules in text databases*, Proceedings of the eighth international conference on Information and knowledge management, 1999, pp. 234–242.

[HDO+98] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf, *Support vector machines*, IEEE Intelligent Systems and their applications **13** (1998), no. 4, 18–28.

[HHLB10] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown, *Automated configuration of mixed integer programming solvers*, International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, Springer, 2010, pp. 186–202.

[HHLB11] ———, *Sequential model-based optimization for general algorithm configuration*, International conference on learning and intelligent optimization, Springer, 2011, pp. 507–523.

[HHLB12] ———, *Parallel algorithm configuration*, International Conference on Learning and Intelligent Optimization, Springer, 2012, pp. 55–70.

[HHS07] Frank Hutter, Holger H Hoos, and Thomas Stützle, *Automatic algorithm configuration based on local search*, Aaai, vol. 7, 2007, pp. 1152–1157.

[HLB14] Holger Hoos and Kevin Leyton-Brown, *An efficient approach for assessing hyperparameter importance*, International conference on machine learning, 2014, pp. 754–762.

[HM09] Pascal Van Hentenryck and Laurent Michel, *Constraint-based local search*, The MIT press, 2009.

[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.

[HS04] Holger H Hoos and Thomas Stützle, *Stochastic local search: Foundations and applications*, Elsevier, 2004.

[IM96] Tomasz Imielinski and Heikki Mannila, *A database perspective on knowledge discovery*, Communications of the ACM **39** (1996), no. 11, 58–64.

[JLBRS12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon, *The international sat solver competitions*, Ai Magazine **33** (2012), no. 1, 89–92.

[JMD+18] Said Jabbour, Fatima Ezzahra Mana, Imen Ouled Dlala, Badran Raddaoui, and Lakhdar Sais, *On maximal frequent itemsets mining with constraints*, International Conference on Principles and Practice of Constraint Programming, Springer, 2018, pp. 554–569.

[JSS15] Said Jabbour, Lakhdar Sais, and Yakoub Salhi, *Decomposition based sat encodings for itemset mining problems*, Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2015, pp. 662–674.

[JSS17] _____, *Mining top-k motifs with a sat-based framework*, Artificial Intelligence **244** (2017), 30–47.

[KADM20] Gökberk Koçak, Özgür Akgün, Nguyen Dang, and Ian Miguel, *Efficient incremental modelling and solving*, The 19th workshop on Constraint Modelling and Reformulation (ModRef 2020), 2020.

[KAGM19] Gökberk Koçak, Özgür Akgün, Tias Guns, and Ian Miguel, *Towards improving solution dominance with incomparability conditions: A case-study using generator itemset mining*, The 18th workshop on Constraint Modelling and Reformulation (ModRef 2019), 2019.

[KAGM20a] _____, *Exploiting incomparability in solution dominance: Improving general purpose constraint-based mining*, ECAI 2020-24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29-September 8, 2020-Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020), vol. 325, IOS Press, 2020, pp. 331–338.

[KAGM20b] Gökberk Koçak, Özgür Akgün, Tias Guns, and Ian Miguel, *Experiments for ECAI2020 CDP paper. DOI: 10.5281/zenodo.3675340*, February 2020.

[KAMN18a] Gökberk Koçak, Özgür Akgün, Ian Miguel, and Peter Nightingale, *Closed frequent itemset mining with arbitrary side constraints*, 2018 IEEE International Conference on Data Mining Workshops (ICDMW), IEEE, 2018, pp. 1224–1232.

[KAMN18b] Gökberk Koçak, Ozgür Akgün, Ian Miguel, and Peter Nightingale, *Maximal frequent itemset mining with non-monotonic side constraints*, International Conference on Principles and Practice of Constraint Programming Doctoral Program Proceedings, 2018.

[Koç22] Gökberk Koçak, *gokberkkocak/phd_experiments: v0.1. DOI: 10.5281/zenodo.5931360*, January 2022.

[KR14] Ludmila I Kuncheva and Juan J Rodríguez, *A weighted voting framework for classifiers ensembles*, Knowledge and Information Systems **38** (2014), no. 2, 259–275.

[Kry98] Marzena Kryszkiewicz, *Representative association rules and minimum condition maximum consequence association rules*, European Symposium on Principles of DM and KD, Springer, 1998, pp. 361–369.

[KS97] Rainer Kolisch and Arno Sprecher, *Psplib-a project scheduling problem library: Or software-orsep operations research software exchange program*, European journal of operational research **96** (1997), no. 1, 205–216.

[LIDLC+16] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle, *The irace package: Iterated racing for automatic algorithm configuration*, Operations Research Perspectives **3** (2016), 43–58.

[LLL+16] Nadjib Lazaar, Yahia Lebbah, Samir Loudni, Mehdi Maamar, Valentin Lemière, Christian Bessière, and Patrice Boizumault, *A global constraint for closed frequent pattern mining*, International Conference on Principles and Practice of Constraint Programming, Springer, 2016, pp. 333–349.

[LM09] Chu Min Li and Felip Manya, *Maxsat, hard and soft constraints*, Handbook of satisfiability, IOS Press, 2009, pp. 613–631.

[LRA10] Florian Lemmerich, Mathias Rohlfs, and Martin Atzmueller, *Fast discovery of relevant subgroup patterns*, Twenty-Third International FLAIRS Conference, 2010.

[LRSV18] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím, *Ibm ilog cp optimizer for scheduling*, Constraints **23** (2018), no. 2, 210–250.

[Man00] Heikki Mannila, *Theoretical frameworks for data mining*, ACM SIGKDD Explorations Newsletter **1** (2000), no. 2, 30–32.

[MJPL92] Steven Minton, Mark D Johnston, Andrew B Philips, and Philip Laird, *Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems*, Artificial intelligence **58** (1992), no. 1-3, 161–205.

[MML14] Ruben Martins, Vasco Manquinho, and Inês Lynce, *Open-wbo: A modular maxsat solver*, International Conference on Theory and Applications of Satisfiability Testing, Springer, 2014, pp. 438–445.

[Moc12] Jonas Mockus, *Bayesian approach to global optimization: theory and applications*, vol. 37, Springer Science & Business Media, 2012.

[MPC98] Rosa Meo, Giuseppe Psaila, and Stefano Ceri, *An extension to sql for mining association rules*, Data mining and knowledge discovery **2** (1998), no. 2, 195–224.

[MSLM09] Joao Marques-Silva, Inês Lynce, and Sharad Malik, *Conflict-driven clause learning sat solvers*, Handbook of satisfiability, ios Press, 2009, pp. 131–153.

[MT97] Masao Mori and Ching Chih Tseng, *A genetic algorithm for multi-mode resource constrained project scheduling problem*, European Journal of Operational Research **100** (1997), no. 1, 134–141.

[NAG+14] Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, and Ian Miguel, *Automatically improving constraint models in savile row through associative-commutative common subexpression elimination*, International Conference on Principles and Practice of Constraint Programming, Springer, 2014, pp. 590–605.

[NAG+17] Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen, *Automatically improving constraint models in savile row*, Artificial Intelligence **251** (2017), 35–61.

[NDGN13] Benjamin Negrevergne, Anton Dries, Tias Guns, and Siegfried Nijssen, *Dominance programming for itemset mining*, 2013 IEEE 13th International Conference on Data Mining, IEEE, 2013, pp. 557–566.

[Nem62] Peter Nemenyi, *Distribution-free multiple comparisons*, Biometrics, vol. 18, International Biometric Soc 1441 I ST, NW, SUITE 700, WASHINGTON, DC 20005-2210, 1962, p. 263.

[NLHP98] Raymond T Ng, Laks VS Lakshmanan, Jiawei Han, and Alex Pang, *Exploratory mining and pruning optimizations of constrained associations rules*, ACM Sigmod Record **27** (1998), no. 2, 13–24.

[NR16] Peter Nightingale and Andrea Rendl, *Essence' description*, arXiv preprint arXiv:1601.02865 (2016).

[NSB+07] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack, *Minizinc: Towards a standard cp modelling language*, International Conference on Principles and Practice of Constraint Programming, Springer, 2007, pp. 529–543.

[NSM15] Peter Nightingale, Patrick Spracklen, and Ian Miguel, *Automatically improving sat encoding of constraint problems through common subexpression elimination in savile row*, International Conference on Principles and Practice of Constraint Programming, Springer, 2015, pp. 330–340.

[OSC07] Olga Ohrimenko, Peter J Stuckey, and Michael Codish, *Propagation= lazy clause generation*, International Conference on Principles and Practice of Constraint Programming, Springer, 2007, pp. 544–558.

[PBTL99] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal, *Discovering frequent closed itemsets for association rules*, International Conference on Database Theory, Springer, 1999, pp. 398–416.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Scikit-learn: Machine learning in Python*, Journal of Machine Learning Research **12** (2011), 2825–2830.

[RN02] Stuart Russell and Peter Norvig, *Artificial intelligence: a modern approach*.

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh, *Handbook of constraint programming*, Elsevier, 2006.

[SAG17] Pierre Schaus, John OR Aoga, and Tias Guns, *Coversize: A global constraint for frequency-based itemset mining*, International Conference on Principles and Practice of Constraint Programming, Springer, 2017, pp. 529–546.

[SE05] Niklas Sorensson and Niklas Een, *Minisat v1. 13-a sat solver with conflict-clause minimization*, SAT **2005** (2005), no. 53, 1–2.

[SLT06] Christian Schulte, Mikael Lagerkvist, and Guido Tack, *Gecode*, Software download and online material at the website: http://www. gecode. org (2006), 11–13.

[SMTdlB16] Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda, *Learning from learning solvers*, International conference on principles and practice of constraint programming, Springer, 2016, pp. 455–472.

[SNV07] Laszlo Szathmary, Amedeo Napoli, and Petko Valtchev, *Towards rare itemset mining*, 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007), vol. 1, IEEE, 2007, pp. 305–312.

[SR14] Arnaud Soulet and François Rioult, *Efficiently depth-first minimal pattern mining*, Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2014, pp. 28–39.

[SS08] Christian Schulte and Peter J Stuckey, *Efficient constraint propagation engines*, ACM Transactions on Programming Languages and Systems (TOPLAS) **31** (2008), no. 1, 1–43.

[Tan15] Ole Tange, *Gnu parallel-the command-line power tool.; login: The usenix magazine, 36 (1): 42–47, feb 2011*, URL http://www.gnu.org/s/parallel **101** (2015).

[THLN01] Anthony KH Tung, Jiawei Han, Laks VS Lakshmanan, and Raymond T Ng, *Constraint-based clustering in large databases*, International Conference on Database Theory, Springer, 2001, pp. 405–419.

[TK01] Pang-Ning Tan and Vipin Kumar, *Mining indirect associations in web data*, International Workshop on Mining Web Log Data Across All Customers Touch Points, Springer, 2001, pp. 145–166.

[TS16] Takahisa Toda and Takehide Soh, *Implementing efficient all solutions sat solvers*, Journal of Experimental Algorithmics (JEA) **21** (2016), 1–44.

[UKA+04] Takeaki Uno, Masashi Kiyomi, Hiroki Arimura, et al., *Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets*, Fimi, vol. 126, 2004.

[Uma01] Christopher Umans, *The minimum equivalent dnf problem and shortest implicants*, Journal of Computer and System Sciences **63** (2001), no. 4, 597–611.

[VH99] Pascal Van Hentenryck, *The opl optimization programming language*, MIT press, 1999.

[XHLB10] Lin Xu, Holger Hoos, and Kevin Leyton-Brown, *Hydra: Automatically configuring algorithms for portfolio-based selection*, Twenty-Fourth AAAI Conference on Artificial Intelligence, 2010.

[Zak00] Mohammed Javeed Zaki, *Scalable algorithms for association mining*, IEEE transactions on knowledge and data engineering **12** (2000), no. 3, 372–390.

Appendix A

Early Ad-Hoc Mining Results

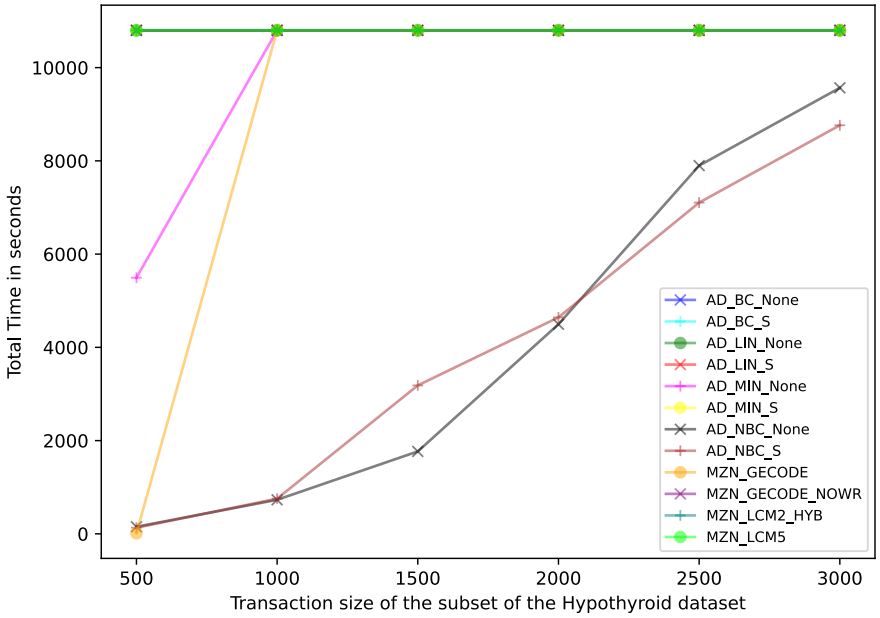


Figure A.1: Full plot of the preliminary results for the ad-hoc iterative miner and its comparison to a handful of MiningZinc options. The transaction size of the dataset indicates the number of transactions in the dataset. Time values are in seconds. The timeout threshold is 3 hours.

Appendix B

Essence CDP+I Models for Pattern Mining Problems

```
language Essence 1.3
given db : mset of set of int
given min_freq : int
letting db_minValue be  min([val | entry <- db, val <- entry])
letting db_maxValue be max([val | entry <- db, val <- entry])
letting db_maxEntrySize be max([ |entry| | entry <- db ])
letting db_row_size be |db|
given utility_values : matrix indexed by [int(db_minValue..db_maxValue)] of int
given cost_values : matrix indexed by [int(db_minValue..db_maxValue)] of int
given min_utility : int
given max_cost : int
find db_minValue_var : int(db_minValue)
find db_maxValue_var : int(db_maxValue)
find db_maxEntrySize_var : int(db_maxEntrySize)
find current_size: int(0..db_maxEntrySize)
find freq_items : (set (maxSize db_maxEntrySize) of int(db_minValue..db_maxValue), int
    (1..db_row_size))
such that
    |freq_items[1]| = current_size
such that
    (sum entry in db . toInt(freq_items[1] subsetEq entry)) = freq_items[2]
such that
    freq_items[2] >= min_freq
such that
    (sum item in freq_items[1] . utility_values[item]) >= min_utility
such that
dominanceRelation (freq_items[1] subsetEq fromSolution(freq_items[1])) -> (freq_items
    [2] > fromSolution(freq_items[2]))
incomparabilityFunction descending current_size
```

Figure B.1: Full ESSENCE CDP+I model for CFIM.

```
language Essence 1.3
given db : mset of set of int
given min_freq : int
letting db_minValue be  min([val | entry <- db, val <- entry])
letting db_maxValue be max([val | entry <- db, val <- entry])
letting db_maxEntrySize be max([ |entry| | entry <- db ])
letting db_row_size be |db|
given utility_values : matrix indexed by [int(db_minValue..db_maxValue)] of int
given cost_values : matrix indexed by [int(db_minValue..db_maxValue)] of int
given min_utility : int
given max_cost : int
find db_minValue_var : int(db_minValue)
find db_maxValue_var : int(db_maxValue)
find db_maxEntrySize_var : int(db_maxEntrySize)
find current_size: int(0..db_maxEntrySize)
find freq_items : (set (maxSize db_maxEntrySize) of int(db_minValue..db_maxValue), int
    (1..db_row_size))
such that
    |freq_items[1]| = current_size
such that
    (sum entry in db . toInt(freq_items[1] subsetEq entry)) = freq_items[2]
such that
    freq_items[2] >= min_freq
such that
    (sum item in freq_items[1] . utility_values[item]) >= min_utility
such that
    (sum item in freq_items[1] . cost_values[item]) <= max_cost
dominanceRelation (fromSolution(freq_items[1]) subsetEq freq_items[1]) -> (freq_items
    [2] < fromSolution(freq_items[2]))
incomparabilityFunction ascending current_size
```

Figure B.2: Full ESSENCE CDP+I model for GFIM.

```
language Essence 1.3
given db : mset of set of int
given min_freq : int
letting db_minValue be  min([val | entry <- db, val <- entry])
letting db_maxValue be max([val | entry <- db, val <- entry])
letting db_maxEntrySize be max([ |entry| | entry <- db ])
letting db_row_size be |db|
given utility_values : matrix indexed by [int(db_minValue..db_maxValue)] of int
given cost_values : matrix indexed by [int(db_minValue..db_maxValue)] of int
given min_utility : int
given max_cost : int
find db_minValue_var : int(db_minValue)
find db_maxValue_var : int(db_maxValue)
find db_maxEntrySize_var : int(db_maxEntrySize)
find current_size: int(0..db_maxEntrySize)
find freq_items : (set (maxSize db_maxEntrySize) of int(db_minValue..db_maxValue), int
    (1..db_row_size))
such that
    |freq_items[1]| = current_size
such that
    (sum entry in db . toInt(freq_items[1] subsetEq entry)) = freq_items[2]
such that
    freq_items[2] < min_freq
such that
    (sum item in freq_items[1] . utility_values[item]) >= min_utility
such that
    (sum item in freq_items[1] . cost_values[item]) <= max_cost
dominanceRelation !(fromSolution(freq_items[1]) subsetEq freq_items[1])
incomparabilityFunction ascending current_size
```

Figure B.3: Full ESSENCE CDP+I model for MRIM.

```
language Essence 1.3
given db : mset of record { itemset : set of int, class : int }
given min_freq : int
letting db_minValue be  min([val | entry <- db, val <- entry[itemset]])
letting db_maxValue be max([val | entry <- db, val <- entry[itemset]])
letting db_maxEntrySize be max([ |entry[itemset]| | entry <- db ])
letting db_row_size be |db|
given utility_values : matrix indexed by [int(db_minValue..db_maxValue)] of int
given cost_values : matrix indexed by [int(db_minValue..db_maxValue)] of int
given min_utility : int
given max_cost : int
letting support_domain be domain int(1..db_row_size)
find freq_items : record {
                        itemset : set (minSize 1, maxSize db_maxEntrySize) of int(
                            db_minValue..db_maxValue),
                        support_pos : support_domain,
                        support_neg : support_domain
                    }
such that
    (sum entry in db . toInt(freq_items[itemset] subsetEq entry[itemset] /\ entry[
        class] = 1)) = freq_items[support_pos]
such that
    (sum entry in db . toInt(freq_items[itemset] subsetEq entry[itemset] /\ entry[
        class] = 0)) = freq_items[support_neg]
such that
    freq_items[support_pos] - freq_items[support_neg] > min_freq

such that
    (sum item in freq_items[itemset] . utility_values[item]) >= min_utility
such that
    (sum item in freq_items[itemset] . cost_values[item]) <= max_cost
dominanceRelation (freq_items[itemset] subsetEq fromSolution(freq_items[itemset])) ->
    (freq_items[support_pos] > fromSolution(freq_items[support_pos]))
incomparabilityFunction descending |freq_items[itemset]|
```

Figure B.4: Full ESSENCE CDP+I model for CDIM.

```
language Essence 1.3
given db : sequence of record { itemset : set of int, class : int }
given min_freq : int
letting db_minValue be  min([val | (_, entry) <- db, val <- entry[itemset]])
letting db_maxValue be max([val | (_, entry) <- db, val <- entry[itemset]])
letting db_maxEntrySize be max([ |entry[itemset]| | (_, entry) <- db ])
letting db_row_size be |db|
given utility_values : matrix indexed by [int(db_minValue..db_maxValue)] of int
given cost_values : matrix indexed by [int(db_minValue..db_maxValue)] of int
given min_utility : int
given max_cost : int
letting support_domain be domain int(1..db_row_size)
find freq_items : record {
                        itemset : set (minSize 1, maxSize db_maxEntrySize) of int(
                            db_minValue..db_maxValue),
                        cover_pos : set (maxSize db_row_size) of support_domain,
                        cover_neg : set (maxSize db_row_size) of support_domain
                    }
such that
    forAll (row, entry) in db .
        row in freq_items[cover_neg]  <-> (entry[itemset] supsetEq freq_items[itemset]
             /\ entry[class] = 0)
such that
    forAll (row, entry) in db .
        row in freq_items[cover_pos]  <-> (entry[itemset] supsetEq freq_items[itemset]
             /\ entry[class] = 1)
such that
    |freq_items[cover_pos]| > min_freq
such that
    (sum item in freq_items[itemset] . utility_values[item]) >= min_utility
such that
    (sum item in freq_items[itemset] . cost_values[item]) <= max_cost
dominanceRelation !((freq_items[cover_pos] subsetEq fromSolution(freq_items[cover_pos
    ]))
/\ ( freq_items[cover_neg] supsetEq fromSolution(freq_items[cover_neg]) )
/\ ((freq_items[cover_pos] union freq_items[cover_neg] = fromSolution(freq_items[
    cover_pos]) union fromSolution(freq_items[cover_neg]) ) -> (freq_items[itemset]
    subsetEq fromSolution(freq_items[itemset]) )))
incomparabilityFunction descending |freq_items[itemset]|
```

Figure B.5: Full ESSENCE CDP+I model for RSD.

APPENDIX C

ESSENCE SPECIFICATION FOR MRCPSP

```
language Essence 1.3
given nonRenewableResources new type enum
given renewableResources new type enum
given jobs new type enum
given startDummy, endDummy : jobs
given modes new type enum
given renewableLimits: function (total) renewableResources --> int
given nonRenewableLimits  : function (total) nonRenewableResources --> int
given successors : function (total) jobs --> set of jobs
given renewableResourceUsage :
    function  (jobs, modes, renewableResources) --> int
given nonRenewableResourceUsage :
    function (jobs, modes, nonRenewableResources) --> int
given duration : function  (jobs,modes) --> int
given horizon : int
letting timesRange be domain int(1..horizon)
find start: function (total)  jobs --> timesRange
find mode: function (total) jobs --> modes
find jobActive: function (total) (jobs,timesRange) --> bool
such that
forAll job : jobs .
    forAll jobSuccessor in successors(job) .
        start(jobSuccessor) >= start(job) + duration((job,mode(job)))
such that
forAll job : jobs .
    forAll time : timesRange .
        jobActive((job,time)) <->(
            time >= start(job) /\ time < start(job) + duration((job,mode(job))))
such that
forAll resource : nonRenewableResources .
    sum([nonRenewableResourceUsage((job, mode(job), resource) )| job : jobs])
        <= nonRenewableLimits(resource)
such that
forAll resource : renewableResources .
    forAll time : timesRange .
        sum([renewableResourceUsage((job,mode(job),resource)) |
            job : jobs, jobActive((job,time))])
                <= renewableLimits(resource)
such that
start(startDummy)=1
minimising start(endDummy)
```

Figure C.1: ESSENCE specification for MRCPSP.

Appendix D

Low-level Data Structures for Experiment Collection

```
#[derive(Serialize, Deserialize)]
#[serde(tag = "type")]
enum SolveInformation {
    #[serde(rename = "SUCCESS")]
    Success {
        total_solver_time: f64,
        total_sr_time: f64,
        total_nodes: Option<u64>,
        nb_solutions: u64,
        seed: Option<f64>,
        memory_limit: u64,
        time_limit: u64,
        machine_info: String,
        level_info: Box<LevelInformation>,
        ...
    },
    #[serde(rename = "TIMEOUT")]
    Timeout {
        seed: Option<f64>,
        memory_limit: u64,
        time_limit: u64,
        machine_info: String,
    },
    ...
}
```

Figure D.1: An excerpt of the `SolveInformation` data structure in `rrr` which benefits from strong typing and enum for heterogeneous parsing.

```
pub struct DBRow {
    exp_id: String,
    config_id: String,
    result_type: Result,
    measured_time: f64,
    nb_solutions: Option<u64>,
    machine_info: String,
    memory_limit: u64,
    seed: Option<f64>,
}
```

Figure D.2: `DBRow` data structure in `rrr` to represent a small subset of the experiment results.

```rust
#[derive(Serialize)]
pub struct PlotConfigView<'a> {
    nb_data_points: u8,
    total_solver_time_mean: Option<f64>,
    total_solver_time_best: Option<f64>,
    total_sr_time_mean: Option<f64>,
    total_sr_time_best: Option<f64>,
    total_nodes_mean: Option<f64>,
    total_nodes_best: Option<f64>,
    nb_solutions: Option<u64>,
    nb_levels: Option<u16>,
    levels_best_solver_time: &'a Option<HashMap<String, f64>>,
    levels_best_sat_clauses: &'a Option<HashMap<String, u64>>,
    levels_best_satv: &'a Option<HashMap<String, u64>>,
    levels_best_sat_learnt_clauses: &'a Option<HashMap<String, u64>>,
    levels_best_nodes: &'a Option<HashMap<String, u64>>,
    levels_best_nb_sols: &'a Option<HashMap<String, u64>>,
}
```

Figure D.3: `PlotConfigView` data structure in `rrr` to represent the experiment data with almost zero copywhere data representation is more suitable for plotting scripts.

Appendix E

Full Essence Features for a Pattern Mining Problem

- "db_cardinality",
- "db_cardinality_ratioToMax",
- "db_cardinality_intIsOffByOne",
- "db_cardinality_intIsRepeated",
- "utility_values_cardinality",
- "utility_values_cardinality_ratioToMax",
- "utility_values_cardinality_intIsOffByOne",
- "utility_values_cardinality_intIsRepeated",
- "utility_values_1_min",
- "utility_values_1_min_ratioToMax",
- "utility_values_1_min_intIsOffByOne",
- "utility_values_1_min_intIsRepeated",
- "utility_values_1_max",
- "utility_values_1_max_ratioToMax",
- "utility_values_1_max_intIsOffByOne",
- "utility_values_1_max_intIsRepeated",
- "cost_values_cardinality",
- "cost_values_cardinality_ratioToMax",
- "cost_values_cardinality_intIsOffByOne",
- "cost_values_cardinality_intIsRepeated",
- "cost_values_1_min",
- "cost_values_1_min_ratioToMax",
- "cost_values_1_min_intIsOffByOne",
- "cost_values_1_min_intIsRepeated",
- "cost_values_1_max",
- "cost_values_1_max_ratioToMax",
- "cost_values_1_max_intIsOffByOne",
- "cost_values_1_max_intIsRepeated",
- "min_utility_intValue",
- "min_utility_intValue_ratioToMax",
- "min_utility_intValue_intIsOffByOne",
- "min_utility_intValue_intIsRepeated",
- "min_utility_isEven",
- "min_utility_isSquare",

- "min_utility_isPrime",
- "max_cost_intValue",
- "max_cost_intValue_ratioToMax",
- "max_cost_intValue_intIsOffByOne",
- "max_cost_intValue_intIsRepeated",
- "max_cost_isEven",
- "max_cost_isSquare",
- "max_cost_isPrime",
- "db_cardinality_utility_values_cardinality_-intIntRatio",
- "db_cardinality_utility_values_1_min_intIntRatio",
- "db_cardinality_utility_values_1_max_intIntRatio",
- "db_cardinality_min_utility_intValue_intIntRatio",
- "db_cardinality_max_cost_intValue_intIntRatio",
- "utility_values_1_min_utility_values_cardinality_-intIntRatio",
- "utility_values_1_max_utility_values_cardinality_-intIntRatio",
- "utility_values_1_max_utility_values_1_min_-intIntRatio",
- "cost_values_cardinality_db_cardinality_-intIntRatio",
- "cost_values_cardinality_utility_values_-cardinality_intIntRatio",
- "cost_values_cardinality_utility_values_1_min_-intIntRatio",
- "cost_values_cardinality_utility_values_1_max_-intIntRatio",
- "cost_values_cardinality_min_utility_intValue_-intIntRatio",
- "cost_values_cardinality_max_cost_intValue_-intIntRatio",
- "cost_values_1_min_db_cardinality_intIntRatio",
- "cost_values_1_min_utility_values_cardinality_-intIntRatio",
- "cost_values_1_min_utility_values_1_min_-intIntRatio",
- "cost_values_1_min_utility_values_1_max_-intIntRatio",
- "cost_values_1_min_cost_values_cardinality_-intIntRatio",
- "cost_values_1_min_min_utility_intValue_-intIntRatio",
- "cost_values_1_min_max_cost_intValue_-intIntRatio",
- "cost_values_1_max_db_cardinality_intIntRatio",
- "cost_values_1_max_utility_values_cardinality_-intIntRatio",
- "cost_values_1_max_utility_values_1_min_-intIntRatio",
- "cost_values_1_max_utility_values_1_max_-intIntRatio",
- "cost_values_1_max_cost_values_cardinality_-intIntRatio",
- "cost_values_1_max_cost_values_1_min_-intIntRatio",
- "cost_values_1_max_min_utility_intValue_-intIntRatio",
- "cost_values_1_max_max_cost_intValue_-intIntRatio",
- "min_utility_intValue_utility_values_cardinality_-intIntRatio",
- "min_utility_intValue_utility_values_1_min_-intIntRatio",
- "min_utility_intValue_utility_values_1_max_-intIntRatio",
- "max_cost_intValue_utility_values_cardinality_-intIntRatio",
- "max_cost_intValue_utility_values_1_min_-intIntRatio",
- "max_cost_intValue_utility_values_1_max_-intIntRatio",
- "max_cost_intValue_min_utility_intValue_-intIntRatio",
- "utility_values_1_median",
- "utility_values_1_mean",
- "utility_values_1_stdDev",
- "utility_values_1_harmonicMean",
- "utility_values_1_geometricMean",
- "utility_values_1_skewness",
- "utility_values_1_kurtosis",
- "cost_values_1_median",
- "cost_values_1_mean",
- "cost_values_1_stdDev",
- "cost_values_1_harmonicMean",
- "cost_values_1_geometricMean",
- "cost_values_1_skewness",
- "cost_values_1_kurtosis",

APPENDIX F

CODE SNIPPETS FOR SKLEARN/AUTOSKLEARN

```
X_train, X_test, y_train, y_test = \
    sklearn.model_selection.train_test_split(X, y, random_state=1)
```

Figure F.1: Training/Test set splits in SKLearn in python. While X represents the feature set for the instances, Y represents the classification output attached to the input.

```
clf = RandomForestClassifier(
    n_estimators=100,
    random_state=rnd_seed,
    n_jobs=10,
)
clf.fit(X_train, y_train)
```

Figure F.2: Random forest classifier initialisation and training using SKLearn with only one significant parameter for the number of estimators.

```
automl = autosklearn.classification.AutoSklearnClassifier(
    time_left_for_this_task=600,
    per_run_time_limit=150,
    n_jobs=10,
    memory_limit=24096,
    seed=rnd_seed,
    # ensemble_size=5,
)
automl.fit(X_train, y_train)
```

Figure F.3: Autosklearn portfolio classification initialisation and training. The commented-out ensemble size parameter is the default value of 5.